A0A0¡”80

LLVM Documentation

Release 3.6

LLVM project

2014-10-20

CONTENTS

Warning: If you are using a released version of LLVM, see the download page to find your documentation.

The LLVM compiler infrastructure supports a wide range of projects, from industrial strength compilers to specialized JIT applications to small research projects.

Similarly, documentation is broken down into several high-level groupings targeted at different audiences:

CHAPTER

ONE

LLVM DESIGN & OVERVIEW

Several introductory papers and presentations.

1.1 LLVM Language Reference Manual

1.1.1 Abstract

This document is a reference manual for the LLVM assembly language. LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy.

1.1.2 Introduction

The LLVM code representation is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. This allows LLVM to provide a powerful intermediate representation for efficient compiler transformations and analysis, while providing a natural means to debug and visualize the transformations. The three different forms of LLVM are all equivalent. This document describes the human readable representation and notation.

The LLVM representation aims to be light-weight and low-level while being expressive, typed, and extensible at the same time. It aims to be a "universal IR" of sorts, by being at a low enough level that high-level ideas may be cleanly mapped to it (similar to how microprocessors are "universal IR's", allowing many source languages to be mapped to them). By providing type information, LLVM can be used as the target of optimizations: for example, through pointer analysis, it can be proven that a C automatic variable is never accessed outside of the current function, allowing it to be promoted to a simple SSA value instead of a memory location.

Well-Formedness

It is important to note that this document describes 'well formed' LLVM assembly language. There is a difference between what the parser accepts and what is considered 'well formed'. For example, the following instruction is syntactically okay, but not well formed:

```
%x = add i32 1, %x
```

because the definition of `%x` does not dominate all of its uses. The LLVM infrastructure provides a verification pass that may be used to verify that an LLVM module is well formed. This pass is automatically run by the parser after parsing input assembly and by the optimizer before it outputs bitcode. The violations pointed out by the verifier pass indicate bugs in transformation passes or input to the parser.

1.1.3 Identifiers

LLVM identifiers come in two basic types: global and local. Global identifiers (functions, global variables) begin with the `'@'` character. Local identifiers (register names, types) begin with the `'%'` character. Additionally, there are three different formats for identifiers, for different purposes:

1. Named values are represented as a string of characters with their prefix. For example, `%foo`, `@DivisionByZero`, `%a.really.long.identifier`. The actual regular expression used is '`[%@][a-zA-Z$._][a-zA-Z$._0-9]*`'. Identifiers that require other characters in their names can be surrounded with quotes. Special characters may be escaped using `"\xx"` where `xx` is the ASCII code for the character in hexadecimal. In this way, any character can be used in a name value, even quotes themselves. The `"\01"` prefix can be used on global variables to suppress mangling.
2. Unnamed values are represented as an unsigned numeric value with their prefix. For example, `%12`, `@2`, `%44`.
3. Constants, which are described in the section Constants below.

LLVM requires that values start with a prefix for two reasons: Compilers don't need to worry about name clashes with reserved words, and the set of reserved words may be expanded in the future without penalty. Additionally, unnamed identifiers allow a compiler to quickly come up with a temporary variable without having to avoid symbol table conflicts.

Reserved words in LLVM are very similar to reserved words in other languages. There are keywords for different opcodes ('`add`', '`bitcast`', '`ret`', etc...), for primitive type names ('`void`', '`i32`', etc...), and others. These reserved words cannot conflict with variable names, because none of them start with a prefix character (`'%'` or `'@'`).

Here is an example of LLVM code to multiply the integer variable '`%X`' by 8:

The easy way:

```
%result = mul i32 %X, 8
```

After strength reduction:

```
%result = shl i32 %X, 3
```

And the hard way:

```
%0 = add i32 %X, %X           ; yields i32:%0
%1 = add i32 %0, %0           ; yields i32:%1
%result = add i32 %1, %1
```

This last way of multiplying `%X` by 8 illustrates several important lexical features of LLVM:

1. Comments are delimited with a '`;`' and go until the end of line.
2. Unnamed temporaries are created when the result of a computation is not assigned to a named value.
3. Unnamed temporaries are numbered sequentially (using a per-function incrementing counter, starting with 0). Note that basic blocks and unnamed function parameters are included in this numbering. For example, if the entry basic block is not given a label name and all function parameters are named, then it will get number 0.

It also shows a convention that we follow in this document. When demonstrating instructions, we will follow an instruction with a comment that defines the type and name of value produced.

1.1.4 High Level Structure

Module Structure

LLVM programs are composed of `Module`'s, each of which is a translation unit of the input programs. Each module consists of functions, global variables, and symbol table entries. Modules may be combined together with the LLVM linker, which merges function (and global variable) definitions, resolves forward declarations, and merges symbol table entries. Here is an example of the "hello world" module:

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() {   ; i32()*
  ; Convert [13 x i8]* to i8  *...
  %cast210 = getelementptr [13 x i8]* @.str, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast210)
  ret i32 0
}

; Named metadata
!0 = metadata !{i32 42, null, metadata !"string"}
!foo = !{!0}
```

This example is made up of a *global variable* named "`.str`", an external declaration of the "`puts`" function, a *function definition* for "`main`" and *named metadata* "`foo`".

In general, a module is made up of a list of global values (where both functions and global variables are global values). Global values are represented by a pointer to a memory location (in this case, a pointer to an array of char, and a pointer to a function), and have one of the following *linkage types*.

Linkage Types

All Global Variables and Functions have one of the following types of linkage:

private Global values with "private" linkage are only directly accessible by objects in the current module. In particular, linking code into a module with an private global value may cause the private to be renamed as necessary to avoid collisions. Because the symbol is private to the module, all references can be updated. This doesn't show up in any symbol table in the object file.

internal Similar to private, but the value shows as a local symbol (STB_LOCAL in the case of ELF) in the object file. This corresponds to the notion of the 'static' keyword in C.

available_externally Globals with "available_externally" linkage are never emitted into the object file corresponding to the LLVM module. They exist to allow inlining and other optimizations to take place given knowledge of the definition of the global, which is known to be somewhere outside the module. Globals with available_externally linkage are allowed to be discarded at will, and are otherwise the same as linkonce_odr. This linkage type is only allowed on definitions, not declarations.

linkonce Globals with "linkonce" linkage are merged with other globals of the same name when linkage occurs. This can be used to implement some forms of inline functions, templates, or other code which must be generated in each translation unit that uses it, but where the body may be overridden with a more definitive definition later. Unreferenced linkonce globals are allowed to be discarded. Note that linkonce linkage does not actually allow the optimizer to inline the body of this function into callers because it doesn't know if this definition of the function is the definitive definition within the program or whether it will be overridden by a stronger definition. To enable inlining and other optimizations, use "linkonce_odr" linkage.

weak "weak" linkage has the same merging semantics as linkonce linkage, except that unreferenced globals with weak linkage may not be discarded. This is used for globals that are declared "weak" in C source code.

common "common" linkage is most similar to "weak" linkage, but they are used for tentative definitions in C, such as "int X;" at global scope. Symbols with "common" linkage are merged in the same way as weak symbols, and they may not be deleted if unreferenced. common symbols may not have an explicit section, must have a zero initializer, and may not be marked '*constant*'. Functions and aliases may not have common linkage.

appending "appending" linkage may only be applied to global variables of pointer to array type. When two global variables with appending linkage are linked together, the two global arrays are appended together. This is the LLVM, typesafe, equivalent of having the system linker append together "sections" with identical names when .o files are linked.

extern_weak The semantics of this linkage follow the ELF object file model: the symbol is weak until linked, if not linked, the symbol becomes null instead of being an undefined reference.

linkonce_odr, weak_odr Some languages allow differing globals to be merged, such as two functions with different semantics. Other languages, such as C++, ensure that only equivalent globals are ever merged (the "one definition rule" — "ODR"). Such languages can use the linkonce_odr and weak_odr linkage types to indicate that the global will only be merged with equivalent globals. These linkage types are otherwise the same as their non-odr versions.

external If none of the above identifiers are used, the global is externally visible, meaning that it participates in linkage and can be used to resolve external symbol references.

It is illegal for a function *declaration* to have any linkage type other than external or extern_weak.

Calling Conventions

LLVM *functions*, *calls* and *invokes* can all have an optional calling convention specified for the call. The calling convention of any pair of dynamic caller/callee must match, or the behavior of the program is undefined. The following calling conventions are supported by LLVM, and more may be added in the future:

"`ccc`" - The C calling convention This calling convention (the default if no other calling convention is specified) matches the target C calling conventions. This calling convention supports varargs function calls and tolerates some mismatch in the declared prototype and implemented declaration of the function (as does normal C).

"`fastcc`" - The fast calling convention This calling convention attempts to make calls as fast as possible (e.g. by passing things in registers). This calling convention allows the target to use whatever tricks it wants to produce fast code for the target, without having to conform to an externally specified ABI (Application Binary Interface). Tail calls can only be optimized when this, the GHC or the HiPE convention is used. This calling convention does not support varargs and requires the prototype of all callees to exactly match the prototype of the function definition.

"`coldcc`" - The cold calling convention This calling convention attempts to make code in the caller as efficient as possible under the assumption that the call is not commonly executed. As such, these calls often preserve all registers so that the call does not break any live ranges in the caller side. This calling convention does not support varargs and requires the prototype of all callees to exactly match the prototype of the function definition. Furthermore the inliner doesn't consider such function calls for inlining.

"`cc 10`" - GHC convention This calling convention has been implemented specifically for use by the Glasgow Haskell Compiler (GHC). It passes everything in registers, going to extremes to achieve this by disabling callee save registers. This calling convention should not be used lightly but only for specific situations such as an alternative to the *register pinning* performance technique often used when implementing functional programming languages. At the moment only X86 supports this convention and it has the following limitations:

- On *X86-32* only supports up to 4 bit type parameters. No floating point types are supported.
- On *X86-64* only supports up to 10 bit type parameters and 6 floating point parameters.

This calling convention supports tail call optimization but requires both the caller and callee are using it.

"`cc 11`" - The HiPE calling convention This calling convention has been implemented specifically for use by the High-Performance Erlang (HiPE) compiler, *the* native code compiler of the Ericsson's Open Source Erlang/OTP system. It uses more registers for argument passing than the ordinary C calling convention and defines no callee-saved registers. The calling convention properly supports tail call optimization but requires that both the caller and the callee use it. It uses a *register pinning* mechanism, similar to GHC's convention, for keeping frequently accessed runtime components pinned to specific hardware registers. At the moment only X86 supports this convention (both 32 and 64 bit).

"`webkit_jscc`" - WebKit's JavaScript calling convention This calling convention has been implemented for WebKit FTL JIT. It passes arguments on the stack right to left (as cdecl does), and returns a value in the platform's customary return register.

"`anyregcc`" - Dynamic calling convention for code patching This is a special convention that supports patching an arbitrary code sequence in place of a call site. This convention forces the call arguments into registers but allows them to be dynamcially allocated. This can currently only be used with calls to llvm.experimental.patchpoint because only this intrinsic records the location of its arguments in a side table. See *Stack maps and patch points in LLVM*.

"`preserve_mostcc`" - The *PreserveMost* calling convention This calling convention attempts to make the code in the caller as little intrusive as possible. This calling convention behaves identical to the *C* calling convention on how arguments and return values are passed, but it uses a different set of caller/callee-saved registers. This alleviates the burden of saving and recovering a large register set before and after the call in the caller. If the arguments are passed in callee-saved registers, then they will be preserved by the callee across the call. This doesn't apply for values returned in callee-saved registers.

- On X86-64 the callee preserves all general purpose registers, except for R11. R11 can be used as a scratch register. Floating-point registers (XMMs/YMMs) are not preserved and need to be saved by the caller.

The idea behind this convention is to support calls to runtime functions that have a hot path and a cold path. The hot path is usually a small piece of code that doesn't many registers. The cold path might need to call out to another function and therefore only needs to preserve the caller-saved registers, which haven't already been saved by the caller. The *PreserveMost* calling convention is very similar to the *cold* calling convention in terms of caller/callee-saved registers, but they are used for different types of function calls. *coldcc* is for function calls that are rarely executed, whereas *preserve_mostcc* function calls are intended to be on the hot path and definitely executed a lot. Furthermore *preserve_mostcc* doesn't prevent the inliner from inlining the function call.

This calling convention will be used by a future version of the ObjectiveC runtime and should therefore still be considered experimental at this time. Although this convention was created to optimize certain runtime calls to the ObjectiveC runtime, it is not limited to this runtime and might be used by other runtimes in the future too. The current implementation only supports X86-64, but the intention is to support more architectures in the future.

"`preserve_allcc`" - The *PreserveAll* calling convention This calling convention attempts to make the code in the caller even less intrusive than the *PreserveMost* calling convention. This calling convention also behaves identical to the *C* calling convention on how arguments and return values are passed, but it uses a different set of caller/callee-saved registers. This removes the burden of saving and recovering a large register set before and after the call in the caller. If the arguments are passed in callee-saved registers, then they will be preserved by the callee across the call. This doesn't apply for values returned in callee-saved registers.

- On X86-64 the callee preserves all general purpose registers, except for R11. R11 can be used as a scratch register. Furthermore it also preserves all floating-point registers (XMMs/YMMs).

The idea behind this convention is to support calls to runtime functions that don't need to call out to any other functions.

This calling convention, like the *PreserveMost* calling convention, will be used by a future version of the ObjectiveC runtime and should be considered experimental at this time.

"`cc <n>`" - Numbered convention Any calling convention may be specified by number, allowing target-specific calling conventions to be used. Target specific calling conventions start at 64.

More calling conventions can be added/defined on an as-needed basis, to support Pascal conventions or any other well-known target-independent convention.

Visibility Styles

All Global Variables and Functions have one of the following visibility styles:

"`default`" - Default style On targets that use the ELF object file format, default visibility means that the declaration is visible to other modules and, in shared libraries, means that the declared entity may be overridden. On Darwin, default visibility means that the declaration is visible to other modules. Default visibility corresponds to "external linkage" in the language.

"`hidden`" - Hidden style Two declarations of an object with hidden visibility refer to the same object if they are in the same shared object. Usually, hidden visibility indicates that the symbol will not be placed into the dynamic symbol table, so no other module (executable or shared library) can reference it directly.

"`protected`" - Protected style On ELF, protected visibility indicates that the symbol will be placed in the dynamic symbol table, but that references within the defining module will bind to the local symbol. That is, the symbol cannot be overridden by another module.

A symbol with `internal` or `private` linkage must have `default` visibility.

DLL Storage Classes

All Global Variables, Functions and Aliases can have one of the following DLL storage class:

dllimport "dllimport" causes the compiler to reference a function or variable via a global pointer to a pointer that is set up by the DLL exporting the symbol. On Microsoft Windows targets, the pointer name is formed by combining __imp_ and the function or variable name.

dllexport "dllexport" causes the compiler to provide a global pointer to a pointer in a DLL, so that it can be referenced with the dllimport attribute. On Microsoft Windows targets, the pointer name is formed by combining __imp_ and the function or variable name. Since this storage class exists for defining a dll interface, the compiler, assembler and linker know it is externally referenced and must refrain from deleting the symbol.

Thread Local Storage Models

A variable may be defined as thread_local, which means that it will not be shared by threads (each thread will have a separated copy of the variable). Not all targets support thread-local variables. Optionally, a TLS model may be specified:

localdynamic For variables that are only used within the current shared library.

initialexec For variables in modules that will not be loaded dynamically.

localexec For variables defined in the executable and only used within it.

If no explicit model is given, the "general dynamic" model is used.

The models correspond to the ELF TLS models; see ELF Handling For Thread-Local Storage for more information on under which circumstances the different models may be used. The target may choose a different TLS model if the specified model is not supported, or if a better choice of model can be made.

A model can also be specified in a alias, but then it only governs how the alias is accessed. It will not have any effect in the aliasee.

Structure Types

LLVM IR allows you to specify both "identified" and "literal" *structure types*. Literal types are uniqued structurally, but identified types are never uniqued. An *opaque structural type* can also be used to forward declare a type that is not yet available.

An example of a identified structure specification is:

```
%mytype = type { %mytype*, i32 }
```

Prior to the LLVM 3.0 release, identified types were structurally uniqued. Only literal types are uniqued in recent versions of LLVM.

Global Variables

Global variables define regions of memory allocated at compilation time instead of run-time.

Global variables definitions must be initialized.

Global variables in other translation units can also be declared, in which case they don't have an initializer.

Either global variable definitions or declarations may have an explicit section to be placed in and may have an optional explicit alignment specified.

A variable may be defined as a global `constant`, which indicates that the contents of the variable will **never** be modified (enabling better optimization, allowing the global data to be placed in the read-only section of an executable, etc). Note that variables that need runtime initialization cannot be marked `constant` as there is a store to the variable.

LLVM explicitly allows *declarations* of global variables to be marked constant, even if the final definition of the global is not. This capability can be used to enable slightly better optimization of the program, but requires the language definition to guarantee that optimizations based on the 'constantness' are valid for the translation units that do not include the definition.

As SSA values, global variables define pointer values that are in scope (i.e. they dominate) all basic blocks in the program. Global variables always define a pointer to their "content" type because they describe a region of memory, and all memory objects in LLVM are accessed through pointers.

Global variables can be marked with `unnamed_addr` which indicates that the address is not significant, only the content. Constants marked like this can be merged with other constants if they have the same initializer. Note that a constant with significant address *can* be merged with a `unnamed_addr` constant, the result being a constant whose address is significant.

A global variable may be declared to reside in a target-specific numbered address space. For targets that support them, address spaces may affect how optimizations are performed and/or what target instructions are used to access the variable. The default address space is zero. The address space qualifier must precede any other attributes.

LLVM allows an explicit section to be specified for globals. If the target supports it, it will emit globals to the section specified. Additionally, the global can placed in a comdat if the target has the necessary support.

By default, global initializers are optimized by assuming that global variables defined within the module are not modified from their initial values before the start of the global initializer. This is true even for variables potentially accessible from outside the module, including those with external linkage or appearing in `@llvm.used` or dllexported variables. This assumption may be suppressed by marking the variable with `externally_initialized`.

An explicit alignment may be specified for a global, which must be a power of 2. If not present, or if the alignment is set to zero, the alignment of the global is set by the target to whatever it feels convenient. If an explicit alignment is specified, the global is forced to have exactly that alignment. Targets and optimizers are not allowed to over-align the global if the global has an assigned section. In this case, the extra alignment could be observable: for example, code could assume that the globals are densely packed in their section and try to iterate over them as an array, alignment padding would break this iteration. The maximum alignment is `1 << 29`.

Globals can also have a *DLL storage class*.

Variables and aliasaes can have a *Thread Local Storage Model*.

Syntax:

```
[@<GlobalVarName> =] [Linkage] [Visibility] [DLLStorageClass] [ThreadLocal]
                     [unnamed_addr] [AddrSpace] [ExternallyInitialized]
                     <global | constant> <Type> [<InitializerConstant>]
                     [, section "name"] [, align <Alignment>]
```

For example, the following defines a global in a numbered address space with an initializer, section, and alignment:

```
@G = addrspace(5) constant float 1.0, section "foo", align 4
```

The following example just declares a global variable

```
@G = external global i32
```

The following example defines a thread-local global with the `initialexec` TLS model:

```
@G = thread_local(initialexec) global i32 0, align 4
```

Functions

LLVM function definitions consist of the "`define`" keyword, an optional *linkage type*, an optional *visibility style*, an optional *DLL storage class*, an optional *calling convention*, an optional `unnamed_addr` attribute, a return type, an optional *parameter attribute* for the return type, a function name, a (possibly empty) argument list (each with optional *parameter attributes*), optional *function attributes*, an optional section, an optional alignment, an optional *comdat*, an optional *garbage collector name*, an optional *prefix*, an opening curly brace, a list of basic blocks, and a closing curly brace.

LLVM function declarations consist of the "`declare`" keyword, an optional *linkage type*, an optional *visibility style*, an optional *DLL storage class*, an optional *calling convention*, an optional `unnamed_addr` attribute, a return type, an optional *parameter attribute* for the return type, a function name, a possibly empty list of arguments, an optional alignment, an optional *garbage collector name* and an optional *prefix*.

A function definition contains a list of basic blocks, forming the CFG (Control Flow Graph) for the function. Each basic block may optionally start with a label (giving the basic block a symbol table entry), contains a list of instructions, and ends with a *terminator* instruction (such as a branch or function return). If an explicit label is not provided, a block is assigned an implicit numbered label, using the next value from the same counter as used for unnamed temporaries (*see above*). For example, if a function entry block does not have an explicit label, it will be assigned label "%0", then the first unnamed temporary in that block will be "%1", etc.

The first basic block in a function is special in two ways: it is immediately executed on entrance to the function, and it is not allowed to have predecessor basic blocks (i.e. there can not be any branches to the entry block of a function). Because the block can have no predecessors, it also cannot have any *PHI nodes*.

LLVM allows an explicit section to be specified for functions. If the target supports it, it will emit functions to the section specified. Additionally, the function can placed in a COMDAT.

An explicit alignment may be specified for a function. If not present, or if the alignment is set to zero, the alignment of the function is set by the target to whatever it feels convenient. If an explicit alignment is specified, the function is forced to have at least that much alignment. All alignments must be a power of 2.

If the `unnamed_addr` attribute is given, the address is know to not be significant and two identical functions can be merged.

Syntax:

```
define [linkage] [visibility] [DLLStorageClass]
       [cconv] [ret attrs]
       <ResultType> @<FunctionName> ([argument list])
       [unnamed_addr] [fn Attrs] [section "name"] [comdat $<ComdatName>]
       [align N] [gc] [prefix Constant] { ... }
```

The argument list is a comma seperated sequence of arguments where each argument is of the following form

Syntax:

```
<type> [parameter Attrs] [name]
```

Aliases

Aliases, unlike function or variables, don't create any new data. They are just a new symbol and metadata for an existing position.

Aliases have a name and an aliasee that is either a global value or a constant expression.

Aliases may have an optional *linkage type*, an optional *visibility style*, an optional *DLL storage class* and an optional *tls model*.

Syntax:

```
@<Name> = [Linkage] [Visibility] [DLLStorageClass] [ThreadLocal] [unnamed_addr] alias <AliaseeTy> @<A
```

The linkage must be one of `private`, `internal`, `linkonce`, `weak`, `linkonce_odr`, `weak_odr`, `external`. Note that some system linkers might not correctly handle dropping a weak symbol that is aliased.

Alias that are not `unnamed_addr` are guaranteed to have the same address as the aliasee expression. `unnamed_addr` ones are only guaranteed to point to the same content.

Since aliases are only a second name, some restrictions apply, of which some can only be checked when producing an object file:

- The expression defining the aliasee must be computable at assembly time. Since it is just a name, no relocations can be used.
- No alias in the expression can be weak as the possibility of the intermediate alias being overridden cannot be represented in an object file.
- No global value in the expression can be a declaration, since that would require a relocation, which is not possible.

Comdats

Comdat IR provides access to COFF and ELF object file COMDAT functionality.

Comdats have a name which represents the COMDAT key. All global objects that specify this key will only end up in the final object file if the linker chooses that key over some other key. Aliases are placed in the same COMDAT that their aliasee computes to, if any.

Comdats have a selection kind to provide input on how the linker should choose between keys in two different object files.

Syntax:

```
$<Name> = comdat SelectionKind
```

The selection kind must be one of the following:

`any` The linker may choose any COMDAT key, the choice is arbitrary.

`exactmatch` The linker may choose any COMDAT key but the sections must contain the same data.

`largest` The linker will choose the section containing the largest COMDAT key.

`noduplicates` The linker requires that only section with this COMDAT key exist.

`samesize` The linker may choose any COMDAT key but the sections must contain the same amount of data.

Note that the Mach-O platform doesn't support COMDATs and ELF only supports `any` as a selection kind.

Here is an example of a COMDAT group where a function will only be selected if the COMDAT key's section is the largest:

```
$foo = comdat largest
@foo = global i32 2, comdat $foo

define void @bar() comdat $foo {
  ret void
}
```

In a COFF object file, this will create a COMDAT section with selection kind `IMAGE_COMDAT_SELECT_LARGEST` containing the contents of the `@foo` symbol and another COMDAT section with selection kind

`IMAGE_COMDAT_SELECT_ASSOCIATIVE` which is associated with the first COMDAT section and contains the contents of the `@bar` symbol.

There are some restrictions on the properties of the global object. It, or an alias to it, must have the same name as the COMDAT group when targeting COFF. The contents and size of this object may be used during link-time to determine which COMDAT groups get selected depending on the selection kind. Because the name of the object must match the name of the COMDAT group, the linkage of the global object must not be local; local symbols can get renamed if a collision occurs in the symbol table.

The combined use of COMDATS and section attributes may yield surprising results. For example:

```
$foo = comdat any
$bar = comdat any
@g1 = global i32 42, section "sec", comdat $foo
@g2 = global i32 42, section "sec", comdat $bar
```

From the object file perspective, this requires the creation of two sections with the same name. This is necessary because both globals belong to different COMDAT groups and COMDATs, at the object file level, are represented by sections.

Note that certain IR constructs like global variables and functions may create COMDATs in the object file in addition to any which are specified using COMDAT IR. This arises, for example, when a global variable has linkonce_odr linkage.

Named Metadata

Named metadata is a collection of metadata. *Metadata nodes* (but not metadata strings) are the only valid operands for a named metadata.

Syntax:

```
; Some unnamed metadata nodes, which are referenced by the named metadata.
!0 = metadata !{metadata !"zero"}
!1 = metadata !{metadata !"one"}
!2 = metadata !{metadata !"two"}
; A named metadata.
!name = !{!0, !1, !2}
```

Parameter Attributes

The return type and each parameter of a function type may have a set of *parameter attributes* associated with them. Parameter attributes are used to communicate additional information about the result or parameters of a function. Parameter attributes are considered to be part of the function, not of the function type, so functions with different parameter attributes can have the same function type.

Parameter attributes are simple keywords that follow the type specified. If multiple parameter attributes are needed, they are space separated. For example:

```
declare i32 @printf(i8* noalias nocapture, ...)
declare i32 @atoi(i8 zeroext)
declare signext i8 @returns_signed_char()
```

Note that any attributes for the function result (`nounwind`, `readonly`) come immediately after the argument list.

Currently, only the following parameter attributes are defined:

zeroext This indicates to the code generator that the parameter or return value should be zero-extended to the extent required by the target's ABI (which is usually 32-bits, but is 8-bits for a i1 on x86-64) by the caller (for a parameter) or the callee (for a return value).

signext This indicates to the code generator that the parameter or return value should be sign-extended to the extent required by the target's ABI (which is usually 32-bits) by the caller (for a parameter) or the callee (for a return value).

inreg This indicates that this parameter or return value should be treated in a special target-dependent fashion during while emitting code for a function call or return (usually, by putting it in a register as opposed to memory, though some targets use it to distinguish between two different kinds of registers). Use of this attribute is target-specific.

byval This indicates that the pointer parameter should really be passed by value to the function. The attribute implies that a hidden copy of the pointee is made between the caller and the callee, so the callee is unable to modify the value in the caller. This attribute is only valid on LLVM pointer arguments. It is generally used to pass structs and arrays by value, but is also valid on pointers to scalars. The copy is considered to belong to the caller not the callee (for example, `readonly` functions should not write to `byval` parameters). This is not a valid attribute for return values.

The byval attribute also supports specifying an alignment with the align attribute. It indicates the alignment of the stack slot to form and the known alignment of the pointer specified to the call site. If the alignment is not specified, then the code generator makes a target-specific assumption.

`inalloca`

The `inalloca` argument attribute allows the caller to take the address of outgoing stack arguments. An `inalloca` argument must be a pointer to stack memory produced by an `alloca` instruction. The alloca, or argument allocation, must also be tagged with the inalloca keyword. Only the last argument may have the `inalloca` attribute, and that argument is guaranteed to be passed in memory.

An argument allocation may be used by a call at most once because the call may deallocate it. The `inalloca` attribute cannot be used in conjunction with other attributes that affect argument storage, like `inreg`, `nest`, `sret`, or `byval`. The `inalloca` attribute also disables LLVM's implicit lowering of large aggregate return values, which means that frontend authors must lower them with `sret` pointers.

When the call site is reached, the argument allocation must have been the most recent stack allocation that is still live, or the results are undefined. It is possible to allocate additional stack space after an argument allocation and before its call site, but it must be cleared off with *llvm.stackrestore*.

See *Design and Usage of the InAlloca Attribute* for more information on how to use this attribute.

sret This indicates that the pointer parameter specifies the address of a structure that is the return value of the function in the source program. This pointer must be guaranteed by the caller to be valid: loads and stores to the structure may be assumed by the callee not to trap and to be properly aligned. This may only be applied to the first parameter. This is not a valid attribute for return values.

align <n> This indicates that the pointer value may be assumed by the optimizer to have the specified alignment.

Note that this attribute has additional semantics when combined with the `byval` attribute.

noalias This indicates that pointer values *based* on the argument or return value do not alias pointer values that are not *based* on it, ignoring certain "irrelevant" dependencies. For a call to the parent function, dependencies between memory references from before or after the call and from those during the call are "irrelevant" to the `noalias` keyword for the arguments and return value used in that call. The caller shares the responsibility with the callee for ensuring that these requirements are met. For further details, please see the discussion of the NoAlias response in *alias analysis*.

Note that this definition of `noalias` is intentionally similar to the definition of `restrict` in C99 for function arguments, though it is slightly weaker.

For function return values, C99's `restrict` is not meaningful, while LLVM's `noalias` is.

nocapture This indicates that the callee does not make any copies of the pointer that outlive the callee itself. This is not a valid attribute for return values.

nest This indicates that the pointer parameter can be excised using the *trampoline intrinsics*. This is not a valid attribute for return values and can only be applied to one parameter.

returned This indicates that the function always returns the argument as its return value. This is an optimization hint to the code generator when generating the caller, allowing tail call optimization and omission of register saves and restores in some cases; it is not checked or enforced when generating the callee. The parameter and the function return type must be valid operands for the *bitcast instruction*. This is not a valid attribute for return values and can only be applied to one parameter.

nonnull This indicates that the parameter or return pointer is not null. This attribute may only be applied to pointer typed parameters. This is not checked or enforced by LLVM, the caller must ensure that the pointer passed in is non-null, or the callee must ensure that the returned pointer is non-null.

dereferenceable(<n>) This indicates that the parameter or return pointer is dereferenceable. This attribute may only be applied to pointer typed parameters. A pointer that is dereferenceable can be loaded from speculatively without a risk of trapping. The number of bytes known to be dereferenceable must be provided in parentheses. It is legal for the number of bytes to be less than the size of the pointee type. The `nonnull` attribute does not imply dereferenceability (consider a pointer to one element past the end of an array), however `dereferenceable(<n>)` does imply `nonnull` in `addrspace(0)` (which is the default address space).

Garbage Collector Names

Each function may specify a garbage collector name, which is simply a string:

```
define void @f() gc "name" { ... }
```

The compiler declares the supported values of *name*. Specifying a collector will cause the compiler to alter its output in order to support the named garbage collection algorithm.

Prefix Data

Prefix data is data associated with a function which the code generator will emit immediately before the function body. The purpose of this feature is to allow frontends to associate language-specific runtime metadata with specific functions and make it available through the function pointer while still allowing the function pointer to be called. To access the data for a given function, a program may bitcast the function pointer to a pointer to the constant's type. This implies that the IR symbol points to the start of the prefix data.

To maintain the semantics of ordinary function calls, the prefix data must have a particular format. Specifically, it must begin with a sequence of bytes which decode to a sequence of machine instructions, valid for the module's target, which transfer control to the point immediately succeeding the prefix data, without performing any other visible action. This allows the inliner and other passes to reason about the semantics of the function definition without needing to reason about the prefix data. Obviously this makes the format of the prefix data highly target dependent.

Prefix data is laid out as if it were an initializer for a global variable of the prefix data's type. No padding is automatically placed between the prefix data and the function body. If padding is required, it must be part of the prefix data.

A trivial example of valid prefix data for the x86 architecture is `i8 144`, which encodes the `nop` instruction:

```
define void @f() prefix i8 144 { ... }
```

Generally prefix data can be formed by encoding a relative branch instruction which skips the metadata, as in this example of valid prefix data for the x86_64 architecture, where the first two bytes encode `jmp .+10`:

```
%0 = type <{ i8, i8, i8* }>

define void @f() prefix %0 <{ i8 235, i8 8, i8* @md}> { ... }
```

A function may have prefix data but no body. This has similar semantics to the `available_externally` linkage in that the data may be used by the optimizers but will not be emitted in the object file.

Attribute Groups

Attribute groups are groups of attributes that are referenced by objects within the IR. They are important for keeping `.ll` files readable, because a lot of functions will use the same set of attributes. In the degenerative case of a `.ll` file that corresponds to a single `.c` file, the single attribute group will capture the important command line flags used to build that file.

An attribute group is a module-level object. To use an attribute group, an object references the attribute group's ID (e.g. `#37`). An object may refer to more than one attribute group. In that situation, the attributes from the different groups are merged.

Here is an example of attribute groups for a function that should always be inlined, has a stack alignment of 4, and which shouldn't use SSE instructions:

```
; Target-independent attributes:
attributes #0 = { alwaysinline alignstack=4 }

; Target-dependent attributes:
attributes #1 = { "no-sse" }

; Function @f has attributes: alwaysinline, alignstack=4, and "no-sse".
define void @f() #0 #1 { ... }
```

Function Attributes

Function attributes are set to communicate additional information about a function. Function attributes are considered to be part of the function, not of the function type, so functions with different function attributes can have the same function type.

Function attributes are simple keywords that follow the type specified. If multiple attributes are needed, they are space separated. For example:

```
define void @f() noinline { ... }
define void @f() alwaysinline { ... }
define void @f() alwaysinline optsize { ... }
define void @f() optsize { ... }
```

`alignstack(<n>)` This attribute indicates that, when emitting the prologue and epilogue, the backend should forcibly align the stack pointer. Specify the desired alignment, which must be a power of two, in parentheses.

`alwaysinline` This attribute indicates that the inliner should attempt to inline this function into callers whenever possible, ignoring any active inlining size threshold for this caller.

`builtin` This indicates that the callee function at a call site should be recognized as a built-in function, even though the function's declaration uses the `nobuiltin` attribute. This is only valid at call sites for direct calls to functions that are declared with the `nobuiltin` attribute.

`cold` This attribute indicates that this function is rarely called. When computing edge weights, basic blocks post-dominated by a cold function call are also considered to be cold; and, thus, given low weight.

inlinehint This attribute indicates that the source code contained a hint that inlining this function is desirable (such as the "inline" keyword in C/C++). It is just a hint; it imposes no requirements on the inliner.

jumptable This attribute indicates that the function should be added to a jump-instruction table at code-generation time, and that all address-taken references to this function should be replaced with a reference to the appropriate jump-instruction-table function pointer. Note that this creates a new pointer for the original function, which means that code that depends on function-pointer identity can break. So, any function annotated with `jumptable` must also be `unnamed_addr`.

minsize This attribute suggests that optimization passes and code generator passes make choices that keep the code size of this function as small as possible and perform optimizations that may sacrifice runtime performance in order to minimize the size of the generated code.

naked This attribute disables prologue / epilogue emission for the function. This can have very system-specific consequences.

nobuiltin This indicates that the callee function at a call site is not recognized as a built-in function. LLVM will retain the original call and not replace it with equivalent code based on the semantics of the built-in function, unless the call site uses the `builtin` attribute. This is valid at call sites and on function declarations and definitions.

noduplicate This attribute indicates that calls to the function cannot be duplicated. A call to a `noduplicate` function may be moved within its parent function, but may not be duplicated within its parent function.

A function containing a `noduplicate` call may still be an inlining candidate, provided that the call is not duplicated by inlining. That implies that the function has internal linkage and only has one call site, so the original call is dead after inlining.

noimplicitfloat This attributes disables implicit floating point instructions.

noinline This attribute indicates that the inliner should never inline this function in any situation. This attribute may not be used together with the `alwaysinline` attribute.

nonlazybind This attribute suppresses lazy symbol binding for the function. This may make calls to the function faster, at the cost of extra program startup time if the function is not called during program startup.

noredzone This attribute indicates that the code generator should not use a red zone, even if the target-specific ABI normally permits it.

noreturn This function attribute indicates that the function never returns normally. This produces undefined behavior at runtime if the function ever does dynamically return.

nounwind This function attribute indicates that the function never returns with an unwind or exceptional control flow. If the function does unwind, its runtime behavior is undefined.

optnone This function attribute indicates that the function is not optimized by any optimization or code generator passes with the exception of interprocedural optimization passes. This attribute cannot be used together with the `alwaysinline` attribute; this attribute is also incompatible with the `minsize` attribute and the `optsize` attribute.

This attribute requires the `noinline` attribute to be specified on the function as well, so the function is never inlined into any caller. Only functions with the `alwaysinline` attribute are valid candidates for inlining into the body of this function.

optsize This attribute suggests that optimization passes and code generator passes make choices that keep the code size of this function low, and otherwise do optimizations specifically to reduce code size as long as they do not significantly impact runtime performance.

readnone On a function, this attribute indicates that the function computes its result (or decides to unwind an exception) based strictly on its arguments, without dereferencing any pointer arguments or otherwise accessing any mutable state (e.g. memory, control registers, etc) visible to caller functions. It does not write through any

pointer arguments (including `byval` arguments) and never changes any state visible to callers. This means that it cannot unwind exceptions by calling the `C++` exception throwing methods.

On an argument, this attribute indicates that the function does not dereference that pointer argument, even though it may read or write the memory that the pointer points to if accessed through other pointers.

`readonly` On a function, this attribute indicates that the function does not write through any pointer arguments (including `byval` arguments) or otherwise modify any state (e.g. memory, control registers, etc) visible to caller functions. It may dereference pointer arguments and read state that may be set in the caller. A readonly function always returns the same value (or unwinds an exception identically) when called with the same set of arguments and global state. It cannot unwind an exception by calling the `C++` exception throwing methods.

On an argument, this attribute indicates that the function does not write through this pointer argument, even though it may write to the memory that the pointer points to.

`returns_twice` This attribute indicates that this function can return twice. The C `setjmp` is an example of such a function. The compiler disables some optimizations (like tail calls) in the caller of these functions.

`sanitize_address` This attribute indicates that AddressSanitizer checks (dynamic address safety analysis) are enabled for this function.

`sanitize_memory` This attribute indicates that MemorySanitizer checks (dynamic detection of accesses to uninitialized memory) are enabled for this function.

`sanitize_thread` This attribute indicates that ThreadSanitizer checks (dynamic thread safety analysis) are enabled for this function.

`ssp` This attribute indicates that the function should emit a stack smashing protector. It is in the form of a "canary" — a random value placed on the stack before the local variables that's checked upon return from the function to see if it has been overwritten. A heuristic is used to determine if a function needs stack protectors or not. The heuristic used will enable protectors for functions with:

- Character arrays larger than `ssp-buffer-size` (default 8).
- Aggregates containing character arrays larger than `ssp-buffer-size`.
- Calls to alloca() with variable sizes or constant sizes greater than `ssp-buffer-size`.

Variables that are identified as requiring a protector will be arranged on the stack such that they are adjacent to the stack protector guard.

If a function that has an `ssp` attribute is inlined into a function that doesn't have an `ssp` attribute, then the resulting function will have an `ssp` attribute.

`sspreq` This attribute indicates that the function should *always* emit a stack smashing protector. This overrides the `ssp` function attribute.

Variables that are identified as requiring a protector will be arranged on the stack such that they are adjacent to the stack protector guard. The specific layout rules are:

1. Large arrays and structures containing large arrays (`>= ssp-buffer-size`) are closest to the stack protector.
2. Small arrays and structures containing small arrays (`< ssp-buffer-size`) are 2nd closest to the protector.
3. Variables that have had their address taken are 3rd closest to the protector.

If a function that has an `sspreq` attribute is inlined into a function that doesn't have an `sspreq` attribute or which has an `ssp` or `sspstrong` attribute, then the resulting function will have an `sspreq` attribute.

`sspstrong` This attribute indicates that the function should emit a stack smashing protector. This attribute causes a strong heuristic to be used when determining if a function needs stack protectors. The strong heuristic will enable protectors for functions with:

- Arrays of any size and type
- Aggregates containing an array of any size and type.
- Calls to alloca().
- Local variables that have had their address taken.

Variables that are identified as requiring a protector will be arranged on the stack such that they are adjacent to the stack protector guard. The specific layout rules are:

1. Large arrays and structures containing large arrays (`>= ssp-buffer-size`) are closest to the stack protector.
2. Small arrays and structures containing small arrays (`< ssp-buffer-size`) are 2nd closest to the protector.
3. Variables that have had their address taken are 3rd closest to the protector.

This overrides the `ssp` function attribute.

If a function that has an `sspstrong` attribute is inlined into a function that doesn't have an `sspstrong` attribute, then the resulting function will have an `sspstrong` attribute.

`uwtable` This attribute indicates that the ABI being targeted requires that an unwind table entry be produce for this function even if we can show that no exceptions passes by it. This is normally the case for the ELF x86-64 abi, but it can be disabled for some compilation units.

Module-Level Inline Assembly

Modules may contain "module-level inline asm" blocks, which corresponds to the GCC "file scope inline asm" blocks. These blocks are internally concatenated by LLVM and treated as a single unit, but may be separated in the `.ll` file if desired. The syntax is very simple:

```
module asm "inline asm code goes here"
module asm "more can go here"
```

The strings can contain any character by escaping non-printable characters. The escape sequence used is simply "\xx" where "xx" is the two digit hex code for the number.

The inline asm code is simply printed to the machine code .s file when assembly code is generated.

Data Layout

A module may specify a target specific data layout string that specifies how data is to be laid out in memory. The syntax for the data layout is simply:

```
target datalayout = "layout specification"
```

The *layout specification* consists of a list of specifications separated by the minus sign character ('-'). Each specification starts with a letter and may include other information after the letter to define some aspect of the data layout. The specifications accepted are as follows:

`E` Specifies that the target lays out data in big-endian form. That is, the bits with the most significance have the lowest address location.

`e` Specifies that the target lays out data in little-endian form. That is, the bits with the least significance have the lowest address location.

S<size> Specifies the natural alignment of the stack in bits. Alignment promotion of stack variables is limited to the natural stack alignment to avoid dynamic stack realignment. The stack alignment must be a multiple of 8-bits. If omitted, the natural stack alignment defaults to "unspecified", which does not prevent any alignment promotions.

p[n]:<size>:<abi>:<pref> This specifies the *size* of a pointer and its `<abi>` and `<pref>`erred alignments for address space `n`. All sizes are in bits. The address space, `n` is optional, and if not specified, denotes the default address space 0. The value of `n` must be in the range [1,2^23).

i<size>:<abi>:<pref> This specifies the alignment for an integer type of a given bit `<size>`. The value of `<size>` must be in the range [1,2^23).

v<size>:<abi>:<pref> This specifies the alignment for a vector type of a given bit `<size>`.

f<size>:<abi>:<pref> This specifies the alignment for a floating point type of a given bit `<size>`. Only values of `<size>` that are supported by the target will work. 32 (float) and 64 (double) are supported on all targets; 80 or 128 (different flavors of long double) are also supported on some targets.

a:<abi>:<pref> This specifies the alignment for an object of aggregate type.

m:<mangling> If present, specifies that llvm names are mangled in the output. The options are

- `e`: ELF mangling: Private symbols get a `.L` prefix.
- `m`: Mips mangling: Private symbols get a `$` prefix.
- `o`: Mach-O mangling: Private symbols get `L` prefix. Other symbols get a _ prefix.
- `w`: Windows COFF prefix: Similar to Mach-O, but stdcall and fastcall functions also get a suffix based on the frame size.

n<size1>:<size2>:<size3>... This specifies a set of native integer widths for the target CPU in bits. For example, it might contain `n32` for 32-bit PowerPC, `n32:64` for PowerPC 64, or `n8:16:32:64` for X86-64. Elements of this set are considered to support most general arithmetic operations efficiently.

On every specification that takes a `<abi>:<pref>`, specifying the `<pref>` alignment is optional. If omitted, the preceding `:` should be omitted too and `<pref>` will be equal to `<abi>`.

When constructing the data layout for a given target, LLVM starts with a default set of specifications which are then (possibly) overridden by the specifications in the `datalayout` keyword. The default specifications are given in this list:

- `E` - big endian
- `p:64:64:64` - 64-bit pointers with 64-bit alignment.
- `p[n]:64:64:64` - Other address spaces are assumed to be the same as the default address space.
- `S0` - natural stack alignment is unspecified
- `i1:8:8` - i1 is 8-bit (byte) aligned
- `i8:8:8` - i8 is 8-bit (byte) aligned
- `i16:16:16` - i16 is 16-bit aligned
- `i32:32:32` - i32 is 32-bit aligned
- `i64:32:64` - i64 has ABI alignment of 32-bits but preferred alignment of 64-bits
- `f16:16:16` - half is 16-bit aligned
- `f32:32:32` - float is 32-bit aligned
- `f64:64:64` - double is 64-bit aligned
- `f128:128:128` - quad is 128-bit aligned

- `v64:64:64` - 64-bit vector is 64-bit aligned
- `v128:128:128` - 128-bit vector is 128-bit aligned
- `a:0:64` - aggregates are 64-bit aligned

When LLVM is determining the alignment for a given type, it uses the following rules:

1. If the type sought is an exact match for one of the specifications, that specification is used.
2. If no match is found, and the type sought is an integer type, then the smallest integer type that is larger than the bitwidth of the sought type is used. If none of the specifications are larger than the bitwidth then the largest integer type is used. For example, given the default specifications above, the i7 type will use the alignment of i8 (next largest) while both i65 and i256 will use the alignment of i64 (largest specified).
3. If no match is found, and the type sought is a vector type, then the largest vector type that is smaller than the sought vector type will be used as a fall back. This happens because <128 x double> can be implemented in terms of 64 <2 x double>, for example.

The function of the data layout string may not be what you expect. Notably, this is not a specification from the frontend of what alignment the code generator should use.

Instead, if specified, the target data layout is required to match what the ultimate *code generator* expects. This string is used by the mid-level optimizers to improve code, and this only works if it matches what the ultimate code generator uses. If you would like to generate IR that does not embed this target-specific detail into the IR, then you don't have to specify the string. This will disable some optimizations that require precise layout information, but this also prevents those optimizations from introducing target specificity into the IR.

Target Triple

A module may specify a target triple string that describes the target host. The syntax for the target triple is simply:

```
target triple = "x86_64-apple-macosx10.7.0"
```

The *target triple* string consists of a series of identifiers delimited by the minus sign character ('-'). The canonical forms are:

```
ARCHITECTURE-VENDOR-OPERATING_SYSTEM
ARCHITECTURE-VENDOR-OPERATING_SYSTEM-ENVIRONMENT
```

This information is passed along to the backend so that it generates code for the proper architecture. It's possible to override this on the command line with the `-mtriple` command line option.

Pointer Aliasing Rules

Any memory access must be done through a pointer value associated with an address range of the memory access, otherwise the behavior is undefined. Pointer values are associated with address ranges according to the following rules:

- A pointer value is associated with the addresses associated with any value it is *based* on.
- An address of a global variable is associated with the address range of the variable's storage.
- The result value of an allocation instruction is associated with the address range of the allocated storage.
- A null pointer in the default address-space is associated with no address.
- An integer constant other than zero or a pointer value returned from a function not defined within LLVM may be associated with address ranges allocated through mechanisms other than those provided by LLVM. Such ranges shall not overlap with any ranges of addresses allocated by mechanisms provided by LLVM.

A pointer value is *based* on another pointer value according to the following rules:

- A pointer value formed from a `getelementptr` operation is *based* on the first operand of the `getelementptr`.
- The result value of a `bitcast` is *based* on the operand of the `bitcast`.
- A pointer value formed by an `inttoptr` is *based* on all pointer values that contribute (directly or indirectly) to the computation of the pointer's value.
- The "*based* on" relationship is transitive.

Note that this definition of *"based"* is intentionally similar to the definition of *"based"* in C99, though it is slightly weaker.

LLVM IR does not associate types with memory. The result type of a `load` merely indicates the size and alignment of the memory from which to load, as well as the interpretation of the value. The first operand type of a `store` similarly only indicates the size and alignment of the store.

Consequently, type-based alias analysis, aka TBAA, aka `-fstrict-aliasing`, is not applicable to general unadorned LLVM IR. *Metadata* may be used to encode additional information which specialized optimization passes may use to implement type-based alias analysis.

Volatile Memory Accesses

Certain memory accesses, such as *load*'s, *store*'s, and *llvm.memcpy*'s may be marked `volatile`. The optimizers must not change the number of volatile operations or change their order of execution relative to other volatile operations. The optimizers *may* change the order of volatile operations relative to non-volatile operations. This is not Java's "volatile" and has no cross-thread synchronization behavior.

IR-level volatile loads and stores cannot safely be optimized into llvm.memcpy or llvm.memmove intrinsics even when those intrinsics are flagged volatile. Likewise, the backend should never split or merge target-legal volatile load/store instructions.

Rationale

Platforms may rely on volatile loads and stores of natively supported data width to be executed as single instruction. For example, in C this holds for an l-value of volatile primitive type with native hardware support, but not necessarily for aggregate types. The frontend upholds these expectations, which are intentionally unspecified in the IR. The rules above ensure that IR transformation do not violate the frontend's contract with the language.

Memory Model for Concurrent Operations

The LLVM IR does not define any way to start parallel threads of execution or to register signal handlers. Nonetheless, there are platform-specific ways to create them, and we define LLVM IR's behavior in their presence. This model is inspired by the C++0x memory model.

For a more informal introduction to this model, see the *LLVM Atomic Instructions and Concurrency Guide*.

We define a *happens-before* partial order as the least partial order that

- Is a superset of single-thread program order, and
- When a *synchronizes-with* `b`, includes an edge from `a` to `b`. *Synchronizes-with* pairs are introduced by platform-specific techniques, like pthread locks, thread creation, thread joining, etc., and by atomic instructions. (See also *Atomic Memory Ordering Constraints*).

Note that program order does not introduce *happens-before* edges between a thread and signals executing inside that thread.

Every (defined) read operation (load instructions, memcpy, atomic loads/read-modify-writes, etc.) R reads a series of bytes written by (defined) write operations (store instructions, atomic stores/read-modify-writes, memcpy, etc.). For the purposes of this section, initialized globals are considered to have a write of the initializer which is atomic and happens before any other read or write of the memory in question. For each byte of a read R, R_{byte} may see any write to the same byte, except:

- If $write_1$ happens before $write_2$, and $write_2$ happens before R_{byte}, then R_{byte} does not see $write_1$.
- If R_{byte} happens before $write_3$, then R_{byte} does not see $write_3$.

Given that definition, R_{byte} is defined as follows:

- If R is volatile, the result is target-dependent. (Volatile is supposed to give guarantees which can support `sig_atomic_t` in C/C++, and may be used for accesses to addresses that do not behave like normal memory. It does not generally provide cross-thread synchronization.)
- Otherwise, if there is no write to the same byte that happens before R_{byte}, R_{byte} returns `undef` for that byte.
- Otherwise, if R_{byte} may see exactly one write, R_{byte} returns the value written by that write.
- Otherwise, if R is atomic, and all the writes R_{byte} may see are atomic, it chooses one of the values written. See the *Atomic Memory Ordering Constraints* section for additional constraints on how the choice is made.
- Otherwise R_{byte} returns `undef`.

R returns the value composed of the series of bytes it read. This implies that some bytes within the value may be `undef` **without** the entire value being `undef`. Note that this only defines the semantics of the operation; it doesn't mean that targets will emit more than one instruction to read the series of bytes.

Note that in cases where none of the atomic intrinsics are used, this model places only one restriction on IR transformations on top of what is required for single-threaded execution: introducing a store to a byte which might not otherwise be stored is not allowed in general. (Specifically, in the case where another thread might write to and read from an address, introducing a store can change a load that may see exactly one write into a load that may see multiple writes.)

Atomic Memory Ordering Constraints

Atomic instructions (*cmpxchg*, *atomicrmw*, *fence*, *atomic load*, and *atomic store*) take ordering parameters that determine which other atomic instructions on the same address they *synchronize with*. These semantics are borrowed from Java and C++0x, but are somewhat more colloquial. If these descriptions aren't precise enough, check those specs (see spec references in the *atomics guide*). *fence* instructions treat these orderings somewhat differently since they don't take an address. See that instruction's documentation for details.

For a simpler introduction to the ordering constraints, see the *LLVM Atomic Instructions and Concurrency Guide*.

`unordered` The set of values that can be read is governed by the happens-before partial order. A value cannot be read unless some operation wrote it. This is intended to provide a guarantee strong enough to model Java's non-volatile shared variables. This ordering cannot be specified for read-modify-write operations; it is not strong enough to make them atomic in any interesting way.

`monotonic` In addition to the guarantees of `unordered`, there is a single total order for modifications by `monotonic` operations on each address. All modification orders must be compatible with the happens-before order. There is no guarantee that the modification orders can be combined to a global total order for the whole program (and this often will not be possible). The read in an atomic read-modify-write operation (*cmpxchg* and *atomicrmw*) reads the value in the modification order immediately before the value it writes. If one atomic read happens before another atomic read of the same address, the later read must see the same value or a later value in the address's modification order. This disallows reordering of `monotonic` (or stronger) operations on

the same address. If an address is written `monotonic`-ally by one thread, and other threads `monotonic`-ally read that address repeatedly, the other threads must eventually see the write. This corresponds to the C++0x/C1x `memory_order_relaxed`.

`acquire` In addition to the guarantees of `monotonic`, a *synchronizes-with* edge may be formed with a `release` operation. This is intended to model C++'s `memory_order_acquire`.

`release` In addition to the guarantees of `monotonic`, if this operation writes a value which is subsequently read by an `acquire` operation, it *synchronizes-with* that operation. (This isn't a complete description; see the C++0x definition of a release sequence.) This corresponds to the C++0x/C1x `memory_order_release`.

`acq_rel` (acquire+release) Acts as both an `acquire` and `release` operation on its address. This corresponds to the C++0x/C1x `memory_order_acq_rel`.

`seq_cst` (sequentially consistent) In addition to the guarantees of `acq_rel` (`acquire` for an operation that only reads, `release` for an operation that only writes), there is a global total order on all sequentially-consistent operations on all addresses, which is consistent with the *happens-before* partial order and with the modification orders of all the affected addresses. Each sequentially-consistent read sees the last preceding write to the same address in this global order. This corresponds to the C++0x/C1x `memory_order_seq_cst` and Java volatile.

If an atomic operation is marked `singlethread`, it only *synchronizes with* or participates in modification and seq_cst total orderings with other operations running in the same thread (for example, in signal handlers).

Fast-Math Flags

LLVM IR floating-point binary ops (*fadd*, *fsub*, *fmul*, *fdiv*, *frem*) have the following flags that can set to enable otherwise unsafe floating point operations

`nnan` No NaNs - Allow optimizations to assume the arguments and result are not NaN. Such optimizations are required to retain defined behavior over NaNs, but the value of the result is undefined.

`ninf` No Infs - Allow optimizations to assume the arguments and result are not +/-Inf. Such optimizations are required to retain defined behavior over +/-Inf, but the value of the result is undefined.

`nsz` No Signed Zeros - Allow optimizations to treat the sign of a zero argument or result as insignificant.

`arcp` Allow Reciprocal - Allow optimizations to use the reciprocal of an argument rather than perform division.

`fast` Fast - Allow algebraically equivalent transformations that may dramatically change results in floating point (e.g. reassociate). This flag implies all the others.

Use-list Order Directives

Use-list directives encode the in-memory order of each use-list, allowing the order to be recreated. `<order-indexes>` is a comma-separated list of indexes that are assigned to the referenced value's uses. The referenced value's use-list is immediately sorted by these indexes.

Use-list directives may appear at function scope or global scope. They are not instructions, and have no effect on the semantics of the IR. When they're at function scope, they must appear after the terminator of the final basic block.

If basic blocks have their address taken via `blockaddress()` expressions, `uselistorder_bb` can be used to reorder their use-lists from outside their function's scope.

Syntax

```
uselistorder <ty> <value>, { <order-indexes> }
uselistorder_bb @function, %block { <order-indexes> }
```

Examples

```
define void @foo(i32 %arg1, i32 %arg2) {
entry:
  ; ... instructions ...
bb:
  ; ... instructions ...

  ; At function scope.
  uselistorder i32 %arg1, { 1, 0, 2 }
  uselistorder label %bb, { 1, 0 }
}

; At global scope.
uselistorder i32* @global, { 1, 2, 0 }
uselistorder i32 7, { 1, 0 }
uselistorder i32 (i32) @bar, { 1, 0 }
uselistorder_bb @foo, %bb, { 5, 1, 3, 2, 0, 4 }
```

1.1.5 Type System

The LLVM type system is one of the most important features of the intermediate representation. Being typed enables a number of optimizations to be performed on the intermediate representation directly, without having to do extra analyses on the side before the transformation. A strong type system makes it easier to read the generated code and enables novel analyses and transformations that are not feasible to perform on normal three address code representations.

Void Type

Overview

The void type does not represent any value and has no size.

Syntax

```
void
```

Function Type

Overview

The function type can be thought of as a function signature. It consists of a return type and a list of formal parameter types. The return type of a function type is a void type or first class type — except for *label* and *metadata* types.

Syntax

```
<returntype> (<parameter list>)
```

...where '`<parameter list>`' is a comma-separated list of type specifiers. Optionally, the parameter list may include a type `...`, which indicates that the function takes a variable number of arguments. Variable argument functions can access their arguments with the *variable argument handling intrinsic* functions. '`<returntype>`' is any type except *label* and *metadata*.

Examples

`i32 (i32)`	function taking an `i32`, returning an `i32`
`float (i16, i32 *) *`	*Pointer* to a function that takes an `i16` and a *pointer* to `i32`, returning `float`.
`i32 (i8*, ...)`	A vararg function that takes at least one *pointer* to `i8` (char in C), which returns an integer. This is the signature for `printf` in LLVM.
`{i32, i32} (i32)`	A function taking an `i32`, returning a *structure* containing two `i32` values

First Class Types

The *first class* types are perhaps the most important. Values of these types are the only ones which can be produced by instructions.

Single Value Types

These are the types that are valid in registers from CodeGen's perspective.

Integer Type

Overview

The integer type is a very simple type that simply specifies an arbitrary bit width for the integer type desired. Any bit width from 1 bit to 2^{23}-1 (about 8 million) can be specified.

Syntax

```
iN
```

The number of bits the integer will occupy is specified by the `N` value.

Examples:

`i1`	a single-bit integer.
`i32`	a 32-bit integer.
`i1942652`	a really big integer of over 1 million bits.

Floating Point Types

Type	Description
`half`	16-bit floating point value
`float`	32-bit floating point value
`double`	64-bit floating point value
`fp128`	128-bit floating point value (112-bit mantissa)
`x86_fp80`	80-bit floating point value (X87)
`ppc_fp128`	128-bit floating point value (two 64-bits)

X86_mmx Type

Overview

The x86_mmx type represents a value held in an MMX register on an x86 machine. The operations allowed on it are quite limited: parameters and return values, load and store, and bitcast. User-specified MMX instructions are represented as intrinsic or asm calls with arguments and/or results of this type. There are no arrays, vectors or constants of this type.

Syntax

`x86_mmx`

Pointer Type

Overview

The pointer type is used to specify memory locations. Pointers are commonly used to reference objects in memory.

Pointer types may have an optional address space attribute defining the numbered address space where the pointed-to object resides. The default address space is number zero. The semantics of non-zero address spaces are target-specific.

Note that LLVM does not permit pointers to void (`void*`) nor does it permit pointers to labels (`label*`). Use `i8*` instead.

Syntax

```
<type> *
```

Examples

`[4 x i32]*`	A *pointer* to *array* of four `i32` values.
`i32 (i32*) *`	A *pointer* to a *function* that takes an `i32*`, returning an `i32`.
`i32 addrspace(5)*`	A *pointer* to an `i32` value that resides in address space #5.

Vector Type

Overview

A vector type is a simple derived type that represents a vector of elements. Vector types are used when multiple primitive data are operated in parallel using a single instruction (SIMD). A vector type requires a size (number of elements) and an underlying primitive data type. Vector types are considered *first class*.

Syntax

```
< <# elements> x <elementtype> >
```

The number of elements is a constant integer value larger than 0; elementtype may be any integer, floating point or pointer type. Vectors of size zero are not allowed.

Examples

`<4 x i32>`	Vector of 4 32-bit integer values.
`<8 x float>`	Vector of 8 32-bit floating-point values.
`<2 x i64>`	Vector of 2 64-bit integer values.
`<4 x i64*>`	Vector of 4 pointers to 64-bit integer values.

Label Type

Overview

The label type represents code labels.

Syntax

```
label
```

Metadata Type

Overview

The metadata type represents embedded metadata. No derived types may be created from metadata except for *function* arguments.

Syntax

```
metadata
```

Aggregate Types

Aggregate Types are a subset of derived types that can contain multiple member types. *Arrays* and *structs* are aggregate types. *Vectors* are not considered to be aggregate types.

Array Type

Overview

The array type is a very simple derived type that arranges elements sequentially in memory. The array type requires a size (number of elements) and an underlying data type.

Syntax

```
[<# elements> x <elementtype>]
```

The number of elements is a constant integer value; `elementtype` may be any type with a size.

Examples

`[40 x i32]`	Array of 40 32-bit integer values.
`[41 x i32]`	Array of 41 32-bit integer values.
`[4 x i8]`	Array of 4 8-bit integer values.

Here are some examples of multidimensional arrays:

`[3 x [4 x i32]]`	3x4 array of 32-bit integer values.
`[12 x [10 x float]]`	12x10 array of single precision floating point values.
`[2 x [3 x [4 x i16]]]`	2x3x4 array of 16-bit integer values.

There is no restriction on indexing beyond the end of the array implied by a static type (though there are restrictions on indexing beyond the bounds of an allocated object in some cases). This means that single-dimension 'variable sized array' addressing can be implemented in LLVM with a zero length array type. An implementation of 'pascal style arrays' in LLVM could use the type "`{ i32, [0 x float]}`", for example.

Structure Type

Overview

The structure type is used to represent a collection of data members together in memory. The elements of a structure may be any type that has a size.

Structures in memory are accessed using '`load`' and '`store`' by getting a pointer to a field with the '`getelementptr`' instruction. Structures in registers are accessed using the '`extractvalue`' and '`insertvalue`' instructions.

Structures may optionally be "packed" structures, which indicate that the alignment of the struct is one byte, and that there is no padding between the elements. In non-packed structs, padding between field types is inserted as defined by the DataLayout string in the module, which is required to match what the underlying code generator expects.

Structures can either be "literal" or "identified". A literal structure is defined inline with other types (e.g. `{i32, i32}*`) whereas identified types are always defined at the top level with a name. Literal types are uniqued by their contents and can never be recursive or opaque since there is no way to write one. Identified types can be recursive, can be opaqued, and are never uniqued.

Syntax

```
%T1 = type { <type list> }     ; Identified normal struct type
%T2 = type <{ <type list> }>   ; Identified packed struct type
```

Examples

`{ i32, i32, i32 }`	A triple of three `i32` values
`{ float, i32 (i32) * }`	A pair, where the first element is a `float` and the second element is a *pointer* to a *function* that takes an `i32`, returning an `i32`.
`<{ i8, i32 }>`	A packed struct known to be 5 bytes in size.

Opaque Structure Types

Overview

Opaque structure types are used to represent named structure types that do not have a body specified. This corresponds (for example) to the C notion of a forward declared structure.

Syntax

```
%X = type opaque
%52 = type opaque
```

Examples

`opaque`	An opaque type.

1.1.6 Constants

LLVM has several different basic types of constants. This section describes them all and their syntax.

Simple Constants

Boolean constants The two strings '`true`' and '`false`' are both valid constants of the `i1` type.

Integer constants Standard integers (such as '4') are constants of the *integer* type. Negative numbers may be used with integer types.

Floating point constants Floating point constants use standard decimal notation (e.g. 123.421), exponential notation (e.g. 1.23421e+2), or a more precise hexadecimal notation (see below). The assembler requires the exact decimal value of a floating-point constant. For example, the assembler accepts 1.25 but rejects 1.3 because 1.3 is a repeating decimal in binary. Floating point constants must have a *floating point* type.

Null pointer constants The identifier '`null`' is recognized as a null pointer constant and must be of *pointer type*.

The one non-intuitive notation for constants is the hexadecimal form of floating point constants. For example, the form '`double 0x432ff973cafa8000`' is equivalent to (but harder to read than) '`double 4.5e+15`'. The only time hexadecimal floating point constants are required (and the only time that they are generated by the disassembler) is

when a floating point constant must be emitted but it cannot be represented as a decimal floating point number in a reasonable number of digits. For example, NaN's, infinities, and other special values are represented in their IEEE hexadecimal format so that assembly and disassembly do not cause any bits to change in the constants.

When using the hexadecimal form, constants of types half, float, and double are represented using the 16-digit form shown above (which matches the IEEE754 representation for double); half and float values must, however, be exactly representable as IEEE 754 half and single precision, respectively. Hexadecimal format is always used for long double, and there are three forms of long double. The 80-bit format used by x86 is represented as `0xK` followed by 20 hexadecimal digits. The 128-bit format used by PowerPC (two adjacent doubles) is represented by `0xM` followed by 32 hexadecimal digits. The IEEE 128-bit format is represented by `0xL` followed by 32 hexadecimal digits. Long doubles will only work if they match the long double format on your target. The IEEE 16-bit format (half precision) is represented by `0xH` followed by 4 hexadecimal digits. All hexadecimal formats are big-endian (sign bit at the left).

There are no constants of type x86_mmx.

Complex Constants

Complex constants are a (potentially recursive) combination of simple constants and smaller complex constants.

Structure constants Structure constants are represented with notation similar to structure type definitions (a comma separated list of elements, surrounded by braces (`{}`)). For example: "`{ i32 4, float 17.0, i32* @G }`", where "`@G`" is declared as "`@G = external global i32`". Structure constants must have *structure type*, and the number and types of elements must match those specified by the type.

Array constants Array constants are represented with notation similar to array type definitions (a comma separated list of elements, surrounded by square brackets (`[]`)). For example: "`[ i32 42, i32 11, i32 74 ]`". Array constants must have *array type*, and the number and types of elements must match those specified by the type. As a special case, character array constants may also be represented as a double-quoted string using the `c` prefix. For example: "`c"Hello World\0A\00"`".

Vector constants Vector constants are represented with notation similar to vector type definitions (a comma separated list of elements, surrounded by less-than/greater-than's (<>)). For example: "`< i32 42, i32 11, i32 74, i32 100 >`". Vector constants must have *vector type*, and the number and types of elements must match those specified by the type.

Zero initialization The string '`zeroinitializer`' can be used to zero initialize a value to zero of *any* type, including scalar and *aggregate* types. This is often used to avoid having to print large zero initializers (e.g. for large arrays) and is always exactly equivalent to using explicit zero initializers.

Metadata node A metadata node is a structure-like constant with *metadata type*. For example: "`metadata !{ i32 0, metadata !"test" }`". Unlike other constants that are meant to be interpreted as part of the instruction stream, metadata is a place to attach additional information such as debug info.

Global Variable and Function Addresses

The addresses of *global variables* and *functions* are always implicitly valid (link-time) constants. These constants are explicitly referenced when the *identifier for the global* is used and always have *pointer* type. For example, the following is a legal LLVM file:

```
@X = global i32 17
@Y = global i32 42
@Z = global [2 x i32*] [ i32* @X, i32* @Y ]
```

Undefined Values

The string '`undef`' can be used anywhere a constant is expected, and indicates that the user of the value may receive an unspecified bit-pattern. Undefined values may be of any type (other than '`label`' or '`void`') and be used anywhere a constant is permitted.

Undefined values are useful because they indicate to the compiler that the program is well defined no matter what value is used. This gives the compiler more freedom to optimize. Here are some examples of (potentially surprising) transformations that are valid (in pseudo IR):

```
  %A = add %X, undef
  %B = sub %X, undef
  %C = xor %X, undef
Safe:
  %A = undef
  %B = undef
  %C = undef
```

This is safe because all of the output bits are affected by the undef bits. Any output bit can have a zero or one depending on the input bits.

```
  %A = or %X, undef
  %B = and %X, undef
Safe:
  %A = -1
  %B = 0
Unsafe:
  %A = undef
  %B = undef
```

These logical operations have bits that are not always affected by the input. For example, if `%X` has a zero bit, then the output of the '`and`' operation will always be a zero for that bit, no matter what the corresponding bit from the '`undef`' is. As such, it is unsafe to optimize or assume that the result of the '`and`' is '`undef`'. However, it is safe to assume that all bits of the '`undef`' could be 0, and optimize the '`and`' to 0. Likewise, it is safe to assume that all the bits of the '`undef`' operand to the '`or`' could be set, allowing the '`or`' to be folded to -1.

```
  %A = select undef, %X, %Y
  %B = select undef, 42, %Y
  %C = select %X, %Y, undef
Safe:
  %A = %X     (or %Y)
  %B = 42     (or %Y)
  %C = %Y
Unsafe:
  %A = undef
  %B = undef
  %C = undef
```

This set of examples shows that undefined '`select`' (and conditional branch) conditions can go *either way*, but they have to come from one of the two operands. In the `%A` example, if `%X` and `%Y` were both known to have a clear low bit, then `%A` would have to have a cleared low bit. However, in the `%C` example, the optimizer is allowed to assume that the '`undef`' operand could be the same as `%Y`, allowing the whole '`select`' to be eliminated.

```
  %A = xor undef, undef

  %B = undef
  %C = xor %B, %B

  %D = undef
```

```
  %E = icmp slt %D, 4
  %F = icmp gte %D, 4

Safe:
  %A = undef
  %B = undef
  %C = undef
  %D = undef
  %E = undef
  %F = undef
```

This example points out that two '`undef`' operands are not necessarily the same. This can be surprising to people (and also matches C semantics) where they assume that "`X^X`" is always zero, even if `X` is undefined. This isn't true for a number of reasons, but the short answer is that an '`undef`' "variable" can arbitrarily change its value over its "live range". This is true because the variable doesn't actually *have a live range*. Instead, the value is logically read from arbitrary registers that happen to be around when needed, so the value is not necessarily consistent over time. In fact, `%A` and `%C` need to have the same semantics or the core LLVM "replace all uses with" concept would not hold.

```
  %A = fdiv undef, %X
  %B = fdiv %X, undef
Safe:
  %A = undef
b: unreachable
```

These examples show the crucial difference between an *undefined value* and *undefined behavior*. An undefined value (like '`undef`') is allowed to have an arbitrary bit-pattern. This means that the `%A` operation can be constant folded to '`undef`', because the '`undef`' could be an SNaN, and `fdiv` is not (currently) defined on SNaN's. However, in the second example, we can make a more aggressive assumption: because the `undef` is allowed to be an arbitrary value, we are allowed to assume that it could be zero. Since a divide by zero has *undefined behavior*, we are allowed to assume that the operation does not execute at all. This allows us to delete the divide and all code after it. Because the undefined operation "can't happen", the optimizer can assume that it occurs in dead code.

```
a:  store undef -> %X
b:  store %X -> undef
Safe:
a: <deleted>
b: unreachable
```

These examples reiterate the `fdiv` example: a store *of* an undefined value can be assumed to not have any effect; we can assume that the value is overwritten with bits that happen to match what was already there. However, a store *to* an undefined location could clobber arbitrary memory, therefore, it has undefined behavior.

Poison Values

Poison values are similar to *undef values*, however they also represent the fact that an instruction or constant expression that cannot evoke side effects has nevertheless detected a condition that results in undefined behavior.

There is currently no way of representing a poison value in the IR; they only exist when produced by operations such as *add* with the `nsw` flag.

Poison value behavior is defined in terms of value *dependence*:

- Values other than *phi* nodes depend on their operands.
- *Phi* nodes depend on the operand corresponding to their dynamic predecessor basic block.
- Function arguments depend on the corresponding actual argument values in the dynamic callers of their functions.

- *Call* instructions depend on the *ret* instructions that dynamically transfer control back to them.
- *Invoke* instructions depend on the *ret*, *resume*, or exception-throwing call instructions that dynamically transfer control back to them.
- Non-volatile loads and stores depend on the most recent stores to all of the referenced memory addresses, following the order in the IR (including loads and stores implied by intrinsics such as *@llvm.memcpy*.)
- An instruction with externally visible side effects depends on the most recent preceding instruction with externally visible side effects, following the order in the IR. (This includes *volatile operations*.)
- An instruction *control-depends* on a *terminator instruction* if the terminator instruction has multiple successors and the instruction is always executed when control transfers to one of the successors, and may not be executed when control is transferred to another.
- Additionally, an instruction also *control-depends* on a terminator instruction if the set of instructions it otherwise depends on would be different if the terminator had transferred control to a different successor.
- Dependence is transitive.

Poison values have the same behavior as *undef values*, with the additional effect that any instruction that has a *dependence* on a poison value has undefined behavior.

Here are some examples:

```
entry:
  %poison = sub nuw i32 0, 1           ; Results in a poison value.
  %still_poison = and i32 %poison, 0   ; 0, but also poison.
  %poison_yet_again = getelementptr i32* @h, i32 %still_poison
  store i32 0, i32* %poison_yet_again  ; memory at @h[0] is poisoned

  store i32 %poison, i32* @g           ; Poison value stored to memory.
  %poison2 = load i32* @g              ; Poison value loaded back from memory.

  store volatile i32 %poison, i32* @g  ; External observation; undefined behavior.

  %narrowaddr = bitcast i32* @g to i16*
  %wideaddr = bitcast i32* @g to i64*
  %poison3 = load i16* %narrowaddr     ; Returns a poison value.
  %poison4 = load i64* %wideaddr       ; Returns a poison value.

  %cmp = icmp slt i32 %poison, 0       ; Returns a poison value.
  br i1 %cmp, label %true, label %end  ; Branch to either destination.

true:
  store volatile i32 0, i32* @g        ; This is control-dependent on %cmp, so
                                       ; it has undefined behavior.
  br label %end

end:
  %p = phi i32 [ 0, %entry ], [ 1, %true ]
                                       ; Both edges into this PHI are
                                       ; control-dependent on %cmp, so this
                                       ; always results in a poison value.

  store volatile i32 0, i32* @g        ; This would depend on the store in %true
                                       ; if %cmp is true, or the store in %entry
                                       ; otherwise, so this is undefined behavior.

  br i1 %cmp, label %second_true, label %second_end
                                       ; The same branch again, but this time the
```

```
                                        ; true block doesn't have side effects.

second_true:
  ; No side effects!
  ret void

second_end:
  store volatile i32 0, i32* @g         ; This time, the instruction always depends
                                        ; on the store in %end. Also, it is
                                        ; control-equivalent to %end, so this is
                                        ; well-defined (ignoring earlier undefined
                                        ; behavior in this example).
```

Addresses of Basic Blocks

`blockaddress(@function, %block)`

The '`blockaddress`' constant computes the address of the specified basic block in the specified function, and always has an `i8*` type. Taking the address of the entry block is illegal.

This value only has defined behavior when used as an operand to the '*indirectbr*' instruction, or for comparisons against null. Pointer equality tests between labels addresses results in undefined behavior — though, again, comparison against null is ok, and no label is equal to the null pointer. This may be passed around as an opaque pointer sized value as long as the bits are not inspected. This allows `ptrtoint` and arithmetic to be performed on these values so long as the original value is reconstituted before the `indirectbr` instruction.

Finally, some targets may provide defined semantics when using the value as the operand to an inline assembly, but that is target specific.

Constant Expressions

Constant expressions are used to allow expressions involving other constants to be used as constants. Constant expressions may be of any *first class* type and may involve any LLVM operation that does not have side effects (e.g. load and call are not supported). The following is the syntax for constant expressions:

`trunc (CST to TYPE)` Truncate a constant to another type. The bit size of CST must be larger than the bit size of TYPE. Both types must be integers.

`zext (CST to TYPE)` Zero extend a constant to another type. The bit size of CST must be smaller than the bit size of TYPE. Both types must be integers.

`sext (CST to TYPE)` Sign extend a constant to another type. The bit size of CST must be smaller than the bit size of TYPE. Both types must be integers.

`fptrunc (CST to TYPE)` Truncate a floating point constant to another floating point type. The size of CST must be larger than the size of TYPE. Both types must be floating point.

`fpext (CST to TYPE)` Floating point extend a constant to another type. The size of CST must be smaller or equal to the size of TYPE. Both types must be floating point.

`fptoui (CST to TYPE)` Convert a floating point constant to the corresponding unsigned integer constant. TYPE must be a scalar or vector integer type. CST must be of scalar or vector floating point type. Both CST and TYPE must be scalars, or vectors of the same number of elements. If the value won't fit in the integer type, the results are undefined.

`fptosi (CST to TYPE)` Convert a floating point constant to the corresponding signed integer constant. TYPE must be a scalar or vector integer type. CST must be of scalar or vector floating point type. Both CST and TYPE

must be scalars, or vectors of the same number of elements. If the value won't fit in the integer type, the results are undefined.

uitofp (CST to TYPE) Convert an unsigned integer constant to the corresponding floating point constant. TYPE must be a scalar or vector floating point type. CST must be of scalar or vector integer type. Both CST and TYPE must be scalars, or vectors of the same number of elements. If the value won't fit in the floating point type, the results are undefined.

sitofp (CST to TYPE) Convert a signed integer constant to the corresponding floating point constant. TYPE must be a scalar or vector floating point type. CST must be of scalar or vector integer type. Both CST and TYPE must be scalars, or vectors of the same number of elements. If the value won't fit in the floating point type, the results are undefined.

ptrtoint (CST to TYPE) Convert a pointer typed constant to the corresponding integer constant. `TYPE` must be an integer type. `CST` must be of pointer type. The `CST` value is zero extended, truncated, or unchanged to make it fit in `TYPE`.

inttoptr (CST to TYPE) Convert an integer constant to a pointer constant. TYPE must be a pointer type. CST must be of integer type. The CST value is zero extended, truncated, or unchanged to make it fit in a pointer size. This one is *really* dangerous!

bitcast (CST to TYPE) Convert a constant, CST, to another TYPE. The constraints of the operands are the same as those for the *bitcast instruction*.

addrspacecast (CST to TYPE) Convert a constant pointer or constant vector of pointer, CST, to another TYPE in a different address space. The constraints of the operands are the same as those for the *addrspacecast instruction*.

getelementptr (CSTPTR, IDX0, IDX1, ...),getelementptr inbounds (CSTPTR, IDX0, IDX1, ...) Perform the *getelementptr operation* on constants. As with the *getelementptr* instruction, the index list may have zero or more indexes, which are required to make sense for the type of "CSTPTR".

select (COND, VAL1, VAL2) Perform the *select operation* on constants.

icmp COND (VAL1, VAL2) Performs the *icmp operation* on constants.

fcmp COND (VAL1, VAL2) Performs the *fcmp operation* on constants.

extractelement (VAL, IDX) Perform the *extractelement operation* on constants.

insertelement (VAL, ELT, IDX) Perform the *insertelement operation* on constants.

shufflevector (VEC1, VEC2, IDXMASK) Perform the *shufflevector operation* on constants.

extractvalue (VAL, IDX0, IDX1, ...) Perform the *extractvalue operation* on constants. The index list is interpreted in a similar manner as indices in a '*getelementptr*' operation. At least one index value must be specified.

insertvalue (VAL, ELT, IDX0, IDX1, ...) Perform the *insertvalue operation* on constants. The index list is interpreted in a similar manner as indices in a '*getelementptr*' operation. At least one index value must be specified.

OPCODE (LHS, RHS) Perform the specified operation of the LHS and RHS constants. OPCODE may be any of the *binary* or *bitwise binary* operations. The constraints on operands are the same as those for the corresponding instruction (e.g. no bitwise operations on floating point values are allowed).

1.1.7 Other Values

Inline Assembler Expressions

LLVM supports inline assembler expressions (as opposed to *Module-Level Inline Assembly*) through the use of a special value. This value represents the inline assembler as a string (containing the instructions to emit), a list of operand constraints (stored as a string), a flag that indicates whether or not the inline asm expression has side effects, and a flag indicating whether the function containing the asm needs to align its stack conservatively. An example inline assembler expression is:

```
i32 (i32) asm "bswap $0", "=r,r"
```

Inline assembler expressions may **only** be used as the callee operand of a *call* or an *invoke* instruction. Thus, typically we have:

```
%X = call i32 asm "bswap $0", "=r,r"(i32 %Y)
```

Inline asms with side effects not visible in the constraint list must be marked as having side effects. This is done through the use of the ‘`sideeffect`‘ keyword, like so:

```
call void asm sideeffect "eieio", ""()
```

In some cases inline asms will contain code that will not work unless the stack is aligned in some way, such as calls or SSE instructions on x86, yet will not contain code that does that alignment within the asm. The compiler should make conservative assumptions about what the asm might contain and should generate its usual stack alignment code in the prologue if the ‘`alignstack`‘ keyword is present:

```
call void asm alignstack "eieio", ""()
```

Inline asms also support using non-standard assembly dialects. The assumed dialect is ATT. When the ‘`inteldialect`‘ keyword is present, the inline asm is using the Intel dialect. Currently, ATT and Intel are the only supported dialects. An example is:

```
call void asm inteldialect "eieio", ""()
```

If multiple keywords appear the ‘`sideeffect`‘ keyword must come first, the ‘`alignstack`‘ keyword second and the ‘`inteldialect`‘ keyword last.

Inline Asm Metadata

The call instructions that wrap inline asm nodes may have a “`!srcloc`” MDNode attached to it that contains a list of constant integers. If present, the code generator will use the integer as the location cookie value when report errors through the `LLVMContext` error reporting mechanisms. This allows a front-end to correlate backend errors that occur with inline asm back to the source code that produced it. For example:

```
call void asm sideeffect "something bad", ""(), !srcloc !42
...
!42 = !{ i32 1234567 }
```

It is up to the front-end to make sense of the magic numbers it places in the IR. If the MDNode contains multiple constants, the code generator will use the one that corresponds to the line of the asm that the error occurs on.

Metadata Nodes and Metadata Strings

LLVM IR allows metadata to be attached to instructions in the program that can convey extra information about the code to the optimizers and code generator. One example application of metadata is source-level debug information.

There are two metadata primitives: strings and nodes. All metadata has the `metadata` type and is identified in syntax by a preceding exclamation point ('`!`').

A metadata string is a string surrounded by double quotes. It can contain any character by escaping non-printable characters with "`\xx`" where "`xx`" is the two digit hex code. For example: "`!"test\00"`".

Metadata nodes are represented with notation similar to structure constants (a comma separated list of elements, surrounded by braces and preceded by an exclamation point). Metadata nodes can have any values as their operand. For example:

```
!{ metadata !"test\00", i32 10}
```

A *named metadata* is a collection of metadata nodes, which can be looked up in the module symbol table. For example:

```
!foo =  metadata !{!4, !3}
```

Metadata can be used as function arguments. Here `llvm.dbg.value` function is using two metadata arguments:

```
call void @llvm.dbg.value(metadata !24, i64 0, metadata !25)
```

Metadata can be attached with an instruction. Here metadata `!21` is attached to the `add` instruction using the `!dbg` identifier:

```
%indvar.next = add i64 %indvar, 1, !dbg !21
```

More information about specific metadata nodes recognized by the optimizers and code generator is found below.

'`tbaa`' Metadata

In LLVM IR, memory does not have types, so LLVM's own type system is not suitable for doing TBAA. Instead, metadata is added to the IR to describe a type system of a higher level language. This can be used to implement typical C/C++ TBAA, but it can also be used to implement custom alias analysis behavior for other languages.

The current metadata format is very simple. TBAA metadata nodes have up to three fields, e.g.:

```
!0 = metadata !{ metadata !"an example type tree" }
!1 = metadata !{ metadata !"int", metadata !0 }
!2 = metadata !{ metadata !"float", metadata !0 }
!3 = metadata !{ metadata !"const float", metadata !2, i64 1 }
```

The first field is an identity field. It can be any value, usually a metadata string, which uniquely identifies the type. The most important name in the tree is the name of the root node. Two trees with different root node names are entirely disjoint, even if they have leaves with common names.

The second field identifies the type's parent node in the tree, or is null or omitted for a root node. A type is considered to alias all of its descendants and all of its ancestors in the tree. Also, a type is considered to alias all types in other trees, so that bitcode produced from multiple front-ends is handled conservatively.

If the third field is present, it's an integer which if equal to 1 indicates that the type is "constant" (meaning `pointsToConstantMemory` should return true; see other useful AliasAnalysis methods).

'`tbaa.struct`' Metadata

The *llvm.memcpy* is often used to implement aggregate assignment operations in C and similar languages, however it is defined to copy a contiguous region of memory, which is more than strictly necessary for aggregate types which contain holes due to padding. Also, it doesn't contain any TBAA information about the fields of the aggregate.

`!tbaa.struct` metadata can describe which memory subregions in a memcpy are padding and what the TBAA tags of the struct are.

The current metadata format is very simple. `!tbaa.struct` metadata nodes are a list of operands which are in conceptual groups of three. For each group of three, the first operand gives the byte offset of a field in bytes, the second gives its size in bytes, and the third gives its tbaa tag. e.g.:

```
!4 = metadata !{ i64 0, i64 4, metadata !1, i64 8, i64 4, metadata !2 }
```

This describes a struct with two fields. The first is at offset 0 bytes with size 4 bytes, and has tbaa tag !1. The second is at offset 8 bytes and has size 4 bytes and has tbaa tag !2.

Note that the fields need not be contiguous. In this example, there is a 4 byte gap between the two fields. This gap represents padding which does not carry useful data and need not be preserved.

‘noalias‘ and ‘alias.scope‘ Metadata

`noalias` and `alias.scope` metadata provide the ability to specify generic noalias memory-access sets. This means that some collection of memory access instructions (loads, stores, memory-accessing calls, etc.) that carry `noalias` metadata can specifically be specified not to alias with some other collection of memory access instructions that carry `alias.scope` metadata. Each type of metadata specifies a list of scopes where each scope has an id and a domain. When evaluating an aliasing query, if for some some domain, the set of scopes with that domain in one instruction's `alias.scope` list is a subset of (or qual to) the set of scopes for that domain in another instruction's `noalias` list, then the two memory accesses are assumed not to alias.

The metadata identifying each domain is itself a list containing one or two entries. The first entry is the name of the domain. Note that if the name is a string then it can be combined accross functions and translation units. A self-reference can be used to create globally unique domain names. A descriptive string may optionally be provided as a second list entry.

The metadata identifying each scope is also itself a list containing two or three entries. The first entry is the name of the scope. Note that if the name is a string then it can be combined accross functions and translation units. A self-reference can be used to create globally unique scope names. A metadata reference to the scope's domain is the second entry. A descriptive string may optionally be provided as a third list entry.

For example,

```
; Two scope domains:
!0 = metadata !{metadata !0}
!1 = metadata !{metadata !1}

; Some scopes in these domains:
!2 = metadata !{metadata !2, metadata !0}
!3 = metadata !{metadata !3, metadata !0}
!4 = metadata !{metadata !4, metadata !1}

; Some scope lists:
!5 = metadata !{metadata !4} ; A list containing only scope !4
!6 = metadata !{metadata !4, metadata !3, metadata !2}
!7 = metadata !{metadata !3}

; These two instructions don't alias:
%0 = load float* %c, align 4, !alias.scope !5
store float %0, float* %arrayidx.i, align 4, !noalias !5

; These two instructions also don't alias (for domain !1, the set of scopes
; in the !alias.scope equals that in the !noalias list):
%2 = load float* %c, align 4, !alias.scope !5
store float %2, float* %arrayidx.i2, align 4, !noalias !6

; These two instructions don't alias (for domain !0, the set of scopes in
```

```
; the !noalias list is not a superset of, or equal to, the scopes in the
; !alias.scope list):
%2 = load float* %c, align 4, !alias.scope !6
store float %0, float* %arrayidx.i, align 4, !noalias !7
```

'fpmath' Metadata

fpmath metadata may be attached to any instruction of floating point type. It can be used to express the maximum acceptable error in the result of that instruction, in ULPs, thus potentially allowing the compiler to use a more efficient but less accurate method of computing it. ULP is defined as follows:

> If x is a real number that lies between two finite consecutive floating-point numbers a and b, without being equal to one of them, then ulp(x) = |b - a|, otherwise ulp(x) is the distance between the two non-equal finite floating-point numbers nearest x. Moreover, ulp(NaN) is NaN.

The metadata node shall consist of a single positive floating point number representing the maximum relative error, for example:

```
!0 = metadata !{ float 2.5 } ; maximum acceptable inaccuracy is 2.5 ULPs
```

'range' Metadata

range metadata may be attached only to load, call and invoke of integer types. It expresses the possible ranges the loaded value or the value returned by the called function at this call site is in. The ranges are represented with a flattened list of integers. The loaded value or the value returned is known to be in the union of the ranges defined by each consecutive pair. Each pair has the following properties:

- The type must match the type loaded by the instruction.
- The pair a,b represents the range [a,b).
- Both a and b are constants.
- The range is allowed to wrap.
- The range should not represent the full or empty set. That is, a!=b.

In addition, the pairs must be in signed order of the lower bound and they must be non-contiguous.

Examples:

```
  %a = load i8* %x, align 1, !range !0 ; Can only be 0 or 1
  %b = load i8* %y, align 1, !range !1 ; Can only be 255 (-1), 0 or 1
  %c = call i8 @foo(),       !range !2 ; Can only be 0, 1, 3, 4 or 5
  %d = invoke i8 @bar() to label %cont
         unwind label %lpad, !range !3 ; Can only be -2, -1, 3, 4 or 5
...
!0 = metadata !{ i8 0, i8 2 }
!1 = metadata !{ i8 255, i8 2 }
!2 = metadata !{ i8 0, i8 2, i8 3, i8 6 }
!3 = metadata !{ i8 -2, i8 0, i8 3, i8 6 }
```

'llvm.loop'

It is sometimes useful to attach information to loop constructs. Currently, loop metadata is implemented as metadata attached to the branch instruction in the loop latch block. This type of metadata refer to a metadata node that is guaranteed to be separate for each loop. The loop identifier metadata is specified with the name llvm.loop.

The loop identifier metadata is implemented using a metadata that refers to itself to avoid merging it with any other identifier metadata, e.g., during module linkage or function inlining. That is, each loop should refer to their own identification metadata even if they reside in separate functions. The following example contains loop identifier metadata for two separate loop constructs:

```
!0 = metadata !{ metadata !0 }
!1 = metadata !{ metadata !1 }
```

The loop identifier metadata can be used to specify additional per-loop metadata. Any operands after the first operand can be treated as user-defined metadata. For example the `llvm.loop.unroll.count` suggests an unroll factor to the loop unroller:

```
  br i1 %exitcond, label %._crit_edge, label %.lr.ph, !llvm.loop !0
...
!0 = metadata !{ metadata !0, metadata !1 }
!1 = metadata !{ metadata !"llvm.loop.unroll.count", i32 4 }
```

'`llvm.loop.vectorize`' and '`llvm.loop.interleave`'

Metadata prefixed with `llvm.loop.vectorize` or `llvm.loop.interleave` are used to control per-loop vectorization and interleaving parameters such as vectorization width and interleave count. These metadata should be used in conjunction with `llvm.loop` loop identification metadata. The `llvm.loop.vectorize` and `llvm.loop.interleave` metadata are only optimization hints and the optimizer will only interleave and vectorize loops if it believes it is safe to do so. The `llvm.mem.parallel_loop_access` metadata which contains information about loop-carried memory dependencies can be helpful in determining the safety of these transformations.

'`llvm.loop.interleave.count`' Metadata

This metadata suggests an interleave count to the loop interleaver. The first operand is the string `llvm.loop.interleave.count` and the second operand is an integer specifying the interleave count. For example:

```
!0 = metadata !{ metadata !"llvm.loop.interleave.count", i32 4 }
```

Note that setting `llvm.loop.interleave.count` to 1 disables interleaving multiple iterations of the loop. If `llvm.loop.interleave.count` is set to 0 then the interleave count will be determined automatically.

'`llvm.loop.vectorize.enable`' Metadata

This metadata selectively enables or disables vectorization for the loop. The first operand is the string `llvm.loop.vectorize.enable` and the second operand is a bit. If the bit operand value is 1 vectorization is enabled. A value of 0 disables vectorization:

```
!0 = metadata !{ metadata !"llvm.loop.vectorize.enable", i1 0 }
!1 = metadata !{ metadata !"llvm.loop.vectorize.enable", i1 1 }
```

'`llvm.loop.vectorize.width`' Metadata

This metadata sets the target width of the vectorizer. The first operand is the string `llvm.loop.vectorize.width` and the second operand is an integer specifying the width. For example:

```
!0 = metadata !{ metadata !"llvm.loop.vectorize.width", i32 4 }
```

Note that setting `llvm.loop.vectorize.width` to 1 disables vectorization of the loop. If `llvm.loop.vectorize.width` is set to 0 or if the loop does not have this metadata the width will be determined automatically.

'llvm.loop.unroll'

Metadata prefixed with `llvm.loop.unroll` are loop unrolling optimization hints such as the unroll factor. `llvm.loop.unroll` metadata should be used in conjunction with `llvm.loop` loop identification metadata. The `llvm.loop.unroll` metadata are only optimization hints and the unrolling will only be performed if the optimizer believes it is safe to do so.

'llvm.loop.unroll.count' Metadata

This metadata suggests an unroll factor to the loop unroller. The first operand is the string `llvm.loop.unroll.count` and the second operand is a positive integer specifying the unroll factor. For example:

```
!0 = metadata !{ metadata !"llvm.loop.unroll.count", i32 4 }
```

If the trip count of the loop is less than the unroll count the loop will be partially unrolled.

'llvm.loop.unroll.disable' Metadata

This metadata either disables loop unrolling. The metadata has a single operand which is the string `llvm.loop.unroll.disable`. For example:

```
!0 = metadata !{ metadata !"llvm.loop.unroll.disable" }
```

'llvm.loop.unroll.full' Metadata

This metadata either suggests that the loop should be unrolled fully. The metadata has a single operand which is the string `llvm.loop.unroll.disable`. For example:

```
!0 = metadata !{ metadata !"llvm.loop.unroll.full" }
```

'llvm.mem'

Metadata types used to annotate memory accesses with information helpful for optimizations are prefixed with `llvm.mem`.

'llvm.mem.parallel_loop_access' Metadata

The `llvm.mem.parallel_loop_access` metadata refers to a loop identifier, or metadata containing a list of loop identifiers for nested loops. The metadata is attached to memory accessing instructions and denotes that no loop carried memory dependence exist between it and other instructions denoted with the same loop identifier.

Precisely, given two instructions `m1` and `m2` that both have the `llvm.mem.parallel_loop_access` metadata, with `L1` and `L2` being the set of loops associated with that metadata, respectively, then there is no loop carried dependence between `m1` and `m2` for loops in both `L1` and `L2`.

As a special case, if all memory accessing instructions in a loop have `llvm.mem.parallel_loop_access` metadata that refers to that loop, then the loop has no loop carried memory dependences and is considered to be a parallel loop.

Note that if not all memory access instructions have such metadata referring to the loop, then the loop is considered not being trivially parallel. Additional memory dependence analysis is required to make that determination. As a fail safe mechanism, this causes loops that were originally parallel to be considered sequential (if optimization passes that are unaware of the parallel semantics insert new memory instructions into the loop body).

Example of a loop that is considered parallel due to its correct use of both `llvm.loop` and `llvm.mem.parallel_loop_access` metadata types that refer to the same loop identifier metadata.

```
for.body:
  ...
  %val0 = load i32* %arrayidx, !llvm.mem.parallel_loop_access !0
  ...
  store i32 %val0, i32* %arrayidx1, !llvm.mem.parallel_loop_access !0
  ...
  br i1 %exitcond, label %for.end, label %for.body, !llvm.loop !0

for.end:
...
!0 = metadata !{ metadata !0 }
```

It is also possible to have nested parallel loops. In that case the memory accesses refer to a list of loop identifier metadata nodes instead of the loop identifier metadata node directly:

```
outer.for.body:
  ...
  %val1 = load i32* %arrayidx3, !llvm.mem.parallel_loop_access !2
  ...
  br label %inner.for.body

inner.for.body:
  ...
  %val0 = load i32* %arrayidx1, !llvm.mem.parallel_loop_access !0
  ...
  store i32 %val0, i32* %arrayidx2, !llvm.mem.parallel_loop_access !0
  ...
  br i1 %exitcond, label %inner.for.end, label %inner.for.body, !llvm.loop !1

inner.for.end:
  ...
  store i32 %val1, i32* %arrayidx4, !llvm.mem.parallel_loop_access !2
  ...
  br i1 %exitcond, label %outer.for.end, label %outer.for.body, !llvm.loop !2

outer.for.end:                                    ; preds = %for.body
...
!0 = metadata !{ metadata !1, metadata !2 } ; a list of loop identifiers
!1 = metadata !{ metadata !1 } ; an identifier for the inner loop
!2 = metadata !{ metadata !2 } ; an identifier for the outer loop
```

1.1.8 Module Flags Metadata

Information about the module as a whole is difficult to convey to LLVM's subsystems. The LLVM IR isn't sufficient to transmit this information. The `llvm.module.flags` named metadata exists in order to facilitate this. These flags are in the form of key / value pairs — much like a dictionary — making it easy for any subsystem who cares about a flag to look it up.

The `llvm.module.flags` metadata contains a list of metadata triplets. Each triplet has the following form:

- The first element is a *behavior* flag, which specifies the behavior when two (or more) modules are merged together, and it encounters two (or more) metadata with the same ID. The supported behaviors are described below.
- The second element is a metadata string that is a unique ID for the metadata. Each module may only have one flag entry for each unique ID (not including entries with the **Require** behavior).
- The third element is the value of the flag.

When two (or more) modules are merged together, the resulting `llvm.module.flags` metadata is the union of the modules' flags. That is, for each unique metadata ID string, there will be exactly one entry in the merged modules `llvm.module.flags` metadata table, and the value for that entry will be determined by the merge behavior flag, as described below. The only exception is that entries with the *Require* behavior are always preserved.

The following behaviors are supported:

Value	Behavior
1	**Error** Emits an error if two values disagree, otherwise the resulting value is that of the operands.
2	**Warning** Emits a warning if two values disagree. The result value will be the operand for the flag from the first module being linked.
3	**Require** Adds a requirement that another module flag be present and have a specified value after linking is performed. The value must be a metadata pair, where the first element of the pair is the ID of the module flag to be restricted, and the second element of the pair is the value the module flag should be restricted to. This behavior can be used to restrict the allowable results (via triggering of an error) of linking IDs with the **Override** behavior.
4	**Override** Uses the specified value, regardless of the behavior or value of the other module. If both modules specify **Override**, but the values differ, an error will be emitted.
5	**Append** Appends the two values, which are required to be metadata nodes.
6	**AppendUnique** Appends the two values, which are required to be metadata nodes. However, duplicate entries in the second list are dropped during the append operation.

It is an error for a particular unique flag ID to have multiple behaviors, except in the case of **Require** (which adds restrictions on another metadata value) or **Override**.

An example of module flags:

```
!0 = metadata !{ i32 1, metadata !"foo", i32 1 }
!1 = metadata !{ i32 4, metadata !"bar", i32 37 }
!2 = metadata !{ i32 2, metadata !"qux", i32 42 }
!3 = metadata !{ i32 3, metadata !"qux",
  metadata !{
    metadata !"foo", i32 1
  }
}
!llvm.module.flags = !{ !0, !1, !2, !3 }
```

- Metadata `!0` has the ID `!"foo"` and the value ‘1’. The behavior if two or more `!"foo"` flags are seen is to emit an error if their values are not equal.
- Metadata `!1` has the ID `!"bar"` and the value ‘37’. The behavior if two or more `!"bar"` flags are seen is to use the value ‘37’.

- Metadata `!2` has the ID `!"qux"` and the value '42'. The behavior if two or more `!"qux"` flags are seen is to emit a warning if their values are not equal.
- Metadata `!3` has the ID `!"qux"` and the value:

  ```
  metadata !{ metadata !"foo", i32 1 }
  ```

  The behavior is to emit an error if the `llvm.module.flags` does not contain a flag with the ID `!"foo"` that has the value '1' after linking is performed.

Objective-C Garbage Collection Module Flags Metadata

On the Mach-O platform, Objective-C stores metadata about garbage collection in a special section called "image info". The metadata consists of a version number and a bitmask specifying what types of garbage collection are supported (if any) by the file. If two or more modules are linked together their garbage collection metadata needs to be merged rather than appended together.

The Objective-C garbage collection module flags metadata consists of the following key-value pairs:

Key	Value
`Objective-C Version`	**[Required]** — The Objective-C ABI version. Valid values are 1 and 2.
`Objective-C Image Info Version`	**[Required]** — The version of the image info section. Currently always 0.
`Objective-C Image Info Section`	**[Required]** — The section to place the metadata. Valid values are `"__OBJC, __image_info, regular"` for Objective-C ABI version 1, and `"__DATA,__objc_imageinfo, regular, no_dead_strip"` for Objective-C ABI version 2.
`Objective-C Garbage Collection`	**[Required]** — Specifies whether garbage collection is supported or not. Valid values are 0, for no garbage collection, and 2, for garbage collection supported.
`Objective-C GC Only`	**[Optional]** — Specifies that only garbage collection is supported. If present, its value must be 6. This flag requires that the `Objective-C Garbage Collection` flag have the value 2.

Some important flag interactions:

- If a module with `Objective-C Garbage Collection` set to 0 is merged with a module with `Objective-C Garbage Collection` set to 2, then the resulting module has the `Objective-C Garbage Collection` flag set to 0.
- A module with `Objective-C Garbage Collection` set to 0 cannot be merged with a module with `Objective-C GC Only` set to 6.

Automatic Linker Flags Module Flags Metadata

Some targets support embedding flags to the linker inside individual object files. Typically this is used in conjunction with language extensions which allow source files to explicitly declare the libraries they depend on, and have these automatically be transmitted to the linker via object files.

These flags are encoded in the IR using metadata in the module flags section, using the `Linker Options` key. The merge behavior for this flag is required to be `AppendUnique`, and the value for the key is expected to be a metadata node which should be a list of other metadata nodes, each of which should be a list of metadata strings defining linker options.

For example, the following metadata section specifies two separate sets of linker options, presumably to link against `libz` and the `Cocoa` framework:

```
!0 = metadata !{ i32 6, metadata !"Linker Options",
   metadata !{
      metadata !{ metadata !"-lz" },
      metadata !{ metadata !"-framework", metadata !"Cocoa" } } }
!llvm.module.flags = !{ !0 }
```

The metadata encoding as lists of lists of options, as opposed to a collapsed list of options, is chosen so that the IR encoding can use multiple option strings to specify e.g., a single library, while still having that specifier be preserved as an atomic element that can be recognized by a target specific assembly writer or object file emitter.

Each individual option is required to be either a valid option for the target's linker, or an option that is reserved by the target specific assembly writer or object file emitter. No other aspect of these options is defined by the IR.

C type width Module Flags Metadata

The ARM backend emits a section into each generated object file describing the options that it was compiled with (in a compiler-independent way) to prevent linking incompatible objects, and to allow automatic library selection. Some of these options are not visible at the IR level, namely wchar_t width and enum width.

To pass this information to the backend, these options are encoded in module flags metadata, using the following key-value pairs:

Key	Value
short_wchar	• 0 — sizeof(wchar_t) == 4 • 1 — sizeof(wchar_t) == 2
short_enum	• 0 — Enums are at least as large as an `int`. • 1 — Enums are stored in the smallest integer type which can represent all of its values.

For example, the following metadata section specifies that the module was compiled with a `wchar_t` width of 4 bytes, and the underlying type of an enum is the smallest type which can represent all of its values:

```
!llvm.module.flags = !{!0, !1}
!0 = metadata !{i32 1, metadata !"short_wchar", i32 1}
!1 = metadata !{i32 1, metadata !"short_enum", i32 0}
```

1.1.9 Intrinsic Global Variables

LLVM has a number of "magic" global variables that contain data that affect code generation or other IR semantics. These are documented here. All globals of this sort should have a section specified as "`llvm.metadata`". This section and all globals that start with "`llvm.`" are reserved for use by LLVM.

The '`llvm.used`' Global Variable

The `@llvm.used` global is an array which has *appending linkage*. This array contains a list of pointers to named global variables, functions and aliases which may optionally have a pointer cast formed of bitcast or getelementptr. For example, a legal use of it is:

```
@X = global i8 4
@Y = global i32 123

@llvm.used = appending global [2 x i8*] [
   i8* @X,
   i8* bitcast (i32* @Y to i8*)
], section "llvm.metadata"
```

If a symbol appears in the `@llvm.used` list, then the compiler, assembler, and linker are required to treat the symbol as if there is a reference to the symbol that it cannot see (which is why they have to be named). For example, if a variable has internal linkage and no references other than that from the `@llvm.used` list, it cannot be deleted. This is commonly used to represent references from inline asms and other things the compiler cannot "see", and corresponds to "`attribute((used))`" in GNU C.

On some targets, the code generator must emit a directive to the assembler or object file to prevent the assembler and linker from molesting the symbol.

The '`llvm.compiler.used`' Global Variable

The `@llvm.compiler.used` directive is the same as the `@llvm.used` directive, except that it only prevents the compiler from touching the symbol. On targets that support it, this allows an intelligent linker to optimize references to the symbol without being impeded as it would be by `@llvm.used`.

This is a rare construct that should only be used in rare circumstances, and should not be exposed to source languages.

The '`llvm.global_ctors`' Global Variable

```
%0 = type { i32, void ()*, i8* }
@llvm.global_ctors = appending global [1 x %0] [%0 { i32 65535, void ()* @ctor, i8* @data }]
```

The `@llvm.global_ctors` array contains a list of constructor functions, priorities, and an optional associated global or function. The functions referenced by this array will be called in ascending order of priority (i.e. lowest first) when the module is loaded. The order of functions with the same priority is not defined.

If the third field is present, non-null, and points to a global variable or function, the initializer function will only run if the associated data from the current module is not discarded.

The '`llvm.global_dtors`' Global Variable

```
%0 = type { i32, void ()*, i8* }
@llvm.global_dtors = appending global [1 x %0] [%0 { i32 65535, void ()* @dtor, i8* @data }]
```

The `@llvm.global_dtors` array contains a list of destructor functions, priorities, and an optional associated global or function. The functions referenced by this array will be called in descending order of priority (i.e. highest first) when the module is unloaded. The order of functions with the same priority is not defined.

If the third field is present, non-null, and points to a global variable or function, the destructor function will only run if the associated data from the current module is not discarded.

1.1.10 Instruction Reference

The LLVM instruction set consists of several different classifications of instructions: *terminator instructions*, *binary instructions*, *bitwise binary instructions*, *memory instructions*, and *other instructions*.

Terminator Instructions

As mentioned *previously*, every basic block in a program ends with a "Terminator" instruction, which indicates which block should be executed after the current block is finished. These terminator instructions typically yield a '`void`' value: they produce control flow, not values (the one exception being the '*invoke*' instruction).

The terminator instructions are: '*ret*', '*br*', '*switch*', '*indirectbr*', '*invoke*', '*resume*', and '*unreachable*'.

'`ret`' Instruction

Syntax:

```
ret <type> <value>       ; Return a value from a non-void function
ret void                 ; Return from void function
```

Overview: The '`ret`' instruction is used to return control flow (and optionally a value) from a function back to the caller.

There are two forms of the '`ret`' instruction: one that returns a value and then causes control flow, and one that just causes control flow to occur.

Arguments: The '`ret`' instruction optionally accepts a single argument, the return value. The type of the return value must be a '*first class*' type.

A function is not *well formed* if it it has a non-void return type and contains a '`ret`' instruction with no return value or a return value with a type that does not match its type, or if it has a void return type and contains a '`ret`' instruction with a return value.

Semantics: When the '`ret`' instruction is executed, control flow returns back to the calling function's context. If the caller is a "*call*" instruction, execution continues at the instruction after the call. If the caller was an "*invoke*" instruction, execution continues at the beginning of the "normal" destination block. If the instruction returns a value, that value shall set the call or invoke instruction's return value.

Example:

```
ret i32 5                       ; Return an integer value of 5
ret void                        ; Return from a void function
ret { i32, i8 } { i32 4, i8 2 } ; Return a struct of values 4 and 2
```

'`br`' Instruction

Syntax:

```
br i1 <cond>, label <iftrue>, label <iffalse>
br label <dest>          ; Unconditional branch
```

Overview: The '`br`' instruction is used to cause control flow to transfer to a different basic block in the current function. There are two forms of this instruction, corresponding to a conditional branch and an unconditional branch.

Arguments: The conditional branch form of the '`br`' instruction takes a single '`i1`' value and two '`label`' values. The unconditional form of the '`br`' instruction takes a single '`label`' value as a target.

Semantics: Upon execution of a conditional '`br`' instruction, the '`i1`' argument is evaluated. If the value is `true`, control flows to the '`iftrue`' `label` argument. If "cond" is `false`, control flows to the '`iffalse`' `label` argument.

Example:

```
Test:
  %cond = icmp eq i32 %a, %b
  br i1 %cond, label %IfEqual, label %IfUnequal
IfEqual:
  ret i32 1
IfUnequal:
  ret i32 0
```

'`switch`' Instruction

Syntax:

```
switch <intty> <value>, label <defaultdest> [ <intty> <val>, label <dest> ... ]
```

Overview: The '`switch`' instruction is used to transfer control flow to one of several different places. It is a generalization of the '`br`' instruction, allowing a branch to occur to one of many possible destinations.

Arguments: The '`switch`' instruction uses three parameters: an integer comparison value '`value`', a default '`label`' destination, and an array of pairs of comparison value constants and '`label`'s. The table is not allowed to contain duplicate constant entries.

Semantics: The `switch` instruction specifies a table of values and destinations. When the '`switch`' instruction is executed, this table is searched for the given value. If the value is found, control flow is transferred to the corresponding destination; otherwise, control flow is transferred to the default destination.

Implementation: Depending on properties of the target machine and the particular `switch` instruction, this instruction may be code generated in different ways. For example, it could be generated as a series of chained conditional branches or with a lookup table.

Example:

```
; Emulate a conditional br instruction
%Val = zext i1 %value to i32
switch i32 %Val, label %truedest [ i32 0, label %falsedest ]

; Emulate an unconditional br instruction
switch i32 0, label %dest [ ]

; Implement a jump table:
switch i32 %val, label %otherwise [ i32 0, label %onzero
                                    i32 1, label %onone
                                    i32 2, label %ontwo ]
```

'indirectbr' Instruction

Syntax:

```
indirectbr <somety>* <address>, [ label <dest1>, label <dest2>, ... ]
```

Overview: The 'indirectbr' instruction implements an indirect branch to a label within the current function, whose address is specified by "address". Address must be derived from a *blockaddress* constant.

Arguments: The 'address' argument is the address of the label to jump to. The rest of the arguments indicate the full set of possible destinations that the address may point to. Blocks are allowed to occur multiple times in the destination list, though this isn't particularly useful.

This destination list is required so that dataflow analysis has an accurate understanding of the CFG.

Semantics: Control transfers to the block specified in the address argument. All possible destination blocks must be listed in the label list, otherwise this instruction has undefined behavior. This implies that jumps to labels defined in other functions have undefined behavior as well.

Implementation: This is typically implemented with a jump through a register.

Example:

```
indirectbr i8* %Addr, [ label %bb1, label %bb2, label %bb3 ]
```

'invoke' Instruction

Syntax:

```
<result> = invoke [cconv] [ret attrs] <ptr to function ty> <function ptr val>(<function args>) [fn at
              to label <normal label> unwind label <exception label>
```

Overview: The 'invoke' instruction causes control to transfer to a specified function, with the possibility of control flow transfer to either the 'normal' label or the 'exception' label. If the callee function returns with the "ret" instruction, control flow will return to the "normal" label. If the callee (or any indirect callees) returns via the "*resume*" instruction or other exception handling mechanism, control is interrupted and continued at the dynamically nearest "exception" label.

The 'exception' label is a landing pad for the exception. As such, 'exception' label is required to have the "*landingpad*" instruction, which contains the information about the behavior of the program after unwinding happens, as its first non-PHI instruction. The restrictions on the "landingpad" instruction's tightly couples it to the "invoke" instruction, so that the important information contained within the "landingpad" instruction can't be lost through normal code motion.

Arguments: This instruction requires several arguments:

1. The optional "cconv" marker indicates which *calling convention* the call should use. If none is specified, the call defaults to using C calling conventions.
2. The optional *Parameter Attributes* list for return values. Only 'zeroext', 'signext', and 'inreg' attributes are valid here.

3. ‘`ptr to function ty`‘: shall be the signature of the pointer to function value being invoked. In most cases, this is a direct function invocation, but indirect `invoke`‘s are just as possible, branching off an arbitrary pointer to function value.
4. ‘`function ptr val`‘: An LLVM value containing a pointer to a function to be invoked.
5. ‘`function args`‘: argument list whose types match the function signature argument types and parameter attributes. All arguments must be of *first class* type. If the function signature indicates the function accepts a variable number of arguments, the extra arguments can be specified.
6. ‘`normal label`‘: the label reached when the called function executes a ‘`ret`‘ instruction.
7. ‘`exception label`‘: the label reached when a callee returns via the *resume* instruction or other exception handling mechanism.
8. The optional *function attributes* list. Only ‘`noreturn`‘, ‘`nounwind`‘, ‘`readonly`‘ and ‘`readnone`‘ attributes are valid here.

Semantics: This instruction is designed to operate as a standard ‘`call`‘ instruction in most regards. The primary difference is that it establishes an association with a label, which is used by the runtime library to unwind the stack.

This instruction is used in languages with destructors to ensure that proper cleanup is performed in the case of either a `longjmp` or a thrown exception. Additionally, this is important for implementation of ‘`catch`‘ clauses in high-level languages that support them.

For the purposes of the SSA form, the definition of the value returned by the ‘`invoke`‘ instruction is deemed to occur on the edge from the current block to the “normal” label. If the callee unwinds then no return value is available.

Example:

```
%retval = invoke i32 @Test(i32 15) to label %Continue
            unwind label %TestCleanup              ; i32:retval set
%retval = invoke coldcc i32 %Testfnptr(i32 15) to label %Continue
            unwind label %TestCleanup              ; i32:retval set
```

‘`resume`‘ Instruction

Syntax:

```
resume <type> <value>
```

Overview: The ‘`resume`‘ instruction is a terminator instruction that has no successors.

Arguments: The ‘`resume`‘ instruction requires one argument, which must have the same type as the result of any ‘`landingpad`‘ instruction in the same function.

Semantics: The ‘`resume`‘ instruction resumes propagation of an existing (in-flight) exception whose unwinding was interrupted with a *landingpad* instruction.

Example:

```
resume { i8*, i32 } %exn
```

'unreachable' Instruction

Syntax:

```
unreachable
```

Overview: The 'unreachable' instruction has no defined semantics. This instruction is used to inform the optimizer that a particular portion of the code is not reachable. This can be used to indicate that the code after a no-return function cannot be reached, and other facts.

Semantics: The 'unreachable' instruction has no defined semantics.

Binary Operations

Binary operators are used to do most of the computation in a program. They require two operands of the same type, execute an operation on them, and produce a single value. The operands might represent multiple data, as is the case with the *vector* data type. The result value has the same type as its operands.

There are several different binary operators:

'add' Instruction

Syntax:

```
<result> = add <ty> <op1>, <op2>           ; yields ty:result
<result> = add nuw <ty> <op1>, <op2>       ; yields ty:result
<result> = add nsw <ty> <op1>, <op2>       ; yields ty:result
<result> = add nuw nsw <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'add' instruction returns the sum of its two operands.

Arguments: The two arguments to the 'add' instruction must be *integer* or *vector* of integer values. Both arguments must have identical types.

Semantics: The value produced is the integer sum of the two operands.

If the sum has unsigned overflow, the result returned is the mathematical result modulo 2^n, where n is the bit width of the result.

Because LLVM integers use a two's complement representation, this instruction is appropriate for both signed and unsigned integers.

nuw and nsw stand for "No Unsigned Wrap" and "No Signed Wrap", respectively. If the nuw and/or nsw keywords are present, the result value of the add is a *poison value* if unsigned and/or signed overflow, respectively, occurs.

Example:

```
<result> = add i32 4, %var          ; yields i32:result = 4 + %var
```

'fadd' Instruction

Syntax:

```
<result> = fadd [fast-math flags]* <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'fadd' instruction returns the sum of its two operands.

Arguments: The two arguments to the 'fadd' instruction must be *floating point* or *vector* of floating point values. Both arguments must have identical types.

Semantics: The value produced is the floating point sum of the two operands. This instruction can also take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations:

Example:

```
<result> = fadd float 4.0, %var          ; yields float:result = 4.0 + %var
```

'sub' Instruction

Syntax:

```
<result> = sub <ty> <op1>, <op2>           ; yields ty:result
<result> = sub nuw <ty> <op1>, <op2>       ; yields ty:result
<result> = sub nsw <ty> <op1>, <op2>       ; yields ty:result
<result> = sub nuw nsw <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'sub' instruction returns the difference of its two operands.

Note that the 'sub' instruction is used to represent the 'neg' instruction present in most other intermediate representations.

Arguments: The two arguments to the 'sub' instruction must be *integer* or *vector* of integer values. Both arguments must have identical types.

Semantics: The value produced is the integer difference of the two operands.

If the difference has unsigned overflow, the result returned is the mathematical result modulo 2^n, where n is the bit width of the result.

Because LLVM integers use a two's complement representation, this instruction is appropriate for both signed and unsigned integers.

nuw and nsw stand for "No Unsigned Wrap" and "No Signed Wrap", respectively. If the nuw and/or nsw keywords are present, the result value of the sub is a *poison value* if unsigned and/or signed overflow, respectively, occurs.

Example:

```
<result> = sub i32 4, %var          ; yields i32:result = 4 - %var
<result> = sub i32 0, %val          ; yields i32:result = -%var
```

'fsub' Instruction

Syntax:

```
<result> = fsub [fast-math flags]* <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'fsub' instruction returns the difference of its two operands.

Note that the 'fsub' instruction is used to represent the 'fneg' instruction present in most other intermediate representations.

Arguments: The two arguments to the 'fsub' instruction must be *floating point* or *vector* of floating point values. Both arguments must have identical types.

Semantics: The value produced is the floating point difference of the two operands. This instruction can also take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations:

Example:

```
<result> = fsub float 4.0, %var           ; yields float:result = 4.0 - %var
<result> = fsub float -0.0, %val          ; yields float:result = -%var
```

'mul' Instruction

Syntax:

```
<result> = mul <ty> <op1>, <op2>          ; yields ty:result
<result> = mul nuw <ty> <op1>, <op2>      ; yields ty:result
<result> = mul nsw <ty> <op1>, <op2>      ; yields ty:result
<result> = mul nuw nsw <ty> <op1>, <op2>  ; yields ty:result
```

Overview: The 'mul' instruction returns the product of its two operands.

Arguments: The two arguments to the 'mul' instruction must be *integer* or *vector* of integer values. Both arguments must have identical types.

Semantics: The value produced is the integer product of the two operands.

If the result of the multiplication has unsigned overflow, the result returned is the mathematical result modulo 2^n, where n is the bit width of the result.

Because LLVM integers use a two's complement representation, and the result is the same width as the operands, this instruction returns the correct result for both signed and unsigned integers. If a full product (e.g. i32 * i32 -> i64) is needed, the operands should be sign-extended or zero-extended as appropriate to the width of the full product.

nuw and nsw stand for "No Unsigned Wrap" and "No Signed Wrap", respectively. If the nuw and/or nsw keywords are present, the result value of the mul is a *poison value* if unsigned and/or signed overflow, respectively, occurs.

Example:

```
<result> = mul i32 4, %var          ; yields i32:result = 4 * %var
```

'fmul' Instruction

Syntax:

```
<result> = fmul [fast-math flags]* <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'fmul' instruction returns the product of its two operands.

Arguments: The two arguments to the 'fmul' instruction must be *floating point* or *vector* of floating point values. Both arguments must have identical types.

Semantics: The value produced is the floating point product of the two operands. This instruction can also take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations:

Example:

```
<result> = fmul float 4.0, %var          ; yields float:result = 4.0 * %var
```

'udiv' Instruction

Syntax:

```
<result> = udiv <ty> <op1>, <op2>         ; yields ty:result
<result> = udiv exact <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'udiv' instruction returns the quotient of its two operands.

Arguments: The two arguments to the 'udiv' instruction must be *integer* or *vector* of integer values. Both arguments must have identical types.

Semantics: The value produced is the unsigned integer quotient of the two operands.

Note that unsigned integer division and signed integer division are distinct operations; for signed integer division, use 'sdiv'.

Division by zero leads to undefined behavior.

If the exact keyword is present, the result value of the udiv is a *poison value* if %op1 is not a multiple of %op2 (as such, "((a udiv exact b) mul b) == a").

Example:

```
<result> = udiv i32 4, %var          ; yields i32:result = 4 / %var
```

'sdiv' Instruction

Syntax:

```
<result> = sdiv <ty> <op1>, <op2>         ; yields ty:result
<result> = sdiv exact <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'sdiv' instruction returns the quotient of its two operands.

Arguments: The two arguments to the 'sdiv' instruction must be *integer* or *vector* of integer values. Both arguments must have identical types.

Semantics: The value produced is the signed integer quotient of the two operands rounded towards zero.

Note that signed integer division and unsigned integer division are distinct operations; for unsigned integer division, use 'udiv'.

Division by zero leads to undefined behavior. Overflow also leads to undefined behavior; this is a rare case, but can occur, for example, by doing a 32-bit division of -2147483648 by -1.

If the exact keyword is present, the result value of the sdiv is a *poison value* if the result would be rounded.

Example:

```
<result> = sdiv i32 4, %var          ; yields i32:result = 4 / %var
```

'fdiv' Instruction

Syntax:

```
<result> = fdiv [fast-math flags]* <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'fdiv' instruction returns the quotient of its two operands.

Arguments: The two arguments to the 'fdiv' instruction must be *floating point* or *vector* of floating point values. Both arguments must have identical types.

Semantics: The value produced is the floating point quotient of the two operands. This instruction can also take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations:

Example:

```
<result> = fdiv float 4.0, %var          ; yields float:result = 4.0 / %var
```

'urem' Instruction

Syntax:

```
<result> = urem <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'urem' instruction returns the remainder from the unsigned division of its two arguments.

Arguments: The two arguments to the 'urem' instruction must be *integer* or *vector* of integer values. Both arguments must have identical types.

Semantics: This instruction returns the unsigned integer *remainder* of a division. This instruction always performs an unsigned division to get the remainder.

Note that unsigned integer remainder and signed integer remainder are distinct operations; for signed integer remainder, use '`srem`'.

Taking the remainder of a division by zero leads to undefined behavior.

Example:

```
<result> = urem i32 4, %var          ; yields i32:result = 4 % %var
```

'`srem`' Instruction

Syntax:

```
<result> = srem <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The '`srem`' instruction returns the remainder from the signed division of its two operands. This instruction can also take *vector* versions of the values in which case the elements must be integers.

Arguments: The two arguments to the '`srem`' instruction must be *integer* or *vector* of integer values. Both arguments must have identical types.

Semantics: This instruction returns the *remainder* of a division (where the result is either zero or has the same sign as the dividend, `op1`), not the *modulo* operator (where the result is either zero or has the same sign as the divisor, `op2`) of a value. For more information about the difference, see The Math Forum. For a table of how this is implemented in various languages, please see Wikipedia: modulo operation.

Note that signed integer remainder and unsigned integer remainder are distinct operations; for unsigned integer remainder, use '`urem`'.

Taking the remainder of a division by zero leads to undefined behavior. Overflow also leads to undefined behavior; this is a rare case, but can occur, for example, by taking the remainder of a 32-bit division of -2147483648 by -1. (The remainder doesn't actually overflow, but this rule lets srem be implemented using instructions that return both the result of the division and the remainder.)

Example:

```
<result> = srem i32 4, %var          ; yields i32:result = 4 % %var
```

'`frem`' Instruction

Syntax:

```
<result> = frem [fast-math flags]* <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The '`frem`' instruction returns the remainder from the division of its two operands.

Arguments: The two arguments to the '`frem`' instruction must be *floating point* or *vector* of floating point values. Both arguments must have identical types.

Semantics: This instruction returns the *remainder* of a division. The remainder has the same sign as the dividend. This instruction can also take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations:

Example:

```
<result> = frem float 4.0, %var          ; yields float:result = 4.0 % %var
```

Bitwise Binary Operations

Bitwise binary operators are used to do various forms of bit-twiddling in a program. They are generally very efficient instructions and can commonly be strength reduced from other instructions. They require two operands of the same type, execute an operation on them, and produce a single value. The resulting value is the same type as its operands.

'shl' Instruction

Syntax:

```
<result> = shl <ty> <op1>, <op2>           ; yields ty:result
<result> = shl nuw <ty> <op1>, <op2>       ; yields ty:result
<result> = shl nsw <ty> <op1>, <op2>       ; yields ty:result
<result> = shl nuw nsw <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'shl' instruction returns the first operand shifted to the left a specified number of bits.

Arguments: Both arguments to the 'shl' instruction must be the same *integer* or *vector* of integer type. 'op2' is treated as an unsigned value.

Semantics: The value produced is op1 * 2^{op2} mod 2^n, where n is the width of the result. If op2 is (statically or dynamically) negative or equal to or larger than the number of bits in op1, the result is undefined. If the arguments are vectors, each vector element of op1 is shifted by the corresponding shift amount in op2.

If the nuw keyword is present, then the shift produces a *poison value* if it shifts out any non-zero bits. If the nsw keyword is present, then the shift produces a *poison value* if it shifts out any bits that disagree with the resultant sign bit. As such, NUW/NSW have the same semantics as they would if the shift were expressed as a mul instruction with the same nsw/nuw bits in (mul %op1, (shl 1, %op2)).

Example:

```
<result> = shl i32 4, %var   ; yields i32: 4 << %var
<result> = shl i32 4, 2      ; yields i32: 16
<result> = shl i32 1, 10     ; yields i32: 1024
<result> = shl i32 1, 32     ; undefined
<result> = shl <2 x i32> < i32 1, i32 1>, < i32 1, i32 2>   ; yields: result=<2 x i32> < i32 2, i32 4
```

'lshr' Instruction

Syntax:

```
<result> = lshr <ty> <op1>, <op2>         ; yields ty:result
<result> = lshr exact <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The '`lshr`' instruction (logical shift right) returns the first operand shifted to the right a specified number of bits with zero fill.

Arguments: Both arguments to the '`lshr`' instruction must be the same *integer* or *vector* of integer type. '`op2`' is treated as an unsigned value.

Semantics: This instruction always performs a logical shift right operation. The most significant bits of the result will be filled with zero bits after the shift. If `op2` is (statically or dynamically) equal to or larger than the number of bits in `op1`, the result is undefined. If the arguments are vectors, each vector element of `op1` is shifted by the corresponding shift amount in `op2`.

If the `exact` keyword is present, the result value of the `lshr` is a *poison value* if any of the bits shifted out are non-zero.

Example:

```
<result> = lshr i32 4, 1   ; yields i32:result = 2
<result> = lshr i32 4, 2   ; yields i32:result = 1
<result> = lshr i8  4, 3   ; yields i8:result = 0
<result> = lshr i8 -2, 1   ; yields i8:result = 0x7F
<result> = lshr i32 1, 32  ; undefined
<result> = lshr <2 x i32> < i32 -2, i32 4>, < i32 1, i32 2>   ; yields: result=<2 x i32> < i32 0x7FF
```

'`ashr`' Instruction

Syntax:

```
<result> = ashr <ty> <op1>, <op2>         ; yields ty:result
<result> = ashr exact <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The '`ashr`' instruction (arithmetic shift right) returns the first operand shifted to the right a specified number of bits with sign extension.

Arguments: Both arguments to the '`ashr`' instruction must be the same *integer* or *vector* of integer type. '`op2`' is treated as an unsigned value.

Semantics: This instruction always performs an arithmetic shift right operation, The most significant bits of the result will be filled with the sign bit of `op1`. If `op2` is (statically or dynamically) equal to or larger than the number of bits in `op1`, the result is undefined. If the arguments are vectors, each vector element of `op1` is shifted by the corresponding shift amount in `op2`.

If the `exact` keyword is present, the result value of the `ashr` is a *poison value* if any of the bits shifted out are non-zero.

Example:

```
<result> = ashr i32 4, 1   ; yields i32:result = 2
<result> = ashr i32 4, 2   ; yields i32:result = 1
<result> = ashr i8  4, 3   ; yields i8:result = 0
<result> = ashr i8 -2, 1   ; yields i8:result = -1
<result> = ashr i32 1, 32  ; undefined
<result> = ashr <2 x i32> < i32 -2, i32 4>, < i32 1, i32 3>   ; yields: result=<2 x i32> < i32 -1, i
```

'and' Instruction

Syntax:

```
<result> = and <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'and' instruction returns the bitwise logical and of its two operands.

Arguments: The two arguments to the 'and' instruction must be *integer* or *vector* of integer values. Both arguments must have identical types.

Semantics: The truth table used for the 'and' instruction is:

In0	In1	Out
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```
<result> = and i32 4, %var         ; yields i32:result = 4 & %var
<result> = and i32 15, 40          ; yields i32:result = 8
<result> = and i32 4, 8            ; yields i32:result = 0
```

'or' Instruction

Syntax:

```
<result> = or <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'or' instruction returns the bitwise logical inclusive or of its two operands.

Arguments: The two arguments to the 'or' instruction must be *integer* or *vector* of integer values. Both arguments must have identical types.

Semantics: The truth table used for the 'or' instruction is:

In0	In1	Out
0	0	0
0	1	1
1	0	1
1	1	1

Example:

```
<result> = or i32 4, %var          ; yields i32:result = 4 | %var
<result> = or i32 15, 40           ; yields i32:result = 47
<result> = or i32 4, 8             ; yields i32:result = 12
```

'xor' Instruction

Syntax:

```
<result> = xor <ty> <op1>, <op2>   ; yields ty:result
```

Overview: The 'xor' instruction returns the bitwise logical exclusive or of its two operands. The xor is used to implement the "one's complement" operation, which is the "~" operator in C.

Arguments: The two arguments to the 'xor' instruction must be *integer* or *vector* of integer values. Both arguments must have identical types.

Semantics: The truth table used for the 'xor' instruction is:

In0	In1	Out
0	0	0
0	1	1
1	0	1
1	1	0

Example:

```
<result> = xor i32 4, %var         ; yields i32:result = 4 ^ %var
<result> = xor i32 15, 40          ; yields i32:result = 39
<result> = xor i32 4, 8            ; yields i32:result = 12
<result> = xor i32 %V, -1          ; yields i32:result = ~%V
```

Vector Operations

LLVM supports several instructions to represent vector operations in a target-independent manner. These instructions cover the element-access and vector-specific operations needed to process vectors effectively. While LLVM does directly support these vector operations, many sophisticated algorithms will want to use target-specific intrinsics to take full advantage of a specific target.

'extractelement' Instruction

Syntax:

```
<result> = extractelement <n x <ty>> <val>, <ty2> <idx>  ; yields <ty>
```

Overview: The 'extractelement' instruction extracts a single scalar element from a vector at a specified index.

Arguments: The first operand of an 'extractelement' instruction is a value of *vector* type. The second operand is an index indicating the position from which to extract the element. The index may be a variable of any integer type.

Semantics: The result is a scalar of the same type as the element type of val. Its value is the value at position idx of val. If idx exceeds the length of val, the results are undefined.

Example:

```
<result> = extractelement <4 x i32> %vec, i32 0    ; yields i32
```

'`insertelement`' Instruction

Syntax:

```
<result> = insertelement <n x <ty>> <val>, <ty> <elt>, <ty2> <idx>    ; yields <n x <ty>>
```

Overview: The '`insertelement`' instruction inserts a scalar element into a vector at a specified index.

Arguments: The first operand of an '`insertelement`' instruction is a value of *vector* type. The second operand is a scalar value whose type must equal the element type of the first operand. The third operand is an index indicating the position at which to insert the value. The index may be a variable of any integer type.

Semantics: The result is a vector of the same type as `val`. Its element values are those of `val` except at position `idx`, where it gets the value `elt`. If `idx` exceeds the length of `val`, the results are undefined.

Example:

```
<result> = insertelement <4 x i32> %vec, i32 1, i32 0    ; yields <4 x i32>
```

'`shufflevector`' Instruction

Syntax:

```
<result> = shufflevector <n x <ty>> <v1>, <n x <ty>> <v2>, <m x i32> <mask>    ; yields <m x <ty>>
```

Overview: The '`shufflevector`' instruction constructs a permutation of elements from two input vectors, returning a vector with the same element type as the input and length that is the same as the shuffle mask.

Arguments: The first two operands of a '`shufflevector`' instruction are vectors with the same type. The third argument is a shuffle mask whose element type is always 'i32'. The result of the instruction is a vector whose length is the same as the shuffle mask and whose element type is the same as the element type of the first two operands.

The shuffle mask operand is required to be a constant vector with either constant integer or undef values.

Semantics: The elements of the two input vectors are numbered from left to right across both of the vectors. The shuffle mask operand specifies, for each element of the result vector, which element of the two input vectors the result element gets. The element selector may be undef (meaning "don't care") and the second operand may be undef if performing a shuffle from only one vector.

Example:

```
<result> = shufflevector <4 x i32> %v1, <4 x i32> %v2,
                         <4 x i32> <i32 0, i32 4, i32 1, i32 5>  ; yields <4 x i32>
<result> = shufflevector <4 x i32> %v1, <4 x i32> undef,
                         <4 x i32> <i32 0, i32 1, i32 2, i32 3>  ; yields <4 x i32> - Identity shuffle
<result> = shufflevector <8 x i32> %v1, <8 x i32> undef,
                         <4 x i32> <i32 0, i32 1, i32 2, i32 3>  ; yields <4 x i32>
```

```
<result> = shufflevector <4 x i32> %v1, <4 x i32> %v2,
                         <8 x i32> <i32 0, i32 1, i32 2, i32 3, i32 4, i32 5, i32 6, i32 7 >  ; yields
```

Aggregate Operations

LLVM supports several instructions for working with *aggregate* values.

‘`extractvalue`‘ Instruction

Syntax:

```
<result> = extractvalue <aggregate type> <val>, <idx>{, <idx>}*
```

Overview: The ‘`extractvalue`‘ instruction extracts the value of a member field from an *aggregate* value.

Arguments: The first operand of an ‘`extractvalue`‘ instruction is a value of *struct* or *array* type. The operands are constant indices to specify which value to extract in a similar manner as indices in a ‘`getelementptr`‘ instruction.

The major differences to `getelementptr` indexing are:

- Since the value being indexed is not a pointer, the first index is omitted and assumed to be zero.
- At least one index must be specified.
- Not only struct indices but also array indices must be in bounds.

Semantics: The result is the value at the position in the aggregate specified by the index operands.

Example:

```
<result> = extractvalue {i32, float} %agg, 0    ; yields i32
```

‘`insertvalue`‘ Instruction

Syntax:

```
<result> = insertvalue <aggregate type> <val>, <ty> <elt>, <idx>{, <idx>}*    ; yields <aggregate typ
```

Overview: The ‘`insertvalue`‘ instruction inserts a value into a member field in an *aggregate* value.

Arguments: The first operand of an ‘`insertvalue`‘ instruction is a value of *struct* or *array* type. The second operand is a first-class value to insert. The following operands are constant indices indicating the position at which to insert the value in a similar manner as indices in a ‘`extractvalue`‘ instruction. The value to insert must have the same type as the value identified by the indices.

Semantics: The result is an aggregate of the same type as `val`. Its value is that of `val` except that the value at the position specified by the indices is that of `elt`.

Example:

```
%agg1 = insertvalue {i32, float} undef, i32 1, 0              ; yields {i32 1, float undef}
%agg2 = insertvalue {i32, float} %agg1, float %val, 1         ; yields {i32 1, float %val}
%agg3 = insertvalue {i32, {float}} undef, float %val, 1, 0    ; yields {i32 undef, {float %val}}
```

Memory Access and Addressing Operations

A key design point of an SSA-based representation is how it represents memory. In LLVM, no memory locations are in SSA form, which makes things very simple. This section describes how to read, write, and allocate memory in LLVM.

'alloca' Instruction

Syntax:

```
<result> = alloca [inalloca] <type> [, <ty> <NumElements>] [, align <alignment>]     ; yields type*:r
```

Overview: The 'alloca' instruction allocates memory on the stack frame of the currently executing function, to be automatically released when this function returns to its caller. The object is always allocated in the generic address space (address space zero).

Arguments: The 'alloca' instruction allocates sizeof(<type>)*NumElements bytes of memory on the runtime stack, returning a pointer of the appropriate type to the program. If "NumElements" is specified, it is the number of elements allocated, otherwise "NumElements" is defaulted to be one. If a constant alignment is specified, the value result of the allocation is guaranteed to be aligned to at least that boundary. The alignment may not be greater than 1 << 29. If not specified, or if zero, the target can choose to align the allocation on any convenient boundary compatible with the type.

'type' may be any sized type.

Semantics: Memory is allocated; a pointer is returned. The operation is undefined if there is insufficient stack space for the allocation. 'alloca'd memory is automatically released when the function returns. The 'alloca' instruction is commonly used to represent automatic variables that must have an address available. When the function returns (either with the ret or resume instructions), the memory is reclaimed. Allocating zero bytes is legal, but the result is undefined. The order in which memory is allocated (ie., which way the stack grows) is not specified.

Example:

```
%ptr = alloca i32                               ; yields i32*:ptr
%ptr = alloca i32, i32 4                        ; yields i32*:ptr
%ptr = alloca i32, i32 4, align 1024            ; yields i32*:ptr
%ptr = alloca i32, align 1024                   ; yields i32*:ptr
```

'load' Instruction

Syntax:

```
<result> = load [volatile] <ty>* <pointer>[, align <alignment>][, !nontemporal !<index>][, !invariant
<result> = load atomic [volatile] <ty>* <pointer> [singlethread] <ordering>, align <alignment>
!<index> = !{ i32 1 }
```

Overview: The '`load`' instruction is used to read from memory.

Arguments: The argument to the `load` instruction specifies the memory address from which to load. The pointer must point to a *first class* type. If the `load` is marked as `volatile`, then the optimizer is not allowed to modify the number or order of execution of this `load` with other *volatile operations*.

If the `load` is marked as `atomic`, it takes an extra *ordering* and optional `singlethread` argument. The `release` and `acq_rel` orderings are not valid on `load` instructions. Atomic loads produce *defined* results when they may see multiple atomic stores. The type of the pointee must be an integer type whose bit width is a power of two greater than or equal to eight and less than or equal to a target-specific size limit. `align` must be explicitly specified on atomic loads, and the load has undefined behavior if the alignment is not set to a value which is at least the size in bytes of the pointee. `!nontemporal` does not have any defined semantics for atomic loads.

The optional constant `align` argument specifies the alignment of the operation (that is, the alignment of the memory address). A value of 0 or an omitted `align` argument means that the operation has the ABI alignment for the target. It is the responsibility of the code emitter to ensure that the alignment information is correct. Overestimating the alignment results in undefined behavior. Underestimating the alignment may produce less efficient code. An alignment of 1 is always safe. The maximum possible alignment is `1 << 29`.

The optional `!nontemporal` metadata must reference a single metadata name `<index>` corresponding to a metadata node with one `i32` entry of value 1. The existence of the `!nontemporal` metadata on the instruction tells the optimizer and code generator that this load is not expected to be reused in the cache. The code generator may select special instructions to save cache bandwidth, such as the `MOVNT` instruction on x86.

The optional `!invariant.load` metadata must reference a single metadata name `<index>` corresponding to a metadata node with no entries. The existence of the `!invariant.load` metadata on the instruction tells the optimizer and code generator that this load address points to memory which does not change value during program execution. The optimizer may then move this load around, for example, by hoisting it out of loops using loop invariant code motion.

Semantics: The location of memory pointed to is loaded. If the value being loaded is of scalar type then the number of bytes read does not exceed the minimum number of bytes needed to hold all bits of the type. For example, loading an `i24` reads at most three bytes. When loading a value of a type like `i20` with a size that is not an integral number of bytes, the result is undefined if the value was not originally written using a store of the same type.

Examples:

```
%ptr = alloca i32                               ; yields i32*:ptr
store i32 3, i32* %ptr                          ; yields void
%val = load i32* %ptr                           ; yields i32:val = i32 3
```

'`store`' Instruction

Syntax:

```
store [volatile] <ty> <value>, <ty>* <pointer>[, align <alignment>][, !nontemporal !<index>]
store atomic [volatile] <ty> <value>, <ty>* <pointer> [singlethread] <ordering>, align <alignment>
```

Overview: The '`store`' instruction is used to write to memory.

Arguments: There are two arguments to the `store` instruction: a value to store and an address at which to store it. The type of the `<pointer>` operand must be a pointer to the *first class* type of the `<value>` operand. If the `store`

is marked as `volatile`, then the optimizer is not allowed to modify the number or order of execution of this `store` with other *volatile operations*.

If the `store` is marked as `atomic`, it takes an extra *ordering* and optional `singlethread` argument. The `acquire` and `acq_rel` orderings aren't valid on `store` instructions. Atomic loads produce *defined* results when they may see multiple atomic stores. The type of the pointee must be an integer type whose bit width is a power of two greater than or equal to eight and less than or equal to a target-specific size limit. `align` must be explicitly specified on atomic stores, and the store has undefined behavior if the alignment is not set to a value which is at least the size in bytes of the pointee. `!nontemporal` does not have any defined semantics for atomic stores.

The optional constant `align` argument specifies the alignment of the operation (that is, the alignment of the memory address). A value of 0 or an omitted `align` argument means that the operation has the ABI alignment for the target. It is the responsibility of the code emitter to ensure that the alignment information is correct. Overestimating the alignment results in undefined behavior. Underestimating the alignment may produce less efficient code. An alignment of 1 is always safe. The maximum possible alignment is `1 << 29`.

The optional `!nontemporal` metadata must reference a single metadata name `<index>` corresponding to a metadata node with one `i32` entry of value 1. The existence of the `!nontemporal` metadata on the instruction tells the optimizer and code generator that this load is not expected to be reused in the cache. The code generator may select special instructions to save cache bandwidth, such as the MOVNT instruction on x86.

Semantics: The contents of memory are updated to contain `<value>` at the location specified by the `<pointer>` operand. If `<value>` is of scalar type then the number of bytes written does not exceed the minimum number of bytes needed to hold all bits of the type. For example, storing an `i24` writes at most three bytes. When writing a value of a type like `i20` with a size that is not an integral number of bytes, it is unspecified what happens to the extra bits that do not belong to the type, but they will typically be overwritten.

Example:

```
%ptr = alloca i32                               ; yields i32*:ptr
store i32 3, i32* %ptr                          ; yields void
%val = load i32* %ptr                           ; yields i32:val = i32 3
```

'`fence`' Instruction

Syntax:

```
fence [singlethread] <ordering>                   ; yields void
```

Overview: The '`fence`' instruction is used to introduce happens-before edges between operations.

Arguments: '`fence`' instructions take an *ordering* argument which defines what *synchronizes-with* edges they add. They can only be given `acquire`, `release`, `acq_rel`, and `seq_cst` orderings.

Semantics: A fence A which has (at least) `release` ordering semantics *synchronizes with* a fence B with (at least) `acquire` ordering semantics if and only if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M (either directly or through some side effect of a sequence headed by X), Y is sequenced before B, and Y observes M. This provides a *happens-before* dependency between A and B. Rather than an explicit `fence`, one (but not both) of the atomic operations X or Y might provide a `release` or `acquire` (resp.) ordering constraint and still *synchronize-with* the explicit `fence` and establish the *happens-before* edge.

A `fence` which has `seq_cst` ordering, in addition to having both `acquire` and `release` semantics specified above, participates in the global program order of other `seq_cst` operations and/or fences.

The optional "*singlethread*" argument specifies that the fence only synchronizes with other fences in the same thread. (This is useful for interacting with signal handlers.)

Example:

```
fence acquire                          ; yields void
fence singlethread seq_cst             ; yields void
```

'`cmpxchg`' Instruction

Syntax:

```
cmpxchg [weak] [volatile] <ty>* <pointer>, <ty> <cmp>, <ty> <new> [singlethread] <success ordering> <
```

Overview: The '`cmpxchg`' instruction is used to atomically modify memory. It loads a value in memory and compares it to a given value. If they are equal, it tries to store a new value into the memory.

Arguments: There are three arguments to the '`cmpxchg`' instruction: an address to operate on, a value to compare to the value currently be at that address, and a new value to place at that address if the compared values are equal. The type of '<cmp>' must be an integer type whose bit width is a power of two greater than or equal to eight and less than or equal to a target-specific size limit. '<cmp>' and '<new>' must have the same type, and the type of '<pointer>' must be a pointer to that type. If the `cmpxchg` is marked as `volatile`, then the optimizer is not allowed to modify the number or order of execution of this `cmpxchg` with other *volatile operations*.

The success and failure *ordering* arguments specify how this `cmpxchg` synchronizes with other atomic operations. Both ordering parameters must be at least `monotonic`, the ordering constraint on failure must be no stronger than that on success, and the failure ordering cannot be either `release` or `acq_rel`.

The optional "`singlethread`" argument declares that the `cmpxchg` is only atomic with respect to code (usually signal handlers) running in the same thread as the `cmpxchg`. Otherwise the cmpxchg is atomic with respect to all other code in the system.

The pointer passed into cmpxchg must have alignment greater than or equal to the size in memory of the operand.

Semantics: The contents of memory at the location specified by the '`<pointer>`' operand is read and compared to '`<cmp>`'; if the read value is the equal, the '`<new>`' is written. The original value at the location is returned, together with a flag indicating success (true) or failure (false).

If the cmpxchg operation is marked as `weak` then a spurious failure is permitted: the operation may not write `<new>` even if the comparison matched.

If the cmpxchg operation is strong (the default), the i1 value is 1 if and only if the value loaded equals `cmp`.

A successful `cmpxchg` is a read-modify-write instruction for the purpose of identifying release sequences. A failed `cmpxchg` is equivalent to an atomic load with an ordering parameter determined the second ordering parameter.

Example:

```
entry:
  %orig = atomic load i32* %ptr unordered                     ; yields i32
  br label %loop

loop:
```

```
  %cmp = phi i32 [ %orig, %entry ], [%old, %loop]
  %squared = mul i32 %cmp, %cmp
  %val_success = cmpxchg i32* %ptr, i32 %cmp, i32 %squared acq_rel monotonic ; yields  { i32, i1 }
  %value_loaded = extractvalue { i32, i1 } %val_success, 0
  %success = extractvalue { i32, i1 } %val_success, 1
  br i1 %success, label %done, label %loop

done:
  ...
```

'atomicrmw' Instruction

Syntax:

```
atomicrmw [volatile] <operation> <ty>* <pointer>, <ty> <value> [singlethread] <ordering>
```

Overview: The 'atomicrmw' instruction is used to atomically modify memory.

Arguments: There are three arguments to the 'atomicrmw' instruction: an operation to apply, an address whose value to modify, an argument to the operation. The operation must be one of the following keywords:

- xchg
- add
- sub
- and
- nand
- or
- xor
- max
- min
- umax
- umin

The type of '<value>' must be an integer type whose bit width is a power of two greater than or equal to eight and less than or equal to a target-specific size limit. The type of the '<pointer>' operand must be a pointer to that type. If the atomicrmw is marked as volatile, then the optimizer is not allowed to modify the number or order of execution of this atomicrmw with other *volatile operations*.

Semantics: The contents of memory at the location specified by the '<pointer>' operand are atomically read, modified, and written back. The original value at the location is returned. The modification is specified by the operation argument:

- xchg: *ptr = val
- add: *ptr = *ptr + val
- sub: *ptr = *ptr - val
- and: *ptr = *ptr & val

- nand: `*ptr = ~(*ptr & val)`
- or: `*ptr = *ptr | val`
- xor: `*ptr = *ptr ^ val`
- max: `*ptr = *ptr > val ? *ptr : val` (using a signed comparison)
- min: `*ptr = *ptr < val ? *ptr : val` (using a signed comparison)
- umax: `*ptr = *ptr > val ? *ptr : val` (using an unsigned comparison)
- umin: `*ptr = *ptr < val ? *ptr : val` (using an unsigned comparison)

Example:

```
%old = atomicrmw add i32* %ptr, i32 1 acquire                        ; yields i32
```

'`getelementptr`' Instruction

Syntax:

```
<result> = getelementptr <pty>* <ptrval>{, <ty> <idx>}*
<result> = getelementptr inbounds <pty>* <ptrval>{, <ty> <idx>}*
<result> = getelementptr <ptr vector> ptrval, <vector index type> idx
```

Overview: The '`getelementptr`' instruction is used to get the address of a subelement of an *aggregate* data structure. It performs address calculation only and does not access memory.

Arguments: The first argument is always a pointer or a vector of pointers, and forms the basis of the calculation. The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed. The interpretation of each index is dependent on the type being indexed into. The first index always indexes the pointer value given as the first argument, the second index indexes a value of the type pointed to (not necessarily the value directly pointed to, since the first index can be non-zero), etc. The first type indexed into must be a pointer value, subsequent types can be arrays, vectors, and structs. Note that subsequent types being indexed into can never be pointers, since that would require loading the pointer before continuing calculation.

The type of each index argument depends on the type it is indexing into. When indexing into a (optionally packed) structure, only `i32` integer **constants** are allowed (when using a vector of indices they must all be the **same** `i32` integer constant). When indexing into an array, pointer or vector, integers of any width are allowed, and they are not required to be constant. These integers are treated as signed values where relevant.

For example, let's consider a C code fragment and how it gets compiled to LLVM:

```
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

The LLVM code generated by Clang is:

```
%struct.RT = type { i8, [10 x [20 x i32]], i8 }
%struct.ST = type { i32, double, %struct.RT }

define i32* @foo(%struct.ST* %s) nounwind uwtable readnone optsize ssp {
entry:
  %arrayidx = getelementptr inbounds %struct.ST* %s, i64 1, i32 2, i32 1, i64 5, i64 13
  ret i32* %arrayidx
}
```

Semantics: In the example above, the first index is indexing into the '`%struct.ST*`' type, which is a pointer, yielding a '`%struct.ST`' = '`{ i32, double, %struct.RT }`' type, a structure. The second index indexes into the third element of the structure, yielding a '`%struct.RT`' = '`{ i8 , [10 x [20 x i32]], i8 }`' type, another structure. The third index indexes into the second element of the structure, yielding a '`[10 x [20 x i32]]`' type, an array. The two dimensions of the array are subscripted into, yielding an '`i32`' type. The '`getelementptr`' instruction returns a pointer to this element, thus computing a value of '`i32*`' type.

Note that it is perfectly legal to index partially through a structure, returning a pointer to an inner element. Because of this, the LLVM code for the given testcase is equivalent to:

```
define i32* @foo(%struct.ST* %s) {
  %t1 = getelementptr %struct.ST* %s, i32 1                        ; yields %struct.ST*:%t1
  %t2 = getelementptr %struct.ST* %t1, i32 0, i32 2                ; yields %struct.RT*:%t2
  %t3 = getelementptr %struct.RT* %t2, i32 0, i32 1                ; yields [10 x [20 x i32]]*:%t3
  %t4 = getelementptr [10 x [20 x i32]]* %t3, i32 0, i32 5         ; yields [20 x i32]*:%t4
  %t5 = getelementptr [20 x i32]* %t4, i32 0, i32 13               ; yields i32*:%t5
  ret i32* %t5
}
```

If the `inbounds` keyword is present, the result value of the `getelementptr` is a *poison value* if the base pointer is not an *in bounds* address of an allocated object, or if any of the addresses that would be formed by successive addition of the offsets implied by the indices to the base address with infinitely precise signed arithmetic are not an *in bounds* address of that allocated object. The *in bounds* addresses for an allocated object are all the addresses that point into the object, plus the address one byte past the end. In cases where the base is a vector of pointers the `inbounds` keyword applies to each of the computations element-wise.

If the `inbounds` keyword is not present, the offsets are added to the base address with silently-wrapping two's complement arithmetic. If the offsets have a different width from the pointer, they are sign-extended or truncated to the width of the pointer. The result value of the `getelementptr` may be outside the object pointed to by the base pointer. The result value may not necessarily be used to access memory though, even if it happens to point into allocated storage. See the *Pointer Aliasing Rules* section for more information.

The getelementptr instruction is often confusing. For some more insight into how it works, see *the getelementptr FAQ*.

Example:

```
; yields [12 x i8]*:aptr
%aptr = getelementptr {i32, [12 x i8]}* %saptr, i64 0, i32 1
; yields i8*:vptr
%vptr = getelementptr {i32, <2 x i8>}* %svptr, i64 0, i32 1, i32 1
; yields i8*:eptr
%eptr = getelementptr [12 x i8]* %aptr, i64 0, i32 1
; yields i32*:iptr
%iptr = getelementptr [10 x i32]* @arr, i16 0, i16 0
```

In cases where the pointer argument is a vector of pointers, each index must be a vector with the same number of elements. For example:

```
%A = getelementptr <4 x i8*> %ptrs, <4 x i64> %offsets,
```

Conversion Operations

The instructions in this category are the conversion instructions (casting) which all take a single operand and a type. They perform various bit conversions on the operand.

'trunc .. to' Instruction

Syntax:

```
<result> = trunc <ty> <value> to <ty2>             ; yields ty2
```

Overview: The 'trunc' instruction truncates its operand to the type ty2.

Arguments: The 'trunc' instruction takes a value to trunc, and a type to trunc it to. Both types must be of *integer* types, or vectors of the same number of integers. The bit size of the value must be larger than the bit size of the destination type, ty2. Equal sized types are not allowed.

Semantics: The 'trunc' instruction truncates the high order bits in value and converts the remaining bits to ty2. Since the source size must be larger than the destination size, trunc cannot be a *no-op cast*. It will always truncate bits.

Example:

```
%X = trunc i32 257 to i8                        ; yields i8:1
%Y = trunc i32 123 to i1                        ; yields i1:true
%Z = trunc i32 122 to i1                        ; yields i1:false
%W = trunc <2 x i16> <i16 8, i16 7> to <2 x i8> ; yields <i8 8, i8 7>
```

'zext .. to' Instruction

Syntax:

```
<result> = zext <ty> <value> to <ty2>             ; yields ty2
```

Overview: The 'zext' instruction zero extends its operand to type ty2.

Arguments: The 'zext' instruction takes a value to cast, and a type to cast it to. Both types must be of *integer* types, or vectors of the same number of integers. The bit size of the value must be smaller than the bit size of the destination type, ty2.

Semantics: The zext fills the high order bits of the value with zero bits until it reaches the size of the destination type, ty2.

When zero extending from i1, the result will always be either 0 or 1.

Example:

```
%X = zext i32 257 to i64              ; yields i64:257
%Y = zext i1 true to i32              ; yields i32:1
%Z = zext <2 x i16> <i16 8, i16 7> to <2 x i32> ; yields <i32 8, i32 7>
```

'sext .. to' Instruction

Syntax:

```
<result> = sext <ty> <value> to <ty2>             ; yields ty2
```

Overview: The 'sext' sign extends value to the type ty2.

Arguments: The 'sext' instruction takes a value to cast, and a type to cast it to. Both types must be of *integer* types, or vectors of the same number of integers. The bit size of the value must be smaller than the bit size of the destination type, ty2.

Semantics: The 'sext' instruction performs a sign extension by copying the sign bit (highest order bit) of the value until it reaches the bit size of the type ty2.

When sign extending from i1, the extension always results in -1 or 0.

Example:

```
%X = sext i8  -1 to i16              ; yields i16   :65535
%Y = sext i1 true to i32             ; yields i32:-1
%Z = sext <2 x i16> <i16 8, i16 7> to <2 x i32> ; yields <i32 8, i32 7>
```

'fptrunc .. to' Instruction

Syntax:

```
<result> = fptrunc <ty> <value> to <ty2>             ; yields ty2
```

Overview: The 'fptrunc' instruction truncates value to type ty2.

Arguments: The 'fptrunc' instruction takes a *floating point* value to cast and a *floating point* type to cast it to. The size of value must be larger than the size of ty2. This implies that fptrunc cannot be used to make a *no-op cast*.

Semantics: The 'fptrunc' instruction truncates a value from a larger *floating point* type to a smaller *floating point* type. If the value cannot fit within the destination type, ty2, then the results are undefined.

Example:

```
%X = fptrunc double 123.0 to float         ; yields float:123.0
%Y = fptrunc double 1.0E+300 to float      ; yields undefined
```

'fpext .. to' Instruction

Syntax:

```
<result> = fpext <ty> <value> to <ty2>             ; yields ty2
```

Overview: The 'fpext' extends a floating point value to a larger floating point value.

Arguments: The 'fpext' instruction takes a *floating point* value to cast, and a *floating point* type to cast it to. The source type must be smaller than the destination type.

Semantics: The 'fpext' instruction extends the value from a smaller *floating point* type to a larger *floating point* type. The fpext cannot be used to make a *no-op cast* because it always changes bits. Use bitcast to make a *no-op cast* for a floating point cast.

Example:

```
%X = fpext float 3.125 to double         ; yields double:3.125000e+00
%Y = fpext double %X to fp128            ; yields fp128:0xL00000000000000004000900000000000
```

'fptoui .. to' Instruction

Syntax:

```
<result> = fptoui <ty> <value> to <ty2>             ; yields ty2
```

Overview: The 'fptoui' converts a floating point value to its unsigned integer equivalent of type ty2.

Arguments: The 'fptoui' instruction takes a value to cast, which must be a scalar or vector *floating point* value, and a type to cast it to ty2, which must be an *integer* type. If ty is a vector floating point type, ty2 must be a vector integer type with the same number of elements as ty

Semantics: The 'fptoui' instruction converts its *floating point* operand into the nearest (rounding towards zero) unsigned integer value. If the value cannot fit in ty2, the results are undefined.

Example:

```
%X = fptoui double 123.0 to i32      ; yields i32:123
%Y = fptoui float 1.0E+300 to i1     ; yields undefined:1
%Z = fptoui float 1.04E+17 to i8     ; yields undefined:1
```

'fptosi .. to' Instruction

Syntax:

```
<result> = fptosi <ty> <value> to <ty2>             ; yields ty2
```

Overview: The 'fptosi' instruction converts *floating point* value to type ty2.

Arguments: The '`fptosi`' instruction takes a value to cast, which must be a scalar or vector *floating point* value, and a type to cast it to `ty2`, which must be an *integer* type. If `ty` is a vector floating point type, `ty2` must be a vector integer type with the same number of elements as `ty`

Semantics: The '`fptosi`' instruction converts its *floating point* operand into the nearest (rounding towards zero) signed integer value. If the value cannot fit in `ty2`, the results are undefined.

Example:

```
%X = fptosi double -123.0 to i32      ; yields i32:-123
%Y = fptosi float 1.0E-247 to i1      ; yields undefined:1
%Z = fptosi float 1.04E+17 to i8      ; yields undefined:1
```

'`uitofp .. to`' Instruction

Syntax:

```
<result> = uitofp <ty> <value> to <ty2>             ; yields ty2
```

Overview: The '`uitofp`' instruction regards `value` as an unsigned integer and converts that value to the `ty2` type.

Arguments: The '`uitofp`' instruction takes a value to cast, which must be a scalar or vector *integer* value, and a type to cast it to `ty2`, which must be an *floating point* type. If `ty` is a vector integer type, `ty2` must be a vector floating point type with the same number of elements as `ty`

Semantics: The '`uitofp`' instruction interprets its operand as an unsigned integer quantity and converts it to the corresponding floating point value. If the value cannot fit in the floating point value, the results are undefined.

Example:

```
%X = uitofp i32 257 to float         ; yields float:257.0
%Y = uitofp i8 -1 to double          ; yields double:255.0
```

'`sitofp .. to`' Instruction

Syntax:

```
<result> = sitofp <ty> <value> to <ty2>             ; yields ty2
```

Overview: The '`sitofp`' instruction regards `value` as a signed integer and converts that value to the `ty2` type.

Arguments: The '`sitofp`' instruction takes a value to cast, which must be a scalar or vector *integer* value, and a type to cast it to `ty2`, which must be an *floating point* type. If `ty` is a vector integer type, `ty2` must be a vector floating point type with the same number of elements as `ty`

Semantics: The '`sitofp`' instruction interprets its operand as a signed integer quantity and converts it to the corresponding floating point value. If the value cannot fit in the floating point value, the results are undefined.

Example:

```
%X = sitofp i32 257 to float         ; yields float:257.0
%Y = sitofp i8 -1 to double          ; yields double:-1.0
```

'ptrtoint .. to' Instruction

Syntax:

```
<result> = ptrtoint <ty> <value> to <ty2>             ; yields ty2
```

Overview: The 'ptrtoint' instruction converts the pointer or a vector of pointers value to the integer (or vector of integers) type ty2.

Arguments: The 'ptrtoint' instruction takes a value to cast, which must be a a value of type *pointer* or a vector of pointers, and a type to cast it to ty2, which must be an *integer* or a vector of integers type.

Semantics: The 'ptrtoint' instruction converts value to integer type ty2 by interpreting the pointer value as an integer and either truncating or zero extending that value to the size of the integer type. If value is smaller than ty2 then a zero extension is done. If value is larger than ty2 then a truncation is done. If they are the same size, then nothing is done (*no-op cast*) other than a type change.

Example:

```
%X = ptrtoint i32* %P to i8                         ; yields truncation on 32-bit architecture
%Y = ptrtoint i32* %P to i64                        ; yields zero extension on 32-bit architecture
%Z = ptrtoint <4 x i32*> %P to <4 x i64>; yields vector zero extension for a vector of addresses on
```

'inttoptr .. to' Instruction

Syntax:

```
<result> = inttoptr <ty> <value> to <ty2>             ; yields ty2
```

Overview: The 'inttoptr' instruction converts an integer value to a pointer type, ty2.

Arguments: The 'inttoptr' instruction takes an *integer* value to cast, and a type to cast it to, which must be a *pointer* type.

Semantics: The 'inttoptr' instruction converts value to type ty2 by applying either a zero extension or a truncation depending on the size of the integer value. If value is larger than the size of a pointer then a truncation is done. If value is smaller than the size of a pointer then a zero extension is done. If they are the same size, nothing is done (*no-op cast*).

Example:

```
%X = inttoptr i32 255 to i32*          ; yields zero extension on 64-bit architecture
%Y = inttoptr i32 255 to i32*          ; yields no-op on 32-bit architecture
%Z = inttoptr i64 0 to i32*            ; yields truncation on 32-bit architecture
%Z = inttoptr <4 x i32> %G to <4 x i8*>; yields truncation of vector G to four pointers
```

'bitcast .. to' Instruction

Syntax:

```
<result> = bitcast <ty> <value> to <ty2>             ; yields ty2
```

Overview: The 'bitcast' instruction converts value to type ty2 without changing any bits.

Arguments: The 'bitcast' instruction takes a value to cast, which must be a non-aggregate first class value, and a type to cast it to, which must also be a non-aggregate *first class* type. The bit sizes of value and the destination type, ty2, must be identical. If the source type is a pointer, the destination type must also be a pointer of the same size. This instruction supports bitwise conversion of vectors to integers and to vectors of other types (as long as they have the same size).

Semantics: The 'bitcast' instruction converts value to type ty2. It is always a *no-op cast* because no bits change with this conversion. The conversion is done as if the value had been stored to memory and read back as type ty2. Pointer (or vector of pointers) types may only be converted to other pointer (or vector of pointers) types with the same address space through this instruction. To convert pointers to other types, use the *inttoptr* or *ptrtoint* instructions first.

Example:

```
%X = bitcast i8 255 to i8              ; yields i8 :-1
%Y = bitcast i32* %x to sint*          ; yields sint*:%x
%Z = bitcast <2 x int> %V to i64;        ; yields i64: %V
%Z = bitcast <2 x i32*> %V to <2 x i64*> ; yields <2 x i64*>
```

'addrspacecast .. to' Instruction

Syntax:

```
<result> = addrspacecast <pty> <ptrval> to <pty2>       ; yields pty2
```

Overview: The 'addrspacecast' instruction converts ptrval from pty in address space n to type pty2 in address space m.

Arguments: The 'addrspacecast' instruction takes a pointer or vector of pointer value to cast and a pointer type to cast it to, which must have a different address space.

Semantics: The 'addrspacecast' instruction converts the pointer value ptrval to type pty2. It can be a *no-op cast* or a complex value modification, depending on the target and the address space pair. Pointer conversions within the same address space must be performed with the bitcast instruction. Note that if the address space conversion is legal then both result and operand refer to the same memory location.

Example:

```
%X = addrspacecast i32* %x to i32 addrspace(1)*    ; yields i32 addrspace(1)*:%x
%Y = addrspacecast i32 addrspace(1)* %y to i64 addrspace(2)*    ; yields i64 addrspace(2)*:%y
%Z = addrspacecast <4 x i32*> %z to <4 x float addrspace(3)*>   ; yields <4 x float addrspace(3)*>:%z
```

Other Operations

The instructions in this category are the "miscellaneous" instructions, which defy better classification.

'icmp' Instruction

Syntax:

```
<result> = icmp <cond> <ty> <op1>, <op2>   ; yields i1 or <N x i1>:result
```

Overview: The 'icmp' instruction returns a boolean value or a vector of boolean values based on comparison of its two integer, integer vector, pointer, or pointer vector operands.

Arguments: The 'icmp' instruction takes three operands. The first operand is the condition code indicating the kind of comparison to perform. It is not a value, just a keyword. The possible condition code are:

1. eq: equal
2. ne: not equal
3. ugt: unsigned greater than
4. uge: unsigned greater or equal
5. ult: unsigned less than
6. ule: unsigned less or equal
7. sgt: signed greater than
8. sge: signed greater or equal
9. slt: signed less than
10. sle: signed less or equal

The remaining two arguments must be *integer* or *pointer* or integer *vector* typed. They must also be identical types.

Semantics: The 'icmp' compares op1 and op2 according to the condition code given as cond. The comparison performed always yields either an *i1* or vector of i1 result, as follows:

1. eq: yields true if the operands are equal, false otherwise. No sign interpretation is necessary or performed.
2. ne: yields true if the operands are unequal, false otherwise. No sign interpretation is necessary or performed.
3. ugt: interprets the operands as unsigned values and yields true if op1 is greater than op2.
4. uge: interprets the operands as unsigned values and yields true if op1 is greater than or equal to op2.
5. ult: interprets the operands as unsigned values and yields true if op1 is less than op2.
6. ule: interprets the operands as unsigned values and yields true if op1 is less than or equal to op2.
7. sgt: interprets the operands as signed values and yields true if op1 is greater than op2.
8. sge: interprets the operands as signed values and yields true if op1 is greater than or equal to op2.
9. slt: interprets the operands as signed values and yields true if op1 is less than op2.
10. sle: interprets the operands as signed values and yields true if op1 is less than or equal to op2.

If the operands are *pointer* typed, the pointer values are compared as if they were integers.

If the operands are integer vectors, then they are compared element by element. The result is an `i1` vector with the same number of elements as the values being compared. Otherwise, the result is an `i1`.

Example:

```
<result> = icmp eq i32 4, 5          ; yields: result=false
<result> = icmp ne float* %X, %X     ; yields: result=false
<result> = icmp ult i16  4, 5        ; yields: result=true
<result> = icmp sgt i16  4, 5        ; yields: result=false
<result> = icmp ule i16 -4, 5        ; yields: result=false
<result> = icmp sge i16  4, 5        ; yields: result=false
```

Note that the code generator does not yet support vector types with the `icmp` instruction.

'`fcmp`' Instruction

Syntax:

```
<result> = fcmp <cond> <ty> <op1>, <op2>     ; yields i1 or <N x i1>:result
```

Overview: The '`fcmp`' instruction returns a boolean value or vector of boolean values based on comparison of its operands.

If the operands are floating point scalars, then the result type is a boolean (*i1*).

If the operands are floating point vectors, then the result type is a vector of boolean with the same number of elements as the operands being compared.

Arguments: The '`fcmp`' instruction takes three operands. The first operand is the condition code indicating the kind of comparison to perform. It is not a value, just a keyword. The possible condition code are:

1. `false`: no comparison, always returns false
2. `oeq`: ordered and equal
3. `ogt`: ordered and greater than
4. `oge`: ordered and greater than or equal
5. `olt`: ordered and less than
6. `ole`: ordered and less than or equal
7. `one`: ordered and not equal
8. `ord`: ordered (no nans)
9. `ueq`: unordered or equal
10. `ugt`: unordered or greater than
11. `uge`: unordered or greater than or equal
12. `ult`: unordered or less than
13. `ule`: unordered or less than or equal
14. `une`: unordered or not equal
15. `uno`: unordered (either nans)

16. `true`: no comparison, always returns true

Ordered means that neither operand is a QNAN while *unordered* means that either operand may be a QNAN.

Each of `val1` and `val2` arguments must be either a *floating point* type or a *vector* of floating point type. They must have identical types.

Semantics: The '`fcmp`' instruction compares `op1` and `op2` according to the condition code given as `cond`. If the operands are vectors, then the vectors are compared element by element. Each comparison performed always yields an *i1* result, as follows:

1. `false`: always yields `false`, regardless of operands.
2. `oeq`: yields `true` if both operands are not a QNAN and `op1` is equal to `op2`.
3. `ogt`: yields `true` if both operands are not a QNAN and `op1` is greater than `op2`.
4. `oge`: yields `true` if both operands are not a QNAN and `op1` is greater than or equal to `op2`.
5. `olt`: yields `true` if both operands are not a QNAN and `op1` is less than `op2`.
6. `ole`: yields `true` if both operands are not a QNAN and `op1` is less than or equal to `op2`.
7. `one`: yields `true` if both operands are not a QNAN and `op1` is not equal to `op2`.
8. `ord`: yields `true` if both operands are not a QNAN.
9. `ueq`: yields `true` if either operand is a QNAN or `op1` is equal to `op2`.
10. `ugt`: yields `true` if either operand is a QNAN or `op1` is greater than `op2`.
11. `uge`: yields `true` if either operand is a QNAN or `op1` is greater than or equal to `op2`.
12. `ult`: yields `true` if either operand is a QNAN or `op1` is less than `op2`.
13. `ule`: yields `true` if either operand is a QNAN or `op1` is less than or equal to `op2`.
14. `une`: yields `true` if either operand is a QNAN or `op1` is not equal to `op2`.
15. `uno`: yields `true` if either operand is a QNAN.
16. `true`: always yields `true`, regardless of operands.

Example:

```
<result> = fcmp oeq float 4.0, 5.0    ; yields: result=false
<result> = fcmp one float 4.0, 5.0    ; yields: result=true
<result> = fcmp olt float 4.0, 5.0    ; yields: result=true
<result> = fcmp ueq double 1.0, 2.0   ; yields: result=false
```

Note that the code generator does not yet support vector types with the `fcmp` instruction.

'`phi`' Instruction

Syntax:

```
<result> = phi <ty> [ <val0>, <label0>], ...
```

Overview: The '`phi`' instruction is used to implement the ϕ node in the SSA graph representing the function.

Arguments: The type of the incoming values is specified with the first type field. After this, the '`phi`' instruction takes a list of pairs as arguments, with one pair for each predecessor basic block of the current block. Only values of *first class* type may be used as the value arguments to the PHI node. Only labels may be used as the label arguments.

There must be no non-phi instructions between the start of a basic block and the PHI instructions: i.e. PHI instructions must be first in a basic block.

For the purposes of the SSA form, the use of each incoming value is deemed to occur on the edge from the corresponding predecessor block to the current block (but after any definition of an '`invoke`' instruction's return value on the same edge).

Semantics: At runtime, the '`phi`' instruction logically takes on the value specified by the pair corresponding to the predecessor basic block that executed just prior to the current block.

Example:

```
Loop:       ; Infinite loop that counts from 0 on up...
  %indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
  %nextindvar = add i32 %indvar, 1
  br label %Loop
```

'`select`' Instruction

Syntax:

```
<result> = select selty <cond>, <ty> <val1>, <ty> <val2>             ; yields ty

selty is either i1 or {<N x i1>}
```

Overview: The '`select`' instruction is used to choose one value based on a condition, without IR-level branching.

Arguments: The '`select`' instruction requires an 'i1' value or a vector of 'i1' values indicating the condition, and two values of the same *first class* type. If the val1/val2 are vectors and the condition is a scalar, then entire vectors are selected, not individual elements.

Semantics: If the condition is an i1 and it evaluates to 1, the instruction returns the first value argument; otherwise, it returns the second value argument.

If the condition is a vector of i1, then the value arguments must be vectors of the same size, and the selection is done element by element.

Example:

```
%X = select i1 true, i8 17, i8 42          ; yields i8:17
```

'`call`' Instruction

Syntax:

```
<result> = [tail | musttail] call [cconv] [ret attrs] <ty> [<fnty>*] <fnptrval>(<function args>) [fn
```

Overview: The '`call`' instruction represents a simple function call.

Arguments: This instruction requires several arguments:

1. The optional `tail` and `musttail` markers indicate that the optimizers should perform tail call optimization. The `tail` marker is a hint that can be ignored. The `musttail` marker means that the call must be tail call optimized in order for the program to be correct. The `musttail` marker provides these guarantees:

   (a) The call will not cause unbounded stack growth if it is part of a recursive cycle in the call graph.

   (b) Arguments with the *inalloca* attribute are forwarded in place.

   Both markers imply that the callee does not access allocas or varargs from the caller. Calls marked `musttail` must obey the following additional rules:

   - The call must immediately precede a *ret* instruction, or a pointer bitcast followed by a ret instruction.
   - The ret instruction must return the (possibly bitcasted) value produced by the call or void.
   - The caller and callee prototypes must match. Pointer types of parameters or return types may differ in pointee type, but not in address space.
   - The calling conventions of the caller and callee must match.
   - All ABI-impacting function attributes, such as sret, byval, inreg, returned, and inalloca, must match.
   - The callee must be varargs iff the caller is varargs. Bitcasting a non-varargs function to the appropriate varargs type is legal so long as the non-varargs prefixes obey the other rules.

   Tail call optimization for calls marked `tail` is guaranteed to occur if the following conditions are met:

   - Caller and callee both have the calling convention `fastcc`.
   - The call is in tail position (ret immediately follows call and ret uses value of call or is void).
   - Option `-tailcallopt` is enabled, or `llvm::GuaranteedTailCallOpt` is `true`.
   - Platform-specific constraints are met.

2. The optional "cconv" marker indicates which *calling convention* the call should use. If none is specified, the call defaults to using C calling conventions. The calling convention of the call must match the calling convention of the target function, or else the behavior is undefined.

3. The optional *Parameter Attributes* list for return values. Only '`zeroext`', '`signext`', and '`inreg`' attributes are valid here.

4. '`ty`': the type of the call instruction itself which is also the type of the return value. Functions that return no value are marked `void`.

5. '`fnty`': shall be the signature of the pointer to function value being invoked. The argument types must match the types implied by this signature. This type can be omitted if the function is not varargs and if the function type does not return a pointer to a function.

6. '`fnptrval`': An LLVM value containing a pointer to a function to be invoked. In most cases, this is a direct function invocation, but indirect `call`'s are just as possible, calling an arbitrary pointer to function value.

7. '`function args`': argument list whose types match the function signature argument types and parameter attributes. All arguments must be of *first class* type. If the function signature indicates the function accepts a variable number of arguments, the extra arguments can be specified.

8. The optional *function attributes* list. Only '`noreturn`', '`nounwind`', '`readonly`' and '`readnone`' attributes are valid here.

Semantics: The '`call`' instruction is used to cause control flow to transfer to a specified function, with its incoming arguments bound to the specified values. Upon a '`ret`' instruction in the called function, control flow continues with the instruction after the function call, and the return value of the function is bound to the result argument.

Example:

```
%retval = call i32 @test(i32 %argc)
call i32 (i8*, ...)* @printf(i8* %msg, i32 12, i8 42)        ; yields i32
%X = tail call i32 @foo()                                    ; yields i32
%Y = tail call fastcc i32 @foo()  ; yields i32
call void %foo(i8 97 signext)

%struct.A = type { i32, i8 }
%r = call %struct.A @foo()                        ; yields { i32, i8 }
%gr = extractvalue %struct.A %r, 0                ; yields i32
%gr1 = extractvalue %struct.A %r, 1               ; yields i8
%Z = call void @foo() noreturn                    ; indicates that %foo never returns normally
%ZZ = call zeroext i32 @bar()                     ; Return value is %zero extended
```

llvm treats calls to some functions with names and arguments that match the standard C99 library as being the C99 library functions, and may perform optimizations or generate code for them under that assumption. This is something we'd like to change in the future to provide better support for freestanding environments and non-C-based languages.

'`va_arg`' Instruction

Syntax:

```
<resultval> = va_arg <va_list*> <arglist>, <argty>
```

Overview: The '`va_arg`' instruction is used to access arguments passed through the "variable argument" area of a function call. It is used to implement the `va_arg` macro in C.

Arguments: This instruction takes a `va_list*` value and the type of the argument. It returns a value of the specified argument type and increments the `va_list` to point to the next argument. The actual type of `va_list` is target specific.

Semantics: The '`va_arg`' instruction loads an argument of the specified type from the specified `va_list` and causes the `va_list` to point to the next argument. For more information, see the variable argument handling *Intrinsic Functions*.

It is legal for this instruction to be called in a function which does not take a variable number of arguments, for example, the `vfprintf` function.

`va_arg` is an LLVM instruction instead of an *intrinsic function* because it takes a type as an argument.

Example: See the *variable argument processing* section.

Note that the code generator does not yet fully support va_arg on many targets. Also, it does not currently support va_arg with aggregate types on any target.

'`landingpad`' Instruction

Syntax:

```
<resultval> = landingpad <resultty> personality <type> <pers_fn> <clause>+
<resultval> = landingpad <resultty> personality <type> <pers_fn> cleanup <clause>*

<clause> := catch <type> <value>
<clause> := filter <array constant type> <array constant>
```

Overview: The '`landingpad`' instruction is used by LLVM's exception handling system to specify that a basic block is a landing pad — one where the exception lands, and corresponds to the code found in the `catch` portion of a `try`/`catch` sequence. It defines values supplied by the personality function (`pers_fn`) upon re-entry to the function. The `resultval` has the type `resultty`.

Arguments: This instruction takes a `pers_fn` value. This is the personality function associated with the unwinding mechanism. The optional `cleanup` flag indicates that the landing pad block is a cleanup.

A `clause` begins with the clause type — `catch` or `filter` — and contains the global variable representing the "type" that may be caught or filtered respectively. Unlike the `catch` clause, the `filter` clause takes an array constant as its argument. Use "`[0 x i8**] undef`" for a filter which cannot throw. The '`landingpad`' instruction must contain *at least* one `clause` or the `cleanup` flag.

Semantics: The '`landingpad`' instruction defines the values which are set by the personality function (`pers_fn`) upon re-entry to the function, and therefore the "result type" of the `landingpad` instruction. As with calling conventions, how the personality function results are represented in LLVM IR is target specific.

The clauses are applied in order from top to bottom. If two `landingpad` instructions are merged together through inlining, the clauses from the calling function are appended to the list of clauses. When the call stack is being unwound due to an exception being thrown, the exception is compared against each `clause` in turn. If it doesn't match any of the clauses, and the `cleanup` flag is not set, then unwinding continues further up the call stack.

The `landingpad` instruction has several restrictions:

- A landing pad block is a basic block which is the unwind destination of an '`invoke`' instruction.
- A landing pad block must have a '`landingpad`' instruction as its first non-PHI instruction.
- There can be only one '`landingpad`' instruction within the landing pad block.
- A basic block that is not a landing pad block may not include a '`landingpad`' instruction.
- All '`landingpad`' instructions in a function must have the same personality function.

Example:

```
;; A landing pad which can catch an integer.
%res = landingpad { i8*, i32 } personality i32 (...)* @__gxx_personality_v0
         catch i8** @_ZTIi
;; A landing pad that is a cleanup.
%res = landingpad { i8*, i32 } personality i32 (...)* @__gxx_personality_v0
         cleanup
;; A landing pad which can catch an integer and can only throw a double.
%res = landingpad { i8*, i32 } personality i32 (...)* @__gxx_personality_v0
         catch i8** @_ZTIi
         filter [1 x i8**] [@_ZTId]
```

1.1.11 Intrinsic Functions

LLVM supports the notion of an "intrinsic function". These functions have well known names and semantics and are required to follow certain restrictions. Overall, these intrinsics represent an extension mechanism for the LLVM language that does not require changing all of the transformations in LLVM when adding to the language (or the bitcode reader/writer, the parser, etc...).

Intrinsic function names must all start with an "`llvm.`" prefix. This prefix is reserved in LLVM for intrinsic names; thus, function names may not begin with this prefix. Intrinsic functions must always be external functions: you cannot define the body of intrinsic functions. Intrinsic functions may only be used in call or invoke instructions: it is illegal

to take the address of an intrinsic function. Additionally, because intrinsic functions are part of the LLVM language, it is required if any are added that they be documented here.

Some intrinsic functions can be overloaded, i.e., the intrinsic represents a family of functions that perform the same operation but on different data types. Because LLVM can represent over 8 million different integer types, overloading is used commonly to allow an intrinsic function to operate on any integer type. One or more of the argument types or the result type can be overloaded to accept any integer type. Argument types may also be defined as exactly matching a previous argument's type or the result type. This allows an intrinsic function which accepts multiple arguments, but needs all of them to be of the same type, to only be overloaded with respect to a single argument or the result.

Overloaded intrinsics will have the names of its overloaded argument types encoded into its function name, each preceded by a period. Only those types which are overloaded result in a name suffix. Arguments whose type is matched against another type do not. For example, the `llvm.ctpop` function can take an integer of any width and returns an integer of exactly the same integer width. This leads to a family of functions such as `i8 @llvm.ctpop.i8(i8 %val)` and `i29 @llvm.ctpop.i29(i29 %val)`. Only one type, the return type, is overloaded, and only one type suffix is required. Because the argument's type is matched against the return type, it does not require its own name suffix.

To learn how to add an intrinsic function, please see the Extending LLVM Guide.

Variable Argument Handling Intrinsics

Variable argument support is defined in LLVM with the *va_arg* instruction and these three intrinsic functions. These functions are related to the similarly named macros defined in the `<stdarg.h>` header file.

All of these functions operate on arguments that use a target-specific value type "`va_list`". The LLVM assembly language reference manual does not define what this type is, so all transformations should be prepared to handle these functions regardless of the type used.

This example shows how the *va_arg* instruction and the variable argument handling intrinsic functions are used.

```
define i32 @test(i32 %X, ...) {
  ; Initialize variable argument processing
  %ap = alloca i8*
  %ap2 = bitcast i8** %ap to i8*
  call void @llvm.va_start(i8* %ap2)

  ; Read a single integer argument
  %tmp = va_arg i8** %ap, i32

  ; Demonstrate usage of llvm.va_copy and llvm.va_end
  %aq = alloca i8*
  %aq2 = bitcast i8** %aq to i8*
  call void @llvm.va_copy(i8* %aq2, i8* %ap2)
  call void @llvm.va_end(i8* %aq2)

  ; Stop processing of arguments.
  call void @llvm.va_end(i8* %ap2)
  ret i32 %tmp
}

declare void @llvm.va_start(i8*)
declare void @llvm.va_copy(i8*, i8*)
declare void @llvm.va_end(i8*)
```

'`llvm.va_start`' Intrinsic

Syntax:

```
declare void @llvm.va_start(i8* <arglist>)
```

Overview: The '`llvm.va_start`' intrinsic initializes `*<arglist>` for subsequent use by `va_arg`.

Arguments: The argument is a pointer to a `va_list` element to initialize.

Semantics: The '`llvm.va_start`' intrinsic works just like the `va_start` macro available in C. In a target-dependent way, it initializes the `va_list` element to which the argument points, so that the next call to `va_arg` will produce the first variable argument passed to the function. Unlike the C `va_start` macro, this intrinsic does not need to know the last argument of the function as the compiler can figure that out.

'`llvm.va_end`' Intrinsic

Syntax:

```
declare void @llvm.va_end(i8* <arglist>)
```

Overview: The '`llvm.va_end`' intrinsic destroys `*<arglist>`, which has been initialized previously with `llvm.va_start` or `llvm.va_copy`.

Arguments: The argument is a pointer to a `va_list` to destroy.

Semantics: The '`llvm.va_end`' intrinsic works just like the `va_end` macro available in C. In a target-dependent way, it destroys the `va_list` element to which the argument points. Calls to *llvm.va_start* and *llvm.va_copy* must be matched exactly with calls to `llvm.va_end`.

'`llvm.va_copy`' Intrinsic

Syntax:

```
declare void @llvm.va_copy(i8* <destarglist>, i8* <srcarglist>)
```

Overview: The '`llvm.va_copy`' intrinsic copies the current argument position from the source argument list to the destination argument list.

Arguments: The first argument is a pointer to a `va_list` element to initialize. The second argument is a pointer to a `va_list` element to copy from.

Semantics: The '`llvm.va_copy`' intrinsic works just like the `va_copy` macro available in C. In a target-dependent way, it copies the source `va_list` element into the destination `va_list` element. This intrinsic is necessary because the '' llvm.va_start'' intrinsic may be arbitrarily complex and require, for example, memory allocation.

Accurate Garbage Collection Intrinsics

LLVM support for Accurate Garbage Collection (GC) requires the implementation and generation of these intrinsics. These intrinsics allow identification of *GC roots on the stack*, as well as garbage collector implementations that require *read* and *write* barriers. Front-ends for type-safe garbage collected languages should generate these intrinsics to make use of the LLVM garbage collectors. For more details, see Accurate Garbage Collection with LLVM.

The garbage collection intrinsics only operate on objects in the generic address space (address space zero).

'`llvm.gcroot`' Intrinsic

Syntax:

```
declare void @llvm.gcroot(i8** %ptrloc, i8* %metadata)
```

Overview: The '`llvm.gcroot`' intrinsic declares the existence of a GC root to the code generator, and allows some metadata to be associated with it.

Arguments: The first argument specifies the address of a stack object that contains the root pointer. The second pointer (which must be either a constant or a global value address) contains the meta-data to be associated with the root.

Semantics: At runtime, a call to this intrinsic stores a null pointer into the "ptrloc" location. At compile-time, the code generator generates information to allow the runtime to find the pointer at GC safe points. The '`llvm.gcroot`' intrinsic may only be used in a function which *specifies a GC algorithm*.

'`llvm.gcread`' Intrinsic

Syntax:

```
declare i8* @llvm.gcread(i8* %ObjPtr, i8** %Ptr)
```

Overview: The '`llvm.gcread`' intrinsic identifies reads of references from heap locations, allowing garbage collector implementations that require read barriers.

Arguments: The second argument is the address to read from, which should be an address allocated from the garbage collector. The first object is a pointer to the start of the referenced object, if needed by the language runtime (otherwise null).

Semantics: The '`llvm.gcread`' intrinsic has the same semantics as a load instruction, but may be replaced with substantially more complex code by the garbage collector runtime, as needed. The '`llvm.gcread`' intrinsic may only be used in a function which *specifies a GC algorithm*.

'`llvm.gcwrite`' Intrinsic

Syntax:

```
declare void @llvm.gcwrite(i8* %P1, i8* %Obj, i8** %P2)
```

Overview: The '`llvm.gcwrite`' intrinsic identifies writes of references to heap locations, allowing garbage collector implementations that require write barriers (such as generational or reference counting collectors).

Arguments: The first argument is the reference to store, the second is the start of the object to store it to, and the third is the address of the field of Obj to store to. If the runtime does not require a pointer to the object, Obj may be null.

Semantics: The '`llvm.gcwrite`' intrinsic has the same semantics as a store instruction, but may be replaced with substantially more complex code by the garbage collector runtime, as needed. The '`llvm.gcwrite`' intrinsic may only be used in a function which *specifies a GC algorithm*.

Code Generator Intrinsics

These intrinsics are provided by LLVM to expose special features that may only be implemented with code generator support.

'`llvm.returnaddress`' Intrinsic

Syntax:

```
declare i8  *@llvm.returnaddress(i32 <level>)
```

Overview: The '`llvm.returnaddress`' intrinsic attempts to compute a target-specific value indicating the return address of the current function or one of its callers.

Arguments: The argument to this intrinsic indicates which function to return the address for. Zero indicates the calling function, one indicates its caller, etc. The argument is **required** to be a constant integer value.

Semantics: The '`llvm.returnaddress`' intrinsic either returns a pointer indicating the return address of the specified call frame, or zero if it cannot be identified. The value returned by this intrinsic is likely to be incorrect or 0 for arguments other than zero, so it should only be used for debugging purposes.

Note that calling this intrinsic does not prevent function inlining or other aggressive transformations, so the value returned may not be that of the obvious source-language caller.

'`llvm.frameaddress`' Intrinsic

Syntax:

```
declare i8* @llvm.frameaddress(i32 <level>)
```

Overview: The '`llvm.frameaddress`' intrinsic attempts to return the target-specific frame pointer value for the specified stack frame.

Arguments: The argument to this intrinsic indicates which function to return the frame pointer for. Zero indicates the calling function, one indicates its caller, etc. The argument is **required** to be a constant integer value.

Semantics: The 'llvm.frameaddress' intrinsic either returns a pointer indicating the frame address of the specified call frame, or zero if it cannot be identified. The value returned by this intrinsic is likely to be incorrect or 0 for arguments other than zero, so it should only be used for debugging purposes.

Note that calling this intrinsic does not prevent function inlining or other aggressive transformations, so the value returned may not be that of the obvious source-language caller.

'llvm.read_register' and 'llvm.write_register' Intrinsics

Syntax:

```
declare i32 @llvm.read_register.i32(metadata)
declare i64 @llvm.read_register.i64(metadata)
declare void @llvm.write_register.i32(metadata, i32 @value)
declare void @llvm.write_register.i64(metadata, i64 @value)
!0 = metadata !{metadata !"sp\00"}
```

Overview: The 'llvm.read_register' and 'llvm.write_register' intrinsics provides access to the named register. The register must be valid on the architecture being compiled to. The type needs to be compatible with the register being read.

Semantics: The 'llvm.read_register' intrinsic returns the current value of the register, where possible. The 'llvm.write_register' intrinsic sets the current value of the register, where possible.

This is useful to implement named register global variables that need to always be mapped to a specific register, as is common practice on bare-metal programs including OS kernels.

The compiler doesn't check for register availability or use of the used register in surrounding code, including inline assembly. Because of that, allocatable registers are not supported.

Warning: So far it only works with the stack pointer on selected architectures (ARM, AArch64, PowerPC and x86_64). Significant amount of work is needed to support other registers and even more so, allocatable registers.

'llvm.stacksave' Intrinsic

Syntax:

```
declare i8* @llvm.stacksave()
```

Overview: The 'llvm.stacksave' intrinsic is used to remember the current state of the function stack, for use with *llvm.stackrestore*. This is useful for implementing language features like scoped automatic variable sized arrays in C99.

Semantics: This intrinsic returns a opaque pointer value that can be passed to *llvm.stackrestore*. When an llvm.stackrestore intrinsic is executed with a value saved from llvm.stacksave, it effectively restores the state of the stack to the state it was in when the llvm.stacksave intrinsic executed. In practice, this pops any *alloca* blocks from the stack that were allocated after the llvm.stacksave was executed.

'llvm.stackrestore' Intrinsic

Syntax:

```
declare void @llvm.stackrestore(i8* %ptr)
```

Overview: The '`llvm.stackrestore`' intrinsic is used to restore the state of the function stack to the state it was in when the corresponding *llvm.stacksave* intrinsic executed. This is useful for implementing language features like scoped automatic variable sized arrays in C99.

Semantics: See the description for *llvm.stacksave*.

'`llvm.prefetch`' Intrinsic

Syntax:

```
declare void @llvm.prefetch(i8* <address>, i32 <rw>, i32 <locality>, i32 <cache type>)
```

Overview: The '`llvm.prefetch`' intrinsic is a hint to the code generator to insert a prefetch instruction if supported; otherwise, it is a noop. Prefetches have no effect on the behavior of the program but can change its performance characteristics.

Arguments: `address` is the address to be prefetched, `rw` is the specifier determining if the fetch should be for a read (0) or write (1), and `locality` is a temporal locality specifier ranging from (0) - no locality, to (3) - extremely local keep in cache. The `cache type` specifies whether the prefetch is performed on the data (1) or instruction (0) cache. The `rw`, `locality` and `cache type` arguments must be constant integers.

Semantics: This intrinsic does not modify the behavior of the program. In particular, prefetches cannot trap and do not produce a value. On targets that support this intrinsic, the prefetch can provide hints to the processor cache for better performance.

'`llvm.pcmarker`' Intrinsic

Syntax:

```
declare void @llvm.pcmarker(i32 <id>)
```

Overview: The '`llvm.pcmarker`' intrinsic is a method to export a Program Counter (PC) in a region of code to simulators and other tools. The method is target specific, but it is expected that the marker will use exported symbols to transmit the PC of the marker. The marker makes no guarantees that it will remain with any specific instruction after optimizations. It is possible that the presence of a marker will inhibit optimizations. The intended use is to be inserted after optimizations to allow correlations of simulation runs.

Arguments: `id` is a numerical id identifying the marker.

Semantics: This intrinsic does not modify the behavior of the program. Backends that do not support this intrinsic may ignore it.

'llvm.readcyclecounter' Intrinsic

Syntax:

```
declare i64 @llvm.readcyclecounter()
```

Overview: The 'llvm.readcyclecounter' intrinsic provides access to the cycle counter register (or similar low latency, high accuracy clocks) on those targets that support it. On X86, it should map to RDTSC. On Alpha, it should map to RPCC. As the backing counters overflow quickly (on the order of 9 seconds on alpha), this should only be used for small timings.

Semantics: When directly supported, reading the cycle counter should not modify any memory. Implementations are allowed to either return a application specific value or a system wide value. On backends without support, this is lowered to a constant 0.

Note that runtime support may be conditional on the privilege-level code is running at and the host platform.

'llvm.clear_cache' Intrinsic

Syntax:

```
declare void @llvm.clear_cache(i8*, i8*)
```

Overview: The 'llvm.clear_cache' intrinsic ensures visibility of modifications in the specified range to the execution unit of the processor. On targets with non-unified instruction and data cache, the implementation flushes the instruction cache.

Semantics: On platforms with coherent instruction and data caches (e.g. x86), this intrinsic is a nop. On platforms with non-coherent instruction and data cache (e.g. ARM, MIPS), the intrinsic is lowered either to appropriate instructions or a system call, if cache flushing requires special privileges.

The default behavior is to emit a call to __clear_cache from the run time library.

This instrinsic does *not* empty the instruction pipeline. Modifications of the current function are outside the scope of the intrinsic.

Standard C Library Intrinsics

LLVM provides intrinsics for a few important standard C library functions. These intrinsics allow source-language front-ends to pass information about the alignment of the pointer arguments to the code generator, providing opportunity for more efficient code generation.

'llvm.memcpy' Intrinsic

Syntax: This is an overloaded intrinsic. You can use llvm.memcpy on any integer bit width and for different address spaces. Not all targets support all bit widths however.

```
declare void @llvm.memcpy.p0i8.p0i8.i32(i8* <dest>, i8* <src>,
                                        i32 <len>, i32 <align>, i1 <isvolatile>)
declare void @llvm.memcpy.p0i8.p0i8.i64(i8* <dest>, i8* <src>,
                                        i64 <len>, i32 <align>, i1 <isvolatile>)
```

Overview: The '`llvm.memcpy.*`' intrinsics copy a block of memory from the source location to the destination location.

Note that, unlike the standard libc function, the `llvm.memcpy.*` intrinsics do not return a value, takes extra alignment/isvolatile arguments and the pointers can be in specified address spaces.

Arguments: The first argument is a pointer to the destination, the second is a pointer to the source. The third argument is an integer argument specifying the number of bytes to copy, the fourth argument is the alignment of the source and destination locations, and the fifth is a boolean indicating a volatile access.

If the call to this intrinsic has an alignment value that is not 0 or 1, then the caller guarantees that both the source and destination pointers are aligned to that boundary.

If the `isvolatile` parameter is `true`, the `llvm.memcpy` call is a *volatile operation*. The detailed access behavior is not very cleanly specified and it is unwise to depend on it.

Semantics: The '`llvm.memcpy.*`' intrinsics copy a block of memory from the source location to the destination location, which are not allowed to overlap. It copies "len" bytes of memory over. If the argument is known to be aligned to some boundary, this can be specified as the fourth argument, otherwise it should be set to 0 or 1 (both meaning no alignment).

'`llvm.memmove`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use llvm.memmove on any integer bit width and for different address space. Not all targets support all bit widths however.

```
declare void @llvm.memmove.p0i8.p0i8.i32(i8* <dest>, i8* <src>,
                                         i32 <len>, i32 <align>, i1 <isvolatile>)
declare void @llvm.memmove.p0i8.p0i8.i64(i8* <dest>, i8* <src>,
                                         i64 <len>, i32 <align>, i1 <isvolatile>)
```

Overview: The '`llvm.memmove.*`' intrinsics move a block of memory from the source location to the destination location. It is similar to the '`llvm.memcpy`' intrinsic but allows the two memory locations to overlap.

Note that, unlike the standard libc function, the `llvm.memmove.*` intrinsics do not return a value, takes extra alignment/isvolatile arguments and the pointers can be in specified address spaces.

Arguments: The first argument is a pointer to the destination, the second is a pointer to the source. The third argument is an integer argument specifying the number of bytes to copy, the fourth argument is the alignment of the source and destination locations, and the fifth is a boolean indicating a volatile access.

If the call to this intrinsic has an alignment value that is not 0 or 1, then the caller guarantees that the source and destination pointers are aligned to that boundary.

If the `isvolatile` parameter is `true`, the `llvm.memmove` call is a *volatile operation*. The detailed access behavior is not very cleanly specified and it is unwise to depend on it.

Semantics: The '`llvm.memmove.*`' intrinsics copy a block of memory from the source location to the destination location, which may overlap. It copies "len" bytes of memory over. If the argument is known to be aligned to some boundary, this can be specified as the fourth argument, otherwise it should be set to 0 or 1 (both meaning no alignment).

'llvm.memset.*' Intrinsics

Syntax: This is an overloaded intrinsic. You can use llvm.memset on any integer bit width and for different address spaces. However, not all targets support all bit widths.

```
declare void @llvm.memset.p0i8.i32(i8* <dest>, i8 <val>,
                                   i32 <len>, i32 <align>, i1 <isvolatile>)
declare void @llvm.memset.p0i8.i64(i8* <dest>, i8 <val>,
                                   i64 <len>, i32 <align>, i1 <isvolatile>)
```

Overview: The 'llvm.memset.*' intrinsics fill a block of memory with a particular byte value.

Note that, unlike the standard libc function, the llvm.memset intrinsic does not return a value and takes extra alignment/volatile arguments. Also, the destination can be in an arbitrary address space.

Arguments: The first argument is a pointer to the destination to fill, the second is the byte value with which to fill it, the third argument is an integer argument specifying the number of bytes to fill, and the fourth argument is the known alignment of the destination location.

If the call to this intrinsic has an alignment value that is not 0 or 1, then the caller guarantees that the destination pointer is aligned to that boundary.

If the isvolatile parameter is true, the llvm.memset call is a *volatile operation*. The detailed access behavior is not very cleanly specified and it is unwise to depend on it.

Semantics: The 'llvm.memset.*' intrinsics fill "len" bytes of memory starting at the destination location. If the argument is known to be aligned to some boundary, this can be specified as the fourth argument, otherwise it should be set to 0 or 1 (both meaning no alignment).

'llvm.sqrt.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use llvm.sqrt on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.sqrt.f32(float %Val)
declare double    @llvm.sqrt.f64(double %Val)
declare x86_fp80  @llvm.sqrt.f80(x86_fp80 %Val)
declare fp128     @llvm.sqrt.f128(fp128 %Val)
declare ppc_fp128 @llvm.sqrt.ppcf128(ppc_fp128 %Val)
```

Overview: The 'llvm.sqrt' intrinsics return the sqrt of the specified operand, returning the same value as the libm 'sqrt' functions would. Unlike sqrt in libm, however, llvm.sqrt has undefined behavior for negative numbers other than -0.0 (which allows for better optimization, because there is no need to worry about errno being set). llvm.sqrt(-0.0) is defined to return -0.0 like IEEE sqrt.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the sqrt of the specified operand if it is a nonnegative floating point number.

'llvm.powi.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.powi` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.powi.f32(float  %Val, i32 %power)
declare double    @llvm.powi.f64(double %Val, i32 %power)
declare x86_fp80  @llvm.powi.f80(x86_fp80  %Val, i32 %power)
declare fp128     @llvm.powi.f128(fp128 %Val, i32 %power)
declare ppc_fp128 @llvm.powi.ppcf128(ppc_fp128  %Val, i32 %power)
```

Overview: The 'llvm.powi.*' intrinsics return the first operand raised to the specified (positive or negative) power. The order of evaluation of multiplications is not defined. When a vector of floating point type is used, the second argument remains a scalar integer value.

Arguments: The second argument is an integer power, and the first is a value to raise to that power.

Semantics: This function returns the first value raised to the second power with an unspecified sequence of rounding operations.

'llvm.sin.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.sin` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.sin.f32(float  %Val)
declare double    @llvm.sin.f64(double %Val)
declare x86_fp80  @llvm.sin.f80(x86_fp80  %Val)
declare fp128     @llvm.sin.f128(fp128 %Val)
declare ppc_fp128 @llvm.sin.ppcf128(ppc_fp128  %Val)
```

Overview: The 'llvm.sin.*' intrinsics return the sine of the operand.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the sine of the specified operand, returning the same values as the libm `sin` functions would, and handles error conditions in the same way.

'llvm.cos.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.cos` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.cos.f32(float  %Val)
declare double    @llvm.cos.f64(double %Val)
declare x86_fp80  @llvm.cos.f80(x86_fp80  %Val)
declare fp128     @llvm.cos.f128(fp128 %Val)
declare ppc_fp128 @llvm.cos.ppcf128(ppc_fp128  %Val)
```

Overview: The '`llvm.cos.*`' intrinsics return the cosine of the operand.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the cosine of the specified operand, returning the same values as the libm `cos` functions would, and handles error conditions in the same way.

'`llvm.pow.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.pow` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.pow.f32(float  %Val, float %Power)
declare double    @llvm.pow.f64(double %Val, double %Power)
declare x86_fp80  @llvm.pow.f80(x86_fp80  %Val, x86_fp80 %Power)
declare fp128     @llvm.pow.f128(fp128 %Val, fp128 %Power)
declare ppc_fp128 @llvm.pow.ppcf128(ppc_fp128  %Val, ppc_fp128 Power)
```

Overview: The '`llvm.pow.*`' intrinsics return the first operand raised to the specified (positive or negative) power.

Arguments: The second argument is a floating point power, and the first is a value to raise to that power.

Semantics: This function returns the first value raised to the second power, returning the same values as the libm `pow` functions would, and handles error conditions in the same way.

'`llvm.exp.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.exp` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.exp.f32(float  %Val)
declare double    @llvm.exp.f64(double %Val)
declare x86_fp80  @llvm.exp.f80(x86_fp80  %Val)
declare fp128     @llvm.exp.f128(fp128 %Val)
declare ppc_fp128 @llvm.exp.ppcf128(ppc_fp128  %Val)
```

Overview: The '`llvm.exp.*`' intrinsics perform the exp function.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `exp` functions would, and handles error conditions in the same way.

'`llvm.exp2.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.exp2` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.exp2.f32(float  %Val)
declare double    @llvm.exp2.f64(double %Val)
declare x86_fp80  @llvm.exp2.f80(x86_fp80  %Val)
declare fp128     @llvm.exp2.f128(fp128 %Val)
declare ppc_fp128 @llvm.exp2.ppcf128(ppc_fp128  %Val)
```

Overview: The '`llvm.exp2.*`' intrinsics perform the exp2 function.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `exp2` functions would, and handles error conditions in the same way.

'`llvm.log.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.log` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.log.f32(float  %Val)
declare double    @llvm.log.f64(double %Val)
declare x86_fp80  @llvm.log.f80(x86_fp80  %Val)
declare fp128     @llvm.log.f128(fp128 %Val)
declare ppc_fp128 @llvm.log.ppcf128(ppc_fp128  %Val)
```

Overview: The '`llvm.log.*`' intrinsics perform the log function.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `log` functions would, and handles error conditions in the same way.

'`llvm.log10.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.log10` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.log10.f32(float  %Val)
declare double    @llvm.log10.f64(double %Val)
declare x86_fp80  @llvm.log10.f80(x86_fp80  %Val)
declare fp128     @llvm.log10.f128(fp128 %Val)
declare ppc_fp128 @llvm.log10.ppcf128(ppc_fp128  %Val)
```

Overview: The '`llvm.log10.*`' intrinsics perform the log10 function.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `log10` functions would, and handles error conditions in the same way.

'llvm.log2.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.log2` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.log2.f32(float  %Val)
declare double    @llvm.log2.f64(double %Val)
declare x86_fp80  @llvm.log2.f80(x86_fp80  %Val)
declare fp128     @llvm.log2.f128(fp128 %Val)
declare ppc_fp128 @llvm.log2.ppcf128(ppc_fp128  %Val)
```

Overview: The 'llvm.log2.*' intrinsics perform the log2 function.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `log2` functions would, and handles error conditions in the same way.

'llvm.fma.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.fma` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.fma.f32(float  %a, float  %b, float  %c)
declare double    @llvm.fma.f64(double %a, double %b, double %c)
declare x86_fp80  @llvm.fma.f80(x86_fp80 %a, x86_fp80 %b, x86_fp80 %c)
declare fp128     @llvm.fma.f128(fp128 %a, fp128 %b, fp128 %c)
declare ppc_fp128 @llvm.fma.ppcf128(ppc_fp128 %a, ppc_fp128 %b, ppc_fp128 %c)
```

Overview: The 'llvm.fma.*' intrinsics perform the fused multiply-add operation.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `fma` functions would, and does not set errno.

'llvm.fabs.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.fabs` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.fabs.f32(float  %Val)
declare double    @llvm.fabs.f64(double %Val)
declare x86_fp80  @llvm.fabs.f80(x86_fp80  %Val)
declare fp128     @llvm.fabs.f128(fp128 %Val)
declare ppc_fp128 @llvm.fabs.ppcf128(ppc_fp128  %Val)
```

Overview: The '`llvm.fabs.*`' intrinsics return the absolute value of the operand.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `fabs` functions would, and handles error conditions in the same way.

'`llvm.copysign.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.copysign` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.copysign.f32(float  %Mag, float  %Sgn)
declare double    @llvm.copysign.f64(double %Mag, double %Sgn)
declare x86_fp80  @llvm.copysign.f80(x86_fp80  %Mag, x86_fp80  %Sgn)
declare fp128     @llvm.copysign.f128(fp128 %Mag, fp128 %Sgn)
declare ppc_fp128 @llvm.copysign.ppcf128(ppc_fp128  %Mag, ppc_fp128  %Sgn)
```

Overview: The '`llvm.copysign.*`' intrinsics return a value with the magnitude of the first operand and the sign of the second operand.

Arguments: The arguments and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `copysign` functions would, and handles error conditions in the same way.

'`llvm.floor.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.floor` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.floor.f32(float  %Val)
declare double    @llvm.floor.f64(double %Val)
declare x86_fp80  @llvm.floor.f80(x86_fp80  %Val)
declare fp128     @llvm.floor.f128(fp128 %Val)
declare ppc_fp128 @llvm.floor.ppcf128(ppc_fp128  %Val)
```

Overview: The '`llvm.floor.*`' intrinsics return the floor of the operand.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `floor` functions would, and handles error conditions in the same way.

'llvm.ceil.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.ceil` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.ceil.f32(float  %Val)
declare double    @llvm.ceil.f64(double %Val)
declare x86_fp80  @llvm.ceil.f80(x86_fp80  %Val)
declare fp128     @llvm.ceil.f128(fp128 %Val)
declare ppc_fp128 @llvm.ceil.ppcf128(ppc_fp128  %Val)
```

Overview: The 'llvm.ceil.*' intrinsics return the ceiling of the operand.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `ceil` functions would, and handles error conditions in the same way.

'llvm.trunc.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.trunc` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.trunc.f32(float  %Val)
declare double    @llvm.trunc.f64(double %Val)
declare x86_fp80  @llvm.trunc.f80(x86_fp80  %Val)
declare fp128     @llvm.trunc.f128(fp128 %Val)
declare ppc_fp128 @llvm.trunc.ppcf128(ppc_fp128  %Val)
```

Overview: The 'llvm.trunc.*' intrinsics returns the operand rounded to the nearest integer not larger in magnitude than the operand.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `trunc` functions would, and handles error conditions in the same way.

'llvm.rint.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.rint` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.rint.f32(float  %Val)
declare double    @llvm.rint.f64(double %Val)
declare x86_fp80  @llvm.rint.f80(x86_fp80  %Val)
declare fp128     @llvm.rint.f128(fp128 %Val)
declare ppc_fp128 @llvm.rint.ppcf128(ppc_fp128  %Val)
```

Overview: The '`llvm.rint.*`' intrinsics returns the operand rounded to the nearest integer. It may raise an inexact floating-point exception if the operand isn't an integer.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `rint` functions would, and handles error conditions in the same way.

'`llvm.nearbyint.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.nearbyint` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.nearbyint.f32(float  %Val)
declare double    @llvm.nearbyint.f64(double %Val)
declare x86_fp80  @llvm.nearbyint.f80(x86_fp80  %Val)
declare fp128     @llvm.nearbyint.f128(fp128 %Val)
declare ppc_fp128 @llvm.nearbyint.ppcf128(ppc_fp128  %Val)
```

Overview: The '`llvm.nearbyint.*`' intrinsics returns the operand rounded to the nearest integer.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `nearbyint` functions would, and handles error conditions in the same way.

'`llvm.round.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.round` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.round.f32(float  %Val)
declare double    @llvm.round.f64(double %Val)
declare x86_fp80  @llvm.round.f80(x86_fp80  %Val)
declare fp128     @llvm.round.f128(fp128 %Val)
declare ppc_fp128 @llvm.round.ppcf128(ppc_fp128  %Val)
```

Overview: The '`llvm.round.*`' intrinsics returns the operand rounded to the nearest integer.

Arguments: The argument and return value are floating point numbers of the same type.

Semantics: This function returns the same values as the libm `round` functions would, and handles error conditions in the same way.

Bit Manipulation Intrinsics

LLVM provides intrinsics for a few important bit manipulation operations. These allow efficient code generation for some algorithms.

'llvm.bswap.*' Intrinsics

Syntax: This is an overloaded intrinsic function. You can use bswap on any integer type that is an even number of bytes (i.e. BitWidth % 16 == 0).

```
declare i16 @llvm.bswap.i16(i16 <id>)
declare i32 @llvm.bswap.i32(i32 <id>)
declare i64 @llvm.bswap.i64(i64 <id>)
```

Overview: The 'llvm.bswap' family of intrinsics is used to byte swap integer values with an even number of bytes (positive multiple of 16 bits). These are useful for performing operations on data that is not in the target's native byte order.

Semantics: The llvm.bswap.i16 intrinsic returns an i16 value that has the high and low byte of the input i16 swapped. Similarly, the llvm.bswap.i32 intrinsic returns an i32 value that has the four bytes of the input i32 swapped, so that if the input bytes are numbered 0, 1, 2, 3 then the returned i32 will have its bytes in 3, 2, 1, 0 order. The llvm.bswap.i48, llvm.bswap.i64 and other intrinsics extend this concept to additional even-byte lengths (6 bytes, 8 bytes and more, respectively).

'llvm.ctpop.*' Intrinsic

Syntax: This is an overloaded intrinsic. You can use llvm.ctpop on any integer bit width, or on any vector with integer elements. Not all targets support all bit widths or vector types, however.

```
declare i8 @llvm.ctpop.i8(i8  <src>)
declare i16 @llvm.ctpop.i16(i16 <src>)
declare i32 @llvm.ctpop.i32(i32 <src>)
declare i64 @llvm.ctpop.i64(i64 <src>)
declare i256 @llvm.ctpop.i256(i256 <src>)
declare <2 x i32> @llvm.ctpop.v2i32(<2 x i32> <src>)
```

Overview: The 'llvm.ctpop' family of intrinsics counts the number of bits set in a value.

Arguments: The only argument is the value to be counted. The argument may be of any integer type, or a vector with integer elements. The return type must match the argument type.

Semantics: The 'llvm.ctpop' intrinsic counts the 1's in a variable, or within each element of a vector.

'`llvm.ctlz.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.ctlz` on any integer bit width, or any vector whose elements are integers. Not all targets support all bit widths or vector types, however.

```
declare i8   @llvm.ctlz.i8  (i8   <src>, i1 <is_zero_undef>)
declare i16  @llvm.ctlz.i16 (i16  <src>, i1 <is_zero_undef>)
declare i32  @llvm.ctlz.i32 (i32  <src>, i1 <is_zero_undef>)
declare i64  @llvm.ctlz.i64 (i64  <src>, i1 <is_zero_undef>)
declare i256 @llvm.ctlz.i256(i256 <src>, i1 <is_zero_undef>)
declase <2 x i32> @llvm.ctlz.v2i32(<2 x i32> <src>, i1 <is_zero_undef>)
```

Overview: The '`llvm.ctlz`' family of intrinsic functions counts the number of leading zeros in a variable.

Arguments: The first argument is the value to be counted. This argument may be of any integer type, or a vectory with integer element type. The return type must match the first argument type.

The second argument must be a constant and is a flag to indicate whether the intrinsic should ensure that a zero as the first argument produces a defined result. Historically some architectures did not provide a defined result for zero values as efficiently, and many algorithms are now predicated on avoiding zero-value inputs.

Semantics: The '`llvm.ctlz`' intrinsic counts the leading (most significant) zeros in a variable, or within each element of the vector. If `src == 0` then the result is the size in bits of the type of `src` if `is_zero_undef == 0` and `undef` otherwise. For example, `llvm.ctlz(i32 2) = 30`.

'`llvm.cttz.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.cttz` on any integer bit width, or any vector of integer elements. Not all targets support all bit widths or vector types, however.

```
declare i8   @llvm.cttz.i8  (i8   <src>, i1 <is_zero_undef>)
declare i16  @llvm.cttz.i16 (i16  <src>, i1 <is_zero_undef>)
declare i32  @llvm.cttz.i32 (i32  <src>, i1 <is_zero_undef>)
declare i64  @llvm.cttz.i64 (i64  <src>, i1 <is_zero_undef>)
declare i256 @llvm.cttz.i256(i256 <src>, i1 <is_zero_undef>)
declase <2 x i32> @llvm.cttz.v2i32(<2 x i32> <src>, i1 <is_zero_undef>)
```

Overview: The '`llvm.cttz`' family of intrinsic functions counts the number of trailing zeros.

Arguments: The first argument is the value to be counted. This argument may be of any integer type, or a vectory with integer element type. The return type must match the first argument type.

The second argument must be a constant and is a flag to indicate whether the intrinsic should ensure that a zero as the first argument produces a defined result. Historically some architectures did not provide a defined result for zero values as efficiently, and many algorithms are now predicated on avoiding zero-value inputs.

Semantics: The '`llvm.cttz`' intrinsic counts the trailing (least significant) zeros in a variable, or within each element of a vector. If `src == 0` then the result is the size in bits of the type of `src` if `is_zero_undef == 0` and `undef` otherwise. For example, `llvm.cttz(2) = 1`.

Arithmetic with Overflow Intrinsics

LLVM provides intrinsics for some arithmetic with overflow operations.

'llvm.sadd.with.overflow.*' Intrinsics

Syntax: This is an overloaded intrinsic. You can use `llvm.sadd.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.sadd.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.sadd.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.sadd.with.overflow.i64(i64 %a, i64 %b)
```

Overview: The 'llvm.sadd.with.overflow' family of intrinsic functions perform a signed addition of the two arguments, and indicate whether an overflow occurred during the signed summation.

Arguments: The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo signed addition.

Semantics: The 'llvm.sadd.with.overflow' family of intrinsic functions perform a signed addition of the two variables. They return a structure — the first element of which is the signed summation, and the second element of which is a bit specifying if the signed summation resulted in an overflow.

Examples:

```
%res = call {i32, i1} @llvm.sadd.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %overflow, label %normal
```

'llvm.uadd.with.overflow.*' Intrinsics

Syntax: This is an overloaded intrinsic. You can use `llvm.uadd.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.uadd.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.uadd.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.uadd.with.overflow.i64(i64 %a, i64 %b)
```

Overview: The 'llvm.uadd.with.overflow' family of intrinsic functions perform an unsigned addition of the two arguments, and indicate whether a carry occurred during the unsigned summation.

Arguments: The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo unsigned addition.

Semantics: The 'llvm.uadd.with.overflow' family of intrinsic functions perform an unsigned addition of the two arguments. They return a structure — the first element of which is the sum, and the second element of which is a bit specifying if the unsigned summation resulted in a carry.

Examples:

```
%res = call {i32, i1} @llvm.uadd.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %carry, label %normal
```

'llvm.ssub.with.overflow.*' Intrinsics

Syntax: This is an overloaded intrinsic. You can use `llvm.ssub.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.ssub.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.ssub.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.ssub.with.overflow.i64(i64 %a, i64 %b)
```

Overview: The 'llvm.ssub.with.overflow' family of intrinsic functions perform a signed subtraction of the two arguments, and indicate whether an overflow occurred during the signed subtraction.

Arguments: The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo signed subtraction.

Semantics: The 'llvm.ssub.with.overflow' family of intrinsic functions perform a signed subtraction of the two arguments. They return a structure — the first element of which is the subtraction, and the second element of which is a bit specifying if the signed subtraction resulted in an overflow.

Examples:

```
%res = call {i32, i1} @llvm.ssub.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %overflow, label %normal
```

'llvm.usub.with.overflow.*' Intrinsics

Syntax: This is an overloaded intrinsic. You can use `llvm.usub.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.usub.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.usub.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.usub.with.overflow.i64(i64 %a, i64 %b)
```

Overview: The 'llvm.usub.with.overflow' family of intrinsic functions perform an unsigned subtraction of the two arguments, and indicate whether an overflow occurred during the unsigned subtraction.

Arguments: The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo unsigned subtraction.

Semantics: The '`llvm.usub.with.overflow`' family of intrinsic functions perform an unsigned subtraction of the two arguments. They return a structure — the first element of which is the subtraction, and the second element of which is a bit specifying if the unsigned subtraction resulted in an overflow.

Examples:

```
%res = call {i32, i1} @llvm.usub.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %overflow, label %normal
```

'`llvm.smul.with.overflow.*`' Intrinsics

Syntax: This is an overloaded intrinsic. You can use `llvm.smul.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.smul.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.smul.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.smul.with.overflow.i64(i64 %a, i64 %b)
```

Overview: The '`llvm.smul.with.overflow`' family of intrinsic functions perform a signed multiplication of the two arguments, and indicate whether an overflow occurred during the signed multiplication.

Arguments: The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo signed multiplication.

Semantics: The '`llvm.smul.with.overflow`' family of intrinsic functions perform a signed multiplication of the two arguments. They return a structure — the first element of which is the multiplication, and the second element of which is a bit specifying if the signed multiplication resulted in an overflow.

Examples:

```
%res = call {i32, i1} @llvm.smul.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %overflow, label %normal
```

'`llvm.umul.with.overflow.*`' Intrinsics

Syntax: This is an overloaded intrinsic. You can use `llvm.umul.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.umul.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.umul.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.umul.with.overflow.i64(i64 %a, i64 %b)
```

Overview: The '`llvm.umul.with.overflow`' family of intrinsic functions perform a unsigned multiplication of the two arguments, and indicate whether an overflow occurred during the unsigned multiplication.

Arguments: The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo unsigned multiplication.

Semantics: The '`llvm.umul.with.overflow`' family of intrinsic functions perform an unsigned multiplication of the two arguments. They return a structure — the first element of which is the multiplication, and the second element of which is a bit specifying if the unsigned multiplication resulted in an overflow.

Examples:

```
%res = call {i32, i1} @llvm.umul.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %overflow, label %normal
```

Specialised Arithmetic Intrinsics

'`llvm.fmuladd.*`' Intrinsic

Syntax:

```
declare float @llvm.fmuladd.f32(float %a, float %b, float %c)
declare double @llvm.fmuladd.f64(double %a, double %b, double %c)
```

Overview: The '`llvm.fmuladd.*`' intrinsic functions represent multiply-add expressions that can be fused if the code generator determines that (a) the target instruction set has support for a fused operation, and (b) that the fused operation is more efficient than the equivalent, separate pair of mul and add instructions.

Arguments: The '`llvm.fmuladd.*`' intrinsics each take three arguments: two multiplicands, a and b, and an addend c.

Semantics: The expression:

```
%0 = call float @llvm.fmuladd.f32(%a, %b, %c)
```

is equivalent to the expression a * b + c, except that rounding will not be performed between the multiplication and addition steps if the code generator fuses the operations. Fusion is not guaranteed, even if the target platform supports it. If a fused multiply-add is required the corresponding llvm.fma.* intrinsic function should be used instead. This never sets errno, just as '`llvm.fma.*`'.

Examples:

```
%r2 = call float @llvm.fmuladd.f32(float %a, float %b, float %c) ; yields float:r2 = (a * b) + c
```

Half Precision Floating Point Intrinsics

For most target platforms, half precision floating point is a storage-only format. This means that it is a dense encoding (in memory) but does not support computation in the format.

This means that code must first load the half-precision floating point value as an i16, then convert it to float with *llvm.convert.from.fp16*. Computation can then be performed on the float value (including extending to double etc). To

store the value back to memory, it is first converted to float if needed, then converted to i16 with *llvm.convert.to.fp16*, then storing as an i16 value.

'llvm.convert.to.fp16' Intrinsic

Syntax:

```
declare i16 @llvm.convert.to.fp16.f32(float %a)
declare i16 @llvm.convert.to.fp16.f64(double %a)
```

Overview: The 'llvm.convert.to.fp16' intrinsic function performs a conversion from a conventional floating point type to half precision floating point format.

Arguments: The intrinsic function contains single argument - the value to be converted.

Semantics: The 'llvm.convert.to.fp16' intrinsic function performs a conversion from a conventional floating point format to half precision floating point format. The return value is an i16 which contains the converted number.

Examples:

```
%res = call i16 @llvm.convert.to.fp16.f32(float %a)
store i16 %res, i16* @x, align 2
```

'llvm.convert.from.fp16' Intrinsic

Syntax:

```
declare float @llvm.convert.from.fp16.f32(i16 %a)
declare double @llvm.convert.from.fp16.f64(i16 %a)
```

Overview: The 'llvm.convert.from.fp16' intrinsic function performs a conversion from half precision floating point format to single precision floating point format.

Arguments: The intrinsic function contains single argument - the value to be converted.

Semantics: The 'llvm.convert.from.fp16' intrinsic function performs a conversion from half single precision floating point format to single precision floating point format. The input half-float value is represented by an i16 value.

Examples:

```
%a = load i16* @x, align 2
%res = call float @llvm.convert.from.fp16(i16 %a)
```

Debugger Intrinsics

The LLVM debugger intrinsics (which all start with llvm.dbg. prefix), are described in the LLVM Source Level Debugging document.

Exception Handling Intrinsics

The LLVM exception handling intrinsics (which all start with `llvm.eh.` prefix), are described in the LLVM Exception Handling document.

Trampoline Intrinsics

These intrinsics make it possible to excise one parameter, marked with the *nest* attribute, from a function. The result is a callable function pointer lacking the nest parameter - the caller does not need to provide a value for it. Instead, the value to use is stored in advance in a "trampoline", a block of memory usually allocated on the stack, which also contains code to splice the nest value into the argument list. This is used to implement the GCC nested function address extension.

For example, if the function is `i32 f(i8* nest %c, i32 %x, i32 %y)` then the resulting function pointer has signature `i32 (i32, i32)*`. It can be created as follows:

```
%tramp = alloca [10 x i8], align 4 ; size and alignment only correct for X86
%tramp1 = getelementptr [10 x i8]* %tramp, i32 0, i32 0
call i8* @llvm.init.trampoline(i8* %tramp1, i8* bitcast (i32 (i8*, i32, i32)* @f to i8*), i8* %nval)
%p = call i8* @llvm.adjust.trampoline(i8* %tramp1)
%fp = bitcast i8* %p to i32 (i32, i32)*
```

The call `%val = call i32 %fp(i32 %x, i32 %y)` is then equivalent to `%val = call i32 %f(i8* %nval, i32 %x, i32 %y)`.

'`llvm.init.trampoline`' Intrinsic

Syntax:

```
declare void @llvm.init.trampoline(i8* <tramp>, i8* <func>, i8* <nval>)
```

Overview: This fills the memory pointed to by `tramp` with executable code, turning it into a trampoline.

Arguments: The `llvm.init.trampoline` intrinsic takes three arguments, all pointers. The `tramp` argument must point to a sufficiently large and sufficiently aligned block of memory; this memory is written to by the intrinsic. Note that the size and the alignment are target-specific - LLVM currently provides no portable way of determining them, so a front-end that generates this intrinsic needs to have some target-specific knowledge. The `func` argument must hold a function bitcast to an `i8*`.

Semantics: The block of memory pointed to by `tramp` is filled with target dependent code, turning it into a function. Then `tramp` needs to be passed to *llvm.adjust.trampoline* to get a pointer which can be *bitcast (to a new function) and called*. The new function's signature is the same as that of `func` with any arguments marked with the `nest` attribute removed. At most one such `nest` argument is allowed, and it must be of pointer type. Calling the new function is equivalent to calling `func` with the same argument list, but with `nval` used for the missing `nest` argument. If, after calling `llvm.init.trampoline`, the memory pointed to by `tramp` is modified, then the effect of any later call to the returned function pointer is undefined.

'`llvm.adjust.trampoline`' Intrinsic

Syntax:

```
declare i8* @llvm.adjust.trampoline(i8* <tramp>)
```

Overview: This performs any required machine-specific adjustment to the address of a trampoline (passed as `tramp`).

Arguments: `tramp` must point to a block of memory which already has trampoline code filled in by a previous call to *llvm.init.trampoline*.

Semantics: On some architectures the address of the code to be executed needs to be different than the address where the trampoline is actually stored. This intrinsic returns the executable address corresponding to `tramp` after performing the required machine specific adjustments. The pointer returned can then be *bitcast and executed*.

Memory Use Markers

This class of intrinsics provides information about the lifetime of memory objects and ranges where variables are immutable.

'`llvm.lifetime.start`' Intrinsic

Syntax:

```
declare void @llvm.lifetime.start(i64 <size>, i8* nocapture <ptr>)
```

Overview: The '`llvm.lifetime.start`' intrinsic specifies the start of a memory object's lifetime.

Arguments: The first argument is a constant integer representing the size of the object, or -1 if it is variable sized. The second argument is a pointer to the object.

Semantics: This intrinsic indicates that before this point in the code, the value of the memory pointed to by `ptr` is dead. This means that it is known to never be used and has an undefined value. A load from the pointer that precedes this intrinsic can be replaced with `'undef'`.

'`llvm.lifetime.end`' Intrinsic

Syntax:

```
declare void @llvm.lifetime.end(i64 <size>, i8* nocapture <ptr>)
```

Overview: The '`llvm.lifetime.end`' intrinsic specifies the end of a memory object's lifetime.

Arguments: The first argument is a constant integer representing the size of the object, or -1 if it is variable sized. The second argument is a pointer to the object.

Semantics: This intrinsic indicates that after this point in the code, the value of the memory pointed to by `ptr` is dead. This means that it is known to never be used and has an undefined value. Any stores into the memory object following this intrinsic may be removed as dead.

'llvm.invariant.start' Intrinsic

Syntax:

```
declare {}* @llvm.invariant.start(i64 <size>, i8* nocapture <ptr>)
```

Overview: The 'llvm.invariant.start' intrinsic specifies that the contents of a memory object will not change.

Arguments: The first argument is a constant integer representing the size of the object, or -1 if it is variable sized. The second argument is a pointer to the object.

Semantics: This intrinsic indicates that until an llvm.invariant.end that uses the return value, the referenced memory location is constant and unchanging.

'llvm.invariant.end' Intrinsic

Syntax:

```
declare void @llvm.invariant.end({}* <start>, i64 <size>, i8* nocapture <ptr>)
```

Overview: The 'llvm.invariant.end' intrinsic specifies that the contents of a memory object are mutable.

Arguments: The first argument is the matching llvm.invariant.start intrinsic. The second argument is a constant integer representing the size of the object, or -1 if it is variable sized and the third argument is a pointer to the object.

Semantics: This intrinsic indicates that the memory is mutable again.

General Intrinsics

This class of intrinsics is designed to be generic and has no specific purpose.

'llvm.var.annotation' Intrinsic

Syntax:

```
declare void @llvm.var.annotation(i8* <val>, i8* <str>, i8* <str>, i32  <int>)
```

Overview: The 'llvm.var.annotation' intrinsic.

Arguments: The first argument is a pointer to a value, the second is a pointer to a global string, the third is a pointer to a global string which is the source file name, and the last argument is the line number.

Semantics: This intrinsic allows annotation of local variables with arbitrary strings. This can be useful for special purpose optimizations that want to look for these annotations. These have no other defined use; they are ignored by code generation and optimization.

'`llvm.ptr.annotation.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use '`llvm.ptr.annotation`' on a pointer to an integer of any width. *NOTE* you must specify an address space for the pointer. The identifier for the default address space is the integer '`0`'.

```
declare i8*   @llvm.ptr.annotation.p<address space>i8(i8* <val>, i8* <str>, i8* <str>, i32  <int>)
declare i16*  @llvm.ptr.annotation.p<address space>i16(i16* <val>, i8* <str>, i8* <str>, i32  <int>)
declare i32*  @llvm.ptr.annotation.p<address space>i32(i32* <val>, i8* <str>, i8* <str>, i32  <int>)
declare i64*  @llvm.ptr.annotation.p<address space>i64(i64* <val>, i8* <str>, i8* <str>, i32  <int>)
declare i256* @llvm.ptr.annotation.p<address space>i256(i256* <val>, i8* <str>, i8* <str>, i32  <int>
```

Overview: The '`llvm.ptr.annotation`' intrinsic.

Arguments: The first argument is a pointer to an integer value of arbitrary bitwidth (result of some expression), the second is a pointer to a global string, the third is a pointer to a global string which is the source file name, and the last argument is the line number. It returns the value of the first argument.

Semantics: This intrinsic allows annotation of a pointer to an integer with arbitrary strings. This can be useful for special purpose optimizations that want to look for these annotations. These have no other defined use; they are ignored by code generation and optimization.

'`llvm.annotation.*`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use '`llvm.annotation`' on any integer bit width.

```
declare i8 @llvm.annotation.i8(i8 <val>, i8* <str>, i8* <str>, i32  <int>)
declare i16 @llvm.annotation.i16(i16 <val>, i8* <str>, i8* <str>, i32  <int>)
declare i32 @llvm.annotation.i32(i32 <val>, i8* <str>, i8* <str>, i32  <int>)
declare i64 @llvm.annotation.i64(i64 <val>, i8* <str>, i8* <str>, i32  <int>)
declare i256 @llvm.annotation.i256(i256 <val>, i8* <str>, i8* <str>, i32  <int>)
```

Overview: The '`llvm.annotation`' intrinsic.

Arguments: The first argument is an integer value (result of some expression), the second is a pointer to a global string, the third is a pointer to a global string which is the source file name, and the last argument is the line number. It returns the value of the first argument.

Semantics: This intrinsic allows annotations to be put on arbitrary expressions with arbitrary strings. This can be useful for special purpose optimizations that want to look for these annotations. These have no other defined use; they are ignored by code generation and optimization.

'`llvm.trap`' Intrinsic

Syntax:

```
declare void @llvm.trap() noreturn nounwind
```

Overview: The '`llvm.trap`' intrinsic.

Arguments: None.

Semantics: This intrinsic is lowered to the target dependent trap instruction. If the target does not have a trap instruction, this intrinsic will be lowered to a call of the `abort()` function.

'`llvm.debugtrap`' Intrinsic

Syntax:

```
declare void @llvm.debugtrap() nounwind
```

Overview: The '`llvm.debugtrap`' intrinsic.

Arguments: None.

Semantics: This intrinsic is lowered to code which is intended to cause an execution trap with the intention of requesting the attention of a debugger.

'`llvm.stackprotector`' Intrinsic

Syntax:

```
declare void @llvm.stackprotector(i8* <guard>, i8** <slot>)
```

Overview: The `llvm.stackprotector` intrinsic takes the `guard` and stores it onto the stack at `slot`. The stack slot is adjusted to ensure that it is placed on the stack before local variables.

Arguments: The `llvm.stackprotector` intrinsic requires two pointer arguments. The first argument is the value loaded from the stack guard `@__stack_chk_guard`. The second variable is an `alloca` that has enough space to hold the value of the guard.

Semantics: This intrinsic causes the prologue/epilogue inserter to force the position of the `AllocaInst` stack slot to be before local variables on the stack. This is to ensure that if a local variable on the stack is overwritten, it will destroy the value of the guard. When the function exits, the guard on the stack is checked against the original guard by `llvm.stackprotectorcheck`. If they are different, then `llvm.stackprotectorcheck` causes the program to abort by calling the `__stack_chk_fail()` function.

'`llvm.stackprotectorcheck`' Intrinsic

Syntax:

```
declare void @llvm.stackprotectorcheck(i8** <guard>)
```

Overview: The `llvm.stackprotectorcheck` intrinsic compares `guard` against an already created stack protector and if they are not equal calls the `__stack_chk_fail()` function.

Arguments: The `llvm.stackprotectorcheck` intrinsic requires one pointer argument, the the variable `@__stack_chk_guard`.

Semantics: This intrinsic is provided to perform the stack protector check by comparing `guard` with the stack slot created by `llvm.stackprotector` and if the values do not match call the `__stack_chk_fail()` function.

The reason to provide this as an IR level intrinsic instead of implementing it via other IR operations is that in order to perform this operation at the IR level without an intrinsic, one would need to create additional basic blocks to handle the success/failure cases. This makes it difficult to stop the stack protector check from disrupting sibling tail calls in Codegen. With this intrinsic, we are able to generate the stack protector basic blocks late in codegen after the tail call decision has occurred.

'`llvm.objectsize`' Intrinsic

Syntax:

```
declare i32 @llvm.objectsize.i32(i8* <object>, i1 <min>)
declare i64 @llvm.objectsize.i64(i8* <object>, i1 <min>)
```

Overview: The `llvm.objectsize` intrinsic is designed to provide information to the optimizers to determine at compile time whether a) an operation (like memcpy) will overflow a buffer that corresponds to an object, or b) that a runtime check for overflow isn't necessary. An object in this context means an allocation of a specific class, structure, array, or other object.

Arguments: The `llvm.objectsize` intrinsic takes two arguments. The first argument is a pointer to or into the `object`. The second argument is a boolean and determines whether `llvm.objectsize` returns 0 (if true) or -1 (if false) when the object size is unknown. The second argument only accepts constants.

Semantics: The `llvm.objectsize` intrinsic is lowered to a constant representing the size of the object concerned. If the size cannot be determined at compile time, `llvm.objectsize` returns `i32/i64 -1 or 0` (depending on the `min` argument).

'`llvm.expect`' Intrinsic

Syntax: This is an overloaded intrinsic. You can use `llvm.expect` on any integer bit width.

```
declare i1 @llvm.expect.i1(i1 <val>, i1 <expected_val>)
declare i32 @llvm.expect.i32(i32 <val>, i32 <expected_val>)
declare i64 @llvm.expect.i64(i64 <val>, i64 <expected_val>)
```

Overview: The `llvm.expect` intrinsic provides information about expected (the most probable) value of `val`, which can be used by optimizers.

Arguments: The `llvm.expect` intrinsic takes two arguments. The first argument is a value. The second argument is an expected value, this needs to be a constant value, variables are not allowed.

Semantics: This intrinsic is lowered to the `val`.

'llvm.assume' Intrinsic

Syntax:

```
declare void @llvm.assume(i1 %cond)
```

Overview: The llvm.assume allows the optimizer to assume that the provided condition is true. This information can then be used in simplifying other parts of the code.

Arguments: The condition which the optimizer may assume is always true.

Semantics: The intrinsic allows the optimizer to assume that the provided condition is always true whenever the control flow reaches the intrinsic call. No code is generated for this intrinsic, and instructions that contribute only to the provided condition are not used for code generation. If the condition is violated during execution, the behavior is undefined.

Please note that optimizer might limit the transformations performed on values used by the llvm.assume intrinsic in order to preserve the instructions only used to form the intrinsic's input argument. This might prove undesirable if the extra information provided by the llvm.assume intrinsic does cause sufficient overall improvement in code quality. For this reason, llvm.assume should not be used to document basic mathematical invariants that the optimizer can otherwise deduce or facts that are of little use to the optimizer.

'llvm.donothing' Intrinsic

Syntax:

```
declare void @llvm.donothing() nounwind readnone
```

Overview: The llvm.donothing intrinsic doesn't perform any operation. It's the only intrinsic that can be called with an invoke instruction.

Arguments: None.

Semantics: This intrinsic does nothing, and it's removed by optimizers and ignored by codegen.

Stack Map Intrinsics

LLVM provides experimental intrinsics to support runtime patching mechanisms commonly desired in dynamic language JITs. These intrinsics are described in *Stack maps and patch points in LLVM*.

LLVM Language Reference Manual Defines the LLVM intermediate representation.

Introduction to the LLVM Compiler Presentation providing a users introduction to LLVM.

Intro to LLVM Book chapter providing a compiler hacker's introduction to LLVM.

LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation Design overview.

LLVM: An Infrastructure for Multi-Stage Optimization More details (quite old now).

Publications mentioning LLVM

CHAPTER

TWO

USER GUIDES

For those new to the LLVM system.

NOTE: If you are a user who is only interested in using LLVM-based compilers, you should look into Clang or DragonEgg instead. The documentation here is intended for users who have a need to work with the intermediate LLVM representation.

2.1 Building LLVM with CMake

2.1.1 Introduction

CMake is a cross-platform build-generator tool. CMake does not build the project, it generates the files needed by your build tool (GNU make, Visual Studio, etc) for building LLVM.

If you are really anxious about getting a functional LLVM build, go to the Quick start section. If you are a CMake novice, start on Basic CMake usage and then go back to the Quick start once you know what you are doing. The Options and variables section is a reference for customizing your build. If you already have experience with CMake, this is the recommended starting point.

2.1.2 Quick start

We use here the command-line, non-interactive CMake interface.

1. Download and install CMake. Version 2.8 is the minimum required.

2. Open a shell. Your development tools must be reachable from this shell through the PATH environment variable.
3. Create a directory for containing the build. It is not supported to build LLVM on the source directory. cd to this directory:

   ```
   $ mkdir mybuilddir
   $ cd mybuilddir
   ```

4. Execute this command on the shell replacing *path/to/llvm/source/root* with the path to the root of your LLVM source tree:

   ```
   $ cmake path/to/llvm/source/root
   ```

   CMake will detect your development environment, perform a series of test and generate the files required for building LLVM. CMake will use default values for all build parameters. See the Options and variables section for fine-tuning your build

   This can fail if CMake can't detect your toolset, or if it thinks that the environment is not sane enough. On this case make sure that the toolset that you intend to use is the only one reachable from the shell and that the shell itself is the correct one for you development environment. CMake will refuse to build MinGW makefiles if you have a POSIX shell reachable through the PATH environment variable, for instance. You can force CMake to use a given build tool, see the Usage section.

2.1.3 Basic CMake usage

This section explains basic aspects of CMake, mostly for explaining those options which you may need on your day-to-day usage.

CMake comes with extensive documentation in the form of html files and on the cmake executable itself. Execute `cmake --help` for further help options.

CMake requires to know for which build tool it shall generate files (GNU make, Visual Studio, Xcode, etc). If not specified on the command line, it tries to guess it based on you environment. Once identified the build tool, CMake uses the corresponding *Generator* for creating files for your build tool. You can explicitly specify the generator with the command line option `-G "Name of the generator"`. For knowing the available generators on your platform, execute

```
$ cmake --help
```

This will list the generator's names at the end of the help text. Generator's names are case-sensitive. Example:

```
$ cmake -G "Visual Studio 11" path/to/llvm/source/root
```

For a given development platform there can be more than one adequate generator. If you use Visual Studio "NMake Makefiles" is a generator you can use for building with NMake. By default, CMake chooses the more specific generator supported by your development environment. If you want an alternative generator, you must tell this to CMake with the `-G` option.

2.1.4 Options and variables

Variables customize how the build will be generated. Options are boolean variables, with possible values ON/OFF. Options and variables are defined on the CMake command line like this:

```
$ cmake -DVARIABLE=value path/to/llvm/source
```

You can set a variable after the initial CMake invocation for changing its value. You can also undefine a variable:

```
$ cmake -UVARIABLE path/to/llvm/source
```

Variables are stored on the CMake cache. This is a file named `CMakeCache.txt` on the root of the build directory. Do not hand-edit it.

Variables are listed here appending its type after a colon. It is correct to write the variable and the type on the CMake command line:

```
$ cmake -DVARIABLE:TYPE=value path/to/llvm/source
```

Frequently-used CMake variables

Here are some of the CMake variables that are used often, along with a brief explanation and LLVM-specific notes. For full documentation, check the CMake docs or execute `cmake --help-variable VARIABLE_NAME`.

CMAKE_BUILD_TYPE:STRING Sets the build type for `make` based generators. Possible values are Release, Debug, RelWithDebInfo and MinSizeRel. On systems like Visual Studio the user sets the build type with the IDE settings.

CMAKE_INSTALL_PREFIX:PATH Path where LLVM will be installed if "make install" is invoked or the "INSTALL" target is built.

LLVM_LIBDIR_SUFFIX:STRING Extra suffix to append to the directory where libraries are to be installed. On a 64-bit architecture, one could use `-DLLVM_LIBDIR_SUFFIX=64` to install libraries to `/usr/lib64`.

CMAKE_C_FLAGS:STRING Extra flags to use when compiling C source files.

CMAKE_CXX_FLAGS:STRING Extra flags to use when compiling C++ source files.

BUILD_SHARED_LIBS:BOOL Flag indicating if shared libraries will be built. Its default value is OFF. Shared libraries are not supported on Windows and not recommended on the other OSes.

LLVM-specific variables

LLVM_TARGETS_TO_BUILD:STRING Semicolon-separated list of targets to build, or *all* for building all targets. Case-sensitive. Defaults to *all*. Example: `-DLLVM_TARGETS_TO_BUILD="X86;PowerPC"`.

LLVM_BUILD_TOOLS:BOOL Build LLVM tools. Defaults to ON. Targets for building each tool are generated in any case. You can build an tool separately by invoking its target. For example, you can build *llvm-as* with a makefile-based system executing *make llvm-as* on the root of your build directory.

LLVM_INCLUDE_TOOLS:BOOL Generate build targets for the LLVM tools. Defaults to ON. You can use that option for disabling the generation of build targets for the LLVM tools.

LLVM_BUILD_EXAMPLES:BOOL Build LLVM examples. Defaults to OFF. Targets for building each example are generated in any case. See documentation for *LLVM_BUILD_TOOLS* above for more details.

LLVM_INCLUDE_EXAMPLES:BOOL Generate build targets for the LLVM examples. Defaults to ON. You can use that option for disabling the generation of build targets for the LLVM examples.

LLVM_BUILD_TESTS:BOOL Build LLVM unit tests. Defaults to OFF. Targets for building each unit test are generated in any case. You can build a specific unit test with the target *UnitTestNameTests* (where at this time *UnitTestName* can be ADT, Analysis, ExecutionEngine, JIT, Support, Transform, VMCore; see the subdirectories of *unittests* for an updated list.) It is possible to build all unit tests with the target *UnitTests*.

LLVM_INCLUDE_TESTS:BOOL Generate build targets for the LLVM unit tests. Defaults to ON. You can use that option for disabling the generation of build targets for the LLVM unit tests.

LLVM_APPEND_VC_REV:BOOL Append version control revision info (svn revision number or Git revision id) to LLVM version string (stored in the PACKAGE_VERSION macro). For this to work cmake must be invoked before the build. Defaults to OFF.

LLVM_ENABLE_THREADS:BOOL Build with threads support, if available. Defaults to ON.

LLVM_ENABLE_CXX1Y:BOOL Build in C++1y mode, if available. Defaults to OFF.

LLVM_ENABLE_ASSERTIONS:BOOL Enables code assertions. Defaults to OFF if and only if `CMAKE_BUILD_TYPE` is *Release*.

LLVM_ENABLE_EH:BOOL Build LLVM with exception handling support. This is necessary if you wish to link against LLVM libraries and make use of C++ exceptions in your own code that need to propagate through LLVM code. Defaults to OFF.

LLVM_ENABLE_PIC:BOOL Add the `-fPIC` flag for the compiler command-line, if the compiler supports this flag. Some systems, like Windows, do not need this flag. Defaults to ON.

LLVM_ENABLE_RTTI:BOOL Build LLVM with run time type information. Defaults to OFF.

LLVM_ENABLE_WARNINGS:BOOL Enable all compiler warnings. Defaults to ON.

LLVM_ENABLE_PEDANTIC:BOOL Enable pedantic mode. This disable compiler specific extensions, is possible. Defaults to ON.

LLVM_ENABLE_WERROR:BOOL Stop and fail build, if a compiler warning is triggered. Defaults to OFF.

LLVM_BUILD_32_BITS:BOOL Build 32-bits executables and libraries on 64-bits systems. This option is available only on some 64-bits unix systems. Defaults to OFF.

LLVM_TARGET_ARCH:STRING LLVM target to use for native code generation. This is required for JIT generation. It defaults to "host", meaning that it shall pick the architecture of the machine where LLVM is being built. If you are cross-compiling, set it to the target architecture name.

LLVM_TABLEGEN:STRING Full path to a native TableGen executable (usually named `tblgen`). This is intended for cross-compiling: if the user sets this variable, no native TableGen will be created.

LLVM_LIT_ARGS:STRING Arguments given to lit. `make check` and `make clang-test` are affected. By default, `'-sv --no-progress-bar'` on Visual C++ and Xcode, `'-sv'` on others.

LLVM_LIT_TOOLS_DIR:PATH The path to GnuWin32 tools for tests. Valid on Windows host. Defaults to "", then Lit seeks tools according to %PATH%. Lit can find tools(eg. grep, sort, &c) on LLVM_LIT_TOOLS_DIR at first, without specifying GnuWin32 to %PATH%.

LLVM_ENABLE_FFI:BOOL Indicates whether LLVM Interpreter will be linked with Foreign Function Interface library. If the library or its headers are installed on a custom location, you can set the variables FFI_INCLUDE_DIR and FFI_LIBRARY_DIR. Defaults to OFF.

LLVM_EXTERNAL_{CLANG,LLD,POLLY}_SOURCE_DIR:PATH Path to `{Clang,lld,Polly}`'s source directory. Defaults to `tools/{clang,lld,polly}`. `{Clang,lld,Polly}` will not be built when it is empty or it does not point to a valid path.

LLVM_USE_OPROFILE:BOOL Enable building OProfile JIT support. Defaults to OFF

LLVM_USE_INTEL_JITEVENTS:BOOL Enable building support for Intel JIT Events API. Defaults to OFF

LLVM_ENABLE_ZLIB:BOOL Build with zlib to support compression/uncompression in LLVM tools. Defaults to ON.

LLVM_USE_SANITIZER:STRING Define the sanitizer used to build LLVM binaries and tests. Possible values are `Address`, `Memory`, `MemoryWithOrigins` and `Undefined`. Defaults to empty string.

LLVM_BUILD_DOCS:BOOL Enables all enabled documentation targets (i.e. Doxgyen and Sphinx targets) to be built as part of the normal build. If the `install` target is run then this also enables all built documentation targets to be installed. Defaults to OFF.

LLVM_ENABLE_DOXYGEN:BOOL Enables the generation of browsable HTML documentation using doxygen. Defaults to OFF.

LLVM_ENABLE_DOXYGEN_QT_HELP:BOOL Enables the generation of a Qt Compressed Help file. Defaults to OFF. This affects the make target `doxygen-llvm`. When enabled, apart from the normal HTML output generated by doxygen, this will produce a QCH file named `org.llvm.qch`. You can then load this file into Qt Creator. This option is only useful in combination with `-DLLVM_ENABLE_DOXYGEN=ON`; otherwise this has no effect.

LLVM_DOXYGEN_QCH_FILENAME:STRING The filename of the Qt Compressed Help file that will be genrated when `-DLLVM_ENABLE_DOXYGEN=ON` and `-DLLVM_ENABLE_DOXYGEN_QT_HELP=ON` are given. Defaults to `org.llvm.qch`. This option is only useful in combination with `-DLLVM_ENABLE_DOXYGEN_QT_HELP=ON`; otherwise this has no effect.

LLVM_DOXYGEN_QHP_NAMESPACE:STRING Namespace under which the intermediate Qt Help Project file lives. See Qt Help Project for more information. Defaults to "org.llvm". This option is only useful in combination with `-DLLVM_ENABLE_DOXYGEN_QT_HELP=ON`; otherwise this has no effect.

LLVM_DOXYGEN_QHP_CUST_FILTER_NAME:STRING See Qt Help Project for more information. Defaults to the CMake variable `${PACKAGE_STRING}` which is a combination of the package name and version string. This filter can then be used in Qt Creator to select only documentation from LLVM when browsing through all the help files that you might have loaded. This option is only useful in combination with `-DLLVM_ENABLE_DOXYGEN_QT_HELP=ON`; otherwise this has no effect.

LLVM_DOXYGEN_QHELPGENERATOR_PATH:STRING The path to the `qhelpgenerator` executable. Defaults to whatever CMake's `find_program()` can find. This option is only useful in combination with `-DLLVM_ENABLE_DOXYGEN_QT_HELP=ON`; otherwise this has no effect.

LLVM_ENABLE_SPHINX:BOOL If enabled CMake will search for the `sphinx-build` executable and will make the `SPHINX_OUTPUT_HTML` and `SPHINX_OUTPUT_MAN` CMake options available. Defaults to OFF.

SPHINX_EXECUTABLE:STRING The path to the `sphinx-build` executable detected by CMake.

SPHINX_OUTPUT_HTML:BOOL If enabled (and `LLVM_ENABLE_SPHINX` is enabled) then the targets for building the documentation as html are added (but not built by default unless `LLVM_BUILD_DOCS` is enabled). There is a target for each project in the source tree that uses sphinx (e.g. `docs-llvm-html`, `docs-clang-html` and `docs-lld-html`). Defaults to ON.

SPHINX_OUTPUT_MAN:BOOL If enabled (and `LLVM_ENABLE_SPHINX` is enabled) the targets for building the man pages are added (but not built by default unless `LLVM_BUILD_DOCS` is enabled). Currently the only target added is `docs-llvm-man`. Defaults to ON.

SPHINX_WARNINGS_AS_ERRORS:BOOL If enabled then sphinx documentation warnings will be treated as errors. Defaults to ON.

2.1.5 Executing the test suite

Testing is performed when the *check* target is built. For instance, if you are using makefiles, execute this command while on the top level of your build directory:

```
$ make check
```

On Visual Studio, you may run tests to build the project "check".

2.1.6 Cross compiling

See this wiki page for generic instructions on how to cross-compile with CMake. It goes into detailed explanations and may seem daunting, but it is not. On the wiki page there are several examples including toolchain files. Go directly to this section for a quick solution.

Also see the LLVM-specific variables section for variables used when cross-compiling.

2.1.7 Embedding LLVM in your project

From LLVM 3.5 onwards both the CMake and autoconf/Makefile build systems export LLVM libraries as importable CMake targets. This means that clients of LLVM can now reliably use CMake to develop their own LLVM based projects against an installed version of LLVM regardless of how it was built.

Here is a simple example of CMakeLists.txt file that imports the LLVM libraries and uses them to build a simple application `simple-tool`.

```
cmake_minimum_required(VERSION 2.8.8)
project(SimpleProject)

find_package(LLVM REQUIRED CONFIG)

message(STATUS "Found LLVM ${LLVM_PACKAGE_VERSION}")
message(STATUS "Using LLVMConfig.cmake in: ${LLVM_DIR}")

# Set your project compile flags.
# E.g. if using the C++ header files
# you will need to enable C++11 support
# for your compiler.

include_directories(${LLVM_INCLUDE_DIRS})
add_definitions(${LLVM_DEFINITIONS})

# Now build our tools
add_excutable(simple-tool tool.cpp)

# Find the libraries that correspond to the LLVM components
# that we wish to use
llvm_map_components_to_libnames(llvm_libs support core irreader)

# Link against LLVM libraries
target_link_libraries(simple-tool ${llvm_libs})
```

The `find_package(...)` directive when used in CONFIG mode (as in the above example) will look for the `LLVMConfig.cmake` file in various locations (see cmake manual for details). It creates a `LLVM_DIR` cache entry to save the directory where `LLVMConfig.cmake` is found or allows the user to specify the directory (e.g. by passing `-DLLVM_DIR=/usr/share/llvm/cmake` to the `cmake` command or by setting it directly in `ccmake` or `cmake-gui`).

This file is available in two different locations.

- `<INSTALL_PREFIX>/share/llvm/cmake/LLVMConfig.cmake` where `<INSTALL_PREFIX>` is the install prefix of an installed version of LLVM. On Linux typically this is `/usr/share/llvm/cmake/LLVMConfig.cmake`.
- `<LLVM_BUILD_ROOT>/share/llvm/cmake/LLVMConfig.cmake` where `<LLVM_BUILD_ROOT>` is the root of the LLVM build tree. **Note this only available when building LLVM with CMake**

If LLVM is installed in your operating system's normal installation prefix (e.g. on Linux this is usually `/usr/`) `find_package(LLVM ...)` will automatically find LLVM if it is installed correctly. If LLVM is not installed or you wish to build directly against the LLVM build tree you can use `LLVM_DIR` as previously mentioned.

The `LLVMConfig.cmake` file sets various useful variables. Notable variables include

LLVM_CMAKE_DIR The path to the LLVM CMake directory (i.e. the directory containing LLVMConfig.cmake).

LLVM_DEFINITIONS A list of preprocessor defines that should be used when building against LLVM.

LLVM_ENABLE_ASSERTIONS This is set to ON if LLVM was built with assertions, otherwise OFF.

LLVM_ENABLE_EH This is set to ON if LLVM was built with exception handling (EH) enabled, otherwise OFF.

LLVM_ENABLE_RTTI This is set to ON if LLVM was built with run time type information (RTTI), otherwise OFF.

LLVM_INCLUDE_DIRS A list of include paths to directories containing LLVM header files.

LLVM_PACKAGE_VERSION The LLVM version. This string can be used with CMake conditionals. E.g. `if (${LLVM_PACKAGE_VERSION} VERSION_LESS "3.5")`.

LLVM_TOOLS_BINARY_DIR The path to the directory containing the LLVM tools (e.g. `llvm-as`).

Notice that in the above example we link `simple-tool` against several LLVM libraries. The list of libraries is determined by using the `llvm_map_components_to_libnames()` CMake function. For a list of available components look at the output of running `llvm-config --components`.

Note that for LLVM < 3.5 `llvm_map_components_to_libraries()` was used instead of `llvm_map_components_to_libnames()`. This is now deprecated and will be removed in a future version of LLVM.

Developing LLVM passes out of source

It is possible to develop LLVM passes out of LLVM's source tree (i.e. against an installed or built LLVM). An example of a project layout is provided below.

```
<project dir>/
    |
    CMakeLists.txt
    <pass name>/
        |
        CMakeLists.txt
        Pass.cpp
        ...
```

Contents of `<project dir>/CMakeLists.txt`:

```
find_package(LLVM REQUIRED CONFIG)

add_definitions(${LLVM_DEFINITIONS})
include_directories(${LLVM_INCLUDE_DIRS})

add_subdirectory(<pass name>)
```

Contents of `<project dir>/<pass name>/CMakeLists.txt`:

```
add_library(LLVMPassname MODULE Pass.cpp)
```

Note if you intend for this pass to be merged into the LLVM source tree at some point in the future it might make more sense to use LLVM's internal add_llvm_loadable_module function instead by...

Adding the following to `<project dir>/CMakeLists.txt` (after `find_package(LLVM ...)`)

```
list(APPEND CMAKE_MODULE_PATH "${LLVM_CMAKE_DIR}")
include(AddLLVM)
```

And then changing `<project dir>/<pass name>/CMakeLists.txt` to

```
add_llvm_loadable_module(LLVMPassname
  Pass.cpp
  )
```

When you are done developing your pass, you may wish to integrate it into LLVM source tree. You can achieve it in two easy steps:

1. Copying `<pass name>` folder into `<LLVM root>/lib/Transform` directory.
2. Adding `add_subdirectory(<pass name>)` line into `<LLVM root>/lib/Transform/CMakeLists.txt`.

2.1.8 Compiler/Platform-specific topics

Notes for specific compilers and/or platforms.

Microsoft Visual C++

LLVM_COMPILER_JOBS:STRING Specifies the maximum number of parallell compiler jobs to use per project when building with msbuild or Visual Studio. Only supported for the Visual Studio 2010 CMake generator. 0 means use all processors. Default is 0.

2.2 How To Build On ARM

2.2.1 Introduction

This document contains information about building/testing LLVM and Clang on an ARM machine.

This document is *NOT* tailored to help you cross-compile LLVM/Clang to ARM on another architecture, for example an x86_64 machine. To find out more about cross-compiling, please check *How To Cross-Compile Clang/LLVM using Clang/LLVM*.

2.2.2 Notes On Building LLVM/Clang on ARM

Here are some notes on building/testing LLVM/Clang on ARM. Note that ARM encompasses a wide variety of CPUs; this advice is primarily based on the ARMv6 and ARMv7 architectures and may be inapplicable to older chips.

1. If you are building LLVM/Clang on an ARM board with 1G of memory or less, please use `gold` rather then GNU `ld`. Building LLVM/Clang with `--enable-optimized` is preferred since it consumes less memory. Otherwise, the building process will very likely fail due to insufficient memory. In any case it is probably a good idea to set up a swap partition.
2. If you want to run `make check-all` after building LLVM/Clang, to avoid false alarms (e.g., ARCMT failure) please use at least the following configuration:

   ```
   $ ../$LLVM_SRC_DIR/configure --with-abi=aapcs-vfp
   ```

3. The most popular Linaro/Ubuntu OS's for ARM boards, e.g., the Pandaboard, have become hard-float platforms. The following set of configuration options appears to be a good choice for this platform:

```
./configure --build=armv7l-unknown-linux-gnueabihf \
--host=armv7l-unknown-linux-gnueabihf \
--target=armv7l-unknown-linux-gnueabihf --with-cpu=cortex-a9 \
--with-float=hard --with-abi=aapcs-vfp --with-fpu=neon \
--enable-targets=arm --enable-optimized --enable-assertions
```

4. ARM development boards can be unstable and you may experience that cores are disappearing, caches being flushed on every big.LITTLE switch, and other similar issues. To help ease the effect of this, set the Linux scheduler to "performance" on **all** cores using this little script:

```
# The code below requires the package 'cpufrequtils' to be installed.
for ((cpu=0; cpu<`grep -c proc /proc/cpuinfo`; cpu++)); do
    sudo cpufreq-set -c $cpu -g performance
done
```

5. Running the build on SD cards is ok, but they are more prone to failures than good quality USB sticks, and those are more prone to failures than external hard-drives (those are also a lot faster). So, at least, you should consider to buy a fast USB stick. On systems with a fast eMMC, that's a good option too.
6. Make sure you have a decent power supply (dozens of dollars worth) that can provide *at least* 4 amperes, this is especially important if you use USB devices with your board.

2.3 How To Cross-Compile Clang/LLVM using Clang/LLVM

2.3.1 Introduction

This document contains information about building LLVM and Clang on host machine, targeting another platform.

For more information on how to use Clang as a cross-compiler, please check http://clang.llvm.org/docs/CrossCompilation.html.

TODO: Add MIPS and other platforms to this document.

2.3.2 Cross-Compiling from x86_64 to ARM

In this use case, we'll be using CMake and Ninja, on a Debian-based Linux system, cross-compiling from an x86_64 host (most Intel and AMD chips nowadays) to a hard-float ARM target (most ARM targets nowadays).

The packages you'll need are:

- `cmake`
- `ninja-build` (from backports in Ubuntu)
- `gcc-4.7-arm-linux-gnueabihf`
- `gcc-4.7-multilib-arm-linux-gnueabihf`
- `binutils-arm-linux-gnueabihf`
- `libgcc1-armhf-cross`
- `libsfgcc1-armhf-cross`
- `libstdc++6-armhf-cross`
- `libstdc++6-4.7-dev-armhf-cross`

Configuring CMake

For more information on how to configure CMake for LLVM/Clang, see *Building LLVM with CMake*.

The CMake options you need to add are:

- `-DCMAKE_CROSSCOMPILING=True`
- `-DCMAKE_INSTALL_PREFIX=<install-dir>`
- `-DLLVM_TABLEGEN=<path-to-host-bin>/llvm-tblgen`
- `-DCLANG_TABLEGEN=<path-to-host-bin>/clang-tblgen`
- `-DLLVM_DEFAULT_TARGET_TRIPLE=arm-linux-gnueabihf`
- `-DLLVM_TARGET_ARCH=ARM`
- `-DLLVM_TARGETS_TO_BUILD=ARM`
- `-DCMAKE_CXX_FLAGS='-target armv7a-linux-gnueabihf -mcpu=cortex-a9 -I/usr/arm-linux-gnueabihf/include/c++/4.7.2/arm-linux-gnueabihf/ -I/usr/arm-linux-gnueabihf/include/ -mfloat-abi=hard -ccc-gcc-name arm-linux-gnueabihf-gcc'`

The TableGen options are required to compile it with the host compiler, so you'll need to compile LLVM (or at least `llvm-tblgen`) to your host platform before you start. The CXX flags define the target, cpu (which defaults to `fpu=VFP3` with NEON), and forcing the hard-float ABI. If you're using Clang as a cross-compiler, you will *also* have to set `-ccc-gcc-name`, to make sure it picks the correct linker.

Most of the time, what you want is to have a native compiler to the platform itself, but not others. It might not even be feasible to produce x86 binaries from ARM targets, so there's no point in compiling all back-ends. For that reason, you should also set the `TARGETS_TO_BUILD` to only build the ARM back-end.

You must set the `CMAKE_INSTALL_PREFIX`, otherwise a `ninja install` will copy ARM binaries to your root filesystem, which is not what you want.

Hacks

There are some bugs in current LLVM, which require some fiddling before running CMake:

1. If you're using Clang as the cross-compiler, there is a problem in the LLVM ARM back-end that is producing absolute relocations on position-independent code (`R_ARM_THM_MOVW_ABS_NC`), so for now, you should disable PIC:

   ```
   -DLLVM_ENABLE_PIC=False
   ```

   This is not a problem, since Clang/LLVM libraries are statically linked anyway, it shouldn't affect much.

2. The ARM libraries won't be installed in your system, and possibly not easily installable anyway, so you'll have to build/download them separately. But the CMake prepare step, which checks for dependencies, will check the *host* libraries, not the *target* ones.

   A quick way of getting the libraries is to download them from a distribution repository, like Debian (http://packages.debian.org/wheezy/), and download the missing libraries. Note that the `libXXX` will have the shared objects (`.so`) and the `libXXX-dev` will give you the headers and the static (`.a`) library. Just in case, download both.

   The ones you need for ARM are: `libtinfo`, `zlib1g`, `libxml2` and `liblzma`. In the Debian repository you'll find downloads for all architectures.

After you download and unpack all `.deb` packages, copy all `.so` and `.a` to a directory, make the appropriate symbolic links (if necessary), and add the relevant `-L` and `-I` paths to `-DCMAKE_CXX_FLAGS` above.

Running CMake and Building

Finally, if you're using your platform compiler, run:

```
$ cmake -G Ninja <source-dir> <options above>
```

If you're using Clang as the cross-compiler, run:

```
$ CC='clang' CXX='clang++' cmake -G Ninja <source-dir> <options above>
```

If you have `clang`/`clang++` on the path, it should just work, and special Ninja files will be created in the build directory. I strongly suggest you to run `cmake` on a separate build directory, *not* inside the source tree.

To build, simply type:

```
$ ninja
```

It should automatically find out how many cores you have, what are the rules that needs building and will build the whole thing.

You can't run `ninja check-all` on this tree because the created binaries are targeted to ARM, not x86_64.

Installing and Using

After the LLVM/Clang has built successfully, you should install it via:

```
$ ninja install
```

which will create a sysroot on the install-dir. You can then tar that directory into a binary with the full triple name (for easy identification), like:

```
$ ln -sf <install-dir> arm-linux-gnueabihf-clang
$ tar zchf arm-linux-gnueabihf-clang.tar.gz arm-linux-gnueabihf-clang
```

If you copy that tarball to your target board, you'll be able to use it for running the test-suite, for example. Follow the guidelines at http://llvm.org/docs/lnt/quickstart.html, unpack the tarball in the test directory, and use options:

```
$ ./sandbox/bin/python sandbox/bin/lnt runtest nt \
    --sandbox sandbox \
    --test-suite `pwd`/test-suite \
    --cc `pwd`/arm-linux-gnueabihf-clang/bin/clang \
    --cxx `pwd`/arm-linux-gnueabihf-clang/bin/clang++
```

Remember to add the `-jN` options to `lnt` to the number of CPUs on your board. Also, the path to your clang has to be absolute, so you'll need the *pwd* trick above.

2.4 LLVM Command Guide

The following documents are command descriptions for all of the LLVM tools. These pages describe how to use the LLVM commands and what their options are. Note that these pages do not describe all of the options available for all tools. To get a complete listing, pass the `--help` (general options) or `--help-hidden` (general and debugging options) arguments to the tool you are interested in.

2.4.1 Basic Commands

llvm-as - LLVM assembler

SYNOPSIS

llvm-as [*options*] [*filename*]

DESCRIPTION

llvm-as is the LLVM assembler. It reads a file containing human-readable LLVM assembly language, translates it to LLVM bitcode, and writes the result into a file or to standard output.

If *filename* is omitted or is −, then **llvm-as** reads its input from standard input.

If an output file is not specified with the **-o** option, then **llvm-as** sends its output to a file or standard output by following these rules:

- If the input is standard input, then the output is standard output.
- If the input is a file that ends with `.ll`, then the output file is of the same name, except that the suffix is changed to `.bc`.
- If the input is a file that does not end with the `.ll` suffix, then the output file has the same name as the input file, except that the `.bc` suffix is appended.

OPTIONS

-f Enable binary output on terminals. Normally, **llvm-as** will refuse to write raw bitcode output if the output stream is a terminal. With this option, **llvm-as** will write raw bitcode regardless of the output device.

-help Print a summary of command line options.

-o *filename* Specify the output file name. If *filename* is −, then **llvm-as** sends its output to standard output.

EXIT STATUS

If **llvm-as** succeeds, it will exit with 0. Otherwise, if an error occurs, it will exit with a non-zero value.

SEE ALSO

llvm-dis|llvm-dis, gccas|gccas

llvm-dis - LLVM disassembler

SYNOPSIS

llvm-dis [*options*] [*filename*]

DESCRIPTION

The **llvm-dis** command is the LLVM disassembler. It takes an LLVM bitcode file and converts it into human-readable LLVM assembly language.

If filename is omitted or specified as −, **llvm-dis** reads its input from standard input.

If the input is being read from standard input, then **llvm-dis** will send its output to standard output by default. Otherwise, the output will be written to a file named after the input file, with a `.ll` suffix added (any existing `.bc` suffix will first be removed). You can override the choice of output file using the **-o** option.

OPTIONS

-f

Enable binary output on terminals. Normally, **llvm-dis** will refuse to write raw bitcode output if the output stream is a terminal. With this option, **llvm-dis** will write raw bitcode regardless of the output device.

-help

Print a summary of command line options.

-o *filename*

Specify the output file name. If *filename* is -, then the output is sent to standard output.

EXIT STATUS

If **llvm-dis** succeeds, it will exit with 0. Otherwise, if an error occurs, it will exit with a non-zero value.

SEE ALSO

llvm-as|llvm-as

opt - LLVM optimizer

SYNOPSIS

opt [*options*] [*filename*]

DESCRIPTION

The **opt** command is the modular LLVM optimizer and analyzer. It takes LLVM source files as input, runs the specified optimizations or analyses on it, and then outputs the optimized file or the analysis results. The function of **opt** depends on whether the `-analyze` option is given.

When `-analyze` is specified, **opt** performs various analyses of the input source. It will usually print the results on standard output, but in a few cases, it will print output to standard error or generate a file with the analysis output, which is usually done when the output is meant for another program.

While `-analyze` is *not* given, **opt** attempts to produce an optimized output file. The optimizations available via **opt** depend upon what libraries were linked into it as well as any additional libraries that have been loaded with the `-load` option. Use the `-help` option to determine what optimizations you can use.

If `filename` is omitted from the command line or is "-", **opt** reads its input from standard input. Inputs can be in either the LLVM assembly language format (`.ll`) or the LLVM bitcode format (`.bc`).

If an output filename is not specified with the *-o* option, **opt** writes its output to the standard output.

OPTIONS

-f
Enable binary output on terminals. Normally, **opt** will refuse to write raw bitcode output if the output stream is a terminal. With this option, **opt** will write raw bitcode regardless of the output device.

-help
Print a summary of command line options.

-o <filename>
Specify the output filename.

-S
Write output in LLVM intermediate language (instead of bitcode).

-{passname}
opt provides the ability to run any of LLVM's optimization or analysis passes in any order. The *-help* option lists all the passes available. The order in which the options occur on the command line are the order in which they are executed (within pass constraints).

-disable-inlining
This option simply removes the inlining pass from the standard list.

-disable-opt
This option is only meaningful when *-std-link-opts* is given. It disables most passes.

-strip-debug
This option causes opt to strip debug information from the module before applying other optimizations. It is essentially the same as *-strip* but it ensures that stripping of debug information is done first.

-verify-each
This option causes opt to add a verify pass after every pass otherwise specified on the command line (including *-verify*). This is useful for cases where it is suspected that a pass is creating an invalid module but it is not clear which pass is doing it.

-stats
Print statistics.

-time-passes
Record the amount of time needed for each pass and print it to standard error.

-debug
If this is a debug build, this option will enable debug printouts from passes which use the `DEBUG()` macro. See the LLVM Programmer's Manual, section `#DEBUG` for more information.

-load=<plugin>
Load the dynamic object `plugin`. This object should register new optimization or analysis passes. Once loaded, the object will add new command line options to enable various optimizations or analyses. To see the new complete list of optimizations, use the *-help* and *-load* options together. For example:

```
opt -load=plugin.so -help
```

-p
Print module after each transformation.

EXIT STATUS

If **opt** succeeds, it will exit with 0. Otherwise, if an error occurs, it will exit with a non-zero value.

llc - LLVM static compiler

SYNOPSIS

llc [*options*] [*filename*]

DESCRIPTION

The **llc** command compiles LLVM source inputs into assembly language for a specified architecture. The assembly language output can then be passed through a native assembler and linker to generate a native executable.

The choice of architecture for the output assembly code is automatically determined from the input file, unless the *-march* option is used to override the default.

OPTIONS

If `filename` is "-" or omitted, **llc** reads from standard input. Otherwise, it will from `filename`. Inputs can be in either the LLVM assembly language format (`.ll`) or the LLVM bitcode format (`.bc`).

If the *-o* option is omitted, then **llc** will send its output to standard output if the input is from standard input. If the *-o* option specifies "-", then the output will also be sent to standard output.

If no *-o* option is specified and an input file other than "-" is specified, then **llc** creates the output filename by taking the input filename, removing any existing `.bc` extension, and adding a `.s` suffix.

Other **llc** options are described below.

End-user Options

-help
: Print a summary of command line options.

-O=uint
: Generate code at different optimization levels. These correspond to the `-O0`, `-O1`, `-O2`, and `-O3` optimization levels used by **clang**.

-mtriple=<target triple>
: Override the target triple specified in the input file with the specified string.

-march=<arch>
: Specify the architecture for which to generate assembly, overriding the target encoded in the input file. See the output of `llc -help` for a list of valid architectures. By default this is inferred from the target triple or autodetected to the current architecture.

-mcpu=<cpuname>
: Specify a specific chip in the current architecture to generate code for. By default this is inferred from the target triple and autodetected to the current architecture. For a list of available CPUs, use:

```
llvm-as < /dev/null | llc -march=xyz -mcpu=help
```

-filetype=<output file type>
Specify what kind of output `llc` should generated. Options are: `asm` for textual assembly (`'.s'`), `obj` for native object files (`'.o'`) and `null` for not emitting anything (for performance testing).

Note that not all targets support all options.

-mattr=a1,+a2,-a3,...
Override or control specific attributes of the target, such as whether SIMD operations are enabled or not. The default set of attributes is set by the current CPU. For a list of available attributes, use:

```
llvm-as < /dev/null | llc -march=xyz -mattr=help
```

-disable-fp-elim
Disable frame pointer elimination optimization.

-disable-excess-fp-precision
Disable optimizations that may produce excess precision for floating point. Note that this option can dramatically slow down code on some systems (e.g. X86).

-enable-no-infs-fp-math
Enable optimizations that assume no Inf values.

-enable-no-nans-fp-math
Enable optimizations that assume no NAN values.

-enable-unsafe-fp-math
Enable optimizations that make unsafe assumptions about IEEE math (e.g. that addition is associative) or may not work for all input ranges. These optimizations allow the code generator to make use of some instructions which would otherwise not be usable (such as `fsin` on X86).

-stats
Print statistics recorded by code-generation passes.

-time-passes
Record the amount of time needed for each pass and print a report to standard error.

-load=<dso_path>
Dynamically load `dso_path` (a path to a dynamically shared object) that implements an LLVM target. This will permit the target name to be used with the *-march* option so that code can be generated for that target.

Tuning/Configuration Options

-print-machineinstrs
Print generated machine code between compilation phases (useful for debugging).

-regalloc=<allocator>
Specify the register allocator to use. Valid register allocators are:

basic

Basic register allocator.

fast

Fast register allocator. It is the default for unoptimized code.

greedy

Greedy register allocator. It is the default for optimized code.

pbqp

Register allocator based on 'Partitioned Boolean Quadratic Programming'.

-spiller=<spiller>
Specify the spiller to use for register allocators that support it. Currently this option is used only by the linear scan register allocator. The default `spiller` is *local*. Valid spillers are:

simple

Simple spiller

local

Local spiller

Intel IA-32-specific Options
-x86-asm-syntax=[att|intel]
Specify whether to emit assembly code in AT&T syntax (the default) or Intel syntax.

EXIT STATUS

If **llc** succeeds, it will exit with 0. Otherwise, if an error occurs, it will exit with a non-zero value.

SEE ALSO

lli

lli - directly execute programs from LLVM bitcode

SYNOPSIS

lli [*options*] [*filename*] [*program args*]

DESCRIPTION

lli directly executes programs in LLVM bitcode format. It takes a program in LLVM bitcode format and executes it using a just-in-time compiler, if one is available for the current architecture, or an interpreter. **lli** takes all of the same code generator options as llc|llc, but they are only effective when **lli** is using the just-in-time compiler.

If *filename* is not specified, then **lli** reads the LLVM bitcode for the program from standard input.

The optional *args* specified on the command line are passed to the program as arguments.

GENERAL OPTIONS

-fake-argv0=*executable*

Override the `argv[0]` value passed into the executing program.

-force-interpreter=*{false,true}*

If set to true, use the interpreter even if a just-in-time compiler is available for this architecture. Defaults to false.

-help

Print a summary of command line options.

-load=*pluginfilename*

Causes **lli** to load the plugin (shared object) named *pluginfilename* and use it for optimization.

-stats

Print statistics from the code-generation passes. This is only meaningful for the just-in-time compiler, at present.

-time-passes

Record the amount of time needed for each code-generation pass and print it to standard error.

-version

Print out the version of **lli** and exit without doing anything else.

TARGET OPTIONS

-mtriple=*target triple*

Override the target triple specified in the input bitcode file with the specified string. This may result in a crash if you pick an architecture which is not compatible with the current system.

-march=*arch*

Specify the architecture for which to generate assembly, overriding the target encoded in the bitcode file. See the output of **llc -help** for a list of valid architectures. By default this is inferred from the target triple or autodetected to the current architecture.

-mcpu=*cpuname*

Specify a specific chip in the current architecture to generate code for. By default this is inferred from the target triple and autodetected to the current architecture. For a list of available CPUs, use: **llvm-as < /dev/null | llc -march=xyz -mcpu=help**

-mattr=*a1,+a2,-a3,...*

Override or control specific attributes of the target, such as whether SIMD operations are enabled or not. The default set of attributes is set by the current CPU. For a list of available attributes, use: **llvm-as < /dev/null | llc -march=xyz -mattr=help**

FLOATING POINT OPTIONS

-disable-excess-fp-precision

Disable optimizations that may increase floating point precision.

-enable-no-infs-fp-math

Enable optimizations that assume no Inf values.

-enable-no-nans-fp-math

Enable optimizations that assume no NAN values.

-enable-unsafe-fp-math

Causes **lli** to enable optimizations that may decrease floating point precision.

-soft-float

Causes **lli** to generate software floating point library calls instead of equivalent hardware instructions.

CODE GENERATION OPTIONS

-code-model=*model*

Choose the code model from:

```
default: Target default code model
small: Small code model
kernel: Kernel code model
medium: Medium code model
large: Large code model
```

-disable-post-RA-scheduler

Disable scheduling after register allocation.

-disable-spill-fusing

Disable fusing of spill code into instructions.

-jit-enable-eh

Exception handling should be enabled in the just-in-time compiler.

-join-liveintervals

Coalesce copies (default=true).

-nozero-initialized-in-bss Don't place zero-initialized symbols into the BSS section.

-pre-RA-sched=*scheduler*

Instruction schedulers available (before register allocation):

```
=default: Best scheduler for the target
=none: No scheduling: breadth first sequencing
=simple: Simple two pass scheduling: minimize critical path and maximize processor utilization
=simple-noitin: Simple two pass scheduling: Same as simple except using generic latency
=list-burr: Bottom-up register reduction list scheduling
=list-tdrr: Top-down register reduction list scheduling
=list-td: Top-down list scheduler -print-machineinstrs - Print generated machine code
```

-regalloc=*allocator*

Register allocator to use (default=linearscan)

```
=bigblock: Big-block register allocator
=linearscan: linear scan register allocator =local -   local register allocator
=simple: simple register allocator
```

-relocation-model=*model*

Choose relocation model from:

```
=default: Target default relocation model
=static: Non-relocatable code =pic -   Fully relocatable, position independent code
=dynamic-no-pic: Relocatable external references, non-relocatable code
```

-spiller

Spiller to use (default=local)

```
=simple: simple spiller
=local: local spiller
```

-x86-asm-syntax=*syntax*

Choose style of code to emit from X86 backend:

```
=att: Emit AT&T-style assembly
=intel: Emit Intel-style assembly
```

EXIT STATUS

If **lli** fails to load the program, it will exit with an exit code of 1. Otherwise, it will return the exit code of the program it executes.

SEE ALSO

llc|llc

llvm-link - LLVM bitcode linker

SYNOPSIS

llvm-link [*options*] *filename ...*

DESCRIPTION

llvm-link takes several LLVM bitcode files and links them together into a single LLVM bitcode file. It writes the output file to standard output, unless the `-o` option is used to specify a filename.

OPTIONS

-f
Enable binary output on terminals. Normally, **llvm-link** will refuse to write raw bitcode output if the output stream is a terminal. With this option, **llvm-link** will write raw bitcode regardless of the output device.

-o `filename`
Specify the output file name. If `filename` is "-", then **llvm-link** will write its output to standard output.

-S
Write output in LLVM intermediate language (instead of bitcode).

-d
If specified, **llvm-link** prints a human-readable version of the output bitcode file to standard error.

-help
Print a summary of command line options.

-v
Verbose mode. Print information about what **llvm-link** is doing. This typically includes a message for each bitcode file linked in and for each library found.

EXIT STATUS

If **llvm-link** succeeds, it will exit with 0. Otherwise, if an error occurs, it will exit with a non-zero value.

llvm-ar - LLVM archiver

SYNOPSIS

llvm-ar [-]{dmpqrtx}[Rabfikou] [relpos] [count] <archive> [files...]

DESCRIPTION

The **llvm-ar** command is similar to the common Unix utility, `ar`. It archives several files together into a single file. The intent for this is to produce archive libraries by LLVM bitcode that can be linked into an LLVM program. However, the archive can contain any kind of file. By default, **llvm-ar** generates a symbol table that makes linking faster because only the symbol table needs to be consulted, not each individual file member of the archive.

The **llvm-ar** command can be used to *read* SVR4, GNU and BSD style archive files. However, right now it can only write in the GNU format. If an SVR4 or BSD style archive is used with the `r` (replace) or `q` (quick update) operations, the archive will be reconstructed in GNU format.

Here's where **llvm-ar** departs from previous `ar` implementations:

Symbol Table

> Since **llvm-ar** supports bitcode files. The symbol table it creates is in GNU format and includes both native and bitcode files.

Long Paths

> Currently **llvm-ar** can read GNU and BSD long file names, but only writes archives with the GNU format.

OPTIONS

The options to **llvm-ar** are compatible with other `ar` implementations. However, there are a few modifiers (*R*) that are not found in other `ar` implementations. The options to **llvm-ar** specify a single basic operation to perform on the archive, a variety of modifiers for that operation, the name of the archive file, and an optional list of file names. These options are used to determine how **llvm-ar** should process the archive file.

The Operations and Modifiers are explained in the sections below. The minimal set of options is at least one operator and the name of the archive. Typically archive files end with a `.a` suffix, but this is not required. Following the *archive-name* comes a list of *files* that indicate the specific members of the archive to operate on. If the *files* option is not specified, it generally means either "none" or "all" members, depending on the operation.

Operations d

> Delete files from the archive. No modifiers are applicable to this operation. The *files* options specify which members should be removed from the archive. It is not an error if a specified file does not appear in the archive. If no *files* are specified, the archive is not modified.

m[abi]

> Move files from one location in the archive to another. The *a*, *b*, and *i* modifiers apply to this operation. The *files* will all be moved to the location given by the modifiers. If no modifiers are used, the files will be moved to the end of the archive. If no *files* are specified, the archive is not modified.

p

> Print files to the standard output. This operation simply prints the *files* indicated to the standard output. If no *files* are specified, the entire archive is printed. Printing bitcode files is ill-advised as they might confuse your terminal settings. The *p* operation never modifies the archive.

q

Quickly append files to the end of the archive. This operation quickly adds the *files* to the archive without checking for duplicates that should be removed first. If no *files* are specified, the archive is not modified. Because of the way that **llvm-ar** constructs the archive file, its dubious whether the *q* operation is any faster than the *r* operation.

r[abu]

Replace or insert file members. The *a*, *b*, and *u* modifiers apply to this operation. This operation will replace existing *files* or insert them at the end of the archive if they do not exist. If no *files* are specified, the archive is not modified.

t[v]

Print the table of contents. Without any modifiers, this operation just prints the names of the members to the standard output. With the *v* modifier, **llvm-ar** also prints out the file type (B=bitcode, S=symbol table, blank=regular file), the permission mode, the owner and group, the size, and the date. If any *files* are specified, the listing is only for those files. If no *files* are specified, the table of contents for the whole archive is printed.

x[oP]

Extract archive members back to files. The *o* modifier applies to this operation. This operation retrieves the indicated *files* from the archive and writes them back to the operating system's file system. If no *files* are specified, the entire archive is extract.

Modifiers (operation specific) The modifiers below are specific to certain operations. See the Operations section (above) to determine which modifiers are applicable to which operations.

[a]

When inserting or moving member files, this option specifies the destination of the new files as being after the *relpos* member. If *relpos* is not found, the files are placed at the end of the archive.

[b]

When inserting or moving member files, this option specifies the destination of the new files as being before the *relpos* member. If *relpos* is not found, the files are placed at the end of the archive. This modifier is identical to the *i* modifier.

[i]

A synonym for the *b* option.

[o]

When extracting files, this option will cause **llvm-ar** to preserve the original modification times of the files it writes.

[u]

When replacing existing files in the archive, only replace those files that have a time stamp than the time stamp of the member in the archive.

Modifiers (generic) The modifiers below may be applied to any operation.

[c]

For all operations, **llvm-ar** will always create the archive if it doesn't exist. Normally, **llvm-ar** will print a warning message indicating that the archive is being created. Using this modifier turns off that warning.

[s]

This modifier requests that an archive index (or symbol table) be added to the archive. This is the default mode of operation. The symbol table will contain all the externally visible functions and global variables defined by all the bitcode files in the archive.

[S]

This modifier is the opposite of the *s* modifier. It instructs **llvm-ar** to not build the symbol table. If both *s* and *S* are used, the last modifier to occur in the options will prevail.

[v]

This modifier instructs **llvm-ar** to be verbose about what it is doing. Each editing operation taken against the archive will produce a line of output saying what is being done.

STANDARDS

The **llvm-ar** utility is intended to provide a superset of the IEEE Std 1003.2 (POSIX.2) functionality for `ar`. **llvm-ar** can read both SVR4 and BSD4.4 (or Mac OS X) archives. If the `f` modifier is given to the `x` or `r` operations then **llvm-ar** will write SVR4 compatible archives. Without this modifier, **llvm-ar** will write BSD4.4 compatible archives that have long names immediately after the header and indicated using the "#1/ddd" notation for the name in the header.

FILE FORMAT

The file format for LLVM Archive files is similar to that of BSD 4.4 or Mac OSX archive files. In fact, except for the symbol table, the `ar` commands on those operating systems should be able to read LLVM archive files. The details of the file format follow.

Each archive begins with the archive magic number which is the eight printable characters "!<arch>n" where n represents the newline character (0x0A). Following the magic number, the file is composed of even length members that begin with an archive header and end with a n padding character if necessary (to make the length even). Each file member is composed of a header (defined below), an optional newline-terminated "long file name" and the contents of the file.

The fields of the header are described in the items below. All fields of the header contain only ASCII characters, are left justified and are right padded with space characters.

name - char[16]

This field of the header provides the name of the archive member. If the name is longer than 15 characters or contains a slash (/) character, then this field contains `#1/nnn` where `nnn` provides the length of the name and the `#1/` is literal. In this case, the actual name of the file is provided in the `nnn` bytes immediately following the header. If the name is 15 characters or less, it is contained directly in this field and terminated with a slash (/) character.

date - char[12]

This field provides the date of modification of the file in the form of a decimal encoded number that provides the number of seconds since the epoch (since 00:00:00 Jan 1, 1970) per Posix specifications.

uid - char[6]

This field provides the user id of the file encoded as a decimal ASCII string. This field might not make much sense on non-Unix systems. On Unix, it is the same value as the st_uid field of the stat structure returned by the stat(2) operating system call.

gid - char[6]

This field provides the group id of the file encoded as a decimal ASCII string. This field might not make much sense on non-Unix systems. On Unix, it is the same value as the st_gid field of the stat structure returned by the stat(2) operating system call.

mode - char[8]

This field provides the access mode of the file encoded as an octal ASCII string. This field might not make much sense on non-Unix systems. On Unix, it is the same value as the st_mode field of the stat structure returned by the stat(2) operating system call.

size - char[10]

This field provides the size of the file, in bytes, encoded as a decimal ASCII string.

fmag - char[2]

This field is the archive file member magic number. Its content is always the two characters back tick (0x60) and newline (0x0A). This provides some measure utility in identifying archive files that have been corrupted.

offset - vbr encoded 32-bit integer

The offset item provides the offset into the archive file where the bitcode member is stored that is associated with the symbol. The offset value is 0 based at the start of the first "normal" file member. To derive the actual file offset of the member, you must add the number of bytes occupied by the file signature (8 bytes) and the symbol tables. The value of this item is encoded using variable bit rate encoding to reduce the size of the symbol table. Variable bit rate encoding uses the high bit (0x80) of each byte to indicate if there are more bytes to follow. The remaining 7 bits in each byte carry bits from the value. The final byte does not have the high bit set.

length - vbr encoded 32-bit integer

The length item provides the length of the symbol that follows. Like this *offset* item, the length is variable bit rate encoded.

symbol - character array

The symbol item provides the text of the symbol that is associated with the *offset*. The symbol is not terminated by any character. Its length is provided by the *length* field. Note that is allowed (but unwise) to use non-printing characters (even 0x00) in the symbol. This allows for multiple encodings of symbol names.

EXIT STATUS

If **llvm-ar** succeeds, it will exit with 0. A usage error, results in an exit code of 1. A hard (file system typically) error results in an exit code of 2. Miscellaneous or unknown errors result in an exit code of 3.

SEE ALSO

ar(1)

llvm-nm - list LLVM bitcode and object file's symbol table

SYNOPSIS

llvm-nm [*options*] [*filenames...*]

DESCRIPTION

The **llvm-nm** utility lists the names of symbols from the LLVM bitcode files, object files, or **ar** archives containing them, named on the command line. Each symbol is listed along with some simple information about its provenance. If no file name is specified, or - is used as a file name, **llvm-nm** will process a file on its standard input stream.

llvm-nm's default output format is the traditional BSD **nm** output format. Each such output record consists of an (optional) 8-digit hexadecimal address, followed by a type code character, followed by a name, for each symbol. One record is printed per line; fields are separated by spaces. When the address is omitted, it is replaced by 8 spaces.

Type code characters currently supported, and their meanings, are as follows:

U
: Named object is referenced but undefined in this bitcode file

C
: Common (multiple definitions link together into one def)

W
: Weak reference (multiple definitions link together into zero or one definitions)

t
: Local function (text) object

T
: Global function (text) object

d
: Local data object

D
: Global data object

?
: Something unrecognizable

Because LLVM bitcode files typically contain objects that are not considered to have addresses until they are linked into an executable image or dynamically compiled "just-in-time", **llvm-nm** does not print an address for any symbol in an LLVM bitcode file, even symbols which are defined in the bitcode file.

OPTIONS

-B (default)
: Use BSD output format. Alias for *--format=bsd*.

-P
: Use POSIX.2 output format. Alias for *--format=posix*.

-debug-syms, -a
: Show all symbols, even debugger only.

-defined-only
: Print only symbols defined in this file (as opposed to symbols which may be referenced by objects in this file, but not defined in this file.)

-dynamic, -D
Display dynamic symbols instead of normal symbols.

-extern-only, -g
Print only symbols whose definitions are external; that is, accessible from other files.

-format=format, **-f** format
Select an output format; *format* may be *sysv*, *posix*, or *bsd*. The default is *bsd*.

-help
Print a summary of command-line options and their meanings.

-no-sort, -p
Shows symbols in order encountered.

-numeric-sort, -n, -v
Sort symbols by address.

-print-file-name, -A, -o
Precede each symbol with the file it came from.

-print-size, -S
Show symbol size instead of address.

-size-sort
Sort symbols by size.

-undefined-only, -u
Print only symbols referenced but not defined in this file.

BUGS

- **llvm-nm** cannot demangle C++ mangled names, like GNU **nm** can.
- **llvm-nm** does not support the full set of arguments that GNU **nm** does.

EXIT STATUS

llvm-nm exits with an exit code of zero.

SEE ALSO

llvm-dis, ar(1), nm(1)

llvm-config - Print LLVM compilation options

SYNOPSIS

llvm-config *option* [*components*...]

DESCRIPTION

llvm-config makes it easier to build applications that use LLVM. It can print the compiler flags, linker flags and object libraries needed to link against LLVM.

EXAMPLES

To link against the JIT:

```
g++ `llvm-config --cxxflags` -o HowToUseJIT.o -c HowToUseJIT.cpp
g++ `llvm-config --ldflags` -o HowToUseJIT HowToUseJIT.o \
    `llvm-config --libs engine bcreader scalaropts`
```

OPTIONS

–version

Print the version number of LLVM.

-help

Print a summary of **llvm-config** arguments.

–prefix

Print the installation prefix for LLVM.

–src-root

Print the source root from which LLVM was built.

–obj-root

Print the object root used to build LLVM.

–bindir

Print the installation directory for LLVM binaries.

–includedir

Print the installation directory for LLVM headers.

–libdir

Print the installation directory for LLVM libraries.

–cxxflags

Print the C++ compiler flags needed to use LLVM headers.

–ldflags

Print the flags needed to link against LLVM libraries.

–libs

Print all the libraries needed to link against the specified LLVM *components*, including any dependencies.

–libnames

Similar to **–libs**, but prints the bare filenames of the libraries without **-l** or pathnames. Useful for linking against a not-yet-installed copy of LLVM.

–libfiles

Similar to **–libs**, but print the full path to each library file. This is useful when creating makefile dependencies, to ensure that a tool is relinked if any library it uses changes.

–components

Print all valid component names.

–targets-built

Print the component names for all targets supported by this copy of LLVM.

–build-mode

Print the build mode used when LLVM was built (e.g. Debug or Release)

COMPONENTS

To print a list of all available components, run **llvm-config –components**. In most cases, components correspond directly to LLVM libraries. Useful "virtual" components include:

all

Includes all LLVM libraries. The default if no components are specified.

backend

Includes either a native backend or the C backend.

engine

Includes either a native JIT or the bitcode interpreter.

EXIT STATUS

If **llvm-config** succeeds, it will exit with 0. Otherwise, if an error occurs, it will exit with a non-zero value.

llvm-diff - LLVM structural 'diff'

SYNOPSIS

llvm-diff [*options*] *module 1 module 2* [*global name ...*]

DESCRIPTION

llvm-diff compares the structure of two LLVM modules, primarily focusing on differences in function definitions. Insignificant differences, such as changes in the ordering of globals or in the names of local values, are ignored.

An input module will be interpreted as an assembly file if its name ends in '.ll'; otherwise it will be read in as a bitcode file.

If a list of global names is given, just the values with those names are compared; otherwise, all global values are compared, and diagnostics are produced for globals which only appear in one module or the other.

llvm-diff compares two functions by comparing their basic blocks, beginning with the entry blocks. If the terminators seem to match, then the corresponding successors are compared; otherwise they are ignored. This algorithm is very sensitive to changes in control flow, which tend to stop any downstream changes from being detected.

llvm-diff is intended as a debugging tool for writers of LLVM passes and frontends. It does not have a stable output format.

EXIT STATUS

If **llvm-diff** finds no differences between the modules, it will exit with 0 and produce no output. Otherwise it will exit with a non-zero value.

BUGS

Many important differences, like changes in linkage or function attributes, are not diagnosed.

Changes in memory behavior (for example, coalescing loads) can cause massive detected differences in blocks.

llvm-cov - emit coverage information

SYNOPSIS

llvm-cov [options] SOURCEFILE

DESCRIPTION

The **llvm-cov** tool reads code coverage data files and displays the coverage information for a specified source file. It is compatible with the `gcov` tool from version 4.2 of `GCC` and may also be compatible with some later versions of `gcov`.

To use llvm-cov, you must first build an instrumented version of your application that collects coverage data as it runs. Compile with the `-fprofile-arcs` and `-ftest-coverage` options to add the instrumentation. (Alternatively, you can use the `--coverage` option, which includes both of those other options.) You should compile with debugging information (`-g`) and without optimization (`-O0`); otherwise, the coverage data cannot be accurately mapped back to the source code.

At the time you compile the instrumented code, a `.gcno` data file will be generated for each object file. These `.gcno` files contain half of the coverage data. The other half of the data comes from `.gcda` files that are generated when you run the instrumented program, with a separate `.gcda` file for each object file. Each time you run the program, the execution counts are summed into any existing `.gcda` files, so be sure to remove any old files if you do not want their contents to be included.

By default, the `.gcda` files are written into the same directory as the object files, but you can override that by setting the `GCOV_PREFIX` and `GCOV_PREFIX_STRIP` environment variables. The `GCOV_PREFIX_STRIP` variable specifies a number of directory components to be removed from the start of the absolute path to the object file directory. After stripping those directories, the prefix from the `GCOV_PREFIX` variable is added. These environment variables allow you to run the instrumented program on a machine where the original object file directories are not accessible, but you will then need to copy the `.gcda` files back to the object file directories where llvm-cov expects to find them.

Once you have generated the coverage data files, run llvm-cov for each main source file where you want to examine the coverage results. This should be run from the same directory where you previously ran the compiler. The results for the specified source file are written to a file named by appending a `.gcov` suffix. A separate output file is also created for each file included by the main source file, also with a `.gcov` suffix added.

The basic content of an llvm-cov output file is a copy of the source file with an execution count and line number prepended to every line. The execution count is shown as `-` if a line does not contain any executable code. If a line contains code but that code was never executed, the count is displayed as `#####`.

OPTIONS

-a, -all-blocks
Display all basic blocks. If there are multiple blocks for a single line of source code, this option causes llvm-cov to show the count for each block instead of just one count for the entire line.

-b, -branch-probabilities
Display conditional branch probabilities and a summary of branch information.

-c, -branch-counts
Display branch counts instead of probabilities (requires -b).

-f, -function-summaries
Show a summary of coverage for each function instead of just one summary for an entire source file.

-help
Display available options (–help-hidden for more).

-l, -long-file-names
For coverage output of files included from the main source file, add the main file name followed by `##` as a prefix to the output file names. This can be combined with the –preserve-paths option to use complete paths for both the main file and the included file.

-n, -no-output
Do not output any `.gcov` files. Summary information is still displayed.

-o=<DIR|FILE>, **-object-directory**=<DIR>, **-object-file**=<FILE>
Find objects in DIR or based on FILE's path. If you specify a particular object file, the coverage data files are expected to have the same base name with `.gcno` and `.gcda` extensions. If you specify a directory, the files are expected in that directory with the same base name as the source file.

-p, -preserve-paths
Preserve path components when naming the coverage output files. In addition to the source file name, include the directories from the path to that file. The directories are separate by `#` characters, with `.` directories removed and `..` directories replaced by `^` characters. When used with the –long-file-names option, this applies to both the main file name and the included file name.

-u, -unconditional-branches
Include unconditional branches in the output for the –branch-probabilities option.

-version
Display the version of llvm-cov.

EXIT STATUS

llvm-cov returns 1 if it cannot read input files. Otherwise, it exits with zero.

llvm-profdata - Profile data tool

SYNOPSIS

llvm-profdata *command* [*args...*]

DESCRIPTION

The **llvm-profdata** tool is a small utility for working with profile data files.

COMMANDS

- merge
- show

MERGE

SYNOPSIS **llvm-profdata merge** [*options*] [*filenames...*]

DESCRIPTION **llvm-profdata merge** takes several profile data files generated by PGO instrumentation and merges them together into a single indexed profile data file.

OPTIONS

-help
Print a summary of command line options.

-output=output, **-o**=output
Specify the output file name. *Output* cannot be − as the resulting indexed profile data can't be written to standard output.

SHOW

SYNOPSIS **llvm-profdata show** [*options*] [*filename*]

DESCRIPTION **llvm-profdata show** takes a profile data file and displays the information about the profile counters for this file and for any of the specified function(s).

If *filename* is omitted or is −, then **llvm-profdata show** reads its input from standard input.

OPTIONS

-all-functions
Print details for every function.

-counts
Print the counter values for the displayed functions.

-function=string
Print details for a function if the function's name contains the given string.

-help
Print a summary of command line options.

-output=output, **-o**=output
Specify the output file name. If *output* is − or it isn't specified, then the output is sent to standard output.

EXIT STATUS

llvm-profdata returns 1 if the command is omitted or is invalid, if it cannot read input files, or if there is a mismatch between their data.

llvm-stress - generate random .ll files

SYNOPSIS

llvm-stress [-size=filesize] [-seed=initialseed] [-o=outfile]

DESCRIPTION

The **llvm-stress** tool is used to generate random `.ll` files that can be used to test different components of LLVM.

OPTIONS

-o `filename`
: Specify the output filename.

-size `size`
: Specify the size of the generated `.ll` file.

-seed `seed`
: Specify the seed to be used for the randomly generated instructions.

EXIT STATUS

llvm-stress returns 0.

llvm-symbolizer - convert addresses into source code locations

SYNOPSIS

llvm-symbolizer [options]

DESCRIPTION

llvm-symbolizer reads object file names and addresses from standard input and prints corresponding source code locations to standard output. If object file is specified in command line, **llvm-symbolizer** reads only addresses from standard input. This program uses debug info sections and symbol table in the object files.

EXAMPLE

```
$ cat addr.txt
a.out 0x4004f4
/tmp/b.out 0x400528
/tmp/c.so 0x710
/tmp/mach_universal_binary:i386 0x1f84
/tmp/mach_universal_binary:x86_64 0x100000f24
$ llvm-symbolizer < addr.txt
main
/tmp/a.cc:4
```

```
f(int, int)
/tmp/b.cc:11

h_inlined_into_g
/tmp/header.h:2
g_inlined_into_f
/tmp/header.h:7
f_inlined_into_main
/tmp/source.cc:3
main
/tmp/source.cc:8

_main
/tmp/source_i386.cc:8

_main
/tmp/source_x86_64.cc:8
$ cat addr2.txt
0x4004f4
0x401000
$ llvm-symbolizer -obj=a.out < addr2.txt
main
/tmp/a.cc:4

foo(int)
/tmp/a.cc:12
```

OPTIONS

-obj
Path to object file to be symbolized.

-functions=[none|short|linkage]
Specify the way function names are printed (omit function name, print short function name, or print full linkage name, respectively). Defaults to `linkage`.

-use-symbol-table
Prefer function names stored in symbol table to function names in debug info sections. Defaults to true.

-demangle
Print demangled function names. Defaults to true.

-inlining
If a source code location is in an inlined function, prints all the inlnied frames. Defaults to true.

-default-arch
If a binary contains object files for multiple architectures (e.g. it is a Mach-O universal binary), symbolize the object file for a given architecture. You can also specify architecture by writing `binary_name:arch_name` in the input (see example above). If architecture is not specified in either way, address will not be symbolized. Defaults to empty string.

-dsym-hint=<path/to/file.dSYM>
(Darwin-only flag). If the debug info for a binary isn't present in the default location, look for the debug info at the .dSYM path provided via the `-dsym-hint` flag. This flag can be used multiple times.

EXIT STATUS

llvm-symbolizer returns 0. Other exit codes imply internal program error.

llvm-dwarfdump - print contents of DWARF sections

SYNOPSIS

llvm-dwarfdump [*options*] [*filenames...*]

DESCRIPTION

llvm-dwarfdump parses DWARF sections in the object files and prints their contents in human-readable form.

OPTIONS

-debug-dump=section
: Specify the DWARF section to dump. For example, use `abbrev` to dump the contents of `.debug_abbrev` section, `loc.dwo` to dump the contents of `.debug_loc.dwo` etc. See `llvm-dwarfdump --help` for the complete list of supported sections. Use `all` to dump all DWARF sections. It is the default.

EXIT STATUS

llvm-dwarfdump returns 0. Other exit codes imply internal program error.

2.4.2 Debugging Tools

bugpoint - automatic test case reduction tool

SYNOPSIS

bugpoint [*options*] [*input LLVM ll/bc files*] [*LLVM passes*] **–args** *program arguments*

DESCRIPTION

bugpoint narrows down the source of problems in LLVM tools and passes. It can be used to debug three types of failures: optimizer crashes, miscompilations by optimizers, or bad native code generation (including problems in the static and JIT compilers). It aims to reduce large test cases to small, useful ones. For more information on the design and inner workings of **bugpoint**, as well as advice for using bugpoint, see *llvm/docs/Bugpoint.html* in the LLVM distribution.

OPTIONS

–additional-so *library*

Load the dynamic shared object *library* into the test program whenever it is run. This is useful if you are debugging programs which depend on non-LLVM libraries (such as the X or curses libraries) to run.

–append-exit-code=*{true,false}*

Append the test programs exit code to the output file so that a change in exit code is considered a test failure. Defaults to false.

–args *program args*

Pass all arguments specified after **–args** to the test program whenever it runs. Note that if any of the *program args* start with a "–", you should use:

```
bugpoint [bugpoint args] --args -- [program args]
```

The "––" right after the **–args** option tells **bugpoint** to consider any options starting with "–" to be part of the **–args** option, not as options to **bugpoint** itself.

–tool-args *tool args*

Pass all arguments specified after **–tool-args** to the LLVM tool under test (**llc**, **lli**, etc.) whenever it runs. You should use this option in the following way:

```
bugpoint [bugpoint args] --tool-args -- [tool args]
```

The "––" right after the **–tool-args** option tells **bugpoint** to consider any options starting with "–" to be part of the **–tool-args** option, not as options to **bugpoint** itself. (See **–args**, above.)

–safe-tool-args *tool args*

Pass all arguments specified after **–safe-tool-args** to the "safe" execution tool.

–gcc-tool-args *gcc tool args*

Pass all arguments specified after **–gcc-tool-args** to the invocation of **gcc**.

–opt-args *opt args*

Pass all arguments specified after **–opt-args** to the invocation of **opt**.

–disable-{dce,simplifycfg}

Do not run the specified passes to clean up and reduce the size of the test program. By default, **bugpoint** uses these passes internally when attempting to reduce test programs. If you're trying to find a bug in one of these passes, **bugpoint** may crash.

–enable-valgrind

Use valgrind to find faults in the optimization phase. This will allow bugpoint to find otherwise asymptomatic problems caused by memory mis-management.

-find-bugs

Continually randomize the specified passes and run them on the test program until a bug is found or the user kills **bugpoint**.

-help

Print a summary of command line options.

–input *filename*

Open *filename* and redirect the standard input of the test program, whenever it runs, to come from that file.

–load *plugin*

Load the dynamic object *plugin* into **bugpoint** itself. This object should register new optimization passes. Once loaded, the object will add new command line options to enable various optimizations. To see the new complete list of optimizations, use the **-help** and **–load** options together; for example:

```
bugpoint --load myNewPass.so -help
```

–mlimit *megabytes*

Specifies an upper limit on memory usage of the optimization and codegen. Set to zero to disable the limit.

–output *filename*

Whenever the test program produces output on its standard output stream, it should match the contents of *filename* (the "reference output"). If you do not use this option, **bugpoint** will attempt to generate a reference output by compiling the program with the "safe" backend and running it.

–run-{int,jit,llc,custom}

Whenever the test program is compiled, **bugpoint** should generate code for it using the specified code generator. These options allow you to choose the interpreter, the JIT compiler, the static native code compiler, or a custom command (see **–exec-command**) respectively.

–safe-{llc,custom}

When debugging a code generator, **bugpoint** should use the specified code generator as the "safe" code generator. This is a known-good code generator used to generate the "reference output" if it has not been provided, and to compile portions of the program that as they are excluded from the testcase. These options allow you to choose the static native code compiler, or a custom command, (see **–exec-command**) respectively. The interpreter and the JIT backends cannot currently be used as the "safe" backends.

–exec-command *command*

This option defines the command to use with the **–run-custom** and **–safe-custom** options to execute the bitcode testcase. This can be useful for cross-compilation.

–compile-command *command*

This option defines the command to use with the **–compile-custom** option to compile the bitcode testcase. This can be useful for testing compiler output without running any link or execute stages. To generate a reduced unit test, you may add CHECK directives to the testcase and pass the name of an executable compile-command script in this form:

```
#!/bin/sh
llc "$@"
not FileCheck [bugpoint input file].ll < bugpoint-test-program.s
```

This script will "fail" as long as FileCheck passes. So the result will be the minimum bitcode that passes FileCheck.

–safe-path *path*

This option defines the path to the command to execute with the **–safe-{int,jit,llc,custom}** option.

EXIT STATUS

If **bugpoint** succeeds in finding a problem, it will exit with 0. Otherwise, if an error occurs, it will exit with a non-zero value.

SEE ALSO

optlopt

llvm-extract - extract a function from an LLVM module

SYNOPSIS

llvm-extract [*options*] **–func** *function-name* [*filename*]

DESCRIPTION

The **llvm-extract** command takes the name of a function and extracts it from the specified LLVM bitcode file. It is primarily used as a debugging tool to reduce test cases from larger programs that are triggering a bug.

In addition to extracting the bitcode of the specified function, **llvm-extract** will also remove unreachable global variables, prototypes, and unused types.

The **llvm-extract** command reads its input from standard input if filename is omitted or if filename is −. The output is always written to standard output, unless the **-o** option is specified (see below).

OPTIONS

-f

Enable binary output on terminals. Normally, **llvm-extract** will refuse to write raw bitcode output if the output stream is a terminal. With this option, **llvm-extract** will write raw bitcode regardless of the output device.

–func *function-name*

Extract the function named *function-name* from the LLVM bitcode. May be specified multiple times to extract multiple functions at once.

–rfunc *function-regular-expr*

Extract the function(s) matching *function-regular-expr* from the LLVM bitcode. All functions matching the regular expression will be extracted. May be specified multiple times.

–glob *global-name*

Extract the global variable named *global-name* from the LLVM bitcode. May be specified multiple times to extract multiple global variables at once.

–rglob *glob-regular-expr*

Extract the global variable(s) matching *global-regular-expr* from the LLVM bitcode. All global variables matching the regular expression will be extracted. May be specified multiple times.

-help

Print a summary of command line options.

-o *filename*

Specify the output filename. If filename is "-" (the default), then **llvm-extract** sends its output to standard output.

-S

Write output in LLVM intermediate language (instead of bitcode).

EXIT STATUS

If **llvm-extract** succeeds, it will exit with 0. Otherwise, if an error occurs, it will exit with a non-zero value.

SEE ALSO

bugpoint

llvm-bcanalyzer - LLVM bitcode analyzer

SYNOPSIS

llvm-bcanalyzer [*options*] [*filename*]

DESCRIPTION

The **llvm-bcanalyzer** command is a small utility for analyzing bitcode files. The tool reads a bitcode file (such as generated with the **llvm-as** tool) and produces a statistical report on the contents of the bitcode file. The tool can also dump a low level but human readable version of the bitcode file. This tool is probably not of much interest or utility except for those working directly with the bitcode file format. Most LLVM users can just ignore this tool.

If *filename* is omitted or is −, then **llvm-bcanalyzer** reads its input from standard input. This is useful for combining the tool into a pipeline. Output is written to the standard output.

OPTIONS

-nodetails
Causes **llvm-bcanalyzer** to abbreviate its output by writing out only a module level summary. The details for individual functions are not displayed.

-dump
Causes **llvm-bcanalyzer** to dump the bitcode in a human readable format. This format is significantly different from LLVM assembly and provides details about the encoding of the bitcode file.

-verify
Causes **llvm-bcanalyzer** to verify the module produced by reading the bitcode. This ensures that the statistics generated are based on a consistent module.

-help
Print a summary of command line options.

EXIT STATUS

If **llvm-bcanalyzer** succeeds, it will exit with 0. Otherwise, if an error occurs, it will exit with a non-zero value, usually 1.

SUMMARY OUTPUT DEFINITIONS

The following items are always printed by llvm-bcanalyzer. They comprize the summary output.

Bitcode Analysis Of Module

This just provides the name of the module for which bitcode analysis is being generated.

Bitcode Version Number

The bitcode version (not LLVM version) of the file read by the analyzer.

File Size

The size, in bytes, of the entire bitcode file.

Module Bytes

The size, in bytes, of the module block. Percentage is relative to File Size.

Function Bytes

The size, in bytes, of all the function blocks. Percentage is relative to File Size.

Global Types Bytes

The size, in bytes, of the Global Types Pool. Percentage is relative to File Size. This is the size of the definitions of all types in the bitcode file.

Constant Pool Bytes

The size, in bytes, of the Constant Pool Blocks Percentage is relative to File Size.

Module Globals Bytes

Ths size, in bytes, of the Global Variable Definitions and their initializers. Percentage is relative to File Size.

Instruction List Bytes

The size, in bytes, of all the instruction lists in all the functions. Percentage is relative to File Size. Note that this value is also included in the Function Bytes.

Compaction Table Bytes

The size, in bytes, of all the compaction tables in all the functions. Percentage is relative to File Size. Note that this value is also included in the Function Bytes.

Symbol Table Bytes

The size, in bytes, of all the symbol tables in all the functions. Percentage is relative to File Size. Note that this value is also included in the Function Bytes.

Dependent Libraries Bytes

The size, in bytes, of the list of dependent libraries in the module. Percentage is relative to File Size. Note that this value is also included in the Module Global Bytes.

Number Of Bitcode Blocks

The total number of blocks of any kind in the bitcode file.

Number Of Functions

The total number of function definitions in the bitcode file.

Number Of Types

The total number of types defined in the Global Types Pool.

Number Of Constants

The total number of constants (of any type) defined in the Constant Pool.

Number Of Basic Blocks

The total number of basic blocks defined in all functions in the bitcode file.

Number Of Instructions

The total number of instructions defined in all functions in the bitcode file.

Number Of Long Instructions

The total number of long instructions defined in all functions in the bitcode file. Long instructions are those taking greater than 4 bytes. Typically long instructions are GetElementPtr with several indices, PHI nodes, and calls to functions with large numbers of arguments.

Number Of Operands

The total number of operands used in all instructions in the bitcode file.

Number Of Compaction Tables

The total number of compaction tables in all functions in the bitcode file.

Number Of Symbol Tables

The total number of symbol tables in all functions in the bitcode file.

Number Of Dependent Libs

The total number of dependent libraries found in the bitcode file.

Total Instruction Size

The total size of the instructions in all functions in the bitcode file.

Average Instruction Size

The average number of bytes per instruction across all functions in the bitcode file. This value is computed by dividing Total Instruction Size by Number Of Instructions.

Maximum Type Slot Number

The maximum value used for a type's slot number. Larger slot number values take more bytes to encode.

Maximum Value Slot Number

The maximum value used for a value's slot number. Larger slot number values take more bytes to encode.

Bytes Per Value

The average size of a Value definition (of any type). This is computed by dividing File Size by the total number of values of any type.

Bytes Per Global

The average size of a global definition (constants and global variables).

Bytes Per Function

The average number of bytes per function definition. This is computed by dividing Function Bytes by Number Of Functions.

# of VBR 32-bit Integers

The total number of 32-bit integers encoded using the Variable Bit Rate encoding scheme.

# of VBR 64-bit Integers

The total number of 64-bit integers encoded using the Variable Bit Rate encoding scheme.

# of VBR Compressed Bytes

The total number of bytes consumed by the 32-bit and 64-bit integers that use the Variable Bit Rate encoding scheme.

# of VBR Expanded Bytes

The total number of bytes that would have been consumed by the 32-bit and 64-bit integers had they not been compressed with the Variable Bit Rage encoding scheme.

Bytes Saved With VBR

The total number of bytes saved by using the Variable Bit Rate encoding scheme. The percentage is relative to # of VBR Expanded Bytes.

DETAILED OUTPUT DEFINITIONS

The following definitions occur only if the -nodetails option was not given. The detailed output provides additional information on a per-function basis.

Type

The type signature of the function.

Byte Size

The total number of bytes in the function's block.

Basic Blocks

The number of basic blocks defined by the function.

Instructions

The number of instructions defined by the function.

Long Instructions

The number of instructions using the long instruction format in the function.

Operands

The number of operands used by all instructions in the function.

Instruction Size

The number of bytes consumed by instructions in the function.

Average Instruction Size

The average number of bytes consumed by the instructions in the function. This value is computed by dividing Instruction Size by Instructions.

Bytes Per Instruction

The average number of bytes used by the function per instruction. This value is computed by dividing Byte Size by Instructions. Note that this is not the same as Average Instruction Size. It computes a number relative to the total function size not just the size of the instruction list.

Number of VBR 32-bit Integers

The total number of 32-bit integers found in this function (for any use).

Number of VBR 64-bit Integers

The total number of 64-bit integers found in this function (for any use).

Number of VBR Compressed Bytes

The total number of bytes in this function consumed by the 32-bit and 64-bit integers that use the Variable Bit Rate encoding scheme.

Number of VBR Expanded Bytes

The total number of bytes in this function that would have been consumed by the 32-bit and 64-bit integers had they not been compressed with the Variable Bit Rate encoding scheme.

Bytes Saved With VBR

The total number of bytes saved in this function by using the Variable Bit Rate encoding scheme. The percentage is relative to # of VBR Expanded Bytes.

SEE ALSO

llvm-dis - LLVM disassembler, *LLVM Bitcode File Format*

2.4.3 Developer Tools

FileCheck - Flexible pattern matching file verifier

SYNOPSIS

FileCheck *match-filename* [*–check-prefix=XXX*] [*–strict-whitespace*]

DESCRIPTION

FileCheck reads two files (one from standard input, and one specified on the command line) and uses one to verify the other. This behavior is particularly useful for the testsuite, which wants to verify that the output of some tool (e.g. **llc**) contains the expected information (for example, a movsd from esp or whatever is interesting). This is similar to using **grep**, but it is optimized for matching multiple different inputs in one file in a specific order.

The `match-filename` file specifies the file that contains the patterns to match. The file to verify is read from standard input unless the *`--input-file`* option is used.

OPTIONS

`-help`
Print a summary of command line options.

`-check-prefix` `prefix`
FileCheck searches the contents of `match-filename` for patterns to match. By default, these patterns are prefixed with "`CHECK:`". If you'd like to use a different prefix (e.g. because the same input file is checking multiple different tool or options), the *`--check-prefix`* argument allows you to specify one or more prefixes to match. Multiple prefixes are useful for tests which might change for different run options, but most lines remain the same.

`-input-file` `filename`
File to check (defaults to stdin).

`-strict-whitespace`
By default, FileCheck canonicalizes input horizontal whitespace (spaces and tabs) which causes it to ignore these differences (a space will match a tab). The *`--strict-whitespace`* argument disables this behavior. End-of-line sequences are canonicalized to UNIX-style `\n` in all modes.

-implicit-check-not check-pattern
: Adds implicit negative checks for the specified patterns between positive checks. The option allows writing stricter tests without stuffing them with CHECK-NOTs.

  For example, "--implicit-check-not warning:" can be useful when testing diagnostic messages from tools that don't have an option similar to clang -verify. With this option FileCheck will verify that input does not contain warnings not covered by any CHECK: patterns.

-version
: Show the version number of this program.

EXIT STATUS

If **FileCheck** verifies that the file matches the expected contents, it exits with 0. Otherwise, if not, or if an error occurs, it will exit with a non-zero value.

TUTORIAL

FileCheck is typically used from LLVM regression tests, being invoked on the RUN line of the test. A simple example of using FileCheck from a RUN line looks like this:

```
; RUN: llvm-as < %s | llc -march=x86-64 | FileCheck %s
```

This syntax says to pipe the current file ("%s") into llvm-as, pipe that into llc, then pipe the output of llc into FileCheck. This means that FileCheck will be verifying its standard input (the llc output) against the filename argument specified (the original .ll file specified by "%s"). To see how this works, let's look at the rest of the .ll file (after the RUN line):

```
define void @sub1(i32* %p, i32 %v) {
entry:
; CHECK: sub1:
; CHECK: subl
        %0 = tail call i32 @llvm.atomic.load.sub.i32.p0i32(i32* %p, i32 %v)
        ret void
}

define void @inc4(i64* %p) {
entry:
; CHECK: inc4:
; CHECK: incq
        %0 = tail call i64 @llvm.atomic.load.add.i64.p0i64(i64* %p, i64 1)
        ret void
}
```

Here you can see some "CHECK:" lines specified in comments. Now you can see how the file is piped into llvm-as, then llc, and the machine code output is what we are verifying. FileCheck checks the machine code output to verify that it matches what the "CHECK:" lines specify.

The syntax of the "CHECK:" lines is very simple: they are fixed strings that must occur in order. FileCheck defaults to ignoring horizontal whitespace differences (e.g. a space is allowed to match a tab) but otherwise, the contents of the "CHECK:" line is required to match some thing in the test file exactly.

One nice thing about FileCheck (compared to grep) is that it allows merging test cases together into logical groups. For example, because the test above is checking for the "sub1:" and "inc4:" labels, it will not match unless there is a "subl" in between those labels. If it existed somewhere else in the file, that would not count: "grep subl" matches if "subl" exists anywhere in the file.

The FileCheck -check-prefix option The FileCheck *-check-prefix* option allows multiple test configurations to be driven from one *.ll* file. This is useful in many circumstances, for example, testing different architectural variants with **llc**. Here's a simple example:

```
; RUN: llvm-as < %s | llc -mtriple=i686-apple-darwin9 -mattr=sse41 \
; RUN:              | FileCheck %s -check-prefix=X32
; RUN: llvm-as < %s | llc -mtriple=x86_64-apple-darwin9 -mattr=sse41 \
; RUN:              | FileCheck %s -check-prefix=X64

define <4 x i32> @pinsrd_1(i32 %s, <4 x i32> %tmp) nounwind {
        %tmp1 = insertelement <4 x i32>; %tmp, i32 %s, i32 1
        ret <4 x i32> %tmp1
; X32: pinsrd_1:
; X32:    pinsrd $1, 4(%esp), %xmm0

; X64: pinsrd_1:
; X64:    pinsrd $1, %edi, %xmm0
}
```

In this case, we're testing that we get the expected code generation with both 32-bit and 64-bit code generation.

The "CHECK-NEXT:" directive Sometimes you want to match lines and would like to verify that matches happen on exactly consecutive lines with no other lines in between them. In this case, you can use "`CHECK:`" and "`CHECK-NEXT:`" directives to specify this. If you specified a custom check prefix, just use "`<PREFIX>-NEXT:`". For example, something like this works as you'd expect:

```
define void @t2(<2 x double>* %r, <2 x double>* %A, double %B) {
     %tmp3 = load <2 x double>* %A, align 16
     %tmp7 = insertelement <2 x double> undef, double %B, i32 0
     %tmp9 = shufflevector <2 x double> %tmp3,
                           <2 x double> %tmp7,
                           <2 x i32> < i32 0, i32 2 >
     store <2 x double> %tmp9, <2 x double>* %r, align 16
     ret void

; CHECK:          t2:
; CHECK:             movl    8(%esp), %eax
; CHECK-NEXT:        movapd  (%eax), %xmm0
; CHECK-NEXT:        movhpd  12(%esp), %xmm0
; CHECK-NEXT:        movl    4(%esp), %eax
; CHECK-NEXT:        movapd  %xmm0, (%eax)
; CHECK-NEXT:        ret
}
```

"`CHECK-NEXT:`" directives reject the input unless there is exactly one newline between it and the previous directive. A "`CHECK-NEXT:`" cannot be the first directive in a file.

The "CHECK-NOT:" directive The "`CHECK-NOT:`" directive is used to verify that a string doesn't occur between two matches (or before the first match, or after the last match). For example, to verify that a load is removed by a transformation, a test like this can be used:

```
define i8 @coerce_offset0(i32 %V, i32* %P) {
  store i32 %V, i32* %P

  %P2 = bitcast i32* %P to i8*
  %P3 = getelementptr i8* %P2, i32 2
```

```
  %A = load i8* %P3
  ret i8 %A
; CHECK: @coerce_offset0
; CHECK-NOT: load
; CHECK: ret i8
}
```

The "CHECK-DAG:" directive If it's necessary to match strings that don't occur in a strictly sequential order, "`CHECK-DAG:`" could be used to verify them between two matches (or before the first match, or after the last match). For example, clang emits vtable globals in reverse order. Using `CHECK-DAG:`, we can keep the checks in the natural order:

```
// RUN: %clang_cc1 %s -emit-llvm -o - | FileCheck %s

struct Foo { virtual void method(); };
Foo f;  // emit vtable
// CHECK-DAG: @_ZTV3Foo =

struct Bar { virtual void method(); };
Bar b;
// CHECK-DAG: @_ZTV3Bar =
```

`CHECK-NOT:` directives could be mixed with `CHECK-DAG:` directives to exclude strings between the surrounding `CHECK-DAG:` directives. As a result, the surrounding `CHECK-DAG:` directives cannot be reordered, i.e. all occurrences matching `CHECK-DAG:` before `CHECK-NOT:` must not fall behind occurrences matching `CHECK-DAG:` after `CHECK-NOT:`. For example,

```
; CHECK-DAG: BEFORE
; CHECK-NOT: NOT
; CHECK-DAG: AFTER
```

This case will reject input strings where `BEFORE` occurs after `AFTER`.

With captured variables, `CHECK-DAG:` is able to match valid topological orderings of a DAG with edges from the definition of a variable to its use. It's useful, e.g., when your test cases need to match different output sequences from the instruction scheduler. For example,

```
; CHECK-DAG: add [[REG1:r[0-9]+]], r1, r2
; CHECK-DAG: add [[REG2:r[0-9]+]], r3, r4
; CHECK:     mul r5, [[REG1]], [[REG2]]
```

In this case, any order of that two `add` instructions will be allowed.

If you are defining *and* using variables in the same `CHECK-DAG:` block, be aware that the definition rule can match *after* its use.

So, for instance, the code below will pass:

```
; CHECK-DAG: vmov.32 [[REG2:d[0-9]+]][0]
; CHECK-DAG: vmov.32 [[REG2]][1]
vmov.32 d0[1]
vmov.32 d0[0]
```

While this other code, will not:

```
; CHECK-DAG: vmov.32 [[REG2:d[0-9]+]][0]
; CHECK-DAG: vmov.32 [[REG2]][1]
vmov.32 d1[1]
vmov.32 d0[0]
```

While this can be very useful, it's also dangerous, because in the case of register sequence, you must have a strong order (read before write, copy before use, etc). If the definition your test is looking for doesn't match (because of a bug in the compiler), it may match further away from the use, and mask real bugs away.

In those cases, to enforce the order, use a non-DAG directive between DAG-blocks.

The "CHECK-LABEL:" directive Sometimes in a file containing multiple tests divided into logical blocks, one or more `CHECK:` directives may inadvertently succeed by matching lines in a later block. While an error will usually eventually be generated, the check flagged as causing the error may not actually bear any relationship to the actual source of the problem.

In order to produce better error messages in these cases, the "`CHECK-LABEL:`" directive can be used. It is treated identically to a normal `CHECK` directive except that FileCheck makes an additional assumption that a line matched by the directive cannot also be matched by any other check present in `match-filename`; this is intended to be used for lines containing labels or other unique identifiers. Conceptually, the presence of `CHECK-LABEL` divides the input stream into separate blocks, each of which is processed independently, preventing a `CHECK:` directive in one block matching a line in another block. For example,

```
define %struct.C* @C_ctor_base(%struct.C* %this, i32 %x) {
entry:
; CHECK-LABEL: C_ctor_base:
; CHECK: mov [[SAVETHIS:r[0-9]+]], r0
; CHECK: bl A_ctor_base
; CHECK: mov r0, [[SAVETHIS]]
  %0 = bitcast %struct.C* %this to %struct.A*
  %call = tail call %struct.A* @A_ctor_base(%struct.A* %0)
  %1 = bitcast %struct.C* %this to %struct.B*
  %call2 = tail call %struct.B* @B_ctor_base(%struct.B* %1, i32 %x)
  ret %struct.C* %this
}

define %struct.D* @D_ctor_base(%struct.D* %this, i32 %x) {
entry:
; CHECK-LABEL: D_ctor_base:
```

The use of `CHECK-LABEL:` directives in this case ensures that the three `CHECK:` directives only accept lines corresponding to the body of the `@C_ctor_base` function, even if the patterns match lines found later in the file. Furthermore, if one of these three `CHECK:` directives fail, FileCheck will recover by continuing to the next block, allowing multiple test failures to be detected in a single invocation.

There is no requirement that `CHECK-LABEL:` directives contain strings that correspond to actual syntactic labels in a source or output language: they must simply uniquely match a single line in the file being verified.

`CHECK-LABEL:` directives cannot contain variable definitions or uses.

FileCheck Pattern Matching Syntax The "`CHECK:`" and "`CHECK-NOT:`" directives both take a pattern to match. For most uses of FileCheck, fixed string matching is perfectly sufficient. For some things, a more flexible form of matching is desired. To support this, FileCheck allows you to specify regular expressions in matching strings, surrounded by double braces: `{{yourregex}}`. Because we want to use fixed string matching for a majority of what we do, FileCheck has been designed to support mixing and matching fixed string matching with regular expressions. This allows you to write things like this:

```
; CHECK: movhpd      {{[0-9]+}}(%esp), {{%xmm[0-7]}}
```

In this case, any offset from the ESP register will be allowed, and any xmm register will be allowed.

Because regular expressions are enclosed with double braces, they are visually distinct, and you don't need to use escape characters within the double braces like you would in C. In the rare case that you want to match double braces explicitly from the input, you can use something ugly like `{{[{][{]}}` as your pattern.

FileCheck Variables It is often useful to match a pattern and then verify that it occurs again later in the file. For codegen tests, this can be useful to allow any register, but verify that that register is used consistently later. To do this, **FileCheck** allows named variables to be defined and substituted into patterns. Here is a simple example:

```
; CHECK: test5:
; CHECK:    notw     [[REGISTER:%[a-z]+]]
; CHECK:    andw     {{.*}}[[REGISTER]]
```

The first check line matches a regex `%[a-z]+` and captures it into the variable `REGISTER`. The second line verifies that whatever is in `REGISTER` occurs later in the file after an "`andw`". **FileCheck** variable references are always contained in `[[ ]]` pairs, and their names can be formed with the regex `[a-zA-Z][a-zA-Z0-9]*`. If a colon follows the name, then it is a definition of the variable; otherwise, it is a use.

FileCheck variables can be defined multiple times, and uses always get the latest value. Variables can also be used later on the same line they were defined on. For example:

```
; CHECK: op [[REG:r[0-9]+]], [[REG]]
```

Can be useful if you want the operands of `op` to be the same register, and don't care exactly which register it is.

FileCheck Expressions Sometimes there's a need to verify output which refers line numbers of the match file, e.g. when testing compiler diagnostics. This introduces a certain fragility of the match file structure, as "`CHECK:`" lines contain absolute line numbers in the same file, which have to be updated whenever line numbers change due to text addition or deletion.

To support this case, FileCheck allows using `[[@LINE]]`, `[[@LINE+<offset>]]`, `[[@LINE-<offset>]]` expressions in patterns. These expressions expand to a number of the line where a pattern is located (with an optional integer offset).

This way match patterns can be put near the relevant test lines and include relative line number references, for example:

```
// CHECK: test.cpp:[[@LINE+4]]:6: error: expected ';' after top level declarator
// CHECK-NEXT: {{^int a}}
// CHECK-NEXT: {{^     \^}}
// CHECK-NEXT: {{^     ;}}
int a
```

tblgen - Target Description To C++ Code Generator

SYNOPSIS

tblgen [*options*] [*filename*]

DESCRIPTION

tblgen translates from target description (`.td`) files into C++ code that can be included in the definition of an LLVM target library. Most users of LLVM will not need to use this program. It is only for assisting with writing an LLVM target backend.

The input and output of **tblgen** is beyond the scope of this short introduction; please see the *introduction to TableGen*.

The *filename* argument specifies the name of a Target Description (`.td`) file to read as input.

OPTIONS

-help
Print a summary of command line options.

-o `filename`
Specify the output file name. If `filename` is `-`, then **tblgen** sends its output to standard output.

-I `directory`
Specify where to find other target description files for inclusion. The `directory` value should be a full or partial path to a directory that contains target description files.

-asmparsernum `N`
Make -gen-asm-parser emit assembly writer number `N`.

-asmwriternum `N`
Make -gen-asm-writer emit assembly writer number `N`.

-class `className`
Print the enumeration list for this class.

-print-records
Print all records to standard output (default).

-print-enums
Print enumeration values for a class.

-print-sets
Print expanded sets for testing DAG exprs.

-gen-emitter
Generate machine code emitter.

-gen-register-info
Generate registers and register classes info.

-gen-instr-info
Generate instruction descriptions.

-gen-asm-writer
Generate the assembly writer.

-gen-disassembler
Generate disassembler.

-gen-pseudo-lowering
Generate pseudo instruction lowering.

-gen-dag-isel
Generate a DAG (Directed Acycle Graph) instruction selector.

-gen-asm-matcher
Generate assembly instruction matcher.

-gen-dfa-packetizer
Generate DFA Packetizer for VLIW targets.

-gen-fast-isel
Generate a "fast" instruction selector.

-gen-subtarget
Generate subtarget enumerations.

-gen-intrinsic
Generate intrinsic information.

-gen-tgt-intrinsic
Generate target intrinsic information.

-gen-enhanced-disassembly-info
Generate enhanced disassembly info.

-version
Show the version number of this program.

EXIT STATUS

If **tblgen** succeeds, it will exit with 0. Otherwise, if an error occurs, it will exit with a non-zero value.

lit - LLVM Integrated Tester

SYNOPSIS

lit [*options*] [*tests*]

DESCRIPTION

lit is a portable tool for executing LLVM and Clang style test suites, summarizing their results, and providing indication of failures. **lit** is designed to be a lightweight testing tool with as simple a user interface as possible.

lit should be run with one or more *tests* to run specified on the command line. Tests can be either individual test files or directories to search for tests (see *TEST DISCOVERY*).

Each specified test will be executed (potentially in parallel) and once all tests have been run **lit** will print summary information on the number of tests which passed or failed (see *TEST STATUS RESULTS*). The **lit** program will execute with a non-zero exit code if any tests fail.

By default **lit** will use a succinct progress display and will only print summary information for test failures. See *OUTPUT OPTIONS* for options controlling the **lit** progress display and output.

lit also includes a number of options for controlling how tests are executed (specific features may depend on the particular test format). See *EXECUTION OPTIONS* for more information.

Finally, **lit** also supports additional options for only running a subset of the options specified on the command line, see *SELECTION OPTIONS* for more information.

Users interested in the **lit** architecture or designing a **lit** testing implementation should see *LIT INFRASTRUCTURE*.

GENERAL OPTIONS

-h, -help
Show the **lit** help message.

-j N, **-threads**=N
Run N tests in parallel. By default, this is automatically chosen to match the number of detected available CPUs.

-config-prefix=NAME
Search for *NAME*.cfg and *NAME*.site.cfg when searching for test suites, instead of lit.cfg and lit.site.cfg.

-param NAME, **-param** NAME=VALUE
Add a user defined parameter NAME with the given VALUE (or the empty string if not given). The meaning and use of these parameters is test suite dependent.

OUTPUT OPTIONS

-q, -quiet
Suppress any output except for test failures.

-s, -succinct
Show less output, for example don't show information on tests that pass.

-v, -verbose
Show more information on test failures, for example the entire test output instead of just the test result.

-no-progress-bar
Do not use curses based progress bar.

-show-unsupported
Show the names of unsupported tests.

-show-xfail
Show the names of tests that were expected to fail.

EXECUTION OPTIONS

-path=PATH
Specify an additional PATH to use when searching for executables in tests.

-vg
Run individual tests under valgrind (using the memcheck tool). The --error-exitcode argument for valgrind is used so that valgrind failures will cause the program to exit with a non-zero status.

When this option is enabled, **lit** will also automatically provide a "valgrind" feature that can be used to conditionally disable (or expect failure in) certain tests.

-vg-arg=ARG
When *--vg* is used, specify an additional argument to pass to **valgrind** itself.

-vg-leak
When *--vg* is used, enable memory leak checks. When this option is enabled, **lit** will also automatically provide a "vg_leak" feature that can be used to conditionally disable (or expect failure in) certain tests.

-time-tests
Track the wall time individual tests take to execute and includes the results in the summary output. This is useful for determining which tests in a test suite take the most time to execute. Note that this option is most useful with -j 1.

SELECTION OPTIONS

-max-tests=N
Run at most N tests and then terminate.

-max-time=N
Spend at most N seconds (approximately) running tests and then terminate.

-shuffle
Run the tests in a random order.

ADDITIONAL OPTIONS

-debug
Run **lit** in debug mode, for debugging configuration issues and **lit** itself.

-show-suites
List the discovered test suites and exit.

-show-tests
List all of the the discovered tests and exit.

EXIT STATUS

lit will exit with an exit code of 1 if there are any FAIL or XPASS results. Otherwise, it will exit with the status 0. Other exit codes are used for non-test related failures (for example a user error or an internal program error).

TEST DISCOVERY

The inputs passed to **lit** can be either individual tests, or entire directories or hierarchies of tests to run. When **lit** starts up, the first thing it does is convert the inputs into a complete list of tests to run as part of *test discovery*.

In the **lit** model, every test must exist inside some *test suite*. **lit** resolves the inputs specified on the command line to test suites by searching upwards from the input path until it finds a `lit.cfg` or `lit.site.cfg` file. These files serve as both a marker of test suites and as configuration files which **lit** loads in order to understand how to find and run the tests inside the test suite.

Once **lit** has mapped the inputs into test suites it traverses the list of inputs adding tests for individual files and recursively searching for tests in directories.

This behavior makes it easy to specify a subset of tests to run, while still allowing the test suite configuration to control exactly how tests are interpreted. In addition, **lit** always identifies tests by the test suite they are in, and their relative path inside the test suite. For appropriately configured projects, this allows **lit** to provide convenient and flexible support for out-of-tree builds.

TEST STATUS RESULTS

Each test ultimately produces one of the following six results:

PASS

The test succeeded.

XFAIL

The test failed, but that is expected. This is used for test formats which allow specifying that a test does not currently work, but wish to leave it in the test suite.

XPASS

The test succeeded, but it was expected to fail. This is used for tests which were specified as expected to fail, but are now succeeding (generally because the feature they test was broken and has been fixed).

FAIL

The test failed.

UNRESOLVED

The test result could not be determined. For example, this occurs when the test could not be run, the test itself is invalid, or the test was interrupted.

UNSUPPORTED

The test is not supported in this environment. This is used by test formats which can report unsupported tests.

Depending on the test format tests may produce additional information about their status (generally only for failures). See the *OUTPUT OPTIONS* section for more information.

LIT INFRASTRUCTURE

This section describes the **lit** testing architecture for users interested in creating a new **lit** testing implementation, or extending an existing one.

lit proper is primarily an infrastructure for discovering and running arbitrary tests, and to expose a single convenient interface to these tests. **lit** itself doesn't know how to run tests, rather this logic is defined by *test suites*.

TEST SUITES As described in *TEST DISCOVERY*, tests are always located inside a *test suite*. Test suites serve to define the format of the tests they contain, the logic for finding those tests, and any additional information to run the tests.

lit identifies test suites as directories containing `lit.cfg` or `lit.site.cfg` files (see also `--config-prefix`). Test suites are initially discovered by recursively searching up the directory hierarchy for all the input files passed on the command line. You can use `--show-suites` to display the discovered test suites at startup.

Once a test suite is discovered, its config file is loaded. Config files themselves are Python modules which will be executed. When the config file is executed, two important global variables are predefined:

lit_config

The global **lit** configuration object (a *LitConfig* instance), which defines the builtin test formats, global configuration parameters, and other helper routines for implementing test configurations.

config

This is the config object (a *TestingConfig* instance) for the test suite, which the config file is expected to populate. The following variables are also available on the *config* object, some of which must be set by the config and others are optional or predefined:

name *[required]* The name of the test suite, for use in reports and diagnostics.

test_format *[required]* The test format object which will be used to discover and run tests in the test suite. Generally this will be a builtin test format available from the *lit.formats* module.

test_source_root The filesystem path to the test suite root. For out-of-dir builds this is the directory that will be scanned for tests.

test_exec_root For out-of-dir builds, the path to the test suite root inside the object directory. This is where tests will be run and temporary output files placed.

environment A dictionary representing the environment to use when executing tests in the suite.

suffixes For **lit** test formats which scan directories for tests, this variable is a list of suffixes to identify test files. Used by: *ShTest*.

substitutions For **lit** test formats which substitute variables into a test script, the list of substitutions to perform. Used by: *ShTest*.

unsupported Mark an unsupported directory, all tests within it will be reported as unsupported. Used by: *ShTest*.

parent The parent configuration, this is the config object for the directory containing the test suite, or None.

root The root configuration. This is the top-most **lit** configuration in the project.

pipefail Normally a test using a shell pipe fails if any of the commands on the pipe fail. If this is not desired, setting this variable to false makes the test fail only if the last command in the pipe fails.

TEST DISCOVERY Once test suites are located, **lit** recursively traverses the source directory (following *test_source_root*) looking for tests. When **lit** enters a sub-directory, it first checks to see if a nested test suite is defined in that directory. If so, it loads that test suite recursively, otherwise it instantiates a local test config for the directory (see *LOCAL CONFIGURATION FILES*).

Tests are identified by the test suite they are contained within, and the relative path inside that suite. Note that the relative path may not refer to an actual file on disk; some test formats (such as *GoogleTest*) define "virtual tests" which have a path that contains both the path to the actual test file and a subpath to identify the virtual test.

LOCAL CONFIGURATION FILES When **lit** loads a subdirectory in a test suite, it instantiates a local test configuration by cloning the configuration for the parent direction — the root of this configuration chain will always be a test suite. Once the test configuration is cloned **lit** checks for a *lit.local.cfg* file in the subdirectory. If present, this file will be loaded and can be used to specialize the configuration for each individual directory. This facility can be used to define subdirectories of optional tests, or to change other configuration parameters — for example, to change the test format, or the suffixes which identify test files.

TEST RUN OUTPUT FORMAT The **lit** output for a test run conforms to the following schema, in both short and verbose modes (although in short mode no PASS lines will be shown). This schema has been chosen to be relatively easy to reliably parse by a machine (for example in buildbot log scraping), and for other tools to generate.

Each test result is expected to appear on a line that matches:

```
<result code>: <test name> (<progress info>)
```

where `<result-code>` is a standard test result such as PASS, FAIL, XFAIL, XPASS, UNRESOLVED, or UNSUPPORTED. The performance result codes of IMPROVED and REGRESSED are also allowed.

The `<test name>` field can consist of an arbitrary string containing no newline.

The `<progress info>` field can be used to report progress information such as (1/300) or can be empty, but even when empty the parentheses are required.

Each test result may include additional (multiline) log information in the following format:

```
<log delineator> TEST '(<test name>)' <trailing delineator>
... log message ...
<log delineator>
```

where `<test name>` should be the name of a preceding reported test, `<log delineator>` is a string of "*" characters *at least* four characters long (the recommended length is 20), and `<trailing delineator>` is an arbitrary (unparsed) string.

The following is an example of a test run output which consists of four tests A, B, C, and D, and a log message for the failing test C:

```
PASS: A (1 of 4)
PASS: B (2 of 4)
FAIL: C (3 of 4)
******************** TEST 'C' FAILED ********************
Test 'C' failed as a result of exit code 1.
********************
PASS: D (4 of 4)
```

LIT EXAMPLE TESTS The **lit** distribution contains several example implementations of test suites in the *ExampleTests* directory.

SEE ALSO

valgrind(1)

llvm-build - LLVM Project Build Utility

SYNOPSIS

llvm-build [*options*]

DESCRIPTION

llvm-build is a tool for working with LLVM projects that use the LLVMBuild system for describing their components.

At heart, **llvm-build** is responsible for loading, verifying, and manipulating the project's component data. The tool is primarily designed for use in implementing build systems and tools which need access to the project structure information.

OPTIONS

-h, **–help**

Print the builtin program help.

–source-root=*PATH*

If given, load the project at the given source root path. If this option is not given, the location of the project sources will be inferred from the location of the **llvm-build** script itself.

–print-tree

Print the component tree for the project.

–write-library-table

Write out the C++ fragment which defines the components, library names, and required libraries. This C++ fragment is built into llvm-configlllvm-config in order to provide clients with the list of required libraries for arbitrary component combinations.

–write-llvmbuild

Write out new *LLVMBuild.txt* files based on the loaded components. This is useful for auto-upgrading the schema of the files. **llvm-build** will try to a limited extent to preserve the comments which were written in the original source file, although at this time it only preserves block comments that precede the section names in the *LLVMBuild* files.

–write-cmake-fragment

Write out the LLVMBuild in the form of a CMake fragment, so it can easily be consumed by the CMake based build system. The exact contents and format of this file are closely tied to how LLVMBuild is integrated with CMake, see LLVM's top-level CMakeLists.txt.

–write-make-fragment

Write out the LLVMBuild in the form of a Makefile fragment, so it can easily be consumed by a Make based build system. The exact contents and format of this file are closely tied to how LLVMBuild is integrated with the Makefiles, see LLVM's Makefile.rules.

–llvmbuild-source-root=*PATH*

If given, expect the *LLVMBuild* files for the project to be rooted at the given path, instead of inside the source tree itself. This option is primarily designed for use in conjunction with **–write-llvmbuild** to test changes to *LLVMBuild* schema.

EXIT STATUS

llvm-build exits with 0 if operation was successful. Otherwise, it will exist with a non-zero value.

llvm-readobj - LLVM Object Reader

SYNOPSIS

llvm-readobj [*options*] [*input...*]

DESCRIPTION

The **llvm-readobj** tool displays low-level format-specific information about one or more object files. The tool and its output is primarily designed for use in FileCheck-based tests.

OPTIONS

If `input` is "`-`" or omitted, **llvm-readobj** reads from standard input. Otherwise, it will read from the specified `filenames`.

`-help`
Print a summary of command line options.

`-version`
Display the version of this program

`-file-headers, -h`
Display file headers.

`-sections, -s`
Display all sections.

-section-data, -sd
When used with `-sections`, display section data for each section shown.

-section-relocations, -sr
When used with `-sections`, display relocations for each section shown.

-section-symbols, -st
When used with `-sections`, display symbols for each section shown.

-relocations, -r
Display the relocation entries in the file.

-symbols, -t
Display the symbol table.

-dyn-symbols
Display the dynamic symbol table (only for ELF object files).

-unwind, -u
Display unwind information.

-expand-relocs
When used with `-relocations`, display each relocation in an expanded multi-line format.

-dynamic-table
Display the ELF .dynamic section table (only for ELF object files).

-needed-libs
Display the needed libraries (only for ELF object files).

-program-headers
Display the ELF program headers (only for ELF object files).

EXIT STATUS

llvm-readobj returns 0.

2.5 Getting Started with the LLVM System

2.5.1 Overview

Welcome to LLVM! In order to get started, you first need to know some basic information.

First, LLVM comes in three pieces. The first piece is the LLVM suite. This contains all of the tools, libraries, and header files needed to use LLVM. It contains an assembler, disassembler, bitcode analyzer and bitcode optimizer. It also contains basic regression tests that can be used to test the LLVM tools and the Clang front end.

The second piece is the Clang front end. This component compiles C, C++, Objective C, and Objective C++ code into LLVM bitcode. Once compiled into LLVM bitcode, a program can be manipulated with the LLVM tools from the LLVM suite.

There is a third, optional piece called Test Suite. It is a suite of programs with a testing harness that can be used to further test LLVM's functionality and performance.

2.5.2 Getting Started Quickly (A Summary)

The LLVM Getting Started documentation may be out of date. So, the Clang Getting Started page might also be a good place to start.

Here's the short story for getting up and running quickly with LLVM:

1. Read the documentation.
2. Read the documentation.
3. Remember that you were warned twice about reading the documentation.
4. Checkout LLVM:
   - `cd where-you-want-llvm-to-live`
   - `svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
5. Checkout Clang:
   - `cd where-you-want-llvm-to-live`
   - `cd llvm/tools`
   - `svn co http://llvm.org/svn/llvm-project/cfe/trunk clang`
6. Checkout Compiler-RT:
   - `cd where-you-want-llvm-to-live`
   - `cd llvm/projects`
   - `svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt`
7. Get the Test Suite Source Code **[Optional]**
   - `cd where-you-want-llvm-to-live`
   - `cd llvm/projects`
   - `svn co http://llvm.org/svn/llvm-project/test-suite/trunk test-suite`
8. Configure and build LLVM and Clang:
   - `cd where-you-want-to-build-llvm`
   - `mkdir build` (for building without polluting the source dir)
   - `cd build`
   - `../llvm/configure [options]` Some common options:
     - `--prefix=directory` — Specify for *directory* the full pathname of where you want the LLVM tools and libraries to be installed (default `/usr/local`).
     - `--enable-optimized` — Compile with optimizations enabled (default is NO).
     - `--enable-assertions` — Compile with assertion checks enabled (default is YES).
   - `make [-j]` — The `-j` specifies the number of jobs (commands) to run simultaneously. This builds both LLVM and Clang for Debug+Asserts mode. The `--enable-optimized` configure option is used to specify a Release build.
   - `make check-all` — This run the regression tests to ensure everything is in working order.
   - It is also possible to use CMake instead of the makefiles. With CMake it is possible to generate project files for several IDEs: Xcode, Eclipse CDT4, CodeBlocks, Qt-Creator (use the CodeBlocks generator), KDevelop3.

- If you get an "internal compiler error (ICE)" or test failures, see *below*.

Consult the Getting Started with LLVM section for detailed information on configuring and compiling LLVM. See Setting Up Your Environment for tips that simplify working with the Clang front end and LLVM tools. Go to Program Layout to learn about the layout of the source code tree.

2.5.3 Requirements

Before you begin to use the LLVM system, review the requirements given below. This may save you some trouble by knowing ahead of time what hardware and software you will need.

Hardware

LLVM is known to work on the following host platforms:

OS	Arch	Compilers
Linux	x86[1]	GCC, Clang
Linux	amd64	GCC, Clang
Linux	ARM[4]	GCC, Clang
Linux	PowerPC	GCC, Clang
Solaris	V9 (Ultrasparc)	GCC
FreeBSD	x86[1]	GCC, Clang
FreeBSD	amd64	GCC, Clang
MacOS X[2]	PowerPC	GCC
MacOS X	x86	GCC, Clang
Cygwin/Win32	x86[1, 3]	GCC
Windows	x86[1]	Visual Studio
Windows x64	x86-64	Visual Studio

Note:

1. Code generation supported for Pentium processors and up
2. Code generation supported for 32-bit ABI only
3. To use LLVM modules on Win32-based system, you may configure LLVM with `--enable-shared`.
4. MCJIT not working well pre-v7, old JIT engine not supported any more.

Note that you will need about 1-3 GB of space for a full LLVM build in Debug mode, depending on the system (it is so large because of all the debugging information and the fact that the libraries are statically linked into multiple tools). If you do not need many of the tools and you are space-conscious, you can pass `ONLY_TOOLS="tools you need"` to make. The Release build requires considerably less space.

The LLVM suite *may* compile on other platforms, but it is not guaranteed to do so. If compilation is successful, the LLVM utilities should be able to assemble, disassemble, analyze, and optimize LLVM bitcode. Code generation should work as well, although the generated native code may not work on your platform.

Software

Compiling LLVM requires that you have several software packages installed. The table below lists those required packages. The Package column is the usual name for the software package that LLVM depends on. The Version column provides "known to work" versions of the package. The Notes column describes how LLVM uses the package and provides other details.

Package	Version	Notes
GNU Make	3.79, 3.79.1	Makefile/build processor
GCC	>=4.7.0	C/C++ compiler[1]
python	>=2.5	Automated test suite[2]
GNU M4	1.4	Macro processor for configuration[3]
GNU Autoconf	2.60	Configuration script builder[3]
GNU Automake	1.9.6	aclocal macro generator[3]
libtool	1.5.22	Shared library manager[3]
zlib	>=1.2.3.4	Compression library[4]

Note:

1. Only the C and C++ languages are needed so there's no need to build the other languages for LLVM's purposes. See *below* for specific version info.
2. Only needed if you want to run the automated test suite in the `llvm/test` directory.
3. If you want to make changes to the configure scripts, you will need GNU autoconf (2.60), and consequently, GNU M4 (version 1.4 or higher). You will also need automake (1.9.6). We only use aclocal from that package.
4. Optional, adds compression / uncompression capabilities to selected LLVM tools.

Additionally, your compilation host is expected to have the usual plethora of Unix utilities. Specifically:

- **ar** — archive library builder
- **bzip2** — bzip2 command for distribution generation
- **bunzip2** — bunzip2 command for distribution checking
- **chmod** — change permissions on a file
- **cat** — output concatenation utility
- **cp** — copy files
- **date** — print the current date/time
- **echo** — print to standard output
- **egrep** — extended regular expression search utility
- **find** — find files/dirs in a file system
- **grep** — regular expression search utility
- **gzip** — gzip command for distribution generation
- **gunzip** — gunzip command for distribution checking
- **install** — install directories/files
- **mkdir** — create a directory
- **mv** — move (rename) files
- **ranlib** — symbol table builder for archive libraries
- **rm** — remove (delete) files and directories
- **sed** — stream editor for transforming output
- **sh** — Bourne shell for make build scripts
- **tar** — tape archive for distribution generation
- **test** — test things in file system

- **unzip** — unzip command for distribution checking
- **zip** — zip command for distribution generation

Host C++ Toolchain, both Compiler and Standard Library

LLVM is very demanding of the host C++ compiler, and as such tends to expose bugs in the compiler. We are also planning to follow improvements and developments in the C++ language and library reasonably closely. As such, we require a modern host C++ toolchain, both compiler and standard library, in order to build LLVM.

For the most popular host toolchains we check for specific minimum versions in our build systems:

- Clang 3.1
- GCC 4.7
- Visual Studio 2012

Anything older than these toolchains *may* work, but will require forcing the build system with a special option and is not really a supported host platform. Also note that older versions of these compilers have often crashed or miscompiled LLVM.

For less widely used host toolchains such as ICC or xlC, be aware that a very recent version may be required to support all of the C++ features used in LLVM.

We track certain versions of software that are *known* to fail when used as part of the host toolchain. These even include linkers at times.

GCC 4.6.3 on ARM: Miscompiles `llvm-readobj` at `-O3`. A test failure in `test/Object/readobj-shared-object.test` is one symptom of the problem.

GNU ld 2.16.X. Some 2.16.X versions of the ld linker will produce very long warning messages complaining that some "`.gnu.linkonce.t.*`" symbol was defined in a discarded section. You can safely ignore these messages as they are erroneous and the linkage is correct. These messages disappear using ld 2.17.

GNU binutils 2.17: Binutils 2.17 contains a bug which causes huge link times (minutes instead of seconds) when building LLVM. We recommend upgrading to a newer version (2.17.50.0.4 or later).

GNU Binutils 2.19.1 Gold: This version of Gold contained a bug which causes intermittent failures when building LLVM with position independent code. The symptom is an error about cyclic dependencies. We recommend upgrading to a newer version of Gold.

Clang 3.0 with libstdc++ 4.7.x: a few Linux distributions (Ubuntu 12.10, Fedora 17) have both Clang 3.0 and libstdc++ 4.7 in their repositories. Clang 3.0 does not implement a few builtins that are used in this library. We recommend using the system GCC to compile LLVM and Clang in this case.

Clang 3.0 on Mageia 2. There's a packaging issue: Clang can not find at least some (`cxxabi.h`) libstdc++ headers.

Clang in C++11 mode and libstdc++ 4.7.2. This version of libstdc++ contained a bug which causes Clang to refuse to compile condition_variable header file. At the time of writing, this breaks LLD build.

Getting a Modern Host C++ Toolchain

This section mostly applies to Linux and older BSDs. On Mac OS X, you should have a sufficiently modern Xcode, or you will likely need to upgrade until you do. On Windows, just use Visual Studio 2012 as the host compiler, it is explicitly supported and widely available. FreeBSD 10.0 and newer have a modern Clang as the system compiler.

However, some Linux distributions and some other or older BSDs sometimes have extremely old versions of GCC. These steps attempt to help you upgrade you compiler even on such a system. However, if at all possible, we encourage you to use a recent version of a distribution with a modern system compiler that meets these requirements. Note that

it is tempting to to install a prior version of Clang and libc++ to be the host compiler, however libc++ was not well tested or set up to build on Linux until relatively recently. As a consequence, this guide suggests just using libstdc++ and a modern GCC as the initial host in a bootstrap, and then using Clang (and potentially libc++).

The first step is to get a recent GCC toolchain installed. The most common distribution on which users have struggled with the version requirements is Ubuntu Precise, 12.04 LTS. For this distribution, one easy option is to install the toolchain testing PPA and use it to install a modern GCC. There is a really nice discussions of this on the ask ubuntu stack exchange. However, not all users can use PPAs and there are many other distributions, so it may be necessary (or just useful, if you're here you *are* doing compiler development after all) to build and install GCC from source. It is also quite easy to do these days.

Easy steps for installing GCC 4.8.2:

```
% wget ftp://ftp.gnu.org/gnu/gcc/gcc-4.8.2/gcc-4.8.2.tar.bz2
% tar -xvjf gcc-4.8.2.tar.bz2
% cd gcc-4.8.2
% ./contrib/download_prerequisites
% cd ..
% mkdir gcc-4.8.2-build
% cd gcc-4.8.2-build
% $PWD/../gcc-4.8.2/configure --prefix=$HOME/toolchains --enable-languages=c,c++
% make -j$(nproc)
% make install
```

For more details, check out the excellent GCC wiki entry, where I got most of this information from.

Once you have a GCC toolchain, configure your build of LLVM to use the new toolchain for your host compiler and C++ standard library. Because the new version of libstdc++ is not on the system library search path, you need to pass extra linker flags so that it can be found at link time (`-L`) and at runtime (`-rpath`). If you are using CMake, this invocation should produce working binaries:

```
% mkdir build
% cd build
% CC=$HOME/toolchains/bin/gcc CXX=$HOME/toolchains/bin/g++ \
  cmake .. -DCMAKE_CXX_LINK_FLAGS="-Wl,-rpath,$HOME/toolchains/lib64 -L$HOME/toolchains/lib64"
```

If you fail to set rpath, most LLVM binaries will fail on startup with a message from the loader similar to `libstdc++.so.6: version 'GLIBCXX_3.4.20' not found`. This means you need to tweak the -rpath linker flag.

When you build Clang, you will need to give *it* access to modern C++11 standard library in order to use it as your new host in part of a bootstrap. There are two easy ways to do this, either build (and install) libc++ along with Clang and then use it with the `-stdlib=libc++` compile and link flag, or install Clang into the same prefix (`$HOME/toolchains` above) as GCC. Clang will look within its own prefix for libstdc++ and use it if found. You can also add an explicit prefix for Clang to look in for a GCC toolchain with the `--gcc-toolchain=/opt/my/gcc/prefix` flag, passing it to both compile and link commands when using your just-built-Clang to bootstrap.

2.5.4 Getting Started with LLVM

The remainder of this guide is meant to get you up and running with LLVM and to give you some basic information about the LLVM environment.

The later sections of this guide describe the general layout of the LLVM source tree, a simple example using the LLVM tool chain, and links to find more information about LLVM or to get help via e-mail.

Terminology and Notation

Throughout this manual, the following names are used to denote paths specific to the local system and working environment. *These are not environment variables you need to set but just strings used in the rest of this document below*. In any of the examples below, simply replace each of these names with the appropriate pathname on your local system. All these paths are absolute:

`SRC_ROOT`

This is the top level directory of the LLVM source tree.

`OBJ_ROOT`

This is the top level directory of the LLVM object tree (i.e. the tree where object files and compiled programs will be placed. It can be the same as SRC_ROOT).

Setting Up Your Environment

In order to compile and use LLVM, you may need to set some environment variables.

`LLVM_LIB_SEARCH_PATH=/path/to/your/bitcode/libs`

[Optional] This environment variable helps LLVM linking tools find the locations of your bitcode libraries. It is provided only as a convenience since you can specify the paths using the -L options of the tools and the C/C++ front-end will automatically use the bitcode files installed in its `lib` directory.

Unpacking the LLVM Archives

If you have the LLVM distribution, you will need to unpack it before you can begin to compile it. LLVM is distributed as a set of two files: the LLVM suite and the LLVM GCC front end compiled for your platform. There is an additional test suite that is optional. Each file is a TAR archive that is compressed with the gzip program.

The files are as follows, with *x.y* marking the version number:

`llvm-x.y.tar.gz`

Source release for the LLVM libraries and tools.

`llvm-test-x.y.tar.gz`

Source release for the LLVM test-suite.

Checkout LLVM from Subversion

If you have access to our Subversion repository, you can get a fresh copy of the entire source code. All you need to do is check it out from Subversion as follows:

- `cd where-you-want-llvm-to-live`
- Read-Only: `svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
- Read-Write:`svn co https://user@llvm.org/svn/llvm-project/llvm/trunk llvm`

This will create an '`llvm`' directory in the current directory and fully populate it with the LLVM source code, Makefiles, test directories, and local copies of documentation files.

If you want to get a specific release (as opposed to the most recent revision), you can checkout it from the '`tags`' directory (instead of '`trunk`'). The following releases are located in the following subdirectories of the '`tags`' directory:

- Release 3.4: **RELEASE_34/final**
- Release 3.3: **RELEASE_33/final**
- Release 3.2: **RELEASE_32/final**
- Release 3.1: **RELEASE_31/final**
- Release 3.0: **RELEASE_30/final**
- Release 2.9: **RELEASE_29/final**
- Release 2.8: **RELEASE_28**
- Release 2.7: **RELEASE_27**
- Release 2.6: **RELEASE_26**
- Release 2.5: **RELEASE_25**
- Release 2.4: **RELEASE_24**
- Release 2.3: **RELEASE_23**
- Release 2.2: **RELEASE_22**
- Release 2.1: **RELEASE_21**
- Release 2.0: **RELEASE_20**
- Release 1.9: **RELEASE_19**
- Release 1.8: **RELEASE_18**
- Release 1.7: **RELEASE_17**
- Release 1.6: **RELEASE_16**
- Release 1.5: **RELEASE_15**
- Release 1.4: **RELEASE_14**
- Release 1.3: **RELEASE_13**
- Release 1.2: **RELEASE_12**
- Release 1.1: **RELEASE_11**
- Release 1.0: **RELEASE_1**

If you would like to get the LLVM test suite (a separate package as of 1.4), you get it from the Subversion repository:

```
% cd llvm/projects
% svn co http://llvm.org/svn/llvm-project/test-suite/trunk test-suite
```

By placing it in the `llvm/projects`, it will be automatically configured by the LLVM configure script as well as automatically updated when you run `svn update`.

Git Mirror

Git mirrors are available for a number of LLVM subprojects. These mirrors sync automatically with each Subversion commit and contain all necessary git-svn marks (so, you can recreate git-svn metadata locally). Note that right now mirrors reflect only `trunk` for each project. You can do the read-only Git clone of LLVM via:

```
% git clone http://llvm.org/git/llvm.git
```

If you want to check out clang too, run:

```
% cd llvm/tools
% git clone http://llvm.org/git/clang.git
```

If you want to check out compiler-rt too, run:

```
% cd llvm/projects
% git clone http://llvm.org/git/compiler-rt.git
```

If you want to check out the Test Suite Source Code (optional), run:

```
% cd llvm/projects
% git clone http://llvm.org/git/test-suite.git
```

Since the upstream repository is in Subversion, you should use `git pull --rebase` instead of `git pull` to avoid generating a non-linear history in your clone. To configure `git pull` to pass `--rebase` by default on the master branch, run the following command:

```
% git config branch.master.rebase true
```

Sending patches with Git

Please read Developer Policy, too.

Assume `master` points the upstream and `mybranch` points your working branch, and `mybranch` is rebased onto `master`. At first you may check sanity of whitespaces:

```
% git diff --check master..mybranch
```

The easiest way to generate a patch is as below:

```
% git diff master..mybranch > /path/to/mybranch.diff
```

It is a little different from svn-generated diff. git-diff-generated diff has prefixes like `a/` and `b/`. Don't worry, most developers might know it could be accepted with `patch -p1 -N`.

But you may generate patchset with git-format-patch. It generates by-each-commit patchset. To generate patch files to attach to your article:

```
% git format-patch --no-attach master..mybranch -o /path/to/your/patchset
```

If you would like to send patches directly, you may use git-send-email or git-imap-send. Here is an example to generate the patchset in Gmail's [Drafts].

```
% git format-patch --attach master..mybranch --stdout | git imap-send
```

Then, your .git/config should have [imap] sections.

```
[imap]
      host = imaps://imap.gmail.com
      user = your.gmail.account@gmail.com
      pass = himitsu!
      port = 993
      sslverify = false
; in English
      folder = "[Gmail]/Drafts"
; example for Japanese, "Modified UTF-7" encoded.
      folder = "[Gmail]/&Tgtm+DBN-"
; example for Traditional Chinese
      folder = "[Gmail]/&g0l6Pw-"
```

For developers to work with git-svn

To set up clone from which you can submit code using `git-svn`, run:

```
% git clone http://llvm.org/git/llvm.git
% cd llvm
% git svn init https://llvm.org/svn/llvm-project/llvm/trunk --username=<username>
% git config svn-remote.svn.fetch :refs/remotes/origin/master
% git svn rebase -l  # -l avoids fetching ahead of the git mirror.

# If you have clang too:
% cd tools
% git clone http://llvm.org/git/clang.git
% cd clang
% git svn init https://llvm.org/svn/llvm-project/cfe/trunk --username=<username>
% git config svn-remote.svn.fetch :refs/remotes/origin/master
% git svn rebase -l
```

Likewise for compiler-rt and test-suite.

To update this clone without generating git-svn tags that conflict with the upstream Git repo, run:

```
% git fetch && (cd tools/clang && git fetch)  # Get matching revisions of both trees.
% git checkout master
% git svn rebase -l
% (cd tools/clang &&
   git checkout master &&
   git svn rebase -l)
```

Likewise for compiler-rt and test-suite.

This leaves your working directories on their master branches, so you'll need to `checkout` each working branch individually and `rebase` it on top of its parent branch.

For those who wish to be able to update an llvm repo/revert patches easily using git-svn, please look in the directory for the scripts `git-svnup` and `git-svnrevert`.

To perform the aforementioned update steps go into your source directory and just type `git-svnup` or `git svnup` and everything will just work.

If one wishes to revert a commit with git-svn, but do not want the git hash to escape into the commit message, one can use the script `git-svnrevert` or `git svnrevert` which will take in the git hash for the commit you want to revert, look up the appropriate svn revision, and output a message where all references to the git hash have been replaced with the svn revision.

To commit back changes via git-svn, use `git svn dcommit`:

```
% git svn dcommit
```

Note that git-svn will create one SVN commit for each Git commit you have pending, so squash and edit each commit before executing `dcommit` to make sure they all conform to the coding standards and the developers' policy.

On success, `dcommit` will rebase against the HEAD of SVN, so to avoid conflict, please make sure your current branch is up-to-date (via fetch/rebase) before proceeding.

The git-svn metadata can get out of sync after you mess around with branches and `dcommit`. When that happens, `git svn dcommit` stops working, complaining about files with uncommitted changes. The fix is to rebuild the metadata:

```
% rm -rf .git/svn
% git svn rebase -l
```

Please, refer to the Git-SVN manual (`man git-svn`) for more information.

Local LLVM Configuration

Once checked out from the Subversion repository, the LLVM suite source code must be configured via the `configure` script. This script sets variables in the various `*.in` files, most notably `llvm/Makefile.config` and `llvm/include/Config/config.h`. It also populates *OBJ_ROOT* with the Makefiles needed to begin building LLVM.

The following environment variables are used by the `configure` script to configure the build system:

Variable	Purpose
CC	Tells `configure` which C compiler to use. By default, `configure` will check `PATH` for `clang` and GCC C compilers (in this order). Use this variable to override `configure`'s default behavior.
CXX	Tells `configure` which C++ compiler to use. By default, `configure` will check `PATH` for `clang++` and GCC C++ compilers (in this order). Use this variable to override `configure`'s default behavior.

The following options can be used to set or enable LLVM specific options:

`--enable-optimized`

Enables optimized compilation (debugging symbols are removed and GCC optimization flags are enabled). Note that this is the default setting if you are using the LLVM distribution. The default behavior of a Subversion checkout is to use an unoptimized build (also known as a debug build).

`--enable-debug-runtime`

Enables debug symbols in the runtime libraries. The default is to strip debug symbols from the runtime libraries.

`--enable-jit`

Compile the Just In Time (JIT) compiler functionality. This is not available on all platforms. The default is dependent on platform, so it is best to explicitly enable it if you want it.

`--enable-targets=target-option`

Controls which targets will be built and linked into llc. The default value for `target_options` is "all" which builds and links all available targets. The "host" target is selected as the target of the build host. You can also specify a comma separated list of target names that you want available in llc. The target names use all lower case. The current set of targets is:

```
aarch64, arm, arm64, cpp, hexagon, mips, mipsel, mips64,
mips64el, msp430, powerpc, nvptx, r600, sparc, systemz, x86,
x86_64, xcore.
```

`--enable-doxygen`

Look for the doxygen program and enable construction of doxygen based documentation from the source code. This is disabled by default because generating the documentation can take a long time and producess 100s of megabytes of output.

To configure LLVM, follow these steps:

1. Change directory into the object root directory:

   ```
   % cd OBJ_ROOT
   ```

2. Run the `configure` script located in the LLVM source tree:

```
% SRC_ROOT/configure --prefix=/install/path [other options]
```

Compiling the LLVM Suite Source Code

Once you have configured LLVM, you can build it. There are three types of builds:

Debug Builds

> These builds are the default when one is using a Subversion checkout and types `gmake` (unless the `--enable-optimized` option was used during configuration). The build system will compile the tools and libraries with debugging information. To get a Debug Build using the LLVM distribution the `--disable-optimized` option must be passed to `configure`.

Release (Optimized) Builds

> These builds are enabled with the `--enable-optimized` option to `configure` or by specifying `ENABLE_OPTIMIZED=1` on the `gmake` command line. For these builds, the build system will compile the tools and libraries with GCC optimizations enabled and strip debugging information from the libraries and executables it generates. Note that Release Builds are default when using an LLVM distribution.

Profile Builds

> These builds are for use with profiling. They compile profiling information into the code for use with programs like `gprof`. Profile builds must be started by specifying `ENABLE_PROFILING=1` on the `gmake` command line.

Once you have LLVM configured, you can build it by entering the *OBJ_ROOT* directory and issuing the following command:

```
% gmake
```

If the build fails, please check here to see if you are using a version of GCC that is known not to compile LLVM.

If you have multiple processors in your machine, you may wish to use some of the parallel build options provided by GNU Make. For example, you could use the command:

```
% gmake -j2
```

There are several special targets which are useful when working with the LLVM source code:

`gmake clean`

> Removes all files generated by the build. This includes object files, generated C/C++ files, libraries, and executables.

`gmake dist-clean`

> Removes everything that `gmake clean` does, but also removes files generated by `configure`. It attempts to return the source tree to the original state in which it was shipped.

`gmake install`

> Installs LLVM header files, libraries, tools, and documentation in a hierarchy under `$PREFIX`, specified with `./configure --prefix=[dir]`, which defaults to `/usr/local`.

`gmake -C runtime install-bytecode`

> Assuming you built LLVM into $OBJDIR, when this command is run, it will install bitcode libraries into the GCC front end's bitcode library directory. If you need to update your bitcode libraries, this is the target to use once you've built them.

Please see the Makefile Guide for further details on these `make` targets and descriptions of other targets available.

It is also possible to override default values from `configure` by declaring variables on the command line. The following are some examples:

`gmake ENABLE_OPTIMIZED=1`

Perform a Release (Optimized) build.

`gmake ENABLE_OPTIMIZED=1 DISABLE_ASSERTIONS=1`

Perform a Release (Optimized) build without assertions enabled.

`gmake ENABLE_OPTIMIZED=0`

Perform a Debug build.

`gmake ENABLE_PROFILING=1`

Perform a Profiling build.

`gmake VERBOSE=1`

Print what `gmake` is doing on standard output.

`gmake TOOL_VERBOSE=1`

Ask each tool invoked by the makefiles to print out what it is doing on the standard output. This also implies `VERBOSE=1`.

Every directory in the LLVM object tree includes a `Makefile` to build it and any subdirectories that it contains. Entering any directory inside the LLVM object tree and typing `gmake` should rebuild anything in or below that directory that is out of date.

This does not apply to building the documentation. LLVM's (non-Doxygen) documentation is produced with the Sphinx documentation generation system. There are some HTML documents that have not yet been converted to the new system (which uses the easy-to-read and easy-to-write reStructuredText plaintext markup language). The generated documentation is built in the `SRC_ROOT/docs` directory using a special makefile. For instructions on how to install Sphinx, see Sphinx Introduction for LLVM Developers. After following the instructions there for installing Sphinx, build the LLVM HTML documentation by doing the following:

```
$ cd SRC_ROOT/docs
$ make -f Makefile.sphinx
```

This creates a `_build/html` sub-directory with all of the HTML files, not just the generated ones. This directory corresponds to `llvm.org/docs`. For example, `_build/html/SphinxQuickstartTemplate.html` corresponds to `llvm.org/docs/SphinxQuickstartTemplate.html`. The *Sphinx Quickstart Template* is useful when creating a new document.

Cross-Compiling LLVM

It is possible to cross-compile LLVM itself. That is, you can create LLVM executables and libraries to be hosted on a platform different from the platform where they are built (a Canadian Cross build). To configure a cross-compile, supply the configure script with `--build` and `--host` options that are different. The values of these options must be legal target triples that your GCC compiler supports.

The result of such a build is executables that are not runnable on on the build host (–build option) but can be executed on the compile host (–host option).

Check *How To Cross-Compile Clang/LLVM using Clang/LLVM* and Clang docs on how to cross-compile in general for more information about cross-compiling.

The Location of LLVM Object Files

The LLVM build system is capable of sharing a single LLVM source tree among several LLVM builds. Hence, it is possible to build LLVM for several different platforms or configurations using the same source tree.

This is accomplished in the typical autoconf manner:

- Change directory to where the LLVM object files should live:

```
% cd OBJ_ROOT
```

- Run the `configure` script found in the LLVM source directory:

```
% SRC_ROOT/configure
```

The LLVM build will place files underneath *OBJ_ROOT* in directories named after the build type:

Debug Builds with assertions enabled (the default)

Tools

`OBJ_ROOT/Debug+Asserts/bin`

Libraries

`OBJ_ROOT/Debug+Asserts/lib`

Release Builds

Tools

`OBJ_ROOT/Release/bin`

Libraries

`OBJ_ROOT/Release/lib`

Profile Builds

Tools

`OBJ_ROOT/Profile/bin`

Libraries

`OBJ_ROOT/Profile/lib`

Optional Configuration Items

If you're running on a Linux system that supports the binfmt_misc module, and you have root access on the system, you can set your system up to execute LLVM bitcode files directly. To do this, use commands like this (the first command may not be required if you are already using the module):

```
% mount -t binfmt_misc none /proc/sys/fs/binfmt_misc
% echo ':llvm:M::BC::/path/to/lli:' > /proc/sys/fs/binfmt_misc/register
% chmod u+x hello.bc   (if needed)
% ./hello.bc
```

This allows you to execute LLVM bitcode files directly. On Debian, you can also use this command instead of the 'echo' command above:

```
% sudo update-binfmts --install llvm /path/to/lli --magic 'BC'
```

2.5.5 Program Layout

One useful source of information about the LLVM source base is the LLVM doxygen documentation available at http://llvm.org/doxygen/. The following is a brief introduction to code layout:

llvm/examples

This directory contains some simple examples of how to use the LLVM IR and JIT.

llvm/include

This directory contains public header files exported from the LLVM library. The three main subdirectories of this directory are:

`llvm/include/llvm`

> This directory contains all of the LLVM specific header files. This directory also has subdirectories for different portions of LLVM: `Analysis`, `CodeGen`, `Target`, `Transforms`, etc...

`llvm/include/llvm/Support`

> This directory contains generic support libraries that are provided with LLVM but not necessarily specific to LLVM. For example, some C++ STL utilities and a Command Line option processing library store their header files here.

`llvm/include/llvm/Config`

> This directory contains header files configured by the `configure` script. They wrap "standard" UNIX and C header files. Source code can include these header files which automatically take care of the conditional #includes that the `configure` script generates.

llvm/lib

This directory contains most of the source files of the LLVM system. In LLVM, almost all code exists in libraries, making it very easy to share code among the different tools.

`llvm/lib/VMCore/`

> This directory holds the core LLVM source files that implement core classes like Instruction and BasicBlock.

`llvm/lib/AsmParser/`

> This directory holds the source code for the LLVM assembly language parser library.

`llvm/lib/Bitcode/`

> This directory holds code for reading and write LLVM bitcode.

`llvm/lib/Analysis/`

> This directory contains a variety of different program analyses, such as Dominator Information, Call Graphs, Induction Variables, Interval Identification, Natural Loop Identification, etc.

`llvm/lib/Transforms/`

> This directory contains the source code for the LLVM to LLVM program transformations, such as Aggressive Dead Code Elimination, Sparse Conditional Constant Propagation, Inlining, Loop Invariant Code Motion, Dead Global Elimination, and many others.

`llvm/lib/Target/`

This directory contains files that describe various target architectures for code generation. For example, the `llvm/lib/Target/X86` directory holds the X86 machine description while `llvm/lib/Target/ARM` implements the ARM backend.

`llvm/lib/CodeGen/`

This directory contains the major parts of the code generator: Instruction Selector, Instruction Scheduling, and Register Allocation.

`llvm/lib/MC/`

(FIXME: T.B.D.)

`llvm/lib/Debugger/`

This directory contains the source level debugger library that makes it possible to instrument LLVM programs so that a debugger could identify source code locations at which the program is executing.

`llvm/lib/ExecutionEngine/`

This directory contains libraries for executing LLVM bitcode directly at runtime in both interpreted and JIT compiled fashions.

`llvm/lib/Support/`

This directory contains the source code that corresponds to the header files located in `llvm/include/ADT/` and `llvm/include/Support/`.

llvm/projects

This directory contains projects that are not strictly part of LLVM but are shipped with LLVM. This is also the directory where you should create your own LLVM-based projects.

llvm/runtime

This directory contains libraries which are compiled into LLVM bitcode and used when linking programs with the Clang front end. Most of these libraries are skeleton versions of real libraries; for example, libc is a stripped down version of glibc.

Unlike the rest of the LLVM suite, this directory needs the LLVM GCC front end to compile.

llvm/test

This directory contains feature and regression tests and other basic sanity checks on the LLVM infrastructure. These are intended to run quickly and cover a lot of territory without being exhaustive.

test-suite

This is not a directory in the normal llvm module; it is a separate Subversion module that must be checked out (usually to `projects/test-suite`). This module contains a comprehensive correctness, performance, and benchmarking test suite for LLVM. It is a separate Subversion module because not every LLVM user is interested in downloading or building such a comprehensive test suite. For further details on this test suite, please see the *Testing Guide* document.

llvm/tools

The **tools** directory contains the executables built out of the libraries above, which form the main part of the user interface. You can always get help for a tool by typing `tool_name -help`. The following is a brief introduction to the most important tools. More detailed information is in the Command Guide.

`bugpoint`

> `bugpoint` is used to debug optimization passes or code generation backends by narrowing down the given test case to the minimum number of passes and/or instructions that still cause a problem, whether it is a crash or miscompilation. See HowToSubmitABug.html for more information on using `bugpoint`.

`llvm-ar`

> The archiver produces an archive containing the given LLVM bitcode files, optionally with an index for faster lookup.

`llvm-as`

> The assembler transforms the human readable LLVM assembly to LLVM bitcode.

`llvm-dis`

> The disassembler transforms the LLVM bitcode to human readable LLVM assembly.

`llvm-link`

> `llvm-link`, not surprisingly, links multiple LLVM modules into a single program.

`lli`

> `lli` is the LLVM interpreter, which can directly execute LLVM bitcode (although very slowly...). For architectures that support it (currently x86, Sparc, and PowerPC), by default, `lli` will function as a Just-In-Time compiler (if the functionality was compiled in), and will execute the code *much* faster than the interpreter.

`llc`

> `llc` is the LLVM backend compiler, which translates LLVM bitcode to a native code assembly file or to C code (with the `-march=c` option).

`opt`

> `opt` reads LLVM bitcode, applies a series of LLVM to LLVM transformations (which are specified on the command line), and then outputs the resultant bitcode. The '`opt -help`' command is a good way to get a list of the program transformations available in LLVM.
>
> `opt` can also be used to run a specific analysis on an input LLVM bitcode file and print out the results. It is primarily useful for debugging analyses, or familiarizing yourself with what an analysis does.

llvm/utils

This directory contains utilities for working with LLVM source code, and some of the utilities are actually required as part of the build process because they are code generators for parts of LLVM infrastructure.

`codegen-diff`

> `codegen-diff` is a script that finds differences between code that LLC generates and code that LLI generates. This is a useful tool if you are debugging one of them, assuming that the other generates correct output. For the full user manual, run `'perldoc codegen-diff'`.

`emacs/`

The `emacs` directory contains syntax-highlighting files which will work with Emacs and XEmacs editors, providing syntax highlighting support for LLVM assembly files and TableGen description files. For information on how to use the syntax files, consult the `README` file in that directory.

`getsrcs.sh`

The `getsrcs.sh` script finds and outputs all non-generated source files, which is useful if one wishes to do a lot of development across directories and does not want to individually find each file. One way to use it is to run, for example: `xemacs 'utils/getsources.sh'` from the top of your LLVM source tree.

`llvmgrep`

This little tool performs an `egrep -H -n` on each source file in LLVM and passes to it a regular expression provided on `llvmgrep`'s command line. This is a very efficient way of searching the source base for a particular regular expression.

`makellvm`

The `makellvm` script compiles all files in the current directory and then compiles and links the tool that is the first argument. For example, assuming you are in the directory `llvm/lib/Target/Sparc`, if `makellvm` is in your path, simply running `makellvm llc` will make a build of the current directory, switch to directory `llvm/tools/llc` and build it, causing a re-linking of LLC.

`TableGen/`

The `TableGen` directory contains the tool used to generate register descriptions, instruction set descriptions, and even assemblers from common TableGen description files.

`vim/`

The `vim` directory contains syntax-highlighting files which will work with the VIM editor, providing syntax highlighting support for LLVM assembly files and TableGen description files. For information on how to use the syntax files, consult the `README` file in that directory.

2.5.6 An Example Using the LLVM Tool Chain

This section gives an example of using LLVM with the Clang front end.

Example with clang

1. First, create a simple C file, name it 'hello.c':

```
#include <stdio.h>

int main() {
  printf("hello world\n");
  return 0;
}
```

2. Next, compile the C file into a native executable:

```
% clang hello.c -o hello
```

Note: Clang works just like GCC by default. The standard -S and -c arguments work as usual (producing a native .s or .o file, respectively).

---

3. Next, compile the C file into an LLVM bitcode file:

```
% clang -O3 -emit-llvm hello.c -c -o hello.bc
```

The -emit-llvm option can be used with the -S or -c options to emit an LLVM `.ll` or `.bc` file (respectively) for the code. This allows you to use the standard LLVM tools on the bitcode file.

4. Run the program in both forms. To run the program, use:

   ```
   % ./hello
   ```

   and

   ```
   % lli hello.bc
   ```

   The second examples shows how to invoke the LLVM JIT, *lli*.

5. Use the `llvm-dis` utility to take a look at the LLVM assembly code:

   ```
   % llvm-dis < hello.bc | less
   ```

6. Compile the program to native assembly using the LLC code generator:

   ```
   % llc hello.bc -o hello.s
   ```

7. Assemble the native assembly language file into a program:

   ```
   % /opt/SUNWspro/bin/cc -xarch=v9 hello.s -o hello.native   # On Solaris

   % gcc hello.s -o hello.native                              # On others
   ```

8. Execute the native code program:

   ```
   % ./hello.native
   ```

   Note that using clang to compile directly to native code (i.e. when the `-emit-llvm` option is not present) does steps 6/7/8 for you.

2.5.7 Common Problems

If you are having problems building or using LLVM, or if you have any other general questions about LLVM, please consult the Frequently Asked Questions page.

2.5.8 Links

This document is just an **introduction** on how to use LLVM to do some simple things... there are many more interesting and complicated things that you can do that aren't documented here (but we'll gladly accept a patch if you want to write something up!). For more information about LLVM, check out:

- LLVM Homepage
- LLVM Doxygen Tree
- Starting a Project that Uses LLVM

2.6 Getting Started with the LLVM System using Microsoft Visual Studio

2.6.1 Overview

Welcome to LLVM on Windows! This document only covers LLVM on Windows using Visual Studio, not mingw or cygwin. In order to get started, you first need to know some basic information.

There are many different projects that compose LLVM. The first is the LLVM suite. This contains all of the tools, libraries, and header files needed to use LLVM. It contains an assembler, disassembler, bitcode analyzer and bitcode optimizer. It also contains a test suite that can be used to test the LLVM tools.

Another useful project on Windows is Clang. Clang is a C family ([Objective]C/C++) compiler. Clang mostly works on Windows, but does not currently understand all of the Microsoft extensions to C and C++. Because of this, clang cannot parse the C++ standard library included with Visual Studio, nor parts of the Windows Platform SDK. However, most standard C programs do compile. Clang can be used to emit bitcode, directly emit object files or even linked executables using Visual Studio's `link.exe`.

The large LLVM test suite cannot be run on the Visual Studio port at this time.

Most of the tools build and work. `bugpoint` does build, but does not work.

Additional information about the LLVM directory structure and tool chain can be found on the main *Getting Started with the LLVM System* page.

2.6.2 Requirements

Before you begin to use the LLVM system, review the requirements given below. This may save you some trouble by knowing ahead of time what hardware and software you will need.

Hardware

Any system that can adequately run Visual Studio 2012 is fine. The LLVM source tree and object files, libraries and executables will consume approximately 3GB.

Software

You will need Visual Studio 2012 or higher.

You will also need the CMake build system since it generates the project files you will use to build with.

If you would like to run the LLVM tests you will need Python. Versions 2.4-2.7 are known to work. You will need GnuWin32 tools, too.

Do not install the LLVM directory tree into a path containing spaces (e.g. `C:\Documents and Settings\...`) as the configure step will fail.

2.6.3 Getting Started

Here's the short story for getting up and running quickly with LLVM:

1. Read the documentation.
2. Seriously, read the documentation.
3. Remember that you were warned twice about reading the documentation.
4. Get the Source Code
   - With the distributed files:
     (a) `cd <where-you-want-llvm-to-live>`
     (b) `gunzip --stdout llvm-VERSION.tar.gz | tar -xvf -` (*or use WinZip*)
     (c) `cd llvm`
   - With anonymous Subversion access:
     (a) `cd <where-you-want-llvm-to-live>`
     (b) `svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
     (c) `cd llvm`
5. Use CMake to generate up-to-date project files:
   - Once CMake is installed then the simplest way is to just start the CMake GUI, select the directory where you have LLVM extracted to, and the default options should all be fine. One option you may really want to change, regardless of anything else, might be the `CMAKE_INSTALL_PREFIX` setting to select a directory to INSTALL to once compiling is complete, although installation is not mandatory for using LLVM. Another important option is `LLVM_TARGETS_TO_BUILD`, which controls the LLVM target architectures that are included on the build.
   - See the *LLVM CMake guide* for detailed information about how to configure the LLVM build.
   - CMake generates project files for all build types. To select a specific build type, use the Configuration manager from the VS IDE or the `/property:Configuration` command line option when using MSBuild.
6. Start Visual Studio
   - In the directory you created the project files will have an `llvm.sln` file, just double-click on that to open Visual Studio.
7. Build the LLVM Suite:
   - The projects may still be built individually, but to build them all do not just select all of them in batch build (as some are meant as configuration projects), but rather select and build just the `ALL_BUILD` project to build everything, or the `INSTALL` project, which first builds the `ALL_BUILD` project, then installs the LLVM headers, libs, and other useful things to the directory set by the `CMAKE_INSTALL_PREFIX` setting when you first configured CMake.
   - The Fibonacci project is a sample program that uses the JIT. Modify the project's debugging properties to provide a numeric command line argument or run it from the command line. The program will print the corresponding fibonacci value.
8. Test LLVM in Visual Studio:
   - If `%PATH%` does not contain GnuWin32, you may specify `LLVM_LIT_TOOLS_DIR` on CMake for the path to GnuWin32.

- You can run LLVM tests by merely building the project "check". The test results will be shown in the VS output window.

9. Test LLVM on the command line:

   - The LLVM tests can be run by changing directory to the llvm source directory and running:

     ```
     C:\..\llvm> python ..\build\bin\llvm-lit --param build_config=Win32 --param build_mode=Debug
     ```

     This example assumes that Python is in your PATH variable, you have built a Win32 Debug version of llvm with a standard out of line build. You should not see any unexpected failures, but will see many unsupported tests and expected failures.

     A specific test or test directory can be run with:

     ```
     C:\..\llvm> python ..\build\bin\llvm-lit --param build_config=Win32 --param build_mode=Debug
     ```

2.6.4 An Example Using the LLVM Tool Chain

1. First, create a simple C file, name it '`hello.c`':

   ```
   #include <stdio.h>
   int main() {
     printf("hello world\n");
     return 0;
   }
   ```

2. Next, compile the C file into an LLVM bitcode file:

   ```
   C:\..> clang -c hello.c -emit-llvm -o hello.bc
   ```

   This will create the result file `hello.bc` which is the LLVM bitcode that corresponds the compiled program and the library facilities that it required. You can execute this file directly using `lli` tool, compile it to native assembly with the `llc`, optimize or analyze it further with the `opt` tool, etc.

   Alternatively you can directly output an executable with clang with:

   ```
   C:\..> clang hello.c -o hello.exe
   ```

   The `-o hello.exe` is required because clang currently outputs `a.out` when neither `-o` nor `-c` are given.

3. Run the program using the just-in-time compiler:

   ```
   C:\..> lli hello.bc
   ```

4. Use the `llvm-dis` utility to take a look at the LLVM assembly code:

   ```
   C:\..> llvm-dis < hello.bc | more
   ```

5. Compile the program to object code using the LLC code generator:

   ```
   C:\..> llc -filetype=obj hello.bc
   ```

6. Link to binary using Microsoft link:

   ```
   C:\..> link hello.obj -defaultlib:libcmt
   ```

7. Execute the native code program:

   ```
   C:\..> hello.exe
   ```

2.6.5 Common Problems

If you are having problems building or using LLVM, or if you have any other general questions about LLVM, please consult the *Frequently Asked Questions* page.

2.6.6 Links

This document is just an **introduction** to how to use LLVM to do some simple things... there are many more interesting and complicated things that you can do that aren't documented here (but we'll gladly accept a patch if you want to write something up!). For more information about LLVM, check out:

- LLVM homepage
- LLVM doxygen tree

2.7 Frequently Asked Questions (FAQ)

2.7.1 License

Does the University of Illinois Open Source License really qualify as an "open source" license?

Yes, the license is certified by the Open Source Initiative (OSI).

Can I modify LLVM source code and redistribute the modified source?

Yes. The modified source distribution must retain the copyright notice and follow the three bulletted conditions listed in the LLVM license.

Can I modify the LLVM source code and redistribute binaries or other tools based on it, without redistributing the source?

Yes. This is why we distribute LLVM under a less restrictive license than GPL, as explained in the first question above.

2.7.2 Source Code

In what language is LLVM written?

All of the LLVM tools and libraries are written in C++ with extensive use of the STL.

How portable is the LLVM source code?

The LLVM source code should be portable to most modern Unix-like operating systems. Most of the code is written in standard C++ with operating system services abstracted to a support library. The tools required to build and test LLVM have been ported to a plethora of platforms.

Some porting problems may exist in the following areas:

- The autoconf/makefile build system relies heavily on UNIX shell tools, like the Bourne Shell and sed. Porting to systems without these tools (MacOS 9, Plan 9) will require more effort.

What API do I use to store a value to one of the virtual registers in LLVM IR's SSA representation?

In short: you can't. It's actually kind of a silly question once you grok what's going on. Basically, in code like:

```
%result = add i32 %foo, %bar
```

, `%result` is just a name given to the `Value` of the `add` instruction. In other words, `%result` *is* the add instruction. The "assignment" doesn't explicitly "store" anything to any "virtual register"; the "=" is more like the mathematical sense of equality.

Longer explanation: In order to generate a textual representation of the IR, some kind of name has to be given to each instruction so that other instructions can textually reference it. However, the isomorphic in-memory representation that you manipulate from C++ has no such restriction since instructions can simply keep pointers to any other `Value`'s that they reference. In fact, the names of dummy numbered temporaries like `%1` are not explicitly represented in the in-memory representation at all (see `Value::getName()`).

2.7.3 Build Problems

When I run configure, it finds the wrong C compiler.

The `configure` script attempts to locate first `gcc` and then `cc`, unless it finds compiler paths set in `CC` and `CXX` for the C and C++ compiler, respectively.

If `configure` finds the wrong compiler, either adjust your `PATH` environment variable or set `CC` and `CXX` explicitly.

The `configure` script finds the right C compiler, but it uses the LLVM tools from a previous build. What do I do?

The `configure` script uses the `PATH` to find executables, so if it's grabbing the wrong linker/assembler/etc, there are two ways to fix it:

1. Adjust your PATH environment variable so that the correct program appears first in the PATH. This may work, but may not be convenient when you want them *first* in your path for other work.
2. Run configure with an alternative PATH that is correct. In a Bourne compatible shell, the syntax would be:

```
% PATH=[the path without the bad program] ./configure ...
```

This is still somewhat inconvenient, but it allows configure to do its work without having to adjust your PATH permanently.

When creating a dynamic library, I get a strange GLIBC error.

Under some operating systems (i.e. Linux), libtool does not work correctly if GCC was compiled with the --disable-shared option. To work around this, install your own version of GCC that has shared libraries enabled by default.

I've updated my source tree from Subversion, and now my build is trying to use a file/directory that doesn't exist.

You need to re-run configure in your object directory. When new Makefiles are added to the source tree, they have to be copied over to the object tree in order to be used by the build.

I've modified a Makefile in my source tree, but my build tree keeps using the old version. What do I do?

If the Makefile already exists in your object tree, you can just run the following command in the top level directory of your object tree:

```
% ./config.status <relative path to Makefile>;
```

If the Makefile is new, you will have to modify the configure script to copy it over.

I've upgraded to a new version of LLVM, and I get strange build errors.

Sometimes, changes to the LLVM source code alters how the build system works. Changes in libtool, autoconf, or header file dependencies are especially prone to this sort of problem.

The best thing to try is to remove the old files and re-build. In most cases, this takes care of the problem. To do this, just type make clean and then make in the directory that fails to build.

I've built LLVM and am testing it, but the tests freeze.

This is most likely occurring because you built a profile or release (optimized) build of LLVM and have not specified the same information on the gmake command line.

For example, if you built LLVM with the command:

```
% gmake ENABLE_PROFILING=1
```

...then you must run the tests with the following commands:

```
% cd llvm/test
% gmake ENABLE_PROFILING=1
```

Why do test results differ when I perform different types of builds?

The LLVM test suite is dependent upon several features of the LLVM tools and libraries.

First, the debugging assertions in code are not enabled in optimized or profiling builds. Hence, tests that used to fail may pass.

Second, some tests may rely upon debugging options or behavior that is only available in the debug build. These tests will fail in an optimized or profile build.

Compiling LLVM with GCC 3.3.2 fails, what should I do?

This is a bug in GCC, and affects projects other than LLVM. Try upgrading or downgrading your GCC.

After Subversion update, rebuilding gives the error "No rule to make target".

If the error is of the form:

```
gmake[2]: *** No rule to make target '/path/to/somefile',
              needed by '/path/to/another/file.d'.
Stop.
```

This may occur anytime files are moved within the Subversion repository or removed entirely. In this case, the best solution is to erase all `.d` files, which list dependencies for source files, and rebuild:

```
% cd $LLVM_OBJ_DIR
% rm -f `find . -name \*\.d`
% gmake
```

In other cases, it may be necessary to run `make clean` before rebuilding.

2.7.4 Source Languages

What source languages are supported?

LLVM currently has full support for C and C++ source languages. These are available through both Clang and DragonEgg.

The PyPy developers are working on integrating LLVM into the PyPy backend so that PyPy language can translate to LLVM.

I'd like to write a self-hosting LLVM compiler. How should I interface with the LLVM middle-end optimizers and back-end code generators?

Your compiler front-end will communicate with LLVM by creating a module in the LLVM intermediate representation (IR) format. Assuming you want to write your language's compiler in the language itself (rather than C++), there are 3 major ways to tackle generating LLVM IR from a front-end:

1. **Call into the LLVM libraries code using your language's FFI (foreign function interface).**

- *for:* best tracks changes to the LLVM IR, .ll syntax, and .bc format
- *for:* enables running LLVM optimization passes without a emit/parse overhead
- *for:* adapts well to a JIT context

- *against:* lots of ugly glue code to write

2. **Emit LLVM assembly from your compiler's native language.**

- *for:* very straightforward to get started
- *against:* the .ll parser is slower than the bitcode reader when interfacing to the middle end
- *against:* it may be harder to track changes to the IR

3. **Emit LLVM bitcode from your compiler's native language.**

- *for:* can use the more-efficient bitcode reader when interfacing to the middle end
- *against:* you'll have to re-engineer the LLVM IR object model and bitcode writer in your language
- *against:* it may be harder to track changes to the IR

If you go with the first option, the C bindings in include/llvm-c should help a lot, since most languages have strong support for interfacing with C. The most common hurdle with calling C from managed code is interfacing with the garbage collector. The C interface was designed to require very little memory management, and so is straightforward in this regard.

What support is there for a higher level source language constructs for building a compiler?

Currently, there isn't much. LLVM supports an intermediate representation which is useful for code representation but will not support the high level (abstract syntax tree) representation needed by most compilers. There are no facilities for lexical nor semantic analysis.

I don't understand the `GetElementPtr` instruction. Help!

See The Often Misunderstood GEP Instruction.

2.7.5 Using the C and C++ Front Ends

Can I compile C or C++ code to platform-independent LLVM bitcode?

No. C and C++ are inherently platform-dependent languages. The most obvious example of this is the preprocessor. A very common way that C code is made portable is by using the preprocessor to include platform-specific code. In practice, information about other platforms is lost after preprocessing, so the result is inherently dependent on the platform that the preprocessing was targeting.

Another example is `sizeof`. It's common for `sizeof(long)` to vary between platforms. In most C front-ends, `sizeof` is expanded to a constant immediately, thus hard-wiring a platform-specific detail.

Also, since many platforms define their ABIs in terms of C, and since LLVM is lower-level than C, front-ends currently must emit platform-specific IR in order to have the result conform to the platform ABI.

2.7.6 Questions about code generated by the demo page

What is this `llvm.global_ctors` and `_GLOBAL__I_a...` stuff that happens when I `#include <iostream>`?

If you `#include` the `<iostream>` header into a C++ translation unit, the file will probably use the `std::cin`/`std::cout`/... global objects. However, C++ does not guarantee an order of initialization between

static objects in different translation units, so if a static ctor/dtor in your .cpp file used `std::cout`, for example, the object would not necessarily be automatically initialized before your use.

To make `std::cout` and friends work correctly in these scenarios, the STL that we use declares a static object that gets created in every translation unit that includes `<iostream>`. This object has a static constructor and destructor that initializes and destroys the global iostream objects before they could possibly be used in the file. The code that you see in the `.ll` file corresponds to the constructor and destructor registration code.

If you would like to make it easier to *understand* the LLVM code generated by the compiler in the demo page, consider using `printf()` instead of `iostreams` to print values.

Where did all of my code go??

If you are using the LLVM demo page, you may often wonder what happened to all of the code that you typed in. Remember that the demo script is running the code through the LLVM optimizers, so if your code doesn't actually do anything useful, it might all be deleted.

To prevent this, make sure that the code is actually needed. For example, if you are computing some expression, return the value from the function instead of leaving it in a local variable. If you really want to constrain the optimizer, you can read from and assign to `volatile` global variables.

What is this "`undef`" thing that shows up in my code?

`undef` is the LLVM way of representing a value that is not defined. You can get these if you do not initialize a variable before you use it. For example, the C function:

```
int X() { int i; return i; }
```

Is compiled to "`ret i32 undef`" because "`i`" never has a value specified for it.

Why does instcombine + simplifycfg turn a call to a function with a mismatched calling convention into "unreachable"? Why not make the verifier reject it?

This is a common problem run into by authors of front-ends that are using custom calling conventions: you need to make sure to set the right calling convention on both the function and on each call to the function. For example, this code:

```
define fastcc void @foo() {
    ret void
}
define void @bar() {
    call void @foo()
    ret void
}
```

Is optimized to:

```
define fastcc void @foo() {
    ret void
}
define void @bar() {
    unreachable
}
```

... with "`opt -instcombine -simplifycfg`". This often bites people because "all their code disappears". Setting the calling convention on the caller and callee is required for indirect calls to work, so people often ask why not make the verifier reject this sort of thing.

The answer is that this code has undefined behavior, but it is not illegal. If we made it illegal, then every transformation that could potentially create this would have to ensure that it doesn't, and there is valid code that can create this sort of construct (in dead code). The sorts of things that can cause this to happen are fairly contrived, but we still need to accept them. Here's an example:

```
define fastcc void @foo() {
    ret void
}
define internal void @bar(void()* %FP, i1 %cond) {
    br i1 %cond, label %T, label %F
T:
    call void %FP()
    ret void
F:
    call fastcc void %FP()
    ret void
}
define void @test() {
    %X = or i1 false, false
    call void @bar(void()* @foo, i1 %X)
    ret void
}
```

In this example, "test" always passes `@foo`/`false` into `bar`, which ensures that it is dynamically called with the right calling conv (thus, the code is perfectly well defined). If you run this through the inliner, you get this (the explicit "or" is there so that the inliner doesn't dead code eliminate a bunch of stuff):

```
define fastcc void @foo() {
    ret void
}
define void @test() {
    %X = or i1 false, false
    br i1 %X, label %T.i, label %F.i
T.i:
    call void @foo()
    br label %bar.exit
F.i:
    call fastcc void @foo()
    br label %bar.exit
bar.exit:
    ret void
}
```

Here you can see that the inlining pass made an undefined call to `@foo` with the wrong calling convention. We really don't want to make the inliner have to know about this sort of thing, so it needs to be valid code. In this case, dead code elimination can trivially remove the undefined code. However, if `%X` was an input argument to `@test`, the inliner would produce this:

```
define fastcc void @foo() {
    ret void
}

define void @test(i1 %X) {
    br i1 %X, label %T.i, label %F.i
T.i:
```

```
    call void @foo()
    br label %bar.exit
F.i:
    call fastcc void @foo()
    br label %bar.exit
bar.exit:
    ret void
}
```

The interesting thing about this is that `%X` *must* be false for the code to be well-defined, but no amount of dead code elimination will be able to delete the broken call as unreachable. However, since `instcombine/simplifycfg` turns the undefined call into unreachable, we end up with a branch on a condition that goes to unreachable: a branch to unreachable can never happen, so "`-inline -instcombine -simplifycfg`" is able to produce:

```
define fastcc void @foo() {
   ret void
}
define void @test(i1 %X) {
F.i:
   call fastcc void @foo()
   ret void
}
```

2.8 The LLVM Lexicon

Note: This document is a work in progress!

2.8.1 Definitions

A

ADCE Aggressive Dead Code Elimination

AST Abstract Syntax Tree.

Due to Clang's influence (mostly the fact that parsing and semantic analysis are so intertwined for C and especially C++), the typical working definition of AST in the LLVM community is roughly "the compiler's first complete symbolic (as opposed to textual) representation of an input program". As such, an "AST" might be a more general graph instead of a "tree" (consider the symbolic representation for the type of a typical "linked list node"). This working definition is closer to what some authors call an "annotated abstract syntax tree".

Consult your favorite compiler book or search engine for more details.

B

BB Vectorization Basic-Block Vectorization

BURS Bottom Up Rewriting System — A method of instruction selection for code generation. An example is the BURG tool.

C

CSE Common Subexpression Elimination. An optimization that removes common subexpression compuation. For example `(a+b)*(a+b)` has two subexpressions that are the same: `(a+b)`. This optimization would perform the addition only once and then perform the multiply (but only if it's computationally correct/safe).

D

DAG Directed Acyclic Graph

Derived Pointer A pointer to the interior of an object, such that a garbage collector is unable to use the pointer for reachability analysis. While a derived pointer is live, the corresponding object pointer must be kept in a root, otherwise the collector might free the referenced object. With copying collectors, derived pointers pose an additional hazard that they may be invalidated at any safe point. This term is used in opposition to object pointer.

DSA Data Structure Analysis

DSE Dead Store Elimination

F

FCA First Class Aggregate

G

GC Garbage Collection. The practice of using reachability analysis instead of explicit memory management to reclaim unused memory.

H

Heap In garbage collection, the region of memory which is managed using reachability analysis.

I

IPA Inter-Procedural Analysis. Refers to any variety of code analysis that occurs between procedures, functions or compilation units (modules).

IPO Inter-Procedural Optimization. Refers to any variety of code optimization that occurs between procedures, functions or compilation units (modules).

ISel Instruction Selection

L

LCSSA Loop-Closed Static Single Assignment Form

LICM Loop Invariant Code Motion

Load-VN Load Value Numbering

LTO Link-Time Optimization

M

MC Machine Code

N

NFC "No functional change". Used in a commit message to indicate that a patch is a pure refactoring/cleanup. Usually used in the first line, so it is visible without opening the actual commit email.

O

Object Pointer A pointer to an object such that the garbage collector is able to trace references contained within the object. This term is used in opposition to derived pointer.

P

PRE Partial Redundancy Elimination

R

RAUW

Replace All Uses With. The functions `User::replaceUsesOfWith()`, `Value::replaceAllUsesWith()`, and `Constant::replaceUsesOfWithOnConstant()` implement the replacement of one Value with another by iterating over its def/use chain and fixing up all of the pointers to point to the new value. See also def/use chains.

Reassociation Rearranging associative expressions to promote better redundancy elimination and other optimization. For example, changing `(A+B-A)` into `(B+A-A)`, permitting it to be optimized into `(B+0)` then `(B)`.

Root In garbage collection, a pointer variable lying outside of the heap from which the collector begins its reachability analysis. In the context of code generation, "root" almost always refers to a "stack root" — a local or temporary variable within an executing function.

RPO Reverse postorder

S

Safe Point In garbage collection, it is necessary to identify stack roots so that reachability analysis may proceed. It may be infeasible to provide this information for every instruction, so instead the information may is calculated only at designated safe points. With a copying collector, derived pointers must not be retained across safe points and object pointers must be reloaded from stack roots.

SDISel Selection DAG Instruction Selection.

SCC Strongly Connected Component

SCCP Sparse Conditional Constant Propagation

SLP Superword-Level Parallelism, same as *Basic-Block Vectorization*.

SRoA Scalar Replacement of Aggregates

SSA Static Single Assignment

Stack Map In garbage collection, metadata emitted by the code generator which identifies roots within the stack frame of an executing function.

T

TBAA Type-Based Alias Analysis

2.9 How To Add Your Build Configuration To LLVM Buildbot Infrastructure

2.9.1 Introduction

This document contains information about adding a build configuration and buildslave to private slave builder to LLVM Buildbot Infrastructure http://lab.llvm.org:8011.

2.9.2 Steps To Add Builder To LLVM Buildbot

Volunteers can provide their build machines to work as build slaves to public LLVM Buildbot.

Here are the steps you can follow to do so:

1. Check the existing build configurations to make sure the one you are interested in is not covered yet or gets built on your computer much faster than on the existing one. We prefer faster builds so developers will get feedback sooner after changes get committed.
2. The computer you will be registering with the LLVM buildbot infrastructure should have all dependencies installed and you can actually build your configuration successfully. Please check what degree of parallelism (-j param) would give the fastest build. You can build multiple configurations on one computer.
3. Install buildslave (currently we are using buildbot version 0.8.5). Depending on the platform, buildslave could be available to download and install with your packet manager, or you can download it directly from http://trac.buildbot.net and install it manually.
4. Create a designated user account, your buildslave will be running under, and set appropriate permissions.
5. Choose the buildslave root directory (all builds will be placed under it), buildslave access name and password the build master will be using to authenticate your buildslave.
6. Create a buildslave in context of that buildslave account. Point it to the **lab.llvm.org** port **9990** (see Buildbot documentation, Creating a slave for more details) by running the following command:

```
$ buildslave create-slave <buildslave-root-directory> \
            lab.llvm.org:9990 \
            <buildslave-access-name> <buildslave-access-password>
```

7. Fill the buildslave description and admin name/e-mail. Here is an example of the buildslave description:

```
Windows 7 x64
Core i7 (2.66GHz), 16GB of RAM

g++.exe (TDM-1 mingw32) 4.4.0
GNU Binutils 2.19.1
cmake version 2.8.4
Microsoft(R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for 80x86
```

8. Make sure you can actually start the buildslave successfully. Then set up your buildslave to start automatically at the start up time. See the buildbot documentation for help. You may want to restart your computer to see if it works.
9. Send a patch which adds your build slave and your builder to zorg.
   - slaves are added to `buildbot/osuosl/master/config/slaves.py`
   - builders are added to `buildbot/osuosl/master/config/builders.py`
10. Send the buildslave access name and the access password directly to Galina Kistanova, and wait till she will let you know that your changes are applied and buildmaster is reconfigured.
11. Check the status of your buildslave on the Waterfall Display to make sure it is connected, and `http://lab.llvm.org:8011/buildslaves/<your-buildslave-name>` to see if administrator contact and slave information are correct.
12. Wait for the first build to succeed and enjoy.

2.10 yaml2obj

yaml2obj takes a YAML description of an object file and converts it to a binary file.

    $ yaml2obj input-file

Outputs the binary to stdout.

2.10.1 COFF Syntax

Here's a sample COFF file.

```
header:
  Machine: IMAGE_FILE_MACHINE_I386 # (0x14C)

sections:
  - Name: .text
    Characteristics: [ IMAGE_SCN_CNT_CODE
                     , IMAGE_SCN_ALIGN_16BYTES
                     , IMAGE_SCN_MEM_EXECUTE
                     , IMAGE_SCN_MEM_READ
                     ] # 0x60500020
    SectionData:
      "\x83\xEC\x0C\xC7\x44\x24\x08\x00\x00\x00\x00\xC7\x04\x24\x00\x00\x00\x00\xE8\x00\x00\x00\x00\x
symbols:
  - Name: .text
    Value: 0
    SectionNumber: 1
    SimpleType: IMAGE_SYM_TYPE_NULL # (0)
    ComplexType: IMAGE_SYM_DTYPE_NULL # (0)
    StorageClass: IMAGE_SYM_CLASS_STATIC # (3)
    NumberOfAuxSymbols: 1
    AuxiliaryData:
      "\x24\x00\x00\x00\x03\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00" # |$.............

  - Name: _main
    Value: 0
    SectionNumber: 1
```

```
    SimpleType: IMAGE_SYM_TYPE_NULL # (0)
    ComplexType: IMAGE_SYM_DTYPE_NULL # (0)
    StorageClass: IMAGE_SYM_CLASS_EXTERNAL # (2)
```

Here's a simplified Kwalify schema with an extension to allow alternate types.

```
type: map
  mapping:
    header:
      type: map
      mapping:
        Machine: [ {type: str, enum:
                                [ IMAGE_FILE_MACHINE_UNKNOWN
                                , IMAGE_FILE_MACHINE_AM33
                                , IMAGE_FILE_MACHINE_AMD64
                                , IMAGE_FILE_MACHINE_ARM
                                , IMAGE_FILE_MACHINE_ARMNT
                                , IMAGE_FILE_MACHINE_EBC
                                , IMAGE_FILE_MACHINE_I386
                                , IMAGE_FILE_MACHINE_IA64
                                , IMAGE_FILE_MACHINE_M32R
                                , IMAGE_FILE_MACHINE_MIPS16
                                , IMAGE_FILE_MACHINE_MIPSFPU
                                , IMAGE_FILE_MACHINE_MIPSFPU16
                                , IMAGE_FILE_MACHINE_POWERPC
                                , IMAGE_FILE_MACHINE_POWERPCFP
                                , IMAGE_FILE_MACHINE_R4000
                                , IMAGE_FILE_MACHINE_SH3
                                , IMAGE_FILE_MACHINE_SH3DSP
                                , IMAGE_FILE_MACHINE_SH4
                                , IMAGE_FILE_MACHINE_SH5
                                , IMAGE_FILE_MACHINE_THUMB
                                , IMAGE_FILE_MACHINE_WCEMIPSV2
                                ]}
                  , {type: int}
                  ]
        Characteristics:
          - type: seq
            sequence:
              - type: str
                enum: [ IMAGE_FILE_RELOCS_STRIPPED
                      , IMAGE_FILE_EXECUTABLE_IMAGE
                      , IMAGE_FILE_LINE_NUMS_STRIPPED
                      , IMAGE_FILE_LOCAL_SYMS_STRIPPED
                      , IMAGE_FILE_AGGRESSIVE_WS_TRIM
                      , IMAGE_FILE_LARGE_ADDRESS_AWARE
                      , IMAGE_FILE_BYTES_REVERSED_LO
                      , IMAGE_FILE_32BIT_MACHINE
                      , IMAGE_FILE_DEBUG_STRIPPED
                      , IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP
                      , IMAGE_FILE_NET_RUN_FROM_SWAP
                      , IMAGE_FILE_SYSTEM
                      , IMAGE_FILE_DLL
                      , IMAGE_FILE_UP_SYSTEM_ONLY
                      , IMAGE_FILE_BYTES_REVERSED_HI
                      ]
          - type: int
    sections:
      type: seq
```

```
    sequence:
      - type: map
        mapping:
          Name: {type: str}
          Characteristics:
            - type: seq
              sequence:
                - type: str
                  enum: [ IMAGE_SCN_TYPE_NO_PAD
                        , IMAGE_SCN_CNT_CODE
                        , IMAGE_SCN_CNT_INITIALIZED_DATA
                        , IMAGE_SCN_CNT_UNINITIALIZED_DATA
                        , IMAGE_SCN_LNK_OTHER
                        , IMAGE_SCN_LNK_INFO
                        , IMAGE_SCN_LNK_REMOVE
                        , IMAGE_SCN_LNK_COMDAT
                        , IMAGE_SCN_GPREL
                        , IMAGE_SCN_MEM_PURGEABLE
                        , IMAGE_SCN_MEM_16BIT
                        , IMAGE_SCN_MEM_LOCKED
                        , IMAGE_SCN_MEM_PRELOAD
                        , IMAGE_SCN_ALIGN_1BYTES
                        , IMAGE_SCN_ALIGN_2BYTES
                        , IMAGE_SCN_ALIGN_4BYTES
                        , IMAGE_SCN_ALIGN_8BYTES
                        , IMAGE_SCN_ALIGN_16BYTES
                        , IMAGE_SCN_ALIGN_32BYTES
                        , IMAGE_SCN_ALIGN_64BYTES
                        , IMAGE_SCN_ALIGN_128BYTES
                        , IMAGE_SCN_ALIGN_256BYTES
                        , IMAGE_SCN_ALIGN_512BYTES
                        , IMAGE_SCN_ALIGN_1024BYTES
                        , IMAGE_SCN_ALIGN_2048BYTES
                        , IMAGE_SCN_ALIGN_4096BYTES
                        , IMAGE_SCN_ALIGN_8192BYTES
                        , IMAGE_SCN_LNK_NRELOC_OVFL
                        , IMAGE_SCN_MEM_DISCARDABLE
                        , IMAGE_SCN_MEM_NOT_CACHED
                        , IMAGE_SCN_MEM_NOT_PAGED
                        , IMAGE_SCN_MEM_SHARED
                        , IMAGE_SCN_MEM_EXECUTE
                        , IMAGE_SCN_MEM_READ
                        , IMAGE_SCN_MEM_WRITE
                        ]
            - type: int
          SectionData: {type: str}
  symbols:
    type: seq
    sequence:
      - type: map
        mapping:
          Name: {type: str}
          Value: {type: int}
          SectionNumber: {type: int}
          SimpleType: [ {type: str, enum: [ IMAGE_SYM_TYPE_NULL
                                          , IMAGE_SYM_TYPE_VOID
                                          , IMAGE_SYM_TYPE_CHAR
                                          , IMAGE_SYM_TYPE_SHORT
```

```
                                    , IMAGE_SYM_TYPE_INT
                                    , IMAGE_SYM_TYPE_LONG
                                    , IMAGE_SYM_TYPE_FLOAT
                                    , IMAGE_SYM_TYPE_DOUBLE
                                    , IMAGE_SYM_TYPE_STRUCT
                                    , IMAGE_SYM_TYPE_UNION
                                    , IMAGE_SYM_TYPE_ENUM
                                    , IMAGE_SYM_TYPE_MOE
                                    , IMAGE_SYM_TYPE_BYTE
                                    , IMAGE_SYM_TYPE_WORD
                                    , IMAGE_SYM_TYPE_UINT
                                    , IMAGE_SYM_TYPE_DWORD
                                    ]}
                , {type: int}
                ]
ComplexType: [ {type: str, enum: [ IMAGE_SYM_DTYPE_NULL
                                 , IMAGE_SYM_DTYPE_POINTER
                                 , IMAGE_SYM_DTYPE_FUNCTION
                                 , IMAGE_SYM_DTYPE_ARRAY
                                 ]}
               , {type: int}
               ]
StorageClass: [ {type: str, enum:
                            [ IMAGE_SYM_CLASS_END_OF_FUNCTION
                            , IMAGE_SYM_CLASS_NULL
                            , IMAGE_SYM_CLASS_AUTOMATIC
                            , IMAGE_SYM_CLASS_EXTERNAL
                            , IMAGE_SYM_CLASS_STATIC
                            , IMAGE_SYM_CLASS_REGISTER
                            , IMAGE_SYM_CLASS_EXTERNAL_DEF
                            , IMAGE_SYM_CLASS_LABEL
                            , IMAGE_SYM_CLASS_UNDEFINED_LABEL
                            , IMAGE_SYM_CLASS_MEMBER_OF_STRUCT
                            , IMAGE_SYM_CLASS_ARGUMENT
                            , IMAGE_SYM_CLASS_STRUCT_TAG
                            , IMAGE_SYM_CLASS_MEMBER_OF_UNION
                            , IMAGE_SYM_CLASS_UNION_TAG
                            , IMAGE_SYM_CLASS_TYPE_DEFINITION
                            , IMAGE_SYM_CLASS_UNDEFINED_STATIC
                            , IMAGE_SYM_CLASS_ENUM_TAG
                            , IMAGE_SYM_CLASS_MEMBER_OF_ENUM
                            , IMAGE_SYM_CLASS_REGISTER_PARAM
                            , IMAGE_SYM_CLASS_BIT_FIELD
                            , IMAGE_SYM_CLASS_BLOCK
                            , IMAGE_SYM_CLASS_FUNCTION
                            , IMAGE_SYM_CLASS_END_OF_STRUCT
                            , IMAGE_SYM_CLASS_FILE
                            , IMAGE_SYM_CLASS_SECTION
                            , IMAGE_SYM_CLASS_WEAK_EXTERNAL
                            , IMAGE_SYM_CLASS_CLR_TOKEN
                            ]}
                , {type: int}
                ]
```

2.11 How to submit an LLVM bug report

2.11.1 Introduction - Got bugs?

If you're working with LLVM and run into a bug, we definitely want to know about it. This document describes what you can do to increase the odds of getting it fixed quickly.

Basically you have to do two things at a minimum. First, decide whether the bug crashes the compiler (or an LLVM pass), or if the compiler is miscompiling the program (i.e., the compiler successfully produces an executable, but it doesn't run right). Based on what type of bug it is, follow the instructions in the linked section to narrow down the bug so that the person who fixes it will be able to find the problem more easily.

Once you have a reduced test-case, go to the LLVM Bug Tracking System and fill out the form with the necessary details (note that you don't need to pick a category, just use the "new-bugs" category if you're not sure). The bug description should contain the following information:

- All information necessary to reproduce the problem.
- The reduced test-case that triggers the bug.
- The location where you obtained LLVM (if not from our Subversion repository).

Thanks for helping us make LLVM better!

2.11.2 Crashing Bugs

More often than not, bugs in the compiler cause it to crash—often due to an assertion failure of some sort. The most important piece of the puzzle is to figure out if it is crashing in the GCC front-end or if it is one of the LLVM libraries (e.g. the optimizer or code generator) that has problems.

To figure out which component is crashing (the front-end, optimizer or code generator), run the `clang` command line as you were when the crash occurred, but with the following extra command line options:

- `-O0 -emit-llvm`: If `clang` still crashes when passed these options (which disable the optimizer and code generator), then the crash is in the front-end. Jump ahead to the section on *front-end bugs*.
- `-emit-llvm`: If `clang` crashes with this option (which disables the code generator), you found an optimizer bug. Jump ahead to compile-time optimization bugs.
- Otherwise, you have a code generator crash. Jump ahead to code generator bugs.

Front-end bugs

If the problem is in the front-end, you should re-run the same `clang` command that resulted in the crash, but add the `-save-temps` option. The compiler will crash again, but it will leave behind a `foo.i` file (containing preprocessed C source code) and possibly `foo.s` for each compiled `foo.c` file. Send us the `foo.i` file, along with the options you passed to `clang`, and a brief description of the error it caused.

The delta tool helps to reduce the preprocessed file down to the smallest amount of code that still replicates the problem. You're encouraged to use delta to reduce the code to make the developers' lives easier. This website has instructions on the best way to use delta.

Compile-time optimization bugs

If you find that a bug crashes in the optimizer, compile your test-case to a `.bc` file by passing "`-emit-llvm -O0 -c -o foo.bc`". Then run:

```
opt -O3 -debug-pass=Arguments foo.bc -disable-output
```

This command should do two things: it should print out a list of passes, and then it should crash in the same way as clang. If it doesn't crash, please follow the instructions for a front-end bug.

If this does crash, then you should be able to debug this with the following bugpoint command:

```
bugpoint foo.bc <list of passes printed by opt>
```

Please run this, then file a bug with the instructions and reduced .bc files that bugpoint emits. If something goes wrong with bugpoint, please submit the "foo.bc" file and the list of passes printed by `opt`.

Code generator bugs

If you find a bug that crashes clang in the code generator, compile your source file to a .bc file by passing "`-emit-llvm -c -o foo.bc`" to clang (in addition to the options you already pass). Once your have foo.bc, one of the following commands should fail:

1. `llc foo.bc`
2. `llc foo.bc -relocation-model=pic`
3. `llc foo.bc -relocation-model=static`

If none of these crash, please follow the instructions for a front-end bug. If one of these do crash, you should be able to reduce this with one of the following bugpoint command lines (use the one corresponding to the command above that failed):

1. `bugpoint -run-llc foo.bc`
2. `bugpoint -run-llc foo.bc --tool-args -relocation-model=pic`
3. `bugpoint -run-llc foo.bc --tool-args -relocation-model=static`

Please run this, then file a bug with the instructions and reduced .bc file that bugpoint emits. If something goes wrong with bugpoint, please submit the "foo.bc" file and the option that llc crashes with.

2.11.3 Miscompilations

If clang successfully produces an executable, but that executable doesn't run right, this is either a bug in the code or a bug in the compiler. The first thing to check is to make sure it is not using undefined behavior (e.g. reading a variable before it is defined). In particular, check to see if the program valgrind's clean, passes purify, or some other memory checker tool. Many of the "LLVM bugs" that we have chased down ended up being bugs in the program being compiled, not LLVM.

Once you determine that the program itself is not buggy, you should choose which code generator you wish to compile the program with (e.g. LLC or the JIT) and optionally a series of LLVM passes to run. For example:

```
bugpoint -run-llc [... optzn passes ...] file-to-test.bc --args -- [program arguments]
```

bugpoint will try to narrow down your list of passes to the one pass that causes an error, and simplify the bitcode file as much as it can to assist you. It will print a message letting you know how to reproduce the resulting error.

2.11.4 Incorrect code generation

Similarly to debugging incorrect compilation by mis-behaving passes, you can debug incorrect code generation by either LLC or the JIT, using `bugpoint`. The process `bugpoint` follows in this case is to try to narrow the code down to a function that is miscompiled by one or the other method, but since for correctness, the entire program must

be run, `bugpoint` will compile the code it deems to not be affected with the C Backend, and then link in the shared object it generates.

To debug the JIT:

```
bugpoint -run-jit -output=[correct output file] [bitcode file]  \
         --tool-args -- [arguments to pass to lli]              \
         --args -- [program arguments]
```

Similarly, to debug the LLC, one would run:

```
bugpoint -run-llc -output=[correct output file] [bitcode file]  \
         --tool-args -- [arguments to pass to llc]              \
         --args -- [program arguments]
```

Special note: if you are debugging MultiSource or SPEC tests that already exist in the `llvm/test` hierarchy, there is an easier way to debug the JIT, LLC, and CBE, using the pre-written Makefile targets, which will pass the program options specified in the Makefiles:

```
cd llvm/test/../../program
make bugpoint-jit
```

At the end of a successful `bugpoint` run, you will be presented with two bitcode files: a *safe* file which can be compiled with the C backend and the *test* file which either LLC or the JIT mis-codegenerates, and thus causes the error.

To reproduce the error that `bugpoint` found, it is sufficient to do the following:

1. Regenerate the shared object from the safe bitcode file:

   ```
   llc -march=c safe.bc -o safe.c
   gcc -shared safe.c -o safe.so
   ```

2. If debugging LLC, compile test bitcode native and link with the shared object:

   ```
   llc test.bc -o test.s
   gcc test.s safe.so -o test.llc
   ./test.llc [program options]
   ```

3. If debugging the JIT, load the shared object and supply the test bitcode:

   ```
   lli -load=safe.so test.bc [program options]
   ```

2.12 Sphinx Quickstart Template

2.12.1 Introduction and Quickstart

This document is meant to get you writing documentation as fast as possible even if you have no previous experience with Sphinx. The goal is to take someone in the state of "I want to write documentation and get it added to LLVM's docs" and turn that into useful documentation mailed to llvm-commits with as little nonsense as possible.

You can find this document in `docs/SphinxQuickstartTemplate.rst`. You should copy it, open the new file in your text editor, write your docs, and then send the new document to llvm-commits for review.

Focus on *content*. It is easy to fix the Sphinx (reStructuredText) syntax later if necessary, although reStructuredText tries to imitate common plain-text conventions so it should be quite natural. A basic knowledge of reStructuredText syntax is useful when writing the document, so the last ~half of this document (starting with Example Section) gives examples which should cover 99% of use cases.

Let me say that again: focus on *content*. But if you really need to verify Sphinx's output, see `docs/README.txt` for information.

Once you have finished with the content, please send the `.rst` file to llvm-commits for review.

2.12.2 Guidelines

Try to answer the following questions in your first section:

1. Why would I want to read this document?
2. What should I know to be able to follow along with this document?
3. What will I have learned by the end of this document?

Common names for the first section are `Introduction`, `Overview`, or `Background`.

If possible, make your document a "how to". Give it a name `HowTo*.rst` like the other "how to" documents. This format is usually the easiest for another person to understand and also the most useful.

You generally should not be writing documentation other than a "how to" unless there is already a "how to" about your topic. The reason for this is that without a "how to" document to read first, it is difficult for a person to understand a more advanced document.

Focus on content (yes, I had to say it again).

The rest of this document shows example reStructuredText markup constructs that are meant to be read by you in your text editor after you have copied this file into a new file for the documentation you are about to write.

2.12.3 Example Section

Your text can be *emphasized*, **bold**, or `monospace`.

Use blank lines to separate paragraphs.

Headings (like `Example Section` just above) give your document its structure. Use the same kind of adornments (e.g. `======` vs. `------`) as are used in this document. The adornment must be the same length as the text above it. For Vim users, variations of `yypVr=` might be handy.

Example Subsection

Make a link like this. There is also a more sophisticated syntax which can be more readable for longer links since it disrupts the flow less. You can put the `.. _`link text`: <URL>` block pretty much anywhere later in the document.

Lists can be made like this:

1. A list starting with `#.` will be automatically numbered.
2. This is a second list element.
   (a) Use indentation to create nested lists.

You can also use unordered lists.

- Stuff.
  - Deeper stuff.
- More stuff.

Example Subsubsection

You can make blocks of code like this:

```
int main() {
  return 0;
}
```

For a shell session, use a `console` code block (some existing docs use `bash`):

```
$ echo "Goodbye cruel world!"
$ rm -rf /
```

If you need to show LLVM IR use the `llvm` code block.

```
define i32 @test1() {
entry:
  ret i32 0
}
```

Some other common code blocks you might need are `c`, `objc`, `make`, and `cmake`. If you need something beyond that, you can look at the full list of supported code blocks.

However, don't waste time fiddling with syntax highlighting when you could be adding meaningful content. When in doubt, show preformatted text without any syntax highlighting like this:

```
                        .
                         +:.
                      ..:: ::
                   .++:+:: ::+:.:.
                  .:+            :
          ::.::..::              .+.
       ..:+    ::                 :
......+:.                         ..
       :++.    ..                 :
         .+:::+::                 :
         ..   . .+                ::
                +.:       .::+.
                 ...+. .: .
                    .++:..
                     ...
```

Hopefully you won't need to be this deep If you need to do fancier things than what has been shown in this document, you can mail the list or check Sphinx's reStructuredText Primer.

2.13 Code Reviews with Phabricator

If you prefer to use a web user interface for code reviews, you can now submit your patches for Clang and LLVM at LLVM's Phabricator instance.

While Phabricator is a useful tool for some, the relevant -commits mailing list is the system of record for all LLVM code review. The mailing list should be added as a subscriber on all reviews, and Phabricator users should be prepared to respond to free-form comments in mail sent to the commits list.

2.13.1 Sign up

To get started with Phabricator, navigate to http://reviews.llvm.org and click the power icon in the top right. You can register with a GitHub account, a Google account, or you can create your own profile.

Make *sure* that the email address registered with Phabricator is subscribed to the relevant -commits mailing list. If your are not subscribed to the commit list, all mail sent by Phabricator on your behalf will be held for moderation.

Note that if you use your Subversion user name as Phabricator user name, Phabricator will automatically connect your submits to your Phabricator user in the Code Repository Browser.

2.13.2 Requesting a review via the command line

Phabricator has a tool called *Arcanist* to upload patches from the command line. To get you set up, follow the Arcanist Quick Start instructions.

You can learn more about how to use arc to interact with Phabricator in the Arcanist User Guide.

2.13.3 Requesting a review via the web interface

The tool to create and review patches in Phabricator is called *Differential*.

Note that you can upload patches created through various diff tools, including git and svn. To make reviews easier, please always include **as much context as possible** with your diff! Don't worry, Phabricator will automatically send a diff with a smaller context in the review email, but having the full file in the web interface will help the reviewer understand your code.

To get a full diff, use one of the following commands (or just use Arcanist to upload your patch):

- `git diff -U999999 other-branch`
- `svn diff --diff-cmd=diff -x -U999999`

To upload a new patch:

- Click *Differential*.
- Click *Create Diff*.
- Paste the text diff or upload the patch file. Note that TODO
- Leave the drop down on *Create a new Revision...* and click *Continue*.
- Enter a descriptive title and summary; add reviewers and mailing lists that you want to be included in the review. If your patch is for LLVM, cc llvm-commits; if your patch is for Clang, cc cfe-commits.
- Click *Save*.

To submit an updated patch:

- Click *Differential*.
- Click *Create Diff*.

- Paste the updated diff.
- Select the review you want to from the *Attach To* dropdown and click *Continue*.
- Click *Save*.

2.13.4 Reviewing code with Phabricator

Phabricator allows you to add inline comments as well as overall comments to a revision. To add an inline comment, select the lines of code you want to comment on by clicking and dragging the line numbers in the diff pane.

You can add overall comments or submit your comments at the bottom of the page.

Phabricator has many useful features, for example allowing you to select diffs between different versions of the patch as it was reviewed in the *Revision Update History*. Most features are self descriptive - explore, and if you have a question, drop by on #llvm in IRC to get help.

Note that as e-mail is the system of reference for code reviews, and some people prefer it over a web interface, we do not generate automated mail when a review changes state, for example by clicking "Accept Revision" in the web interface. Thus, please type LGTM into the comment box to accept a change from Phabricator.

2.13.5 Committing a change

Arcanist can manage the commit transparently. It will retrieve the description, reviewers, the `Differential Revision`, etc from the review and commit it to the repository.

```
arc patch D<Revision>
arc commit --revision D<Revision>
```

When committing an LLVM change that has been reviewed using Phabricator, the convention is for the commit message to end with the line:

```
Differential Revision: <URL>
```

where `<URL>` is the URL for the code review, starting with `http://reviews.llvm.org/`.

Note that Arcanist will add this automatically.

This allows people reading the version history to see the review for context. This also allows Phabricator to detect the commit, close the review, and add a link from the review to the commit.

2.13.6 Status

Please let us know whether you like it and what could be improved!

2.14 LLVM Testing Infrastructure Guide

2.14.1 LLVM test-suite Makefile Guide

Overview

This document describes the features of the Makefile-based LLVM test-suite. This way of interacting with the test-suite is deprecated in favor of running the test-suite using LNT, but may continue to prove useful for some users. See the Testing Guide's *test-suite Quickstart* section for more information.

Test suite Structure

The `test-suite` module contains a number of programs that can be compiled with LLVM and executed. These programs are compiled using the native compiler and various LLVM backends. The output from the program compiled with the native compiler is assumed correct; the results from the other programs are compared to the native program output and pass if they match.

When executing tests, it is usually a good idea to start out with a subset of the available tests or programs. This makes test run times smaller at first and later on this is useful to investigate individual test failures. To run some test only on a subset of programs, simply change directory to the programs you want tested and run `gmake` there. Alternatively, you can run a different test using the `TEST` variable to change what tests or run on the selected programs (see below for more info).

In addition for testing correctness, the `test-suite` directory also performs timing tests of various LLVM optimizations. It also records compilation times for the compilers and the JIT. This information can be used to compare the effectiveness of LLVM's optimizations and code generation.

`test-suite` tests are divided into three types of tests: MultiSource, SingleSource, and External.

- `test-suite/SingleSource`

  The SingleSource directory contains test programs that are only a single source file in size. These are usually small benchmark programs or small programs that calculate a particular value. Several such programs are grouped together in each directory.

- `test-suite/MultiSource`

  The MultiSource directory contains subdirectories which contain entire programs with multiple source files. Large benchmarks and whole applications go here.

- `test-suite/External`

  The External directory contains Makefiles for building code that is external to (i.e., not distributed with) LLVM. The most prominent members of this directory are the SPEC 95 and SPEC 2000 benchmark suites. The `External` directory does not contain these actual tests, but only the Makefiles that know how to properly compile these programs from somewhere else. The presence and location of these external programs is configured by the test-suite `configure` script.

Each tree is then subdivided into several categories, including applications, benchmarks, regression tests, code that is strange grammatically, etc. These organizations should be relatively self explanatory.

Some tests are known to fail. Some are bugs that we have not fixed yet; others are features that we haven't added yet (or may never add). In the regression tests, the result for such tests will be XFAIL (eXpected FAILure). In this way, you can tell the difference between an expected and unexpected failure.

The tests in the test suite have no such feature at this time. If the test passes, only warnings and other miscellaneous output will be generated. If a test fails, a large <program> FAILED message will be displayed. This will help you separate benign warnings from actual test failures.

Running the test suite

First, all tests are executed within the LLVM object directory tree. They *are not* executed inside of the LLVM source tree. This is because the test suite creates temporary files during execution.

To run the test suite, you need to use the following steps:

1. `cd` into the `llvm/projects` directory in your source tree.
2. Check out the `test-suite` module with:

   ```
   % svn co http://llvm.org/svn/llvm-project/test-suite/trunk test-suite
   ```

   This will get the test suite into `llvm/projects/test-suite`.
3. Configure and build `llvm`.
4. Configure and build `llvm-gcc`.
5. Install `llvm-gcc` somewhere.
6. *Re-configure* `llvm` from the top level of each build tree (LLVM object directory tree) in which you want to run the test suite, just as you do before building LLVM.

   During the *re-configuration*, you must either: (1) have `llvm-gcc` you just built in your path, or (2) specify the directory where your just-built `llvm-gcc` is installed using `--with-llvmgccdir=$LLVM_GCC_DIR`.

You must also tell the configure machinery that the test suite is available so it can be configured for your build tree:

```
% cd $LLVM_OBJ_ROOT ; $LLVM_SRC_ROOT/configure [--with-llvmgccdir=$LLVM_GCC_DIR]
```

[Remember that `$LLVM_GCC_DIR` is the directory where you *installed* llvm-gcc, not its src or obj directory.]

7. You can now run the test suite from your build tree as follows:

```
% cd $LLVM_OBJ_ROOT/projects/test-suite
% make
```

Note that the second and third steps only need to be done once. After you have the suite checked out and configured, you don't need to do it again (unless the test code or configure script changes).

Configuring External Tests

In order to run the External tests in the `test-suite` module, you must specify *–with-externals*. This must be done during the *re-configuration* step (see above), and the `llvm` re-configuration must recognize the previously-built `llvm-gcc`. If any of these is missing or neglected, the External tests won't work.

- *–with-externals*
- *–with-externals*=<*directory*>

This tells LLVM where to find any external tests. They are expected to be in specifically named subdirectories of <`directory`>. If `directory` is left unspecified, `configure` uses the default value `/home/vadve/shared/benchmarks/speccpu2000/benchspec`. Subdirectory names known to LLVM include:

- spec95
- speccpu2000
- speccpu2006
- povray31

Others are added from time to time, and can be determined from `configure`.

Running different tests

In addition to the regular "whole program" tests, the `test-suite` module also provides a mechanism for compiling the programs in different ways. If the variable TEST is defined on the `gmake` command line, the test system will include a Makefile named `TEST.<value of TEST variable>.Makefile`. This Makefile can modify build rules to yield different results.

For example, the LLVM nightly tester uses `TEST.nightly.Makefile` to create the nightly test reports. To run the nightly tests, run `gmake TEST=nightly`.

There are several TEST Makefiles available in the tree. Some of them are designed for internal LLVM research and will not work outside of the LLVM research group. They may still be valuable, however, as a guide to writing your own TEST Makefile for any optimization or analysis passes that you develop with LLVM.

Generating test output

There are a number of ways to run the tests and generate output. The most simple one is simply running `gmake` with no arguments. This will compile and run all programs in the tree using a number of different methods and compare

results. Any failures are reported in the output, but are likely drowned in the other output. Passes are not reported explicitly.

Somewhat better is running `gmake TEST=sometest test`, which runs the specified test and usually adds per-program summaries to the output (depending on which sometest you use). For example, the `nightly` test explicitly outputs TEST-PASS or TEST-FAIL for every test after each program. Though these lines are still drowned in the output, it's easy to grep the output logs in the Output directories.

Even better are the `report` and `report.format` targets (where `format` is one of `html`, `csv`, `text` or `graphs`). The exact contents of the report are dependent on which `TEST` you are running, but the text results are always shown at the end of the run and the results are always stored in the `report.<type>.format` file (when running with `TEST=<type>`). The `report` also generate a file called `report.<type>.raw.out` containing the output of the entire test run.

Writing custom tests for the test suite

Assuming you can run the test suite, (e.g. "`gmake TEST=nightly report`" should work), it is really easy to run optimizations or code generator components against every program in the tree, collecting statistics or running custom checks for correctness. At base, this is how the nightly tester works, it's just one example of a general framework.

Lets say that you have an LLVM optimization pass, and you want to see how many times it triggers. First thing you should do is add an LLVM statistic to your pass, which will tally counts of things you care about.

Following this, you can set up a test and a report that collects these and formats them for easy viewing. This consists of two files, a "`test-suite/TEST.XXX.Makefile`" fragment (where XXX is the name of your test) and a "`test-suite/TEST.XXX.report`" file that indicates how to format the output into a table. There are many example reports of various levels of sophistication included with the test suite, and the framework is very general.

If you are interested in testing an optimization pass, check out the "libcalls" test as an example. It can be run like this:

```
% cd llvm/projects/test-suite/MultiSource/Benchmarks  # or some other level
% make TEST=libcalls report
```

This will do a bunch of stuff, then eventually print a table like this:

```
Name                                  | total | #exit |
...
FreeBench/analyzer/analyzer           | 51    | 6     |
FreeBench/fourinarow/fourinarow       | 1     | 1     |
FreeBench/neural/neural               | 19    | 9     |
FreeBench/pifft/pifft                 | 5     | 3     |
MallocBench/cfrac/cfrac               | 1     | *     |
MallocBench/espresso/espresso         | 52    | 12    |
MallocBench/gs/gs                     | 4     | *     |
Prolangs-C/TimberWolfMC/timberwolfmc  | 302   | *     |
Prolangs-C/agrep/agrep                | 33    | 12    |
Prolangs-C/allroots/allroots          | *     | *     |
Prolangs-C/assembler/assembler        | 47    | *     |
Prolangs-C/bison/mybison              | 74    | *     |
...
```

This basically is grepping the -stats output and displaying it in a table. You can also use the "TEST=libcalls report.html" target to get the table in HTML form, similarly for report.csv and report.tex.

The source for this is in `test-suite/TEST.libcalls.*`. The format is pretty simple: the Makefile indicates how to run the test (in this case, "`opt -simplify-libcalls -stats`"), and the report contains one line for each column of the output. The first value is the header for the column and the second is the regex to grep the output of the command for. There are lots of example reports that can do fancy stuff.

2.14.2 Overview

This document is the reference manual for the LLVM testing infrastructure. It documents the structure of the LLVM testing infrastructure, the tools needed to use it, and how to add and run tests.

2.14.3 Requirements

In order to use the LLVM testing infrastructure, you will need all of the software required to build LLVM, as well as Python 2.5 or later.

2.14.4 LLVM testing infrastructure organization

The LLVM testing infrastructure contains two major categories of tests: regression tests and whole programs. The regression tests are contained inside the LLVM repository itself under `llvm/test` and are expected to always pass – they should be run before every commit.

The whole programs tests are referred to as the "LLVM test suite" (or "test-suite") and are in the `test-suite` module in subversion. For historical reasons, these tests are also referred to as the "nightly tests" in places, which is less ambiguous than "test-suite" and remains in use although we run them much more often than nightly.

Regression tests

The regression tests are small pieces of code that test a specific feature of LLVM or trigger a specific bug in LLVM. The language they are written in depends on the part of LLVM being tested. These tests are driven by the *Lit* testing tool (which is part of LLVM), and are located in the `llvm/test` directory.

Typically when a bug is found in LLVM, a regression test containing just enough code to reproduce the problem should be written and placed somewhere underneath this directory. For example, it can be a small piece of LLVM IR distilled from an actual application or benchmark.

`test-suite`

The test suite contains whole programs, which are pieces of code which can be compiled and linked into a stand-alone program that can be executed. These programs are generally written in high level languages such as C or C++.

These programs are compiled using a user specified compiler and set of flags, and then executed to capture the program output and timing information. The output of these programs is compared to a reference output to ensure that the program is being compiled correctly.

In addition to compiling and executing programs, whole program tests serve as a way of benchmarking LLVM performance, both in terms of the efficiency of the programs generated as well as the speed with which LLVM compiles, optimizes, and generates code.

The test-suite is located in the `test-suite` Subversion module.

Debugging Information tests

The test suite contains tests to check quality of debugging information. The test are written in C based languages or in LLVM assembly language.

These tests are compiled and run under a debugger. The debugger output is checked to validate of debugging information. See README.txt in the test suite for more information . This test suite is located in the `debuginfo-tests` Subversion module.

2.14.5 Quick start

The tests are located in two separate Subversion modules. The regressions tests are in the main "llvm" module under the directory `llvm/test` (so you get these tests for free with the main LLVM tree). Use `make check-all` to run the regression tests after building LLVM.

The more comprehensive test suite that includes whole programs in C and C++ is in the `test-suite` module. See *test-suite Quickstart* for more information on running these tests.

Regression tests

To run all of the LLVM regression tests, use the master Makefile in the `llvm/test` directory. LLVM Makefiles require GNU Make (read the *LLVM Makefile Guide* for more details):

```
% make -C llvm/test
```

or:

```
% make check
```

If you have Clang checked out and built, you can run the LLVM and Clang tests simultaneously using:

```
% make check-all
```

To run the tests with Valgrind (Memcheck by default), use the `LIT_ARGS` make variable to pass the required options to lit. For example, you can use:

```
% make check LIT_ARGS="-v --vg --vg-leak"
```

to enable testing with valgrind and with leak checking enabled.

To run individual tests or subsets of tests, you can use the `llvm-lit` script which is built as part of LLVM. For example, to run the `Integer/BitPacked.ll` test by itself you can run:

```
% llvm-lit ~/llvm/test/Integer/BitPacked.ll
```

or to run all of the ARM CodeGen tests:

```
% llvm-lit ~/llvm/test/CodeGen/ARM
```

For more information on using the **lit** tool, see `llvm-lit --help` or the *lit man page*.

Debugging Information tests

To run debugging information tests simply checkout the tests inside clang/test directory.

```
% cd clang/test
% svn co http://llvm.org/svn/llvm-project/debuginfo-tests/trunk debuginfo-tests
```

These tests are already set up to run as part of clang regression tests.

2.14.6 Regression test structure

The LLVM regression tests are driven by **lit** and are located in the `llvm/test` directory.

This directory contains a large array of small tests that exercise various features of LLVM and to ensure that regressions do not occur. The directory is broken into several sub-directories, each focused on a particular area of LLVM.

Writing new regression tests

The regression test structure is very simple, but does require some information to be set. This information is gathered via `configure` and is written to a file, `test/lit.site.cfg` in the build directory. The `llvm/test` Makefile does this work for you.

In order for the regression tests to work, each directory of tests must have a `lit.local.cfg` file. **lit** looks for this file to determine how to run the tests. This file is just Python code and thus is very flexible, but we've standardized it for the LLVM regression tests. If you're adding a directory of tests, just copy `lit.local.cfg` from another directory to get running. The standard `lit.local.cfg` simply specifies which files to look in for tests. Any directory that contains only directories does not need the `lit.local.cfg` file. Read the *Lit documentation* for more information.

Each test file must contain lines starting with "RUN:" that tell **lit** how to run it. If there are no RUN lines, **lit** will issue an error while running a test.

RUN lines are specified in the comments of the test program using the keyword `RUN` followed by a colon, and lastly the command (pipeline) to execute. Together, these lines form the "script" that **lit** executes to run the test case. The syntax of the RUN lines is similar to a shell's syntax for pipelines including I/O redirection and variable substitution. However, even though these lines may *look* like a shell script, they are not. RUN lines are interpreted by **lit**. Consequently, the syntax differs from shell in a few ways. You can specify as many RUN lines as needed.

lit performs substitution on each RUN line to replace LLVM tool names with the full paths to the executable built for each tool (in `$(LLVM_OBJ_ROOT)/$(BuildMode)/bin`). This ensures that **lit** does not invoke any stray LLVM tools in the user's path during testing.

Each RUN line is executed on its own, distinct from other lines unless its last character is `\`. This continuation character causes the RUN line to be concatenated with the next one. In this way you can build up long pipelines of commands without making huge line lengths. The lines ending in `\` are concatenated until a RUN line that doesn't end in `\` is found. This concatenated set of RUN lines then constitutes one execution. **lit** will substitute variables and arrange for the pipeline to be executed. If any process in the pipeline fails, the entire line (and test case) fails too.

Below is an example of legal RUN lines in a `.ll` file:

```
; RUN: llvm-as < %s | llvm-dis > %t1
; RUN: llvm-dis < %s.bc-13 > %t2
; RUN: diff %t1 %t2
```

As with a Unix shell, the RUN lines permit pipelines and I/O redirection to be used.

There are some quoting rules that you must pay attention to when writing your RUN lines. In general nothing needs to be quoted. **lit** won't strip off any quote characters so they will get passed to the invoked program. To avoid this use curly braces to tell **lit** that it should treat everything enclosed as one value.

In general, you should strive to keep your RUN lines as simple as possible, using them only to run tools that generate textual output you can then examine. The recommended way to examine output to figure out if the test passes is using the *FileCheck tool*. *[The usage of grep in RUN lines is deprecated - please do not send or commit patches that use it.]*

Fragile tests

It is easy to write a fragile test that would fail spuriously if the tool being tested outputs a full path to the input file. For example, **opt** by default outputs a `ModuleID`:

```
$ cat example.ll
define i32 @main() nounwind {
    ret i32 0
}

$ opt -S /path/to/example.ll
; ModuleID = '/path/to/example.ll'
```

```
define i32 @main() nounwind {
    ret i32 0
}
```

`ModuleID` can unexpetedly match against `CHECK` lines. For example:

```
; RUN: opt -S %s | FileCheck

define i32 @main() nounwind {
    ; CHECK-NOT: load
    ret i32 0
}
```

This test will fail if placed into a `download` directory.

To make your tests robust, always use `opt ... < %s` in the RUN line. **opt** does not output a `ModuleID` when input comes from stdin.

Platform-Specific Tests

Whenever adding tests that require the knowledge of a specific platform, either related to code generated, specific output or back-end features, you must make sure to isolate the features, so that buildbots that run on different architectures (and don't even compile all back-ends), don't fail.

The first problem is to check for target-specific output, for example sizes of structures, paths and architecture names, for example:

- Tests containing Windows paths will fail on Linux and vice-versa.
- Tests that check for `x86_64` somewhere in the text will fail anywhere else.
- Tests where the debug information calculates the size of types and structures.

Also, if the test rely on any behaviour that is coded in any back-end, it must go in its own directory. So, for instance, code generator tests for ARM go into `test/CodeGen/ARM` and so on. Those directories contain a special `lit` configuration file that ensure all tests in that directory will only run if a specific back-end is compiled and available.

For instance, on `test/CodeGen/ARM`, the `lit.local.cfg` is:

```
config.suffixes = ['.ll', '.c', '.cpp', '.test']
if not 'ARM' in config.root.targets:
  config.unsupported = True
```

Other platform-specific tests are those that depend on a specific feature of a specific sub-architecture, for example only to Intel chips that support `AVX2`.

For instance, `test/CodeGen/X86/psubus.ll` tests three sub-architecture variants:

```
; RUN: llc -mcpu=core2 < %s | FileCheck %s -check-prefix=SSE2
; RUN: llc -mcpu=corei7-avx < %s | FileCheck %s -check-prefix=AVX1
; RUN: llc -mcpu=core-avx2 < %s | FileCheck %s -check-prefix=AVX2
```

And the checks are different:

```
; SSE2: @test1
; SSE2: psubusw LCPI0_0(%rip), %xmm0
; AVX1: @test1
; AVX1: vpsubusw LCPI0_0(%rip), %xmm0, %xmm0
; AVX2: @test1
; AVX2: vpsubusw LCPI0_0(%rip), %xmm0, %xmm0
```

So, if you're testing for a behaviour that you know is platform-specific or depends on special features of sub-architectures, you must add the specific triple, test with the specific FileCheck and put it into the specific directory that will filter out all other architectures.

Substitutions

Besides replacing LLVM tool names the following substitutions are performed in RUN lines:

%% Replaced by a single %. This allows escaping other substitutions.

%s File path to the test case's source. This is suitable for passing on the command line as the input to an LLVM tool.

Example: /home/user/llvm/test/MC/ELF/foo_test.s

%S Directory path to the test case's source.

Example: /home/user/llvm/test/MC/ELF

%t File path to a temporary file name that could be used for this test case. The file name won't conflict with other test cases. You can append to it if you need multiple temporaries. This is useful as the destination of some redirected output.

Example: /home/user/llvm.build/test/MC/ELF/Output/foo_test.s.tmp

%T Directory of %t.

Example: /home/user/llvm.build/test/MC/ELF/Output

%{pathsep}

Expands to the path separator, i.e. **:** (or **;** on Windows).

LLVM-specific substitutions:

%shlibext The suffix for the host platforms shared library files. This includes the period as the first character.

Example: .so (Linux), .dylib (OS X), .dll (Windows)

%exeext The suffix for the host platforms executable files. This includes the period as the first character.

Example: .exe (Windows), empty on Linux.

%(line), %(line+<number>), %(line-<number>) The number of the line where this substitution is used, with an optional integer offset. This can be used in tests with multiple RUN lines, which reference test file's line numbers.

Clang-specific substitutions:

%clang Invokes the Clang driver.

%clang_cpp Invokes the Clang driver for C++.

%clang_cl Invokes the CL-compatible Clang driver.

%clangxx Invokes the G++-compatible Clang driver.

%clang_cc1 Invokes the Clang frontend.

%itanium_abi_triple, %ms_abi_triple These substitutions can be used to get the current target triple adjusted to the desired ABI. For example, if the test suite is running with the i686-pc-win32 target, %itanium_abi_triple will expand to i686-pc-mingw32. This allows a test to run with a specific ABI without constraining it to a specific triple.

To add more substituations, look at test/lit.cfg or lit.local.cfg.

Other Features

To make RUN line writing easier, there are several helper programs. These helpers are in the PATH when running tests, so you can just call them using their name. For example:

not This program runs its arguments and then inverts the result code from it. Zero result codes become 1. Non-zero result codes become 0.

Sometimes it is necessary to mark a test case as "expected fail" or XFAIL. You can easily mark a test as XFAIL just by including `XFAIL:` on a line near the top of the file. This signals that the test case should succeed if the test fails. Such test cases are counted separately by the testing tool. To specify an expected fail, use the XFAIL keyword in the comments of the test program followed by a colon and one or more failure patterns. Each failure pattern can be either `*` (to specify fail everywhere), or a part of a target triple (indicating the test should fail on that platform), or the name of a configurable feature (for example, `loadable_module`). If there is a match, the test is expected to fail. If not, the test is expected to succeed. To XFAIL everywhere just specify `XFAIL: *`. Here is an example of an `XFAIL` line:

```
; XFAIL: darwin,sun
```

To make the output more useful, **lit** will scan the lines of the test case for ones that contain a pattern that matches `PR[0-9]+`. This is the syntax for specifying a PR (Problem Report) number that is related to the test case. The number after "PR" specifies the LLVM bugzilla number. When a PR number is specified, it will be used in the pass/fail reporting. This is useful to quickly get some context when a test fails.

Finally, any line that contains "END." will cause the special interpretation of lines to terminate. This is generally done right after the last RUN: line. This has two side effects:

1. it prevents special interpretation of lines that are part of the test program, not the instructions to the test case, and
2. it speeds things up for really big test cases by avoiding interpretation of the remainder of the file.

2.14.7 `test-suite` Overview

The `test-suite` module contains a number of programs that can be compiled and executed. The `test-suite` includes reference outputs for all of the programs, so that the output of the executed program can be checked for correctness.

`test-suite` tests are divided into three types of tests: MultiSource, SingleSource, and External.

- `test-suite/SingleSource`

  The SingleSource directory contains test programs that are only a single source file in size. These are usually small benchmark programs or small programs that calculate a particular value. Several such programs are grouped together in each directory.

- `test-suite/MultiSource`

  The MultiSource directory contains subdirectories which contain entire programs with multiple source files. Large benchmarks and whole applications go here.

- `test-suite/External`

  The External directory contains Makefiles for building code that is external to (i.e., not distributed with) LLVM. The most prominent members of this directory are the SPEC 95 and SPEC 2000 benchmark suites. The `External` directory does not contain these actual tests, but only the Makefiles that know how to properly compile these programs from somewhere else. When using `LNT`, use the `--test-externals` option to include these tests in the results.

`test-suite` Quickstart

The modern way of running the `test-suite` is focused on testing and benchmarking complete compilers using the LNT testing infrastructure.

For more information on using LNT to execute the `test-suite`, please see the LNT Quickstart documentation.

`test-suite` Makefiles

Historically, the `test-suite` was executed using a complicated setup of Makefiles. The LNT based approach above is recommended for most users, but there are some testing scenarios which are not supported by the LNT approach. In addition, LNT currently uses the Makefile setup under the covers and so developers who are interested in how LNT works under the hood may want to understand the Makefile based setup.

For more information on the `test-suite` Makefile setup, please see the *Test Suite Makefile Guide*.

2.15 LLVM Tutorial: Table of Contents

2.15.1 Kaleidoscope: Implementing a Language with LLVM

Kaleidoscope: Tutorial Introduction and the Lexer

Tutorial Introduction

Welcome to the "Implementing a language with LLVM" tutorial. This tutorial runs through the implementation of a simple language, showing how fun and easy it can be. This tutorial will get you up and started as well as help to build a framework you can extend to other languages. The code in this tutorial can also be used as a playground to hack on other LLVM specific things.

The goal of this tutorial is to progressively unveil our language, describing how it is built up over time. This will let us cover a fairly broad range of language design and LLVM-specific usage issues, showing and explaining the code for it all along the way, without overwhelming you with tons of details up front.

It is useful to point out ahead of time that this tutorial is really about teaching compiler techniques and LLVM specifically, *not* about teaching modern and sane software engineering principles. In practice, this means that we'll take a number of shortcuts to simplify the exposition. For example, the code leaks memory, uses global variables all over the place, doesn't use nice design patterns like visitors, etc... but it is very simple. If you dig in and use the code as a basis for future projects, fixing these deficiencies shouldn't be hard.

I've tried to put this tutorial together in a way that makes chapters easy to skip over if you are already familiar with or are uninterested in the various pieces. The structure of the tutorial is:

- Chapter #1: Introduction to the Kaleidoscope language, and the definition of its Lexer - This shows where we are going and the basic functionality that we want it to do. In order to make this tutorial maximally understandable and hackable, we choose to implement everything in C++ instead of using lexer and parser generators. LLVM obviously works just fine with such tools, feel free to use one if you prefer.

- Chapter #2: Implementing a Parser and AST - With the lexer in place, we can talk about parsing techniques and basic AST construction. This tutorial describes recursive descent parsing and operator precedence parsing. Nothing in Chapters 1 or 2 is LLVM-specific, the code doesn't even link in LLVM at this point. :)
- Chapter #3: Code generation to LLVM IR - With the AST ready, we can show off how easy generation of LLVM IR really is.
- Chapter #4: Adding JIT and Optimizer Support - Because a lot of people are interested in using LLVM as a JIT, we'll dive right into it and show you the 3 lines it takes to add JIT support. LLVM is also useful in many other ways, but this is one simple and "sexy" way to show off its power. :)
- Chapter #5: Extending the Language: Control Flow - With the language up and running, we show how to extend it with control flow operations (if/then/else and a 'for' loop). This gives us a chance to talk about simple SSA construction and control flow.
- Chapter #6: Extending the Language: User-defined Operators - This is a silly but fun chapter that talks about extending the language to let the user program define their own arbitrary unary and binary operators (with assignable precedence!). This lets us build a significant piece of the "language" as library routines.
- Chapter #7: Extending the Language: Mutable Variables - This chapter talks about adding user-defined local variables along with an assignment operator. The interesting part about this is how easy and trivial it is to construct SSA form in LLVM: no, LLVM does *not* require your front-end to construct SSA form!
- Chapter #8: Conclusion and other useful LLVM tidbits - This chapter wraps up the series by talking about potential ways to extend the language, but also includes a bunch of pointers to info about "special topics" like adding garbage collection support, exceptions, debugging, support for "spaghetti stacks", and a bunch of other tips and tricks.

By the end of the tutorial, we'll have written a bit less than 700 lines of non-comment, non-blank, lines of code. With this small amount of code, we'll have built up a very reasonable compiler for a non-trivial language including a hand-written lexer, parser, AST, as well as code generation support with a JIT compiler. While other systems may have interesting "hello world" tutorials, I think the breadth of this tutorial is a great testament to the strengths of LLVM and why you should consider it if you're interested in language or compiler design.

A note about this tutorial: we expect you to extend the language and play with it on your own. Take the code and go crazy hacking away at it, compilers don't need to be scary creatures - it can be a lot of fun to play with languages!

The Basic Language

This tutorial will be illustrated with a toy language that we'll call "Kaleidoscope" (derived from "meaning beautiful, form, and view"). Kaleidoscope is a procedural language that allows you to define functions, use conditionals, math, etc. Over the course of the tutorial, we'll extend Kaleidoscope to support the if/then/else construct, a for loop, user defined operators, JIT compilation with a simple command line interface, etc.

Because we want to keep things simple, the only datatype in Kaleidoscope is a 64-bit floating point type (aka 'double' in C parlance). As such, all values are implicitly double precision and the language doesn't require type declarations. This gives the language a very nice and simple syntax. For example, the following simple example computes Fibonacci numbers:

```
# Compute the x'th fibonacci number.
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2)

# This expression will compute the 40th number.
fib(40)
```

We also allow Kaleidoscope to call into standard library functions (the LLVM JIT makes this completely trivial). This means that you can use the 'extern' keyword to define a function before you use it (this is also useful for mutually recursive functions). For example:

```
extern sin(arg);
extern cos(arg);
extern atan2(arg1 arg2);

atan2(sin(.4), cos(42))
```

A more interesting example is included in Chapter 6 where we write a little Kaleidoscope application that displays a Mandelbrot Set at various levels of magnification.

Lets dive into the implementation of this language!

The Lexer

When it comes to implementing a language, the first thing needed is the ability to process a text file and recognize what it says. The traditional way to do this is to use a "lexer" (aka 'scanner') to break the input up into "tokens". Each token returned by the lexer includes a token code and potentially some metadata (e.g. the numeric value of a number). First, we define the possibilities:

```
// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2, tok_extern = -3,

  // primary
  tok_identifier = -4, tok_number = -5,
};

static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number
```

Each token returned by our lexer will either be one of the Token enum values or it will be an 'unknown' character like '+', which is returned as its ASCII value. If the current token is an identifier, the `IdentifierStr` global variable holds the name of the identifier. If the current token is a numeric literal (like 1.0), `NumVal` holds its value. Note that we use global variables for simplicity, this is not the best choice for a real language implementation :).

The actual implementation of the lexer is a single function named `gettok`. The `gettok` function is called to return the next token from standard input. Its definition starts as:

```
/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();
```

`gettok` works by calling the C `getchar()` function to read characters one at a time from standard input. It eats them as it recognizes them and stores the last character read, but not processed, in LastChar. The first thing that it has to do is ignore whitespace between tokens. This is accomplished with the loop above.

The next thing `gettok` needs to do is recognize identifiers and specific keywords like "def". Kaleidoscope does this with this simple loop:

```
if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
  IdentifierStr = LastChar;
  while (isalnum((LastChar = getchar())))
    IdentifierStr += LastChar;

  if (IdentifierStr == "def") return tok_def;
  if (IdentifierStr == "extern") return tok_extern;
  return tok_identifier;
}
```

Note that this code sets the '`IdentifierStr`' global whenever it lexes an identifier. Also, since language keywords are matched by the same loop, we handle them here inline. Numeric values are similar:

```
if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
  std::string NumStr;
  do {
    NumStr += LastChar;
    LastChar = getchar();
  } while (isdigit(LastChar) || LastChar == '.');

  NumVal = strtod(NumStr.c_str(), 0);
  return tok_number;
}
```

This is all pretty straight-forward code for processing input. When reading a numeric value from input, we use the C `strtod` function to convert it to a numeric value that we store in `NumVal`. Note that this isn't doing sufficient error checking: it will incorrectly read "1.23.45.67" and handle it as if you typed in "1.23". Feel free to extend it :). Next we handle comments:

```
if (LastChar == '#') {
  // Comment until end of line.
  do LastChar = getchar();
  while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

  if (LastChar != EOF)
    return gettok();
}
```

We handle comments by skipping to the end of the line and then return the next token. Finally, if the input doesn't match one of the above cases, it is either an operator character like '+' or the end of the file. These are handled with this code:

```
  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}
```

With this, we have the complete lexer for the basic Kaleidoscope language (the full code listing for the Lexer is available in the next chapter of the tutorial). Next we'll build a simple parser that uses this to build an Abstract Syntax Tree. When we have that, we'll include a driver so that you can use the lexer and parser together.

Next: Implementing a Parser and AST

Kaleidoscope: Implementing a Parser and AST

Chapter 2 Introduction

Welcome to Chapter 2 of the "Implementing a language with LLVM" tutorial. This chapter shows you how to use the lexer, built in Chapter 1, to build a full parser for our Kaleidoscope language. Once we have a parser, we'll define and build an Abstract Syntax Tree (AST).

The parser we will build uses a combination of Recursive Descent Parsing and Operator-Precedence Parsing to parse the Kaleidoscope language (the latter for binary expressions and the former for everything else). Before we get to parsing though, lets talk about the output of the parser: the Abstract Syntax Tree.

The Abstract Syntax Tree (AST)

The AST for a program captures its behavior in such a way that it is easy for later stages of the compiler (e.g. code generation) to interpret. We basically want one object for each construct in the language, and the AST should closely model the language. In Kaleidoscope, we have expressions, a prototype, and a function object. We'll start with expressions first:

```
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val) : Val(val) {}
};
```

The code above shows the definition of the base ExprAST class and one subclass which we use for numeric literals. The important thing to note about this code is that the NumberExprAST class captures the numeric value of the literal as an instance variable. This allows later phases of the compiler to know what the stored numeric value is.

Right now we only create the AST, so there are no useful accessor methods on them. It would be very easy to add a virtual method to pretty print the code, for example. Here are the other expression AST node definitions that we'll use in the basic form of the Kaleidoscope language:

```
/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;
```

```
public:
  VariableExprAST(const std::string &name) : Name(name) {}
};

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  ExprAST *LHS, *RHS;
public:
  BinaryExprAST(char op, ExprAST *lhs, ExprAST *rhs)
    : Op(op), LHS(lhs), RHS(rhs) {}
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<ExprAST*> Args;
public:
  CallExprAST(const std::string &callee, std::vector<ExprAST*> &args)
    : Callee(callee), Args(args) {}
};
```

This is all (intentionally) rather straight-forward: variables capture the variable name, binary operators capture their opcode (e.g. '+'), and calls capture a function name as well as a list of any argument expressions. One thing that is nice about our AST is that it captures the language features without talking about the syntax of the language. Note that there is no discussion about precedence of binary operators, lexical structure, etc.

For our basic language, these are all of the expression nodes we'll define. Because it doesn't have conditional control flow, it isn't Turing-complete; we'll fix that in a later installment. The two things we need next are a way to talk about the interface to a function, and a way to talk about functions themselves:

```
/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its name, and its argument names (thus implicitly the number
/// of arguments the function takes).
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args)
    : Name(name), Args(args) {}
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  PrototypeAST *Proto;
  ExprAST *Body;
public:
  FunctionAST(PrototypeAST *proto, ExprAST *body)
    : Proto(proto), Body(body) {}
};
```

In Kaleidoscope, functions are typed with just a count of their arguments. Since all values are double precision floating point, the type of each argument doesn't need to be stored anywhere. In a more aggressive and realistic language, the "ExprAST" class would probably have a type field.

With this scaffolding, we can now talk about parsing expressions and function bodies in Kaleidoscope.

Parser Basics

Now that we have an AST to build, we need to define the parser code to build it. The idea here is that we want to parse something like "x+y" (which is returned as three tokens by the lexer) into an AST that could be generated with calls like this:

```
ExprAST *X = new VariableExprAST("x");
ExprAST *Y = new VariableExprAST("y");
ExprAST *Result = new BinaryExprAST('+', X, Y);
```

In order to do this, we'll start by defining some basic helper routines:

```
/// CurTok/getNextToken - Provide a simple token buffer.  CurTok is the current
/// token the parser is looking at.  getNextToken reads another token from the
/// lexer and updates CurTok with its results.
static int CurTok;
static int getNextToken() {
  return CurTok = gettok();
}
```

This implements a simple token buffer around the lexer. This allows us to look one token ahead at what the lexer is returning. Every function in our parser will assume that CurTok is the current token that needs to be parsed.

```
/// Error* - These are little helper functions for error handling.
ExprAST *Error(const char *Str) { fprintf(stderr, "Error: %s\n", Str);return 0;}
PrototypeAST *ErrorP(const char *Str) { Error(Str); return 0; }
FunctionAST *ErrorF(const char *Str) { Error(Str); return 0; }
```

The `Error` routines are simple helper routines that our parser will use to handle errors. The error recovery in our parser will not be the best and is not particular user-friendly, but it will be enough for our tutorial. These routines make it easier to handle errors in routines that have various return types: they always return null.

With these basic helper functions, we can implement the first piece of our grammar: numeric literals.

Basic Expression Parsing

We start with numeric literals, because they are the simplest to process. For each production in our grammar, we'll define a function which parses that production. For numeric literals, we have:

```
/// numberexpr ::= number
static ExprAST *ParseNumberExpr() {
  ExprAST *Result = new NumberExprAST(NumVal);
  getNextToken(); // consume the number
  return Result;
}
```

This routine is very simple: it expects to be called when the current token is a `tok_number` token. It takes the current number value, creates a `NumberExprAST` node, advances the lexer to the next token, and finally returns.

There are some interesting aspects to this. The most important one is that this routine eats all of the tokens that correspond to the production and returns the lexer buffer with the next token (which is not part of the grammar production) ready to go. This is a fairly standard way to go for recursive descent parsers. For a better example, the parenthesis operator is defined like this:

```
/// parenexpr ::= '(' expression ')'
static ExprAST *ParseParenExpr() {
  getNextToken();  // eat (.
  ExprAST *V = ParseExpression();
```

```
  if (!V) return 0;

  if (CurTok != ')')
    return Error("expected ')'");
  getNextToken();  // eat ).
  return V;
}
```

This function illustrates a number of interesting things about the parser:

1) It shows how we use the Error routines. When called, this function expects that the current token is a '(' token, but after parsing the subexpression, it is possible that there is no ')' waiting. For example, if the user types in "(4 x" instead of "(4)", the parser should emit an error. Because errors can occur, the parser needs a way to indicate that they happened: in our parser, we return null on an error.

2) Another interesting aspect of this function is that it uses recursion by calling `ParseExpression` (we will soon see that `ParseExpression` can call `ParseParenExpr`). This is powerful because it allows us to handle recursive grammars, and keeps each production very simple. Note that parentheses do not cause construction of AST nodes themselves. While we could do it this way, the most important role of parentheses are to guide the parser and provide grouping. Once the parser constructs the AST, parentheses are not needed.

The next simple production is for handling variable references and function calls:

```
/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static ExprAST *ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return new VariableExprAST(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<ExprAST*> Args;
  if (CurTok != ')') {
    while (1) {
      ExprAST *Arg = ParseExpression();
      if (!Arg) return 0;
      Args.push_back(Arg);

      if (CurTok == ')') break;

      if (CurTok != ',')
        return Error("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return new CallExprAST(IdName, Args);
}
```

This routine follows the same style as the other routines. (It expects to be called if the current token is a `tok_identifier` token). It also has recursion and error handling. One interesting aspect of this is that it uses *look-*

ahead to determine if the current identifier is a stand alone variable reference or if it is a function call expression. It handles this by checking to see if the token after the identifier is a '(' token, constructing either a `VariableExprAST` or `CallExprAST` node as appropriate.

Now that we have all of our simple expression-parsing logic in place, we can define a helper function to wrap it together into one entry point. We call this class of expressions "primary" expressions, for reasons that will become more clear later in the tutorial. In order to parse an arbitrary primary expression, we need to determine what sort of expression it is:

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  }
}
```

Now that you see the definition of this function, it is more obvious why we can assume the state of CurTok in the various functions. This uses look-ahead to determine which sort of expression is being inspected, and then parses it with a function call.

Now that basic expressions are handled, we need to handle binary expressions. They are a bit more complex.

Binary Expression Parsing

Binary expressions are significantly harder to parse because they are often ambiguous. For example, when given the string "x+y*z", the parser can choose to parse it as either "(x+y)*z" or "x+(y*z)". With common definitions from mathematics, we expect the later parse, because "*" (multiplication) has higher *precedence* than "+" (addition).

There are many ways to handle this, but an elegant and efficient way is to use Operator-Precedence Parsing. This parsing technique uses the precedence of binary operators to guide recursion. To start with, we need a table of precedences:

```
/// BinopPrecedence - This holds the precedence for each binary operator that is
/// defined.
static std::map<char, int> BinopPrecedence;

/// GetTokPrecedence - Get the precedence of the pending binary operator token.
static int GetTokPrecedence() {
  if (!isascii(CurTok))
    return -1;

  // Make sure it's a declared binop.
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0) return -1;
  return TokPrec;
}

int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
```

```
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.
  ...
}
```

For the basic form of Kaleidoscope, we will only support 4 binary operators (this can obviously be extended by you, our brave and intrepid reader). The `GetTokPrecedence` function returns the precedence for the current token, or -1 if the token is not a binary operator. Having a map makes it easy to add new operators and makes it clear that the algorithm doesn't depend on the specific operators involved, but it would be easy enough to eliminate the map and do the comparisons in the `GetTokPrecedence` function. (Or just use a fixed-size array).

With the helper above defined, we can now start parsing binary expressions. The basic idea of operator precedence parsing is to break down an expression with potentially ambiguous binary operators into pieces. Consider ,for example, the expression "a+b+(c+d)*e*f+g". Operator precedence parsing considers this as a stream of primary expressions separated by binary operators. As such, it will first parse the leading primary expression "a", then it will see the pairs [+, b] [+, (c+d)] [*, e] [*, f] and [+, g]. Note that because parentheses are primary expressions, the binary expression parser doesn't need to worry about nested subexpressions like (c+d) at all.

To start, an expression is a primary expression potentially followed by a sequence of [binop,primaryexpr] pairs:

```
/// expression
///   ::= primary binoprhs
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParsePrimary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
}
```

`ParseBinOpRHS` is the function that parses the sequence of pairs for us. It takes a precedence and a pointer to an expression for the part that has been parsed so far. Note that "x" is a perfectly valid expression: As such, "binoprhs" is allowed to be empty, in which case it returns the expression that is passed into it. In our example above, the code passes the expression for "a" into `ParseBinOpRHS` and the current token is "+".

The precedence value passed into `ParseBinOpRHS` indicates the *minimal operator precedence* that the function is allowed to eat. For example, if the current pair stream is [+, x] and `ParseBinOpRHS` is passed in a precedence of 40, it will not consume any tokens (because the precedence of '+' is only 20). With this in mind, `ParseBinOpRHS` starts with:

```
/// binoprhs
///   ::= ('+' primary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  // If this is a binop, find its precedence.
  while (1) {
    int TokPrec = GetTokPrecedence();

    // If this is a binop that binds at least as tightly as the current binop,
    // consume it, otherwise we are done.
    if (TokPrec < ExprPrec)
      return LHS;
```

This code gets the precedence of the current token and checks to see if if is too low. Because we defined invalid tokens to have a precedence of -1, this check implicitly knows that the pair-stream ends when the token stream runs out of binary operators. If this check succeeds, we know that the token is a binary operator and that it will be included in this expression:

```
// Okay, we know this is a binop.
int BinOp = CurTok;
getNextToken();  // eat binop

// Parse the primary expression after the binary operator.
ExprAST *RHS = ParsePrimary();
if (!RHS) return 0;
```

As such, this code eats (and remembers) the binary operator and then parses the primary expression that follows. This builds up the whole pair, the first of which is [+, b] for the running example.

Now that we parsed the left-hand side of an expression and one pair of the RHS sequence, we have to decide which way the expression associates. In particular, we could have "(a+b) binop unparsed" or "a + (b binop unparsed)". To determine this, we look ahead at "binop" to determine its precedence and compare it to BinOp's precedence (which is '+' in this case):

```
// If BinOp binds less tightly with RHS than the operator after RHS, let
// the pending operator take RHS as its LHS.
int NextPrec = GetTokPrecedence();
if (TokPrec < NextPrec) {
```

If the precedence of the binop to the right of "RHS" is lower or equal to the precedence of our current operator, then we know that the parentheses associate as "(a+b) binop ...". In our example, the current operator is "+" and the next operator is "+", we know that they have the same precedence. In this case we'll create the AST node for "a+b", and then continue parsing:

```
      ... if body omitted ...
    }

    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }  // loop around to the top of the while loop.
}
```

In our example above, this will turn "a+b+" into "(a+b)" and execute the next iteration of the loop, with "+" as the current token. The code above will eat, remember, and parse "(c+d)" as the primary expression, which makes the current pair equal to [+, (c+d)]. It will then evaluate the 'if' conditional above with "*" as the binop to the right of the primary. In this case, the precedence of "*" is higher than the precedence of "+" so the if condition will be entered.

The critical question left here is "how can the if condition parse the right hand side in full"? In particular, to build the AST correctly for our example, it needs to get all of "(c+d)*e*f" as the RHS expression variable. The code to do this is surprisingly simple (code from the above two blocks duplicated for context):

```
    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec+1, RHS);
      if (RHS == 0) return 0;
    }
    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }  // loop around to the top of the while loop.
}
```

At this point, we know that the binary operator to the RHS of our primary has higher precedence than the binop we are currently parsing. As such, we know that any sequence of pairs whose operators are all higher precedence than "+" should be parsed together and returned as "RHS". To do this, we recursively invoke the `ParseBinOpRHS` function

specifying "TokPrec+1" as the minimum precedence required for it to continue. In our example above, this will cause it to return the AST node for "(c+d)*e*f" as RHS, which is then set as the RHS of the '+' expression.

Finally, on the next iteration of the while loop, the "+g" piece is parsed and added to the AST. With this little bit of code (14 non-trivial lines), we correctly handle fully general binary expression parsing in a very elegant way. This was a whirlwind tour of this code, and it is somewhat subtle. I recommend running through it with a few tough examples to see how it works.

This wraps up handling of expressions. At this point, we can point the parser at an arbitrary token stream and build an expression from it, stopping at the first token that is not part of the expression. Next up we need to handle function definitions, etc.

Parsing the Rest

The next thing missing is handling of function prototypes. In Kaleidoscope, these are used both for 'extern' function declarations as well as function body definitions. The code to do this is straight-forward and not very interesting (once you've survived expressions):

```
/// prototype
///   ::= id '(' id* ')'
static PrototypeAST *ParsePrototype() {
  if (CurTok != tok_identifier)
    return ErrorP("Expected function name in prototype");

  std::string FnName = IdentifierStr;
  getNextToken();

  if (CurTok != '(')
    return ErrorP("Expected '(' in prototype");

  // Read the list of argument names.
  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return ErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  return new PrototypeAST(FnName, ArgNames);
}
```

Given this, a function definition is very simple, just a prototype plus an expression to implement the body:

```
/// definition ::= 'def' prototype expression
static FunctionAST *ParseDefinition() {
  getNextToken();  // eat def.
  PrototypeAST *Proto = ParsePrototype();
  if (Proto == 0) return 0;

  if (ExprAST *E = ParseExpression())
    return new FunctionAST(Proto, E);
  return 0;
}
```

In addition, we support 'extern' to declare functions like 'sin' and 'cos' as well as to support forward declaration of user functions. These 'extern's are just prototypes with no body:

```
/// external ::= 'extern' prototype
static PrototypeAST *ParseExtern() {
  getNextToken();  // eat extern.
  return ParsePrototype();
}
```

Finally, we'll also let the user type in arbitrary top-level expressions and evaluate them on the fly. We will handle this by defining anonymous nullary (zero argument) functions for them:

```
/// toplevelexpr ::= expression
static FunctionAST *ParseTopLevelExpr() {
  if (ExprAST *E = ParseExpression()) {
    // Make an anonymous proto.
    PrototypeAST *Proto = new PrototypeAST("", std::vector<std::string>());
    return new FunctionAST(Proto, E);
  }
  return 0;
}
```

Now that we have all the pieces, let's build a little driver that will let us actually *execute* this code we've built!

The Driver

The driver for this simply invokes all of the parsing pieces with a top-level dispatch loop. There isn't much interesting here, so I'll just include the top-level loop. See below for full code in the "Top-Level Parsing" section.

```
/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (1) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:    return;
    case ';':        getNextToken(); break;  // ignore top-level semicolons.
    case tok_def:    HandleDefinition(); break;
    case tok_extern: HandleExtern(); break;
    default:         HandleTopLevelExpression(); break;
    }
  }
}
```

The most interesting part of this is that we ignore top-level semicolons. Why is this, you ask? The basic reason is that if you type "4 + 5" at the command line, the parser doesn't know whether that is the end of what you will type or not. For example, on the next line you could type "def foo..." in which case 4+5 is the end of a top-level expression. Alternatively you could type "* 6", which would continue the expression. Having top-level semicolons allows you to type "4+5;", and the parser will know you are done.

Conclusions

With just under 400 lines of commented code (240 lines of non-comment, non-blank code), we fully defined our minimal language, including a lexer, parser, and AST builder. With this done, the executable will validate Kaleidoscope code and tell us if it is grammatically invalid. For example, here is a sample interaction:

```
$ ./a.out
ready> def foo(x y) x+foo(y, 4.0);
Parsed a function definition.
ready> def foo(x y) x+y y;
```

```
Parsed a function definition.
Parsed a top-level expr
ready> def foo(x y) x+y );
Parsed a function definition.
Error: unknown token when expecting an expression
ready> extern sin(a);
ready> Parsed an extern
ready> ^D
$
```

There is a lot of room for extension here. You can define new AST nodes, extend the language in many ways, etc. In the next installment, we will describe how to generate LLVM Intermediate Representation (IR) from the AST.

Full Code Listing

Here is the complete code listing for this and the previous chapter. Note that it is fully self-contained: you don't need LLVM or any external libraries at all for this. (Besides the C and C++ standard libraries, of course.) To build this, just compile with:

```
# Compile
clang++ -g -O3 toy.cpp
# Run
./a.out
```

Here is the code:

```
#include <cctype>
#include <cstdio>
#include <cstdlib>
#include <map>
#include <string>
#include <vector>

//===----------------------------------------------------------------------===//
// Lexer
//===----------------------------------------------------------------------===//

// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2, tok_extern = -3,

  // primary
  tok_identifier = -4, tok_number = -5
};

static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number

/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
```

```
  while (isspace(LastChar))
    LastChar = getchar();

  if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getchar())))
      IdentifierStr += LastChar;

    if (IdentifierStr == "def") return tok_def;
    if (IdentifierStr == "extern") return tok_extern;
    return tok_identifier;
  }

  if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
    std::string NumStr;
    do {
      NumStr += LastChar;
      LastChar = getchar();
    } while (isdigit(LastChar) || LastChar == '.');

    NumVal = strtod(NumStr.c_str(), 0);
    return tok_number;
  }

  if (LastChar == '#') {
    // Comment until end of line.
    do LastChar = getchar();
    while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

    if (LastChar != EOF)
      return gettok();
  }

  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}

//===----------------------------------------------------------------------===//
// Abstract Syntax Tree (aka Parse Tree)
//===----------------------------------------------------------------------===//
namespace {
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
public:
  NumberExprAST(double val) {}
};
```

```
/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;
public:
  VariableExprAST(const std::string &name) : Name(name) {}
};

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
public:
  BinaryExprAST(char op, ExprAST *lhs, ExprAST *rhs) {}
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<ExprAST*> Args;
public:
  CallExprAST(const std::string &callee, std::vector<ExprAST*> &args)
    : Callee(callee), Args(args) {}
};

/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its name, and its argument names (thus implicitly the number
/// of arguments the function takes).
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args)
    : Name(name), Args(args) {}

};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
public:
  FunctionAST(PrototypeAST *proto, ExprAST *body) {}
};
} // end anonymous namespace

//===----------------------------------------------------------------------===//
// Parser
//===----------------------------------------------------------------------===//

/// CurTok/getNextToken - Provide a simple token buffer.  CurTok is the current
/// token the parser is looking at.  getNextToken reads another token from the
/// lexer and updates CurTok with its results.
static int CurTok;
static int getNextToken() {
  return CurTok = gettok();
}

/// BinopPrecedence - This holds the precedence for each binary operator that is
/// defined.
static std::map<char, int> BinopPrecedence;

/// GetTokPrecedence - Get the precedence of the pending binary operator token.
```

```
static int GetTokPrecedence() {
  if (!isascii(CurTok))
    return -1;

  // Make sure it's a declared binop.
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0) return -1;
  return TokPrec;
}

/// Error* - These are little helper functions for error handling.
ExprAST *Error(const char *Str) { fprintf(stderr, "Error: %s\n", Str);return 0;}
PrototypeAST *ErrorP(const char *Str) { Error(Str); return 0; }

static ExprAST *ParseExpression();

/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static ExprAST *ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return new VariableExprAST(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<ExprAST*> Args;
  if (CurTok != ')') {
    while (1) {
      ExprAST *Arg = ParseExpression();
      if (!Arg) return 0;
      Args.push_back(Arg);

      if (CurTok == ')') break;

      if (CurTok != ',')
        return Error("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return new CallExprAST(IdName, Args);
}

/// numberexpr ::= number
static ExprAST *ParseNumberExpr() {
  ExprAST *Result = new NumberExprAST(NumVal);
  getNextToken(); // consume the number
  return Result;
}

/// parenexpr ::= '(' expression ')'
```

```
static ExprAST *ParseParenExpr() {
  getNextToken();  // eat (.
  ExprAST *V = ParseExpression();
  if (!V) return 0;

  if (CurTok != ')')
    return Error("expected ')'");
  getNextToken();  // eat ).
  return V;
}

/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  }
}

/// binoprhs
///   ::= ('+' primary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  // If this is a binop, find its precedence.
  while (1) {
    int TokPrec = GetTokPrecedence();

    // If this is a binop that binds at least as tightly as the current binop,
    // consume it, otherwise we are done.
    if (TokPrec < ExprPrec)
      return LHS;

    // Okay, we know this is a binop.
    int BinOp = CurTok;
    getNextToken();  // eat binop

    // Parse the primary expression after the binary operator.
    ExprAST *RHS = ParsePrimary();
    if (!RHS) return 0;

    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec+1, RHS);
      if (RHS == 0) return 0;
    }

    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }
}

/// expression
```

```
///   ::= primary binoprhs
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParsePrimary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
}

/// prototype
///   ::= id '(' id* ')'
static PrototypeAST *ParsePrototype() {
  if (CurTok != tok_identifier)
    return ErrorP("Expected function name in prototype");

  std::string FnName = IdentifierStr;
  getNextToken();

  if (CurTok != '(')
    return ErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return ErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  return new PrototypeAST(FnName, ArgNames);
}

/// definition ::= 'def' prototype expression
static FunctionAST *ParseDefinition() {
  getNextToken();  // eat def.
  PrototypeAST *Proto = ParsePrototype();
  if (Proto == 0) return 0;

  if (ExprAST *E = ParseExpression())
    return new FunctionAST(Proto, E);
  return 0;
}

/// toplevelexpr ::= expression
static FunctionAST *ParseTopLevelExpr() {
  if (ExprAST *E = ParseExpression()) {
    // Make an anonymous proto.
    PrototypeAST *Proto = new PrototypeAST("", std::vector<std::string>());
    return new FunctionAST(Proto, E);
  }
  return 0;
}

/// external ::= 'extern' prototype
static PrototypeAST *ParseExtern() {
  getNextToken();  // eat extern.
  return ParsePrototype();
```

```
}

//===----------------------------------------------------------------------===//
// Top-Level parsing
//===----------------------------------------------------------------------===//

static void HandleDefinition() {
  if (ParseDefinition()) {
    fprintf(stderr, "Parsed a function definition.\n");
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleExtern() {
  if (ParseExtern()) {
    fprintf(stderr, "Parsed an extern\n");
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (ParseTopLevelExpr()) {
    fprintf(stderr, "Parsed a top-level expr\n");
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (1) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:    return;
    case ';':        getNextToken(); break;  // ignore top-level semicolons.
    case tok_def:    HandleDefinition(); break;
    case tok_extern: HandleExtern(); break;
    default:         HandleTopLevelExpression(); break;
    }
  }
}

//===----------------------------------------------------------------------===//
// Main driver code.
//===----------------------------------------------------------------------===//

int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
```

```
  BinopPrecedence['*'] = 40;  // highest.

  // Prime the first token.
  fprintf(stderr, "ready> ");
  getNextToken();

  // Run the main "interpreter loop" now.
  MainLoop();

  return 0;
}
```

Next: Implementing Code Generation to LLVM IR

Kaleidoscope: Code generation to LLVM IR

Chapter 3 Introduction

Welcome to Chapter 3 of the "Implementing a language with LLVM" tutorial. This chapter shows you how to transform the Abstract Syntax Tree, built in Chapter 2, into LLVM IR. This will teach you a little bit about how LLVM does things, as well as demonstrate how easy it is to use. It's much more work to build a lexer and parser than it is to generate LLVM IR code. :)

Please note: the code in this chapter and later require LLVM 2.2 or later. LLVM 2.1 and before will not work with it. Also note that you need to use a version of this tutorial that matches your LLVM release: If you are using an official LLVM release, use the version of the documentation included with your release or on the llvm.org releases page.

Code Generation Setup

In order to generate LLVM IR, we want some simple setup to get started. First we define virtual code generation (codegen) methods in each AST class:

```
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
  virtual Value *Codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val) : Val(val) {}
  virtual Value *Codegen();
```

```
};
...
```

The Codegen() method says to emit IR for that AST node along with all the things it depends on, and they all return an LLVM Value object. "Value" is the class used to represent a "Static Single Assignment (SSA) register" or "SSA value" in LLVM. The most distinct aspect of SSA values is that their value is computed as the related instruction executes, and it does not get a new value until (and if) the instruction re-executes. In other words, there is no way to "change" an SSA value. For more information, please read up on Static Single Assignment - the concepts are really quite natural once you grok them.

Note that instead of adding virtual methods to the ExprAST class hierarchy, it could also make sense to use a visitor pattern or some other way to model this. Again, this tutorial won't dwell on good software engineering practices: for our purposes, adding a virtual method is simplest.

The second thing we want is an "Error" method like we used for the parser, which will be used to report errors found during code generation (for example, use of an undeclared parameter):

```
Value *ErrorV(const char *Str) { Error(Str); return 0; }

static Module *TheModule;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*> NamedValues;
```

The static variables will be used during code generation. `TheModule` is the LLVM construct that contains all of the functions and global variables in a chunk of code. In many ways, it is the top-level structure that the LLVM IR uses to contain code.

The `Builder` object is a helper object that makes it easy to generate LLVM instructions. Instances of the `IRBuilder <http://llvm.org/doxygen/IRBuilder_8h-source.html>`_ class template keep track of the current place to insert instructions and has methods to create new instructions.

The `NamedValues` map keeps track of which values are defined in the current scope and what their LLVM representation is. (In other words, it is a symbol table for the code). In this form of Kaleidoscope, the only things that can be referenced are function parameters. As such, function parameters will be in this map when generating code for their function body.

With these basics in place, we can start talking about how to generate code for each expression. Note that this assumes that the `Builder` has been set up to generate code *into* something. For now, we'll assume that this has already been done, and we'll just use it to emit code.

Expression Code Generation

Generating LLVM code for expression nodes is very straightforward: less than 45 lines of commented code for all four of our expression nodes. First we'll do numeric literals:

```
Value *NumberExprAST::Codegen() {
  return ConstantFP::get(getGlobalContext(), APFloat(Val));
}
```

In the LLVM IR, numeric constants are represented with the `ConstantFP` class, which holds the numeric value in an `APFloat` internally (`APFloat` has the capability of holding floating point constants of Arbitrary Precision). This code basically just creates and returns a `ConstantFP`. Note that in the LLVM IR that constants are all uniqued together and shared. For this reason, the API uses the "foo::get(...)" idiom instead of "new foo(..)" or "foo::Create(..)".

```
Value *VariableExprAST::Codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
```

```
  return V ? V : ErrorV("Unknown variable name");
}
```

References to variables are also quite simple using LLVM. In the simple version of Kaleidoscope, we assume that the variable has already been emitted somewhere and its value is available. In practice, the only values that can be in the `NamedValues` map are function arguments. This code simply checks to see that the specified name is in the map (if not, an unknown variable is being referenced) and returns the value for it. In future chapters, we'll add support for loop induction variables in the symbol table, and for local variables.

```
Value *BinaryExprAST::Codegen() {
  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateFAdd(L, R, "addtmp");
  case '-': return Builder.CreateFSub(L, R, "subtmp");
  case '*': return Builder.CreateFMul(L, R, "multtmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                "booltmp");
  default: return ErrorV("invalid binary operator");
  }
}
```

Binary operators start to get more interesting. The basic idea here is that we recursively emit code for the left-hand side of the expression, then the right-hand side, then we compute the result of the binary expression. In this code, we do a simple switch on the opcode to create the right LLVM instruction.

In the example above, the LLVM builder class is starting to show its value. IRBuilder knows where to insert the newly created instruction, all you have to do is specify what instruction to create (e.g. with `CreateFAdd`), which operands to use (`L` and `R` here) and optionally provide a name for the generated instruction.

One nice thing about LLVM is that the name is just a hint. For instance, if the code above emits multiple "addtmp" variables, LLVM will automatically provide each one with an increasing, unique numeric suffix. Local value names for instructions are purely optional, but it makes it much easier to read the IR dumps.

LLVM instructions are constrained by strict rules: for example, the Left and Right operators of an add instruction must have the same type, and the result type of the add must match the operand types. Because all values in Kaleidoscope are doubles, this makes for very simple code for add, sub and mul.

On the other hand, LLVM specifies that the fcmp instruction always returns an 'i1' value (a one bit integer). The problem with this is that Kaleidoscope wants the value to be a 0.0 or 1.0 value. In order to get these semantics, we combine the fcmp instruction with a uitofp instruction. This instruction converts its input integer into a floating point value by treating the input as an unsigned value. In contrast, if we used the sitofp instruction, the Kaleidoscope '<' operator would return 0.0 and -1.0, depending on the input value.

```
Value *CallExprAST::Codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (CalleeF == 0)
    return ErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return ErrorV("Incorrect # arguments passed");
```

```
  std::vector<Value*> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->Codegen());
    if (ArgsV.back() == 0) return 0;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}
```

Code generation for function calls is quite straightforward with LLVM. The code above initially does a function name lookup in the LLVM Module's symbol table. Recall that the LLVM Module is the container that holds all of the functions we are JIT'ing. By giving each function the same name as what the user specifies, we can use the LLVM symbol table to resolve function names for us.

Once we have the function to call, we recursively codegen each argument that is to be passed in, and create an LLVM call instruction. Note that LLVM uses the native C calling conventions by default, allowing these calls to also call into standard library functions like "sin" and "cos", with no additional effort.

This wraps up our handling of the four basic expressions that we have so far in Kaleidoscope. Feel free to go in and add some more. For example, by browsing the LLVM language reference you'll find several other interesting instructions that are really easy to plug into our basic framework.

Function Code Generation

Code generation for prototypes and functions must handle a number of details, which make their code less beautiful than expression code generation, but allows us to illustrate some important points. First, lets talk about code generation for prototypes: they are used both for function bodies and external function declarations. The code starts with:

```
Function *PrototypeAST::Codegen() {
  // Make the function type:  double(double,double) etc.
  std::vector<Type*> Doubles(Args.size(),
                             Type::getDoubleTy(getGlobalContext()));
  FunctionType *FT = FunctionType::get(Type::getDoubleTy(getGlobalContext()),
                                       Doubles, false);

  Function *F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule);
```

This code packs a lot of power into a few lines. Note first that this function returns a "Function*" instead of a "Value*". Because a "prototype" really talks about the external interface for a function (not the value computed by an expression), it makes sense for it to return the LLVM Function it corresponds to when codegen'd.

The call to `FunctionType::get` creates the `FunctionType` that should be used for a given Prototype. Since all function arguments in Kaleidoscope are of type double, the first line creates a vector of "N" LLVM double types. It then uses the `Functiontype::get` method to create a function type that takes "N" doubles as arguments, returns one double as a result, and that is not vararg (the false parameter indicates this). Note that Types in LLVM are uniqued just like Constants are, so you don't "new" a type, you "get" it.

The final line above actually creates the function that the prototype will correspond to. This indicates the type, linkage and name to use, as well as which module to insert into. "external linkage" means that the function may be defined outside the current module and/or that it is callable by functions outside the module. The Name passed in is the name the user specified: since "`TheModule`" is specified, this name is registered in "`TheModule`"s symbol table, which is used by the function call code above.

```
// If F conflicted, there was already something named 'Name'.  If it has a
// body, don't allow redefinition or reextern.
if (F->getName() != Name) {
  // Delete the one we just made and get the existing one.
```

```
  F->eraseFromParent();
  F = TheModule->getFunction(Name);
```

The Module symbol table works just like the Function symbol table when it comes to name conflicts: if a new function is created with a name that was previously added to the symbol table, the new function will get implicitly renamed when added to the Module. The code above exploits this fact to determine if there was a previous definition of this function.

In Kaleidoscope, I choose to allow redefinitions of functions in two cases: first, we want to allow 'extern'ing a function more than once, as long as the prototypes for the externs match (since all arguments have the same type, we just have to check that the number of arguments match). Second, we want to allow 'extern'ing a function and then defining a body for it. This is useful when defining mutually recursive functions.

In order to implement this, the code above first checks to see if there is a collision on the name of the function. If so, it deletes the function we just created (by calling `eraseFromParent`) and then calling `getFunction` to get the existing function with the specified name. Note that many APIs in LLVM have "erase" forms and "remove" forms. The "remove" form unlinks the object from its parent (e.g. a Function from a Module) and returns it. The "erase" form unlinks the object and then deletes it.

```
  // If F already has a body, reject this.
  if (!F->empty()) {
    ErrorF("redefinition of function");
    return 0;
  }

  // If F took a different number of args, reject.
  if (F->arg_size() != Args.size()) {
    ErrorF("redefinition of function with different # args");
    return 0;
  }
}
```

In order to verify the logic above, we first check to see if the pre-existing function is "empty". In this case, empty means that it has no basic blocks in it, which means it has no body. If it has no body, it is a forward declaration. Since we don't allow anything after a full definition of the function, the code rejects this case. If the previous reference to a function was an 'extern', we simply verify that the number of arguments for that definition and this one match up. If not, we emit an error.

```
  // Set names for all arguments.
  unsigned Idx = 0;
  for (Function::arg_iterator AI = F->arg_begin(); Idx != Args.size();
       ++AI, ++Idx) {
    AI->setName(Args[Idx]);

    // Add arguments to variable symbol table.
    NamedValues[Args[Idx]] = AI;
  }
  return F;
}
```

The last bit of code for prototypes loops over all of the arguments in the function, setting the name of the LLVM Argument objects to match, and registering the arguments in the `NamedValues` map for future use by the `VariableExprAST` AST node. Once this is set up, it returns the Function object to the caller. Note that we don't check for conflicting argument names here (e.g. "extern foo(a b a)"). Doing so would be very straight-forward with the mechanics we have already used above.

```
Function *FunctionAST::Codegen() {
  NamedValues.clear();
```

```
  Function *TheFunction = Proto->Codegen();
  if (TheFunction == 0)
    return 0;
```

Code generation for function definitions starts out simply enough: we just codegen the prototype (Proto) and verify that it is ok. We then clear out the `NamedValues` map to make sure that there isn't anything in it from the last function we compiled. Code generation of the prototype ensures that there is an LLVM Function object that is ready to go for us.

```
// Create a new basic block to start insertion into.
BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);
Builder.SetInsertPoint(BB);

if (Value *RetVal = Body->Codegen()) {
```

Now we get to the point where the `Builder` is set up. The first line creates a new basic block (named "entry"), which is inserted into `TheFunction`. The second line then tells the builder that new instructions should be inserted into the end of the new basic block. Basic blocks in LLVM are an important part of functions that define the Control Flow Graph. Since we don't have any control flow, our functions will only contain one block at this point. We'll fix this in Chapter 5 :).

```
if (Value *RetVal = Body->Codegen()) {
  // Finish off the function.
  Builder.CreateRet(RetVal);

  // Validate the generated code, checking for consistency.
  verifyFunction(*TheFunction);

  return TheFunction;
}
```

Once the insertion point is set up, we call the `CodeGen()` method for the root expression of the function. If no error happens, this emits code to compute the expression into the entry block and returns the value that was computed. Assuming no error, we then create an LLVM ret instruction, which completes the function. Once the function is built, we call `verifyFunction`, which is provided by LLVM. This function does a variety of consistency checks on the generated code, to determine if our compiler is doing everything right. Using this is important: it can catch a lot of bugs. Once the function is finished and validated, we return it.

```
  // Error reading body, remove function.
  TheFunction->eraseFromParent();
  return 0;
}
```

The only piece left here is handling of the error case. For simplicity, we handle this by merely deleting the function we produced with the `eraseFromParent` method. This allows the user to redefine a function that they incorrectly typed in before: if we didn't delete it, it would live in the symbol table, with a body, preventing future redefinition.

This code does have a bug, though. Since the `PrototypeAST::Codegen` can return a previously defined forward declaration, our code can actually delete a forward declaration. There are a number of ways to fix this bug, see what you can come up with! Here is a testcase:

```
extern foo(a b);      # ok, defines foo.
def foo(a b) c;       # error, 'c' is invalid.
def bar() foo(1, 2); # error, unknown function "foo"
```

Driver Changes and Closing Thoughts

For now, code generation to LLVM doesn't really get us much, except that we can look at the pretty IR calls. The sample code inserts calls to Codegen into the "`HandleDefinition`", "`HandleExtern`" etc functions, and then dumps out the LLVM IR. This gives a nice way to look at the LLVM IR for simple functions. For example:

```
ready> 4+5;
Read top-level expression:
define double @0() {
entry:
  ret double 9.000000e+00
}
```

Note how the parser turns the top-level expression into anonymous functions for us. This will be handy when we add JIT support in the next chapter. Also note that the code is very literally transcribed, no optimizations are being performed except simple constant folding done by IRBuilder. We will add optimizations explicitly in the next chapter.

```
ready> def foo(a b) a*a + 2*a*b + b*b;
Read function definition:
define double @foo(double %a, double %b) {
entry:
  %multmp = fmul double %a, %a
  %multmp1 = fmul double 2.000000e+00, %a
  %multmp2 = fmul double %multmp1, %b
  %addtmp = fadd double %multmp, %multmp2
  %multmp3 = fmul double %b, %b
  %addtmp4 = fadd double %addtmp, %multmp3
  ret double %addtmp4
}
```

This shows some simple arithmetic. Notice the striking similarity to the LLVM builder calls that we use to create the instructions.

```
ready> def bar(a) foo(a, 4.0) + bar(31337);
Read function definition:
define double @bar(double %a) {
entry:
  %calltmp = call double @foo(double %a, double 4.000000e+00)
  %calltmp1 = call double @bar(double 3.133700e+04)
  %addtmp = fadd double %calltmp, %calltmp1
  ret double %addtmp
}
```

This shows some function calls. Note that this function will take a long time to execute if you call it. In the future we'll add conditional control flow to actually make recursion useful :).

```
ready> extern cos(x);
Read extern:
declare double @cos(double)

ready> cos(1.234);
Read top-level expression:
define double @1() {
entry:
  %calltmp = call double @cos(double 1.234000e+00)
  ret double %calltmp
}
```

This shows an extern for the libm "cos" function, and a call to it.

```
ready> ^D
; ModuleID = 'my cool jit'

define double @0() {
entry:
  %addtmp = fadd double 4.000000e+00, 5.000000e+00
  ret double %addtmp
}

define double @foo(double %a, double %b) {
entry:
  %multmp = fmul double %a, %a
  %multmp1 = fmul double 2.000000e+00, %a
  %multmp2 = fmul double %multmp1, %b
  %addtmp = fadd double %multmp, %multmp2
  %multmp3 = fmul double %b, %b
  %addtmp4 = fadd double %addtmp, %multmp3
  ret double %addtmp4
}

define double @bar(double %a) {
entry:
  %calltmp = call double @foo(double %a, double 4.000000e+00)
  %calltmp1 = call double @bar(double 3.133700e+04)
  %addtmp = fadd double %calltmp, %calltmp1
  ret double %addtmp
}

declare double @cos(double)

define double @1() {
entry:
  %calltmp = call double @cos(double 1.234000e+00)
  ret double %calltmp
}
```

When you quit the current demo, it dumps out the IR for the entire module generated. Here you can see the big picture with all the functions referencing each other.

This wraps up the third chapter of the Kaleidoscope tutorial. Up next, we'll describe how to add JIT codegen and optimizer support to this so we can actually start running code!

Full Code Listing

Here is the complete code listing for our running example, enhanced with the LLVM code generator. Because this uses the LLVM libraries, we need to link them in. To do this, we use the llvm-config tool to inform our makefile/command line about which options to use:

```
# Compile
clang++ -g -O3 toy.cpp `llvm-config --cppflags --ldflags --libs core` -o toy
# Run
./toy
```

Here is the code:

```
#include "llvm/IR/Verifier.h"
#include "llvm/IR/DerivedTypes.h"
```

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include <cctype>
#include <cstdio>
#include <map>
#include <string>
#include <vector>
using namespace llvm;

//===----------------------------------------------------------------------===//
// Lexer
//===----------------------------------------------------------------------===//

// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2, tok_extern = -3,

  // primary
  tok_identifier = -4, tok_number = -5
};

static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number

/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();

  if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getchar())))
      IdentifierStr += LastChar;

    if (IdentifierStr == "def") return tok_def;
    if (IdentifierStr == "extern") return tok_extern;
    return tok_identifier;
  }

  if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
    std::string NumStr;
    do {
      NumStr += LastChar;
      LastChar = getchar();
    } while (isdigit(LastChar) || LastChar == '.');

    NumVal = strtod(NumStr.c_str(), 0);
    return tok_number;
  }
```

```
  if (LastChar == '#') {
    // Comment until end of line.
    do LastChar = getchar();
    while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

    if (LastChar != EOF)
      return gettok();
  }

  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}

//===----------------------------------------------------------------------===//
// Abstract Syntax Tree (aka Parse Tree)
//===----------------------------------------------------------------------===//
namespace {
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
  virtual Value *Codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val) : Val(val) {}
  virtual Value *Codegen();
};

/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;
public:
  VariableExprAST(const std::string &name) : Name(name) {}
  virtual Value *Codegen();
};

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  ExprAST *LHS, *RHS;
public:
  BinaryExprAST(char op, ExprAST *lhs, ExprAST *rhs)
    : Op(op), LHS(lhs), RHS(rhs) {}
  virtual Value *Codegen();
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
```

```
  std::string Callee;
  std::vector<ExprAST*> Args;
public:
  CallExprAST(const std::string &callee, std::vector<ExprAST*> &args)
    : Callee(callee), Args(args) {}
  virtual Value *Codegen();
};

/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its name, and its argument names (thus implicitly the number
/// of arguments the function takes).
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args)
    : Name(name), Args(args) {}

  Function *Codegen();
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  PrototypeAST *Proto;
  ExprAST *Body;
public:
  FunctionAST(PrototypeAST *proto, ExprAST *body)
    : Proto(proto), Body(body) {}

  Function *Codegen();
};
} // end anonymous namespace

//===----------------------------------------------------------------------===//
// Parser
//===----------------------------------------------------------------------===//

/// CurTok/getNextToken - Provide a simple token buffer.  CurTok is the current
/// token the parser is looking at.  getNextToken reads another token from the
/// lexer and updates CurTok with its results.
static int CurTok;
static int getNextToken() {
  return CurTok = gettok();
}

/// BinopPrecedence - This holds the precedence for each binary operator that is
/// defined.
static std::map<char, int> BinopPrecedence;

/// GetTokPrecedence - Get the precedence of the pending binary operator token.
static int GetTokPrecedence() {
  if (!isascii(CurTok))
    return -1;

  // Make sure it's a declared binop.
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0) return -1;
  return TokPrec;
```

```
}

/// Error* - These are little helper functions for error handling.
ExprAST *Error(const char *Str) { fprintf(stderr, "Error: %s\n", Str);return 0;}
PrototypeAST *ErrorP(const char *Str) { Error(Str); return 0; }
FunctionAST *ErrorF(const char *Str) { Error(Str); return 0; }

static ExprAST *ParseExpression();

/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static ExprAST *ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return new VariableExprAST(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<ExprAST*> Args;
  if (CurTok != ')') {
    while (1) {
      ExprAST *Arg = ParseExpression();
      if (!Arg) return 0;
      Args.push_back(Arg);

      if (CurTok == ')') break;

      if (CurTok != ',')
        return Error("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return new CallExprAST(IdName, Args);
}

/// numberexpr ::= number
static ExprAST *ParseNumberExpr() {
  ExprAST *Result = new NumberExprAST(NumVal);
  getNextToken(); // consume the number
  return Result;
}

/// parenexpr ::= '(' expression ')'
static ExprAST *ParseParenExpr() {
  getNextToken();  // eat (.
  ExprAST *V = ParseExpression();
  if (!V) return 0;

  if (CurTok != ')')
    return Error("expected ')'");
```

```
  getNextToken();  // eat ).
  return V;
}

/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  }
}

/// binoprhs
///   ::= ('+' primary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  // If this is a binop, find its precedence.
  while (1) {
    int TokPrec = GetTokPrecedence();

    // If this is a binop that binds at least as tightly as the current binop,
    // consume it, otherwise we are done.
    if (TokPrec < ExprPrec)
      return LHS;

    // Okay, we know this is a binop.
    int BinOp = CurTok;
    getNextToken();  // eat binop

    // Parse the primary expression after the binary operator.
    ExprAST *RHS = ParsePrimary();
    if (!RHS) return 0;

    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec+1, RHS);
      if (RHS == 0) return 0;
    }

    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }
}

/// expression
///   ::= primary binoprhs
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParsePrimary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
```

```
}

/// prototype
///   ::= id '(' id* ')'
static PrototypeAST *ParsePrototype() {
  if (CurTok != tok_identifier)
    return ErrorP("Expected function name in prototype");

  std::string FnName = IdentifierStr;
  getNextToken();

  if (CurTok != '(')
    return ErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return ErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  return new PrototypeAST(FnName, ArgNames);
}

/// definition ::= 'def' prototype expression
static FunctionAST *ParseDefinition() {
  getNextToken();  // eat def.
  PrototypeAST *Proto = ParsePrototype();
  if (Proto == 0) return 0;

  if (ExprAST *E = ParseExpression())
    return new FunctionAST(Proto, E);
  return 0;
}

/// toplevelexpr ::= expression
static FunctionAST *ParseTopLevelExpr() {
  if (ExprAST *E = ParseExpression()) {
    // Make an anonymous proto.
    PrototypeAST *Proto = new PrototypeAST("", std::vector<std::string>());
    return new FunctionAST(Proto, E);
  }
  return 0;
}

/// external ::= 'extern' prototype
static PrototypeAST *ParseExtern() {
  getNextToken();  // eat extern.
  return ParsePrototype();
}

//===----------------------------------------------------------------------===//
// Code Generation
//===----------------------------------------------------------------------===//

static Module *TheModule;
```

```
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*> NamedValues;

Value *ErrorV(const char *Str) { Error(Str); return 0; }

Value *NumberExprAST::Codegen() {
  return ConstantFP::get(getGlobalContext(), APFloat(Val));
}

Value *VariableExprAST::Codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  return V ? V : ErrorV("Unknown variable name");
}

Value *BinaryExprAST::Codegen() {
  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateFAdd(L, R, "addtmp");
  case '-': return Builder.CreateFSub(L, R, "subtmp");
  case '*': return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                "booltmp");
  default: return ErrorV("invalid binary operator");
  }
}

Value *CallExprAST::Codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (CalleeF == 0)
    return ErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return ErrorV("Incorrect # arguments passed");

  std::vector<Value*> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->Codegen());
    if (ArgsV.back() == 0) return 0;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}

Function *PrototypeAST::Codegen() {
  // Make the function type:  double(double,double) etc.
  std::vector<Type*> Doubles(Args.size(),
                             Type::getDoubleTy(getGlobalContext()));
  FunctionType *FT = FunctionType::get(Type::getDoubleTy(getGlobalContext()),
                                       Doubles, false);
```

```
  Function *F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule);

  // If F conflicted, there was already something named 'Name'.  If it has a
  // body, don't allow redefinition or reextern.
  if (F->getName() != Name) {
    // Delete the one we just made and get the existing one.
    F->eraseFromParent();
    F = TheModule->getFunction(Name);

    // If F already has a body, reject this.
    if (!F->empty()) {
      ErrorF("redefinition of function");
      return 0;
    }

    // If F took a different number of args, reject.
    if (F->arg_size() != Args.size()) {
      ErrorF("redefinition of function with different # args");
      return 0;
    }
  }

  // Set names for all arguments.
  unsigned Idx = 0;
  for (Function::arg_iterator AI = F->arg_begin(); Idx != Args.size();
       ++AI, ++Idx) {
    AI->setName(Args[Idx]);

    // Add arguments to variable symbol table.
    NamedValues[Args[Idx]] = AI;
  }

  return F;
}

Function *FunctionAST::Codegen() {
  NamedValues.clear();

  Function *TheFunction = Proto->Codegen();
  if (TheFunction == 0)
    return 0;

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);
  Builder.SetInsertPoint(BB);

  if (Value *RetVal = Body->Codegen()) {
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);

    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();
```

```
  return 0;
}

//===----------------------------------------------------------------------===//
// Top-Level parsing and JIT Driver
//===----------------------------------------------------------------------===//

static void HandleDefinition() {
  if (FunctionAST *F = ParseDefinition()) {
    if (Function *LF = F->Codegen()) {
      fprintf(stderr, "Read function definition:");
      LF->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleExtern() {
  if (PrototypeAST *P = ParseExtern()) {
    if (Function *F = P->Codegen()) {
      fprintf(stderr, "Read extern: ");
      F->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (FunctionAST *F = ParseTopLevelExpr()) {
    if (Function *LF = F->Codegen()) {
      fprintf(stderr, "Read top-level expression:");
      LF->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (1) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:    return;
    case ';':        getNextToken(); break;  // ignore top-level semicolons.
    case tok_def:    HandleDefinition(); break;
    case tok_extern: HandleExtern(); break;
    default:         HandleTopLevelExpression(); break;
    }
  }
}
```

```
//===----------------------------------------------------------------------===//
// "Library" functions that can be "extern'd" from user code.
//===----------------------------------------------------------------------===//

/// putchard - putchar that takes a double and returns 0.
extern "C"
double putchard(double X) {
  putchar((char)X);
  return 0;
}

//===----------------------------------------------------------------------===//
// Main driver code.
//===----------------------------------------------------------------------===//

int main() {
  LLVMContext &Context = getGlobalContext();

  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.

  // Prime the first token.
  fprintf(stderr, "ready> ");
  getNextToken();

  // Make the module, which holds all the code.
  TheModule = new Module("my cool jit", Context);

  // Run the main "interpreter loop" now.
  MainLoop();

  // Print out all of the generated code.
  TheModule->dump();

  return 0;
}
```

Next: Adding JIT and Optimizer Support

Kaleidoscope: Adding JIT and Optimizer Support

Chapter 4 Introduction

Welcome to Chapter 4 of the "Implementing a language with LLVM" tutorial. Chapters 1-3 described the implementation of a simple language and added support for generating LLVM IR. This chapter describes two new techniques: adding optimizer support to your language, and adding JIT compiler support. These additions will demonstrate how to get nice, efficient code for the Kaleidoscope language.

Trivial Constant Folding

Our demonstration for Chapter 3 is elegant and easy to extend. Unfortunately, it does not produce wonderful code. The IRBuilder, however, does give us obvious optimizations when compiling simple code:

```
ready> def test(x) 1+2+x;
Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double 3.000000e+00, %x
        ret double %addtmp
}
```

This code is not a literal transcription of the AST built by parsing the input. That would be:

```
ready> def test(x) 1+2+x;
Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double 2.000000e+00, 1.000000e+00
        %addtmp1 = fadd double %addtmp, %x
        ret double %addtmp1
}
```

Constant folding, as seen above, in particular, is a very common and very important optimization: so much so that many language implementors implement constant folding support in their AST representation.

With LLVM, you don't need this support in the AST. Since all calls to build LLVM IR go through the LLVM IR builder, the builder itself checked to see if there was a constant folding opportunity when you call it. If so, it just does the constant fold and return the constant instead of creating an instruction.

Well, that was easy :). In practice, we recommend always using `IRBuilder` when generating code like this. It has no "syntactic overhead" for its use (you don't have to uglify your compiler with constant checks everywhere) and it can dramatically reduce the amount of LLVM IR that is generated in some cases (particular for languages with a macro preprocessor or that use a lot of constants).

On the other hand, the `IRBuilder` is limited by the fact that it does all of its analysis inline with the code as it is built. If you take a slightly more complex example:

```
ready> def test(x) (1+2+x)*(x+(1+2));
ready> Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double 3.000000e+00, %x
        %addtmp1 = fadd double %x, 3.000000e+00
        %multmp = fmul double %addtmp, %addtmp1
        ret double %multmp
}
```

In this case, the LHS and RHS of the multiplication are the same value. We'd really like to see this generate "`tmp = x+3; result = tmp*tmp;`" instead of computing "`x+3`" twice.

Unfortunately, no amount of local analysis will be able to detect and correct this. This requires two transformations: reassociation of expressions (to make the add's lexically identical) and Common Subexpression Elimination (CSE) to delete the redundant add instruction. Fortunately, LLVM provides a broad range of optimizations that you can use, in the form of "passes".

LLVM Optimization Passes

LLVM provides many optimization passes, which do many different sorts of things and have different tradeoffs. Unlike other systems, LLVM doesn't hold to the mistaken notion that one set of optimizations is right for all languages and for all situations. LLVM allows a compiler implementor to make complete decisions about what optimizations to use, in which order, and in what situation.

As a concrete example, LLVM supports both "whole module" passes, which look across as large of body of code as they can (often a whole file, but if run at link time, this can be a substantial portion of the whole program). It also supports and includes "per-function" passes which just operate on a single function at a time, without looking at other functions. For more information on passes and how they are run, see the How to Write a Pass document and the List of LLVM Passes.

For Kaleidoscope, we are currently generating functions on the fly, one at a time, as the user types them in. We aren't shooting for the ultimate optimization experience in this setting, but we also want to catch the easy and quick stuff where possible. As such, we will choose to run a few per-function optimizations as the user types the function in. If we wanted to make a "static Kaleidoscope compiler", we would use exactly the code we have now, except that we would defer running the optimizer until the entire file has been parsed.

In order to get per-function optimizations going, we need to set up a FunctionPassManager to hold and organize the LLVM optimizations that we want to run. Once we have that, we can add a set of optimizations to run. The code looks like this:

```
FunctionPassManager OurFPM(TheModule);

// Set up the optimizer pipeline.  Start with registering info about how the
// target lays out data structures.
OurFPM.add(new DataLayout(*TheExecutionEngine->getDataLayout()));
// Provide basic AliasAnalysis support for GVN.
OurFPM.add(createBasicAliasAnalysisPass());
// Do simple "peephole" optimizations and bit-twiddling optzns.
OurFPM.add(createInstructionCombiningPass());
// Reassociate expressions.
OurFPM.add(createReassociatePass());
// Eliminate Common SubExpressions.
OurFPM.add(createGVNPass());
// Simplify the control flow graph (deleting unreachable blocks, etc).
OurFPM.add(createCFGSimplificationPass());

OurFPM.doInitialization();

// Set the global so the code gen can use this.
TheFPM = &OurFPM;

// Run the main "interpreter loop" now.
MainLoop();
```

This code defines a `FunctionPassManager`, "`OurFPM`". It requires a pointer to the `Module` to construct itself. Once it is set up, we use a series of "add" calls to add a bunch of LLVM passes. The first pass is basically boilerplate, it adds a pass so that later optimizations know how the data structures in the program are laid out. The "`TheExecutionEngine`" variable is related to the JIT, which we will get to in the next section.

In this case, we choose to add 4 optimization passes. The passes we chose here are a pretty standard set of "cleanup" optimizations that are useful for a wide variety of code. I won't delve into what they do but, believe me, they are a good starting place :).

Once the PassManager is set up, we need to make use of it. We do this by running it after our newly created function is constructed (in `FunctionAST::Codegen`), but before it is returned to the client:

```
if (Value *RetVal = Body->Codegen()) {
  // Finish off the function.
  Builder.CreateRet(RetVal);

  // Validate the generated code, checking for consistency.
  verifyFunction(*TheFunction);

  // Optimize the function.
  TheFPM->run(*TheFunction);

  return TheFunction;
}
```

As you can see, this is pretty straightforward. The `FunctionPassManager` optimizes and updates the LLVM Function* in place, improving (hopefully) its body. With this in place, we can try our test above again:

```
ready> def test(x) (1+2+x)*(x+(1+2));
ready> Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double %x, 3.000000e+00
        %multmp = fmul double %addtmp, %addtmp
        ret double %multmp
}
```

As expected, we now get our nicely optimized code, saving a floating point add instruction from every execution of this function.

LLVM provides a wide variety of optimizations that can be used in certain circumstances. Some documentation about the various passes is available, but it isn't very complete. Another good source of ideas can come from looking at the passes that `Clang` runs to get started. The "`opt`" tool allows you to experiment with passes from the command line, so you can see if they do anything.

Now that we have reasonable code coming out of our front-end, lets talk about executing it!

Adding a JIT Compiler

Code that is available in LLVM IR can have a wide variety of tools applied to it. For example, you can run optimizations on it (as we did above), you can dump it out in textual or binary forms, you can compile the code to an assembly file (.s) for some target, or you can JIT compile it. The nice thing about the LLVM IR representation is that it is the "common currency" between many different parts of the compiler.

In this section, we'll add JIT compiler support to our interpreter. The basic idea that we want for Kaleidoscope is to have the user enter function bodies as they do now, but immediately evaluate the top-level expressions they type in. For example, if they type in "1 + 2;", we should evaluate and print out 3. If they define a function, they should be able to call it from the command line.

In order to do this, we first declare and initialize the JIT. This is done by adding a global variable and a call in `main`:

```
static ExecutionEngine *TheExecutionEngine;
...
int main() {
```

```
  ..
  // Create the JIT.  This takes ownership of the module.
  TheExecutionEngine = EngineBuilder(TheModule).create();
  ..
}
```

This creates an abstract "Execution Engine" which can be either a JIT compiler or the LLVM interpreter. LLVM will automatically pick a JIT compiler for you if one is available for your platform, otherwise it will fall back to the interpreter.

Once the `ExecutionEngine` is created, the JIT is ready to be used. There are a variety of APIs that are useful, but the simplest one is the "`getPointerToFunction(F)`" method. This method JIT compiles the specified LLVM Function and returns a function pointer to the generated machine code. In our case, this means that we can change the code that parses a top-level expression to look like this:

```
static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (FunctionAST *F = ParseTopLevelExpr()) {
    if (Function *LF = F->Codegen()) {
      LF->dump();  // Dump the function for exposition purposes.

      // JIT the function, returning a function pointer.
      void *FPtr = TheExecutionEngine->getPointerToFunction(LF);

      // Cast it to the right type (takes no arguments, returns a double) so we
      // can call it as a native function.
      double (*FP)() = (double (*)())(intptr_t)FPtr;
      fprintf(stderr, "Evaluated to %f\n", FP());
    }
```

Recall that we compile top-level expressions into a self-contained LLVM function that takes no arguments and returns the computed double. Because the LLVM JIT compiler matches the native platform ABI, this means that you can just cast the result pointer to a function pointer of that type and call it directly. This means, there is no difference between JIT compiled code and native machine code that is statically linked into your application.

With just these two changes, lets see how Kaleidoscope works now!

```
ready> 4+5;
Read top-level expression:
define double @0() {
entry:
  ret double 9.000000e+00
}

Evaluated to 9.000000
```

Well this looks like it is basically working. The dump of the function shows the "no argument function that always returns double" that we synthesize for each top-level expression that is typed in. This demonstrates very basic functionality, but can we do more?

```
ready> def testfunc(x y) x + y*2;
Read function definition:
define double @testfunc(double %x, double %y) {
entry:
  %multmp = fmul double %y, 2.000000e+00
  %addtmp = fadd double %multmp, %x
  ret double %addtmp
}

ready> testfunc(4, 10);
```

```
Read top-level expression:
define double @1() {
entry:
  %calltmp = call double @testfunc(double 4.000000e+00, double 1.000000e+01)
  ret double %calltmp
}

Evaluated to 24.000000
```

This illustrates that we can now call user code, but there is something a bit subtle going on here. Note that we only invoke the JIT on the anonymous functions that *call testfunc*, but we never invoked it on *testfunc* itself. What actually happened here is that the JIT scanned for all non-JIT'd functions transitively called from the anonymous function and compiled all of them before returning from `getPointerToFunction()`.

The JIT provides a number of other more advanced interfaces for things like freeing allocated machine code, rejit'ing functions to update them, etc. However, even with this simple code, we get some surprisingly powerful capabilities - check this out (I removed the dump of the anonymous functions, you should get the idea by now :) :

```
ready> extern sin(x);
Read extern:
declare double @sin(double)

ready> extern cos(x);
Read extern:
declare double @cos(double)

ready> sin(1.0);
Read top-level expression:
define double @2() {
entry:
  ret double 0x3FEAED548F090CEE
}

Evaluated to 0.841471

ready> def foo(x) sin(x)*sin(x) + cos(x)*cos(x);
Read function definition:
define double @foo(double %x) {
entry:
  %calltmp = call double @sin(double %x)
  %multmp = fmul double %calltmp, %calltmp
  %calltmp2 = call double @cos(double %x)
  %multmp4 = fmul double %calltmp2, %calltmp2
  %addtmp = fadd double %multmp, %multmp4
  ret double %addtmp
}

ready> foo(4.0);
Read top-level expression:
define double @3() {
entry:
  %calltmp = call double @foo(double 4.000000e+00)
  ret double %calltmp
}

Evaluated to 1.000000
```

Whoa, how does the JIT know about sin and cos? The answer is surprisingly simple: in this example, the JIT started execution of a function and got to a function call. It realized that the function was not yet JIT compiled and invoked the

standard set of routines to resolve the function. In this case, there is no body defined for the function, so the JIT ended up calling "`dlsym("sin")`" on the Kaleidoscope process itself. Since "`sin`" is defined within the JIT's address space, it simply patches up calls in the module to call the libm version of `sin` directly.

The LLVM JIT provides a number of interfaces (look in the `ExecutionEngine.h` file) for controlling how unknown functions get resolved. It allows you to establish explicit mappings between IR objects and addresses (useful for LLVM global variables that you want to map to static tables, for example), allows you to dynamically decide on the fly based on the function name, and even allows you to have the JIT compile functions lazily the first time they're called.

One interesting application of this is that we can now extend the language by writing arbitrary C++ code to implement operations. For example, if we add:

```
/// putchard - putchar that takes a double and returns 0.
extern "C"
double putchard(double X) {
  putchar((char)X);
  return 0;
}
```

Now we can produce simple output to the console by using things like: "`extern putchard(x); putchard(120);`", which prints a lowercase 'x' on the console (120 is the ASCII code for 'x'). Similar code could be used to implement file I/O, console input, and many other capabilities in Kaleidoscope.

This completes the JIT and optimizer chapter of the Kaleidoscope tutorial. At this point, we can compile a non-Turing-complete programming language, optimize and JIT compile it in a user-driven way. Next up we'll look into extending the language with control flow constructs, tackling some interesting LLVM IR issues along the way.

Full Code Listing

Here is the complete code listing for our running example, enhanced with the LLVM JIT and optimizer. To build this example, use:

```
# Compile
clang++ -g toy.cpp `llvm-config --cppflags --ldflags --libs core jit native` -O3 -o toy
# Run
./toy
```

If you are compiling this on Linux, make sure to add the "-rdynamic" option as well. This makes sure that the external functions are resolved properly at runtime.

Here is the code:

```
#include "llvm/Analysis/Passes.h"
#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include "llvm/PassManager.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Transforms/Scalar.h"
#include <cctype>
#include <cstdio>
#include <map>
#include <string>
```

```
#include <vector>
using namespace llvm;

//===----------------------------------------------------------------------===//
// Lexer
//===----------------------------------------------------------------------===//

// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2, tok_extern = -3,

  // primary
  tok_identifier = -4, tok_number = -5
};

static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number

/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();

  if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getchar())))
      IdentifierStr += LastChar;

    if (IdentifierStr == "def") return tok_def;
    if (IdentifierStr == "extern") return tok_extern;
    return tok_identifier;
  }

  if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
    std::string NumStr;
    do {
      NumStr += LastChar;
      LastChar = getchar();
    } while (isdigit(LastChar) || LastChar == '.');

    NumVal = strtod(NumStr.c_str(), 0);
    return tok_number;
  }

  if (LastChar == '#') {
    // Comment until end of line.
    do LastChar = getchar();
    while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

    if (LastChar != EOF)
      return gettok();
```

```
  }

  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}

//===----------------------------------------------------------------------===//
// Abstract Syntax Tree (aka Parse Tree)
//===----------------------------------------------------------------------===//
namespace {
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
  virtual Value *Codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val) : Val(val) {}
  virtual Value *Codegen();
};

/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;
public:
  VariableExprAST(const std::string &name) : Name(name) {}
  virtual Value *Codegen();
};

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  ExprAST *LHS, *RHS;
public:
  BinaryExprAST(char op, ExprAST *lhs, ExprAST *rhs)
    : Op(op), LHS(lhs), RHS(rhs) {}
  virtual Value *Codegen();
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<ExprAST*> Args;
public:
  CallExprAST(const std::string &callee, std::vector<ExprAST*> &args)
    : Callee(callee), Args(args) {}
  virtual Value *Codegen();
};
```

```
/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its name, and its argument names (thus implicitly the number
/// of arguments the function takes).
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args)
    : Name(name), Args(args) {}

  Function *Codegen();
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  PrototypeAST *Proto;
  ExprAST *Body;
public:
  FunctionAST(PrototypeAST *proto, ExprAST *body)
    : Proto(proto), Body(body) {}

  Function *Codegen();
};
} // end anonymous namespace

//===----------------------------------------------------------------------===//
// Parser
//===----------------------------------------------------------------------===//

/// CurTok/getNextToken - Provide a simple token buffer.  CurTok is the current
/// token the parser is looking at.  getNextToken reads another token from the
/// lexer and updates CurTok with its results.
static int CurTok;
static int getNextToken() {
  return CurTok = gettok();
}

/// BinopPrecedence - This holds the precedence for each binary operator that is
/// defined.
static std::map<char, int> BinopPrecedence;

/// GetTokPrecedence - Get the precedence of the pending binary operator token.
static int GetTokPrecedence() {
  if (!isascii(CurTok))
    return -1;

  // Make sure it's a declared binop.
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0) return -1;
  return TokPrec;
}

/// Error* - These are little helper functions for error handling.
ExprAST *Error(const char *Str) { fprintf(stderr, "Error: %s\n", Str);return 0;}
PrototypeAST *ErrorP(const char *Str) { Error(Str); return 0; }
FunctionAST *ErrorF(const char *Str) { Error(Str); return 0; }

static ExprAST *ParseExpression();
```

```
/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static ExprAST *ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return new VariableExprAST(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<ExprAST*> Args;
  if (CurTok != ')') {
    while (1) {
      ExprAST *Arg = ParseExpression();
      if (!Arg) return 0;
      Args.push_back(Arg);

      if (CurTok == ')') break;

      if (CurTok != ',')
        return Error("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return new CallExprAST(IdName, Args);
}

/// numberexpr ::= number
static ExprAST *ParseNumberExpr() {
  ExprAST *Result = new NumberExprAST(NumVal);
  getNextToken(); // consume the number
  return Result;
}

/// parenexpr ::= '(' expression ')'
static ExprAST *ParseParenExpr() {
  getNextToken();  // eat (.
  ExprAST *V = ParseExpression();
  if (!V) return 0;

  if (CurTok != ')')
    return Error("expected ')'");
  getNextToken();  // eat ).
  return V;
}

/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
static ExprAST *ParsePrimary() {
```

```
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  }
}

/// binoprhs
///   ::= ('+' primary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  // If this is a binop, find its precedence.
  while (1) {
    int TokPrec = GetTokPrecedence();

    // If this is a binop that binds at least as tightly as the current binop,
    // consume it, otherwise we are done.
    if (TokPrec < ExprPrec)
      return LHS;

    // Okay, we know this is a binop.
    int BinOp = CurTok;
    getNextToken();  // eat binop

    // Parse the primary expression after the binary operator.
    ExprAST *RHS = ParsePrimary();
    if (!RHS) return 0;

    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec+1, RHS);
      if (RHS == 0) return 0;
    }

    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }
}

/// expression
///   ::= primary binoprhs
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParsePrimary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
}

/// prototype
///   ::= id '(' id* ')'
static PrototypeAST *ParsePrototype() {
  if (CurTok != tok_identifier)
    return ErrorP("Expected function name in prototype");

  std::string FnName = IdentifierStr;
```

```
  getNextToken();

  if (CurTok != '(')
    return ErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return ErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  return new PrototypeAST(FnName, ArgNames);
}

/// definition ::= 'def' prototype expression
static FunctionAST *ParseDefinition() {
  getNextToken();  // eat def.
  PrototypeAST *Proto = ParsePrototype();
  if (Proto == 0) return 0;

  if (ExprAST *E = ParseExpression())
    return new FunctionAST(Proto, E);
  return 0;
}

/// toplevelexpr ::= expression
static FunctionAST *ParseTopLevelExpr() {
  if (ExprAST *E = ParseExpression()) {
    // Make an anonymous proto.
    PrototypeAST *Proto = new PrototypeAST("", std::vector<std::string>());
    return new FunctionAST(Proto, E);
  }
  return 0;
}

/// external ::= 'extern' prototype
static PrototypeAST *ParseExtern() {
  getNextToken();  // eat extern.
  return ParsePrototype();
}

//===----------------------------------------------------------------------===//
// Code Generation
//===----------------------------------------------------------------------===//

static Module *TheModule;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*> NamedValues;
static FunctionPassManager *TheFPM;

Value *ErrorV(const char *Str) { Error(Str); return 0; }

Value *NumberExprAST::Codegen() {
  return ConstantFP::get(getGlobalContext(), APFloat(Val));
}
```

```
Value *VariableExprAST::Codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  return V ? V : ErrorV("Unknown variable name");
}

Value *BinaryExprAST::Codegen() {
  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateFAdd(L, R, "addtmp");
  case '-': return Builder.CreateFSub(L, R, "subtmp");
  case '*': return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                "booltmp");
  default: return ErrorV("invalid binary operator");
  }
}

Value *CallExprAST::Codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (CalleeF == 0)
    return ErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return ErrorV("Incorrect # arguments passed");

  std::vector<Value*> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->Codegen());
    if (ArgsV.back() == 0) return 0;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}

Function *PrototypeAST::Codegen() {
  // Make the function type:  double(double,double) etc.
  std::vector<Type*> Doubles(Args.size(),
                             Type::getDoubleTy(getGlobalContext()));
  FunctionType *FT = FunctionType::get(Type::getDoubleTy(getGlobalContext()),
                                       Doubles, false);

  Function *F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule);

  // If F conflicted, there was already something named 'Name'.  If it has a
  // body, don't allow redefinition or reextern.
  if (F->getName() != Name) {
    // Delete the one we just made and get the existing one.
    F->eraseFromParent();
    F = TheModule->getFunction(Name);
```

```
    // If F already has a body, reject this.
    if (!F->empty()) {
      ErrorF("redefinition of function");
      return 0;
    }

    // If F took a different number of args, reject.
    if (F->arg_size() != Args.size()) {
      ErrorF("redefinition of function with different # args");
      return 0;
    }
  }

  // Set names for all arguments.
  unsigned Idx = 0;
  for (Function::arg_iterator AI = F->arg_begin(); Idx != Args.size();
       ++AI, ++Idx) {
    AI->setName(Args[Idx]);

    // Add arguments to variable symbol table.
    NamedValues[Args[Idx]] = AI;
  }

  return F;
}

Function *FunctionAST::Codegen() {
  NamedValues.clear();

  Function *TheFunction = Proto->Codegen();
  if (TheFunction == 0)
    return 0;

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);
  Builder.SetInsertPoint(BB);

  if (Value *RetVal = Body->Codegen()) {
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);

    // Optimize the function.
    TheFPM->run(*TheFunction);

    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();
  return 0;
}

//===----------------------------------------------------------------------===//
// Top-Level parsing and JIT Driver
//===----------------------------------------------------------------------===//
```

```
static ExecutionEngine *TheExecutionEngine;

static void HandleDefinition() {
  if (FunctionAST *F = ParseDefinition()) {
    if (Function *LF = F->Codegen()) {
      fprintf(stderr, "Read function definition:");
      LF->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleExtern() {
  if (PrototypeAST *P = ParseExtern()) {
    if (Function *F = P->Codegen()) {
      fprintf(stderr, "Read extern: ");
      F->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (FunctionAST *F = ParseTopLevelExpr()) {
    if (Function *LF = F->Codegen()) {
      // JIT the function, returning a function pointer.
      void *FPtr = TheExecutionEngine->getPointerToFunction(LF);

      // Cast it to the right type (takes no arguments, returns a double) so we
      // can call it as a native function.
      double (*FP)() = (double (*)())(intptr_t)FPtr;
      fprintf(stderr, "Evaluated to %f\n", FP());
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (1) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:    return;
    case ';':        getNextToken(); break;  // ignore top-level semicolons.
    case tok_def:    HandleDefinition(); break;
    case tok_extern: HandleExtern(); break;
    default:         HandleTopLevelExpression(); break;
    }
  }
}
```

```
//===----------------------------------------------------------------------===//
// "Library" functions that can be "extern'd" from user code.
//===----------------------------------------------------------------------===//

/// putchard - putchar that takes a double and returns 0.
extern "C"
double putchard(double X) {
  putchar((char)X);
  return 0;
}

//===----------------------------------------------------------------------===//
// Main driver code.
//===----------------------------------------------------------------------===//

int main() {
  InitializeNativeTarget();
  LLVMContext &Context = getGlobalContext();

  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.

  // Prime the first token.
  fprintf(stderr, "ready> ");
  getNextToken();

  // Make the module, which holds all the code.
  std::unique_ptr<Module> Owner = make_unique<Module>("my cool jit", Context);
  TheModule = Owner.get();

  // Create the JIT.  This takes ownership of the module.
  std::string ErrStr;
  TheExecutionEngine =
      EngineBuilder(std::move(Owner)).setErrorStr(&ErrStr).create();
  if (!TheExecutionEngine) {
    fprintf(stderr, "Could not create ExecutionEngine: %s\n", ErrStr.c_str());
    exit(1);
  }

  FunctionPassManager OurFPM(TheModule);

  // Set up the optimizer pipeline.  Start with registering info about how the
  // target lays out data structures.
  TheModule->setDataLayout(TheExecutionEngine->getDataLayout());
  OurFPM.add(new DataLayoutPass());
  // Provide basic AliasAnalysis support for GVN.
  OurFPM.add(createBasicAliasAnalysisPass());
  // Do simple "peephole" optimizations and bit-twiddling optzns.
  OurFPM.add(createInstructionCombiningPass());
  // Reassociate expressions.
  OurFPM.add(createReassociatePass());
  // Eliminate Common SubExpressions.
  OurFPM.add(createGVNPass());
  // Simplify the control flow graph (deleting unreachable blocks, etc).
```

```
  OurFPM.add(createCFGSimplificationPass());

  OurFPM.doInitialization();

  // Set the global so the code gen can use this.
  TheFPM = &OurFPM;

  // Run the main "interpreter loop" now.
  MainLoop();

  TheFPM = 0;

  // Print out all of the generated code.
  TheModule->dump();

  return 0;
}
```

Next: Extending the language: control flow

Kaleidoscope: Extending the Language: Control Flow

Chapter 5 Introduction

Welcome to Chapter 5 of the "Implementing a language with LLVM" tutorial. Parts 1-4 described the implementation of the simple Kaleidoscope language and included support for generating LLVM IR, followed by optimizations and a JIT compiler. Unfortunately, as presented, Kaleidoscope is mostly useless: it has no control flow other than call and return. This means that you can't have conditional branches in the code, significantly limiting its power. In this episode of "build that compiler", we'll extend Kaleidoscope to have an if/then/else expression plus a simple 'for' loop.

If/Then/Else

Extending Kaleidoscope to support if/then/else is quite straightforward. It basically requires adding support for this "new" concept to the lexer, parser, AST, and LLVM code emitter. This example is nice, because it shows how easy it is to "grow" a language over time, incrementally extending it as new ideas are discovered.

Before we get going on "how" we add this extension, lets talk about "what" we want. The basic idea is that we want to be able to write this sort of thing:

```
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2);
```

In Kaleidoscope, every construct is an expression: there are no statements. As such, the if/then/else expression needs to return a value like any other. Since we're using a mostly functional form, we'll have it evaluate its conditional, then return the 'then' or 'else' value based on how the condition was resolved. This is very similar to the C "?:" expression.

The semantics of the if/then/else expression is that it evaluates the condition to a boolean equality value: 0.0 is considered to be false and everything else is considered to be true. If the condition is true, the first subexpression is evaluated and returned, if the condition is false, the second subexpression is evaluated and returned. Since Kaleidoscope allows side-effects, this behavior is important to nail down.

Now that we know what we "want", lets break this down into its constituent pieces.

Lexer Extensions for If/Then/Else The lexer extensions are straightforward. First we add new enum values for the relevant tokens:

```
// control
tok_if = -6, tok_then = -7, tok_else = -8,
```

Once we have that, we recognize the new keywords in the lexer. This is pretty simple stuff:

```
...
if (IdentifierStr == "def") return tok_def;
if (IdentifierStr == "extern") return tok_extern;
if (IdentifierStr == "if") return tok_if;
if (IdentifierStr == "then") return tok_then;
if (IdentifierStr == "else") return tok_else;
return tok_identifier;
```

AST Extensions for If/Then/Else To represent the new expression we add a new AST node for it:

```
/// IfExprAST - Expression class for if/then/else.
class IfExprAST : public ExprAST {
  ExprAST *Cond, *Then, *Else;
public:
  IfExprAST(ExprAST *cond, ExprAST *then, ExprAST *_else)
    : Cond(cond), Then(then), Else(_else) {}
  virtual Value *Codegen();
};
```

The AST node just has pointers to the various subexpressions.

Parser Extensions for If/Then/Else Now that we have the relevant tokens coming from the lexer and we have the AST node to build, our parsing logic is relatively straightforward. First we define a new parsing function:

```
/// ifexpr ::= 'if' expression 'then' expression 'else' expression
static ExprAST *ParseIfExpr() {
  getNextToken();  // eat the if.

  // condition.
```

```
  ExprAST *Cond = ParseExpression();
  if (!Cond) return 0;

  if (CurTok != tok_then)
    return Error("expected then");
  getNextToken();  // eat the then

  ExprAST *Then = ParseExpression();
  if (Then == 0) return 0;

  if (CurTok != tok_else)
    return Error("expected else");

  getNextToken();

  ExprAST *Else = ParseExpression();
  if (!Else) return 0;

  return new IfExprAST(Cond, Then, Else);
}
```

Next we hook it up as a primary expression:

```
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  case tok_if:         return ParseIfExpr();
  }
}
```

LLVM IR for If/Then/Else Now that we have it parsing and building the AST, the final piece is adding LLVM code generation support. This is the most interesting part of the if/then/else example, because this is where it starts to introduce new concepts. All of the code above has been thoroughly described in previous chapters.

To motivate the code we want to produce, lets take a look at a simple example. Consider:

```
extern foo();
extern bar();
def baz(x) if x then foo() else bar();
```

If you disable optimizations, the code you'll (soon) get from Kaleidoscope looks like this:

```
declare double @foo()

declare double @bar()

define double @baz(double %x) {
entry:
  %ifcond = fcmp one double %x, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:       ; preds = %entry
  %calltmp = call double @foo()
  br label %ifcont
```

```
else:       ; preds = %entry
  %calltmp1 = call double @bar()
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ %calltmp, %then ], [ %calltmp1, %else ]
  ret double %iftmp
}
```

To visualize the control flow graph, you can use a nifty feature of the LLVM 'opt' tool. If you put this LLVM IR into "t.ll" and run "`llvm-as < t.ll | opt -analyze -view-cfg`", a window will pop up and you'll see this graph:

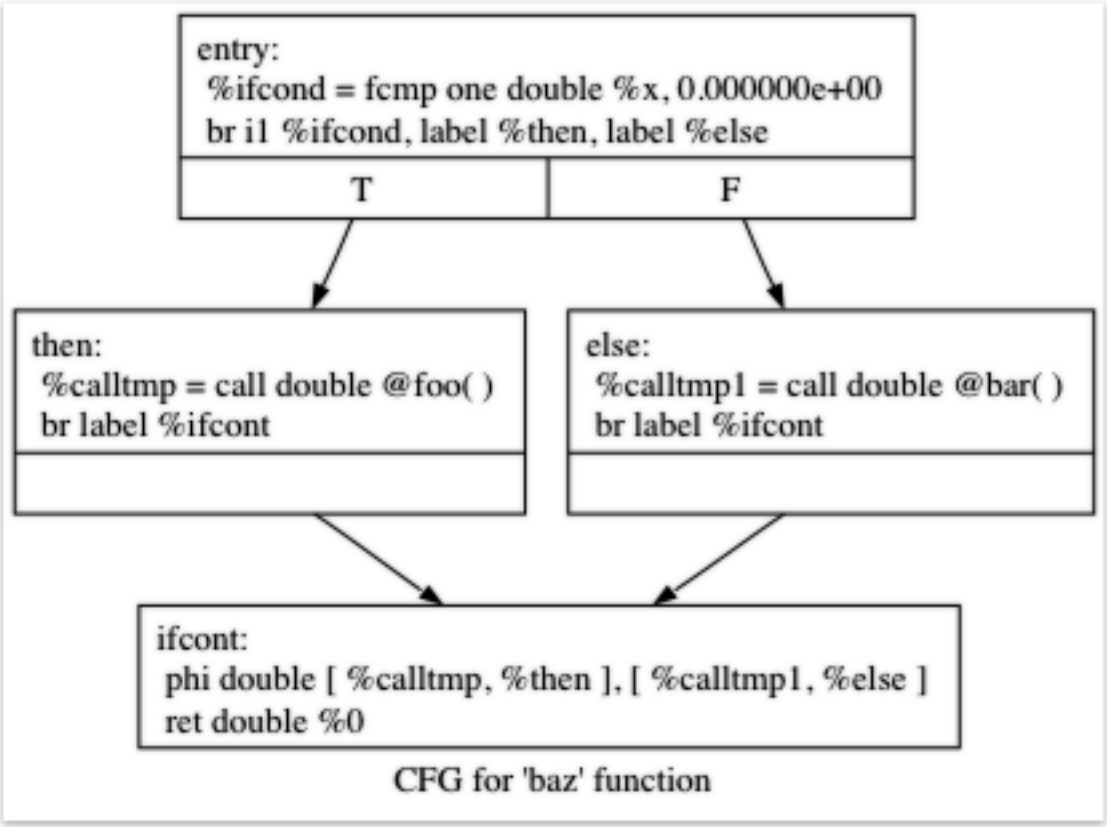


Figure 2.1: Example CFG

Another way to get this is to call "`F->viewCFG()`" or "`F->viewCFGOnly()`" (where F is a "`Function*`") either by inserting actual calls into the code and recompiling or by calling these in the debugger. LLVM has many nice features for visualizing various graphs.

Getting back to the generated code, it is fairly simple: the entry block evaluates the conditional expression ("x" in our case here) and compares the result to 0.0 with the "`fcmp one`" instruction ('one' is "Ordered and Not Equal"). Based on the result of this expression, the code jumps to either the "then" or "else" blocks, which contain the expressions for the true/false cases.

Once the then/else blocks are finished executing, they both branch back to the 'ifcont' block to execute the code that happens after the if/then/else. In this case the only thing left to do is to return to the caller of the function. The question then becomes: how does the code know which expression to return?

The answer to this question involves an important SSA operation: the Phi operation. If you're not familiar with SSA, the wikipedia article is a good introduction and there are various other introductions to it available on your favorite search engine. The short version is that "execution" of the Phi operation requires "remembering" which block control came from. The Phi operation takes on the value corresponding to the input control block. In this case, if control comes in from the "then" block, it gets the value of "calltmp". If control comes from the "else" block, it gets the value of "calltmp1".

At this point, you are probably starting to think "Oh no! This means my simple and elegant front-end will have to start generating SSA form in order to use LLVM!". Fortunately, this is not the case, and we strongly advise *not* implementing an SSA construction algorithm in your front-end unless there is an amazingly good reason to do so. In practice, there are two sorts of values that float around in code written for your average imperative programming language that might need Phi nodes:

1. Code that involves user variables: `x = 1; x = x + 1;`
2. Values that are implicit in the structure of your AST, such as the Phi node in this case.

In Chapter 7 of this tutorial ("mutable variables"), we'll talk about #1 in depth. For now, just believe me that you don't need SSA construction to handle this case. For #2, you have the choice of using the techniques that we will describe for #1, or you can insert Phi nodes directly, if convenient. In this case, it is really really easy to generate the Phi node, so we choose to do it directly.

Okay, enough of the motivation and overview, lets generate code!

Code Generation for If/Then/Else In order to generate code for this, we implement the `Codegen` method for `IfExprAST`:

```
Value *IfExprAST::Codegen() {
  Value *CondV = Cond->Codegen();
  if (CondV == 0) return 0;

  // Convert condition to a bool by comparing equal to 0.0.
  CondV = Builder.CreateFCmpONE(CondV,
                                ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                  "ifcond");
```

This code is straightforward and similar to what we saw before. We emit the expression for the condition, then compare that value to zero to get a truth value as a 1-bit (bool) value.

```
Function *TheFunction = Builder.GetInsertBlock()->getParent();

// Create blocks for the then and else cases.  Insert the 'then' block at the
// end of the function.
BasicBlock *ThenBB = BasicBlock::Create(getGlobalContext(), "then", TheFunction);
BasicBlock *ElseBB = BasicBlock::Create(getGlobalContext(), "else");
BasicBlock *MergeBB = BasicBlock::Create(getGlobalContext(), "ifcont");

Builder.CreateCondBr(CondV, ThenBB, ElseBB);
```

This code creates the basic blocks that are related to the if/then/else statement, and correspond directly to the blocks in the example above. The first line gets the current Function object that is being built. It gets this by asking the builder for the current BasicBlock, and asking that block for its "parent" (the function it is currently embedded into).

Once it has that, it creates three blocks. Note that it passes "TheFunction" into the constructor for the "then" block. This causes the constructor to automatically insert the new block into the end of the specified function. The other two blocks are created, but aren't yet inserted into the function.

Once the blocks are created, we can emit the conditional branch that chooses between them. Note that creating new blocks does not implicitly affect the IRBuilder, so it is still inserting into the block that the condition went into. Also

note that it is creating a branch to the "then" block and the "else" block, even though the "else" block isn't inserted into the function yet. This is all ok: it is the standard way that LLVM supports forward references.

```
// Emit then value.
Builder.SetInsertPoint(ThenBB);

Value *ThenV = Then->Codegen();
if (ThenV == 0) return 0;

Builder.CreateBr(MergeBB);
// Codegen of 'Then' can change the current block, update ThenBB for the PHI.
ThenBB = Builder.GetInsertBlock();
```

After the conditional branch is inserted, we move the builder to start inserting into the "then" block. Strictly speaking, this call moves the insertion point to be at the end of the specified block. However, since the "then" block is empty, it also starts out by inserting at the beginning of the block. :)

Once the insertion point is set, we recursively codegen the "then" expression from the AST. To finish off the "then" block, we create an unconditional branch to the merge block. One interesting (and very important) aspect of the LLVM IR is that it requires all basic blocks to be "terminated" with a control flow instruction such as return or branch. This means that all control flow, *including fall throughs* must be made explicit in the LLVM IR. If you violate this rule, the verifier will emit an error.

The final line here is quite subtle, but is very important. The basic issue is that when we create the Phi node in the merge block, we need to set up the block/value pairs that indicate how the Phi will work. Importantly, the Phi node expects to have an entry for each predecessor of the block in the CFG. Why then, are we getting the current block when we just set it to ThenBB 5 lines above? The problem is that the "Then" expression may actually itself change the block that the Builder is emitting into if, for example, it contains a nested "if/then/else" expression. Because calling Codegen recursively could arbitrarily change the notion of the current block, we are required to get an up-to-date value for code that will set up the Phi node.

```
// Emit else block.
TheFunction->getBasicBlockList().push_back(ElseBB);
Builder.SetInsertPoint(ElseBB);

Value *ElseV = Else->Codegen();
if (ElseV == 0) return 0;

Builder.CreateBr(MergeBB);
// Codegen of 'Else' can change the current block, update ElseBB for the PHI.
ElseBB = Builder.GetInsertBlock();
```

Code generation for the 'else' block is basically identical to codegen for the 'then' block. The only significant difference is the first line, which adds the 'else' block to the function. Recall previously that the 'else' block was created, but not added to the function. Now that the 'then' and 'else' blocks are emitted, we can finish up with the merge code:

```
  // Emit merge block.
  TheFunction->getBasicBlockList().push_back(MergeBB);
  Builder.SetInsertPoint(MergeBB);
  PHINode *PN = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()), 2,
                                  "iftmp");

  PN->addIncoming(ThenV, ThenBB);
  PN->addIncoming(ElseV, ElseBB);
  return PN;
}
```

The first two lines here are now familiar: the first adds the "merge" block to the Function object (it was previously floating, like the else block above). The second block changes the insertion point so that newly created code will go

into the "merge" block. Once that is done, we need to create the PHI node and set up the block/value pairs for the PHI.

Finally, the CodeGen function returns the phi node as the value computed by the if/then/else expression. In our example above, this returned value will feed into the code for the top-level function, which will create the return instruction.

Overall, we now have the ability to execute conditional code in Kaleidoscope. With this extension, Kaleidoscope is a fairly complete language that can calculate a wide variety of numeric functions. Next up we'll add another useful expression that is familiar from non-functional languages...

'for' Loop Expression

Now that we know how to add basic control flow constructs to the language, we have the tools to add more powerful things. Lets add something more aggressive, a 'for' expression:

```
extern putchard(char)
def printstar(n)
  for i = 1, i < n, 1.0 in
    putchard(42);  # ascii 42 = '*'

# print 100 '*' characters
printstar(100);
```

This expression defines a new variable ("i" in this case) which iterates from a starting value, while the condition ("i < n" in this case) is true, incrementing by an optional step value ("1.0" in this case). If the step value is omitted, it defaults to 1.0. While the loop is true, it executes its body expression. Because we don't have anything better to return, we'll just define the loop as always returning 0.0. In the future when we have mutable variables, it will get more useful.

As before, lets talk about the changes that we need to Kaleidoscope to support this.

Lexer Extensions for the 'for' Loop The lexer extensions are the same sort of thing as for if/then/else:

```
... in enum Token ...
// control
tok_if = -6, tok_then = -7, tok_else = -8,
tok_for = -9, tok_in = -10

... in gettok ...
if (IdentifierStr == "def") return tok_def;
if (IdentifierStr == "extern") return tok_extern;
if (IdentifierStr == "if") return tok_if;
if (IdentifierStr == "then") return tok_then;
if (IdentifierStr == "else") return tok_else;
if (IdentifierStr == "for") return tok_for;
if (IdentifierStr == "in") return tok_in;
return tok_identifier;
```

AST Extensions for the 'for' Loop The AST node is just as simple. It basically boils down to capturing the variable name and the constituent expressions in the node.

```
/// ForExprAST - Expression class for for/in.
class ForExprAST : public ExprAST {
  std::string VarName;
  ExprAST *Start, *End, *Step, *Body;
public:
```

```
  ForExprAST(const std::string &varname, ExprAST *start, ExprAST *end,
             ExprAST *step, ExprAST *body)
    : VarName(varname), Start(start), End(end), Step(step), Body(body) {}
  virtual Value *Codegen();
};
```

Parser Extensions for the 'for' Loop The parser code is also fairly standard. The only interesting thing here is handling of the optional step value. The parser code handles it by checking to see if the second comma is present. If not, it sets the step value to null in the AST node:

```
/// forexpr ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression
static ExprAST *ParseForExpr() {
  getNextToken();  // eat the for.

  if (CurTok != tok_identifier)
    return Error("expected identifier after for");

  std::string IdName = IdentifierStr;
  getNextToken();  // eat identifier.

  if (CurTok != '=')
    return Error("expected '=' after for");
  getNextToken();  // eat '='.


  ExprAST *Start = ParseExpression();
  if (Start == 0) return 0;
  if (CurTok != ',')
    return Error("expected ',' after for start value");
  getNextToken();

  ExprAST *End = ParseExpression();
  if (End == 0) return 0;

  // The step value is optional.
  ExprAST *Step = 0;
  if (CurTok == ',') {
    getNextToken();
    Step = ParseExpression();
    if (Step == 0) return 0;
  }

  if (CurTok != tok_in)
    return Error("expected 'in' after for");
  getNextToken();  // eat 'in'.

  ExprAST *Body = ParseExpression();
  if (Body == 0) return 0;

  return new ForExprAST(IdName, Start, End, Step, Body);
}
```

LLVM IR for the 'for' Loop Now we get to the good part: the LLVM IR we want to generate for this thing. With the simple example above, we get this LLVM IR (note that this dump is generated with optimizations disabled for clarity):

```
declare double @putchard(double)

define double @printstar(double %n) {
entry:
  ; initial value = 1.0 (inlined into phi)
  br label %loop

loop:       ; preds = %loop, %entry
  %i = phi double [ 1.000000e+00, %entry ], [ %nextvar, %loop ]
  ; body
  %calltmp = call double @putchard(double 4.200000e+01)
  ; increment
  %nextvar = fadd double %i, 1.000000e+00

  ; termination test
  %cmptmp = fcmp ult double %i, %n
  %booltmp = uitofp i1 %cmptmp to double
  %loopcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %loopcond, label %loop, label %afterloop

afterloop:      ; preds = %loop
  ; loop always returns 0.0
  ret double 0.000000e+00
}
```

This loop contains all the same constructs we saw before: a phi node, several expressions, and some basic blocks. Lets see how this fits together.

Code Generation for the 'for' Loop The first part of Codegen is very simple: we just output the start expression for the loop value:

```
Value *ForExprAST::Codegen() {
  // Emit the start code first, without 'variable' in scope.
  Value *StartVal = Start->Codegen();
  if (StartVal == 0) return 0;
```

With this out of the way, the next step is to set up the LLVM basic block for the start of the loop body. In the case above, the whole loop body is one block, but remember that the body code itself could consist of multiple blocks (e.g. if it contains an if/then/else or a for/in expression).

```
// Make the new basic block for the loop header, inserting after current
// block.
Function *TheFunction = Builder.GetInsertBlock()->getParent();
BasicBlock *PreheaderBB = Builder.GetInsertBlock();
BasicBlock *LoopBB = BasicBlock::Create(getGlobalContext(), "loop", TheFunction);

// Insert an explicit fall through from the current block to the LoopBB.
Builder.CreateBr(LoopBB);
```

This code is similar to what we saw for if/then/else. Because we will need it to create the Phi node, we remember the block that falls through into the loop. Once we have that, we create the actual block that starts the loop and create an unconditional branch for the fall-through between the two blocks.

```
// Start insertion in LoopBB.
Builder.SetInsertPoint(LoopBB);

// Start the PHI node with an entry for Start.
```

```
PHINode *Variable = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()), 2, VarName.c_str());
Variable->addIncoming(StartVal, PreheaderBB);
```

Now that the "preheader" for the loop is set up, we switch to emitting code for the loop body. To begin with, we move the insertion point and create the PHI node for the loop induction variable. Since we already know the incoming value for the starting value, we add it to the Phi node. Note that the Phi will eventually get a second value for the backedge, but we can't set it up yet (because it doesn't exist!).

```
// Within the loop, the variable is defined equal to the PHI node.  If it
// shadows an existing variable, we have to restore it, so save it now.
Value *OldVal = NamedValues[VarName];
NamedValues[VarName] = Variable;

// Emit the body of the loop.  This, like any other expr, can change the
// current BB.  Note that we ignore the value computed by the body, but don't
// allow an error.
if (Body->Codegen() == 0)
  return 0;
```

Now the code starts to get more interesting. Our 'for' loop introduces a new variable to the symbol table. This means that our symbol table can now contain either function arguments or loop variables. To handle this, before we codegen the body of the loop, we add the loop variable as the current value for its name. Note that it is possible that there is a variable of the same name in the outer scope. It would be easy to make this an error (emit an error and return null if there is already an entry for VarName) but we choose to allow shadowing of variables. In order to handle this correctly, we remember the Value that we are potentially shadowing in `OldVal` (which will be null if there is no shadowed variable).

Once the loop variable is set into the symbol table, the code recursively codegen's the body. This allows the body to use the loop variable: any references to it will naturally find it in the symbol table.

```
// Emit the step value.
Value *StepVal;
if (Step) {
  StepVal = Step->Codegen();
  if (StepVal == 0) return 0;
} else {
  // If not specified, use 1.0.
  StepVal = ConstantFP::get(getGlobalContext(), APFloat(1.0));
}

Value *NextVar = Builder.CreateFAdd(Variable, StepVal, "nextvar");
```

Now that the body is emitted, we compute the next value of the iteration variable by adding the step value, or 1.0 if it isn't present. '`NextVar`' will be the value of the loop variable on the next iteration of the loop.

```
// Compute the end condition.
Value *EndCond = End->Codegen();
if (EndCond == 0) return EndCond;

// Convert condition to a bool by comparing equal to 0.0.
EndCond = Builder.CreateFCmpONE(EndCond,
                            ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                "loopcond");
```

Finally, we evaluate the exit value of the loop, to determine whether the loop should exit. This mirrors the condition evaluation for the if/then/else statement.

```
// Create the "after loop" block and insert it.
BasicBlock *LoopEndBB = Builder.GetInsertBlock();
```

```
BasicBlock *AfterBB = BasicBlock::Create(getGlobalContext(), "afterloop", TheFunction);

// Insert the conditional branch into the end of LoopEndBB.
Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

// Any new code will be inserted in AfterBB.
Builder.SetInsertPoint(AfterBB);
```

With the code for the body of the loop complete, we just need to finish up the control flow for it. This code remembers the end block (for the phi node), then creates the block for the loop exit ("afterloop"). Based on the value of the exit condition, it creates a conditional branch that chooses between executing the loop again and exiting the loop. Any future code is emitted in the "afterloop" block, so it sets the insertion position to it.

```
  // Add a new entry to the PHI node for the backedge.
  Variable->addIncoming(NextVar, LoopEndBB);

  // Restore the unshadowed variable.
  if (OldVal)
    NamedValues[VarName] = OldVal;
  else
    NamedValues.erase(VarName);

  // for expr always returns 0.0.
  return Constant::getNullValue(Type::getDoubleTy(getGlobalContext()));
}
```

The final code handles various cleanups: now that we have the "NextVar" value, we can add the incoming value to the loop PHI node. After that, we remove the loop variable from the symbol table, so that it isn't in scope after the for loop. Finally, code generation of the for loop always returns 0.0, so that is what we return from `ForExprAST::Codegen`.

With this, we conclude the "adding control flow to Kaleidoscope" chapter of the tutorial. In this chapter we added two control flow constructs, and used them to motivate a couple of aspects of the LLVM IR that are important for front-end implementors to know. In the next chapter of our saga, we will get a bit crazier and add user-defined operators to our poor innocent language.

Full Code Listing

Here is the complete code listing for our running example, enhanced with the if/then/else and for expressions.. To build this example, use:

```
# Compile
clang++ -g toy.cpp `llvm-config --cppflags --ldflags --libs core jit native` -O3 -o toy
# Run
./toy
```

Here is the code:

```
#include "llvm/Analysis/Passes.h"
#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include "llvm/PassManager.h"
#include "llvm/Support/TargetSelect.h"
```

```
#include "llvm/Transforms/Scalar.h"
#include <cctype>
#include <cstdio>
#include <map>
#include <string>
#include <vector>
using namespace llvm;

//===----------------------------------------------------------------------===//
// Lexer
//===----------------------------------------------------------------------===//

// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2, tok_extern = -3,

  // primary
  tok_identifier = -4, tok_number = -5,

  // control
  tok_if = -6, tok_then = -7, tok_else = -8,
  tok_for = -9, tok_in = -10
};

static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number

/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();

  if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getchar())))
      IdentifierStr += LastChar;

    if (IdentifierStr == "def") return tok_def;
    if (IdentifierStr == "extern") return tok_extern;
    if (IdentifierStr == "if") return tok_if;
    if (IdentifierStr == "then") return tok_then;
    if (IdentifierStr == "else") return tok_else;
    if (IdentifierStr == "for") return tok_for;
    if (IdentifierStr == "in") return tok_in;
    return tok_identifier;
  }

  if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
    std::string NumStr;
    do {
      NumStr += LastChar;
```

```
      LastChar = getchar();
    } while (isdigit(LastChar) || LastChar == '.');

    NumVal = strtod(NumStr.c_str(), 0);
    return tok_number;
  }

  if (LastChar == '#') {
    // Comment until end of line.
    do LastChar = getchar();
    while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

    if (LastChar != EOF)
      return gettok();
  }

  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}

//===----------------------------------------------------------------------===//
// Abstract Syntax Tree (aka Parse Tree)
//===----------------------------------------------------------------------===//
namespace {
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
  virtual Value *Codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val) : Val(val) {}
  virtual Value *Codegen();
};

/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;
public:
  VariableExprAST(const std::string &name) : Name(name) {}
  virtual Value *Codegen();
};

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  ExprAST *LHS, *RHS;
public:
```

```
  BinaryExprAST(char op, ExprAST *lhs, ExprAST *rhs)
    : Op(op), LHS(lhs), RHS(rhs) {}
  virtual Value *Codegen();
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<ExprAST*> Args;
public:
  CallExprAST(const std::string &callee, std::vector<ExprAST*> &args)
    : Callee(callee), Args(args) {}
  virtual Value *Codegen();
};

/// IfExprAST - Expression class for if/then/else.
class IfExprAST : public ExprAST {
  ExprAST *Cond, *Then, *Else;
public:
  IfExprAST(ExprAST *cond, ExprAST *then, ExprAST *_else)
  : Cond(cond), Then(then), Else(_else) {}
  virtual Value *Codegen();
};

/// ForExprAST - Expression class for for/in.
class ForExprAST : public ExprAST {
  std::string VarName;
  ExprAST *Start, *End, *Step, *Body;
public:
  ForExprAST(const std::string &varname, ExprAST *start, ExprAST *end,
             ExprAST *step, ExprAST *body)
    : VarName(varname), Start(start), End(end), Step(step), Body(body) {}
  virtual Value *Codegen();
};

/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its name, and its argument names (thus implicitly the number
/// of arguments the function takes).
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args)
    : Name(name), Args(args) {}

  Function *Codegen();
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  PrototypeAST *Proto;
  ExprAST *Body;
public:
  FunctionAST(PrototypeAST *proto, ExprAST *body)
    : Proto(proto), Body(body) {}

  Function *Codegen();
};
```

```
} // end anonymous namespace

//===----------------------------------------------------------------------===//
// Parser
//===----------------------------------------------------------------------===//

/// CurTok/getNextToken - Provide a simple token buffer.  CurTok is the current
/// token the parser is looking at.  getNextToken reads another token from the
/// lexer and updates CurTok with its results.
static int CurTok;
static int getNextToken() {
  return CurTok = gettok();
}

/// BinopPrecedence - This holds the precedence for each binary operator that is
/// defined.
static std::map<char, int> BinopPrecedence;

/// GetTokPrecedence - Get the precedence of the pending binary operator token.
static int GetTokPrecedence() {
  if (!isascii(CurTok))
    return -1;

  // Make sure it's a declared binop.
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0) return -1;
  return TokPrec;
}

/// Error* - These are little helper functions for error handling.
ExprAST *Error(const char *Str) { fprintf(stderr, "Error: %s\n", Str);return 0;}
PrototypeAST *ErrorP(const char *Str) { Error(Str); return 0; }
FunctionAST *ErrorF(const char *Str) { Error(Str); return 0; }

static ExprAST *ParseExpression();

/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static ExprAST *ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return new VariableExprAST(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<ExprAST*> Args;
  if (CurTok != ')') {
    while (1) {
      ExprAST *Arg = ParseExpression();
      if (!Arg) return 0;
      Args.push_back(Arg);

      if (CurTok == ')') break;
```

```
      if (CurTok != ',')
        return Error("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return new CallExprAST(IdName, Args);
}

/// numberexpr ::= number
static ExprAST *ParseNumberExpr() {
  ExprAST *Result = new NumberExprAST(NumVal);
  getNextToken(); // consume the number
  return Result;
}

/// parenexpr ::= '(' expression ')'
static ExprAST *ParseParenExpr() {
  getNextToken();  // eat (.
  ExprAST *V = ParseExpression();
  if (!V) return 0;

  if (CurTok != ')')
    return Error("expected ')'");
  getNextToken();  // eat ).
  return V;
}

/// ifexpr ::= 'if' expression 'then' expression 'else' expression
static ExprAST *ParseIfExpr() {
  getNextToken();  // eat the if.

  // condition.
  ExprAST *Cond = ParseExpression();
  if (!Cond) return 0;

  if (CurTok != tok_then)
    return Error("expected then");
  getNextToken();  // eat the then

  ExprAST *Then = ParseExpression();
  if (Then == 0) return 0;

  if (CurTok != tok_else)
    return Error("expected else");

  getNextToken();

  ExprAST *Else = ParseExpression();
  if (!Else) return 0;

  return new IfExprAST(Cond, Then, Else);
}

/// forexpr ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression
```

```
static ExprAST *ParseForExpr() {
  getNextToken();  // eat the for.

  if (CurTok != tok_identifier)
    return Error("expected identifier after for");

  std::string IdName = IdentifierStr;
  getNextToken();  // eat identifier.

  if (CurTok != '=')
    return Error("expected '=' after for");
  getNextToken();  // eat '='.


  ExprAST *Start = ParseExpression();
  if (Start == 0) return 0;
  if (CurTok != ',')
    return Error("expected ',' after for start value");
  getNextToken();

  ExprAST *End = ParseExpression();
  if (End == 0) return 0;

  // The step value is optional.
  ExprAST *Step = 0;
  if (CurTok == ',') {
    getNextToken();
    Step = ParseExpression();
    if (Step == 0) return 0;
  }

  if (CurTok != tok_in)
    return Error("expected 'in' after for");
  getNextToken();  // eat 'in'.

  ExprAST *Body = ParseExpression();
  if (Body == 0) return 0;

  return new ForExprAST(IdName, Start, End, Step, Body);
}

/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
///   ::= ifexpr
///   ::= forexpr
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  case tok_if:         return ParseIfExpr();
  case tok_for:        return ParseForExpr();
  }
}
```

```
/// binoprhs
///   ::= ('+' primary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  // If this is a binop, find its precedence.
  while (1) {
    int TokPrec = GetTokPrecedence();

    // If this is a binop that binds at least as tightly as the current binop,
    // consume it, otherwise we are done.
    if (TokPrec < ExprPrec)
      return LHS;

    // Okay, we know this is a binop.
    int BinOp = CurTok;
    getNextToken();  // eat binop

    // Parse the primary expression after the binary operator.
    ExprAST *RHS = ParsePrimary();
    if (!RHS) return 0;

    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec+1, RHS);
      if (RHS == 0) return 0;
    }

    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }
}

/// expression
///   ::= primary binoprhs
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParsePrimary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
}

/// prototype
///   ::= id '(' id* ')'
static PrototypeAST *ParsePrototype() {
  if (CurTok != tok_identifier)
    return ErrorP("Expected function name in prototype");

  std::string FnName = IdentifierStr;
  getNextToken();

  if (CurTok != '(')
    return ErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
```

```
  if (CurTok != ')')
    return ErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  return new PrototypeAST(FnName, ArgNames);
}

/// definition ::= 'def' prototype expression
static FunctionAST *ParseDefinition() {
  getNextToken();  // eat def.
  PrototypeAST *Proto = ParsePrototype();
  if (Proto == 0) return 0;

  if (ExprAST *E = ParseExpression())
    return new FunctionAST(Proto, E);
  return 0;
}

/// toplevelexpr ::= expression
static FunctionAST *ParseTopLevelExpr() {
  if (ExprAST *E = ParseExpression()) {
    // Make an anonymous proto.
    PrototypeAST *Proto = new PrototypeAST("", std::vector<std::string>());
    return new FunctionAST(Proto, E);
  }
  return 0;
}

/// external ::= 'extern' prototype
static PrototypeAST *ParseExtern() {
  getNextToken();  // eat extern.
  return ParsePrototype();
}

//===----------------------------------------------------------------------===//
// Code Generation
//===----------------------------------------------------------------------===//

static Module *TheModule;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*> NamedValues;
static FunctionPassManager *TheFPM;

Value *ErrorV(const char *Str) { Error(Str); return 0; }

Value *NumberExprAST::Codegen() {
  return ConstantFP::get(getGlobalContext(), APFloat(Val));
}

Value *VariableExprAST::Codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  return V ? V : ErrorV("Unknown variable name");
}

Value *BinaryExprAST::Codegen() {
```

```
  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateFAdd(L, R, "addtmp");
  case '-': return Builder.CreateFSub(L, R, "subtmp");
  case '*': return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                "booltmp");
  default: return ErrorV("invalid binary operator");
  }
}

Value *CallExprAST::Codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (CalleeF == 0)
    return ErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return ErrorV("Incorrect # arguments passed");

  std::vector<Value*> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->Codegen());
    if (ArgsV.back() == 0) return 0;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}

Value *IfExprAST::Codegen() {
  Value *CondV = Cond->Codegen();
  if (CondV == 0) return 0;

  // Convert condition to a bool by comparing equal to 0.0.
  CondV = Builder.CreateFCmpONE(CondV,
                                ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                  "ifcond");

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Create blocks for the then and else cases.  Insert the 'then' block at the
  // end of the function.
  BasicBlock *ThenBB = BasicBlock::Create(getGlobalContext(), "then", TheFunction);
  BasicBlock *ElseBB = BasicBlock::Create(getGlobalContext(), "else");
  BasicBlock *MergeBB = BasicBlock::Create(getGlobalContext(), "ifcont");

  Builder.CreateCondBr(CondV, ThenBB, ElseBB);

  // Emit then value.
  Builder.SetInsertPoint(ThenBB);
```

```
  Value *ThenV = Then->Codegen();
  if (ThenV == 0) return 0;

  Builder.CreateBr(MergeBB);
  // Codegen of 'Then' can change the current block, update ThenBB for the PHI.
  ThenBB = Builder.GetInsertBlock();

  // Emit else block.
  TheFunction->getBasicBlockList().push_back(ElseBB);
  Builder.SetInsertPoint(ElseBB);

  Value *ElseV = Else->Codegen();
  if (ElseV == 0) return 0;

  Builder.CreateBr(MergeBB);
  // Codegen of 'Else' can change the current block, update ElseBB for the PHI.
  ElseBB = Builder.GetInsertBlock();

  // Emit merge block.
  TheFunction->getBasicBlockList().push_back(MergeBB);
  Builder.SetInsertPoint(MergeBB);
  PHINode *PN = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()), 2,
                                  "iftmp");

  PN->addIncoming(ThenV, ThenBB);
  PN->addIncoming(ElseV, ElseBB);
  return PN;
}

Value *ForExprAST::Codegen() {
  // Output this as:
  //   ...
  //   start = startexpr
  //   goto loop
  // loop:
  //   variable = phi [start, loopheader], [nextvariable, loopend]
  //   ...
  //   bodyexpr
  //   ...
  // loopend:
  //   step = stepexpr
  //   nextvariable = variable + step
  //   endcond = endexpr
  //   br endcond, loop, endloop
  // outloop:

  // Emit the start code first, without 'variable' in scope.
  Value *StartVal = Start->Codegen();
  if (StartVal == 0) return 0;

  // Make the new basic block for the loop header, inserting after current
  // block.
  Function *TheFunction = Builder.GetInsertBlock()->getParent();
  BasicBlock *PreheaderBB = Builder.GetInsertBlock();
  BasicBlock *LoopBB = BasicBlock::Create(getGlobalContext(), "loop", TheFunction);

  // Insert an explicit fall through from the current block to the LoopBB.
  Builder.CreateBr(LoopBB);
```

```
// Start insertion in LoopBB.
Builder.SetInsertPoint(LoopBB);

// Start the PHI node with an entry for Start.
PHINode *Variable = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()), 2, VarName.c_str());
Variable->addIncoming(StartVal, PreheaderBB);

// Within the loop, the variable is defined equal to the PHI node.  If it
// shadows an existing variable, we have to restore it, so save it now.
Value *OldVal = NamedValues[VarName];
NamedValues[VarName] = Variable;

// Emit the body of the loop.  This, like any other expr, can change the
// current BB.  Note that we ignore the value computed by the body, but don't
// allow an error.
if (Body->Codegen() == 0)
  return 0;

// Emit the step value.
Value *StepVal;
if (Step) {
  StepVal = Step->Codegen();
  if (StepVal == 0) return 0;
} else {
  // If not specified, use 1.0.
  StepVal = ConstantFP::get(getGlobalContext(), APFloat(1.0));
}

Value *NextVar = Builder.CreateFAdd(Variable, StepVal, "nextvar");

// Compute the end condition.
Value *EndCond = End->Codegen();
if (EndCond == 0) return EndCond;

// Convert condition to a bool by comparing equal to 0.0.
EndCond = Builder.CreateFCmpONE(EndCond,
                            ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                "loopcond");

// Create the "after loop" block and insert it.
BasicBlock *LoopEndBB = Builder.GetInsertBlock();
BasicBlock *AfterBB = BasicBlock::Create(getGlobalContext(), "afterloop", TheFunction);

// Insert the conditional branch into the end of LoopEndBB.
Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

// Any new code will be inserted in AfterBB.
Builder.SetInsertPoint(AfterBB);

// Add a new entry to the PHI node for the backedge.
Variable->addIncoming(NextVar, LoopEndBB);

// Restore the unshadowed variable.
if (OldVal)
  NamedValues[VarName] = OldVal;
else
  NamedValues.erase(VarName);
```

```
  // for expr always returns 0.0.
  return Constant::getNullValue(Type::getDoubleTy(getGlobalContext()));
}

Function *PrototypeAST::Codegen() {
  // Make the function type:  double(double,double) etc.
  std::vector<Type*> Doubles(Args.size(),
                             Type::getDoubleTy(getGlobalContext()));
  FunctionType *FT = FunctionType::get(Type::getDoubleTy(getGlobalContext()),
                                       Doubles, false);

  Function *F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule);

  // If F conflicted, there was already something named 'Name'.  If it has a
  // body, don't allow redefinition or reextern.
  if (F->getName() != Name) {
    // Delete the one we just made and get the existing one.
    F->eraseFromParent();
    F = TheModule->getFunction(Name);

    // If F already has a body, reject this.
    if (!F->empty()) {
      ErrorF("redefinition of function");
      return 0;
    }

    // If F took a different number of args, reject.
    if (F->arg_size() != Args.size()) {
      ErrorF("redefinition of function with different # args");
      return 0;
    }
  }

  // Set names for all arguments.
  unsigned Idx = 0;
  for (Function::arg_iterator AI = F->arg_begin(); Idx != Args.size();
       ++AI, ++Idx) {
    AI->setName(Args[Idx]);

    // Add arguments to variable symbol table.
    NamedValues[Args[Idx]] = AI;
  }

  return F;
}

Function *FunctionAST::Codegen() {
  NamedValues.clear();

  Function *TheFunction = Proto->Codegen();
  if (TheFunction == 0)
    return 0;

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);
  Builder.SetInsertPoint(BB);

  if (Value *RetVal = Body->Codegen()) {
```

```
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);

    // Optimize the function.
    TheFPM->run(*TheFunction);

    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();
  return 0;
}

//===----------------------------------------------------------------------===//
// Top-Level parsing and JIT Driver
//===----------------------------------------------------------------------===//

static ExecutionEngine *TheExecutionEngine;

static void HandleDefinition() {
  if (FunctionAST *F = ParseDefinition()) {
    if (Function *LF = F->Codegen()) {
      fprintf(stderr, "Read function definition:");
      LF->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleExtern() {
  if (PrototypeAST *P = ParseExtern()) {
    if (Function *F = P->Codegen()) {
      fprintf(stderr, "Read extern: ");
      F->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (FunctionAST *F = ParseTopLevelExpr()) {
    if (Function *LF = F->Codegen()) {
      // JIT the function, returning a function pointer.
      void *FPtr = TheExecutionEngine->getPointerToFunction(LF);

      // Cast it to the right type (takes no arguments, returns a double) so we
      // can call it as a native function.
      double (*FP)() = (double (*)())(intptr_t)FPtr;
      fprintf(stderr, "Evaluated to %f\n", FP());
```

```
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (1) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:    return;
    case ';':        getNextToken(); break;  // ignore top-level semicolons.
    case tok_def:    HandleDefinition(); break;
    case tok_extern: HandleExtern(); break;
    default:         HandleTopLevelExpression(); break;
    }
  }
}

//===----------------------------------------------------------------------===//
// "Library" functions that can be "extern'd" from user code.
//===----------------------------------------------------------------------===//

/// putchard - putchar that takes a double and returns 0.
extern "C"
double putchard(double X) {
  putchar((char)X);
  return 0;
}

//===----------------------------------------------------------------------===//
// Main driver code.
//===----------------------------------------------------------------------===//

int main() {
  InitializeNativeTarget();
  LLVMContext &Context = getGlobalContext();

  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.

  // Prime the first token.
  fprintf(stderr, "ready> ");
  getNextToken();

  // Make the module, which holds all the code.
  std::unique_ptr<Module> Owner = make_unique<Module>("my cool jit", Context);
  TheModule = Owner.get();

  // Create the JIT.  This takes ownership of the module.
  std::string ErrStr;
  TheExecutionEngine =
```

```
      EngineBuilder(std::move(Owner)).setErrorStr(&ErrStr).create();
  if (!TheExecutionEngine) {
    fprintf(stderr, "Could not create ExecutionEngine: %s\n", ErrStr.c_str());
    exit(1);
  }

  FunctionPassManager OurFPM(TheModule);

  // Set up the optimizer pipeline.  Start with registering info about how the
  // target lays out data structures.
  TheModule->setDataLayout(TheExecutionEngine->getDataLayout());
  OurFPM.add(new DataLayoutPass());
  // Provide basic AliasAnalysis support for GVN.
  OurFPM.add(createBasicAliasAnalysisPass());
  // Do simple "peephole" optimizations and bit-twiddling optzns.
  OurFPM.add(createInstructionCombiningPass());
  // Reassociate expressions.
  OurFPM.add(createReassociatePass());
  // Eliminate Common SubExpressions.
  OurFPM.add(createGVNPass());
  // Simplify the control flow graph (deleting unreachable blocks, etc).
  OurFPM.add(createCFGSimplificationPass());

  OurFPM.doInitialization();

  // Set the global so the code gen can use this.
  TheFPM = &OurFPM;

  // Run the main "interpreter loop" now.
  MainLoop();

  TheFPM = 0;

  // Print out all of the generated code.
  TheModule->dump();

  return 0;
}
```

Next: Extending the language: user-defined operators

Kaleidoscope: Extending the Language: User-defined Operators

Chapter 6 Introduction

Welcome to Chapter 6 of the "Implementing a language with LLVM" tutorial. At this point in our tutorial, we now have a fully functional language that is fairly minimal, but also useful. There is still one big problem with it, however. Our language doesn't have many useful operators (like division, logical negation, or even any comparisons besides less-than).

This chapter of the tutorial takes a wild digression into adding user-defined operators to the simple and beautiful Kaleidoscope language. This digression now gives us a simple and ugly language in some ways, but also a powerful one at the same time. One of the great things about creating your own language is that you get to decide what is good or bad. In this tutorial we'll assume that it is okay to use this as a way to show some interesting parsing techniques.

At the end of this tutorial, we'll run through an example Kaleidoscope application that renders the Mandelbrot set. This gives an example of what you can build with Kaleidoscope and its feature set.

User-defined Operators: the Idea

The "operator overloading" that we will add to Kaleidoscope is more general than languages like C++. In C++, you are only allowed to redefine existing operators: you can't programatically change the grammar, introduce new operators, change precedence levels, etc. In this chapter, we will add this capability to Kaleidoscope, which will let the user round out the set of operators that are supported.

The point of going into user-defined operators in a tutorial like this is to show the power and flexibility of using a hand-written parser. Thus far, the parser we have been implementing uses recursive descent for most parts of the grammar and operator precedence parsing for the expressions. See Chapter 2 for details. Without using operator precedence parsing, it would be very difficult to allow the programmer to introduce new operators into the grammar: the grammar is dynamically extensible as the JIT runs.

The two specific features we'll add are programmable unary operators (right now, Kaleidoscope has no unary operators at all) as well as binary operators. An example of this is:

```
# Logical unary not.
def unary!(v)
  if v then
    0
  else
    1;

# Define > with the same precedence as <.
def binary> 10 (LHS RHS)
  RHS < LHS;

# Binary "logical or", (note that it does not "short circuit")
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;

# Define = with slightly lower precedence than relationals.
def binary= 9 (LHS RHS)
  !(LHS < RHS | LHS > RHS);
```

Many languages aspire to being able to implement their standard runtime library in the language itself. In Kaleidoscope, we can implement significant parts of the language in the library!

We will break down implementation of these features into two parts: implementing support for user-defined binary operators and adding unary operators.

User-defined Binary Operators

Adding support for user-defined binary operators is pretty simple with our current framework. We'll first add support for the unary/binary keywords:

```
enum Token {
  ...
  // operators
  tok_binary = -11, tok_unary = -12
};
...
static int gettok() {
...
    if (IdentifierStr == "for") return tok_for;
    if (IdentifierStr == "in") return tok_in;
    if (IdentifierStr == "binary") return tok_binary;
    if (IdentifierStr == "unary") return tok_unary;
    return tok_identifier;
```

This just adds lexer support for the unary and binary keywords, like we did in previous chapters. One nice thing about our current AST, is that we represent binary operators with full generalisation by using their ASCII code as the opcode. For our extended operators, we'll use this same representation, so we don't need any new AST or parser support.

On the other hand, we have to be able to represent the definitions of these new operators, in the "def binary| 5" part of the function definition. In our grammar so far, the "name" for the function definition is parsed as the "prototype" production and into the `PrototypeAST` AST node. To represent our new user-defined operators as prototypes, we have to extend the `PrototypeAST` AST node like this:

```
/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its argument names as well as if it is an operator.
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
  bool isOperator;
  unsigned Precedence;  // Precedence if a binary op.
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args,
               bool isoperator = false, unsigned prec = 0)
  : Name(name), Args(args), isOperator(isoperator), Precedence(prec) {}

  bool isUnaryOp() const { return isOperator && Args.size() == 1; }
  bool isBinaryOp() const { return isOperator && Args.size() == 2; }

  char getOperatorName() const {
    assert(isUnaryOp() || isBinaryOp());
    return Name[Name.size()-1];
  }

  unsigned getBinaryPrecedence() const { return Precedence; }

  Function *Codegen();
};
```

Basically, in addition to knowing a name for the prototype, we now keep track of whether it was an operator, and if it was, what precedence level the operator is at. The precedence is only used for binary operators (as you'll see below, it

just doesn't apply for unary operators). Now that we have a way to represent the prototype for a user-defined operator, we need to parse it:

```
/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
static PrototypeAST *ParsePrototype() {
  std::string FnName;

  unsigned Kind = 0;  // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return ErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_binary:
    getNextToken();
    if (!isascii(CurTok))
      return ErrorP("Expected binary operator");
    FnName = "binary";
    FnName += (char)CurTok;
    Kind = 2;
    getNextToken();

    // Read the precedence if present.
    if (CurTok == tok_number) {
      if (NumVal < 1 || NumVal > 100)
        return ErrorP("Invalid precedecnce: must be 1..100");
      BinaryPrecedence = (unsigned)NumVal;
      getNextToken();
    }
    break;
  }

  if (CurTok != '(')
    return ErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return ErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  // Verify right number of names for operator.
  if (Kind && ArgNames.size() != Kind)
    return ErrorP("Invalid number of operands for operator");

  return new PrototypeAST(FnName, ArgNames, Kind != 0, BinaryPrecedence);
}
```

This is all fairly straightforward parsing code, and we have already seen a lot of similar code in the past. One inter-

esting part about the code above is the couple lines that set up `FnName` for binary operators. This builds names like "binary@" for a newly defined "@" operator. This then takes advantage of the fact that symbol names in the LLVM symbol table are allowed to have any character in them, including embedded nul characters.

The next interesting thing to add, is codegen support for these binary operators. Given our current structure, this is a simple addition of a default case for our existing binary operator node:

```
Value *BinaryExprAST::Codegen() {
  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateFAdd(L, R, "addtmp");
  case '-': return Builder.CreateFSub(L, R, "subtmp");
  case '*': return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                "booltmp");
  default: break;
  }

  // If it wasn't a builtin binary operator, it must be a user defined one. Emit
  // a call to it.
  Function *F = TheModule->getFunction(std::string("binary")+Op);
  assert(F && "binary operator not found!");

  Value *Ops[2] = { L, R };
  return Builder.CreateCall(F, Ops, "binop");
}
```

As you can see above, the new code is actually really simple. It just does a lookup for the appropriate operator in the symbol table and generates a function call to it. Since user-defined operators are just built as normal functions (because the "prototype" boils down to a function with the right name) everything falls into place.

The final piece of code we are missing, is a bit of top-level magic:

```
Function *FunctionAST::Codegen() {
  NamedValues.clear();

  Function *TheFunction = Proto->Codegen();
  if (TheFunction == 0)
    return 0;

  // If this is an operator, install it.
  if (Proto->isBinaryOp())
    BinopPrecedence[Proto->getOperatorName()] = Proto->getBinaryPrecedence();

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);
  Builder.SetInsertPoint(BB);

  if (Value *RetVal = Body->Codegen()) {
    ...
```

Basically, before codegening a function, if it is a user-defined operator, we register it in the precedence table. This allows the binary operator parsing logic we already have in place to handle it. Since we are working on a fully-general operator precedence parser, this is all we need to do to "extend the grammar".

Now we have useful user-defined binary operators. This builds a lot on the previous framework we built for other operators. Adding unary operators is a bit more challenging, because we don't have any framework for it yet - lets see what it takes.

User-defined Unary Operators

Since we don't currently support unary operators in the Kaleidoscope language, we'll need to add everything to support them. Above, we added simple support for the 'unary' keyword to the lexer. In addition to that, we need an AST node:

```
/// UnaryExprAST - Expression class for a unary operator.
class UnaryExprAST : public ExprAST {
  char Opcode;
  ExprAST *Operand;
public:
  UnaryExprAST(char opcode, ExprAST *operand)
    : Opcode(opcode), Operand(operand) {}
  virtual Value *Codegen();
};
```

This AST node is very simple and obvious by now. It directly mirrors the binary operator AST node, except that it only has one child. With this, we need to add the parsing logic. Parsing a unary operator is pretty simple: we'll add a new function to do it:

```
/// unary
///   ::= primary
///   ::= '!' unary
static ExprAST *ParseUnary() {
  // If the current token is not an operator, it must be a primary expr.
  if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
    return ParsePrimary();

  // If this is a unary operator, read it.
  int Opc = CurTok;
  getNextToken();
  if (ExprAST *Operand = ParseUnary())
    return new UnaryExprAST(Opc, Operand);
  return 0;
}
```

The grammar we add is pretty straightforward here. If we see a unary operator when parsing a primary operator, we eat the operator as a prefix and parse the remaining piece as another unary operator. This allows us to handle multiple unary operators (e.g. "!!x"). Note that unary operators can't have ambiguous parses like binary operators can, so there is no need for precedence information.

The problem with this function, is that we need to call ParseUnary from somewhere. To do this, we change previous callers of ParsePrimary to call ParseUnary instead:

```
/// binoprhs
///   ::= ('+' unary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  ...
    // Parse the unary expression after the binary operator.
    ExprAST *RHS = ParseUnary();
    if (!RHS) return 0;
  ...
}
/// expression
///   ::= unary binoprhs
```

```
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParseUnary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
}
```

With these two simple changes, we are now able to parse unary operators and build the AST for them. Next up, we need to add parser support for prototypes, to parse the unary operator prototype. We extend the binary operator code above with:

```
/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
///   ::= unary LETTER (id)
static PrototypeAST *ParsePrototype() {
  std::string FnName;

  unsigned Kind = 0;  // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return ErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_unary:
    getNextToken();
    if (!isascii(CurTok))
      return ErrorP("Expected unary operator");
    FnName = "unary";
    FnName += (char)CurTok;
    Kind = 1;
    getNextToken();
    break;
  case tok_binary:
    ...
```

As with binary operators, we name unary operators with a name that includes the operator character. This assists us at code generation time. Speaking of, the final piece we need to add is codegen support for unary operators. It looks like this:

```
Value *UnaryExprAST::Codegen() {
  Value *OperandV = Operand->Codegen();
  if (OperandV == 0) return 0;

  Function *F = TheModule->getFunction(std::string("unary")+Opcode);
  if (F == 0)
    return ErrorV("Unknown unary operator");

  return Builder.CreateCall(F, OperandV, "unop");
}
```

This code is similar to, but simpler than, the code for binary operators. It is simpler primarily because it doesn't need to handle any predefined operators.

Kicking the Tires

It is somewhat hard to believe, but with a few simple extensions we've covered in the last chapters, we have grown a real-ish language. With this, we can do a lot of interesting things, including I/O, math, and a bunch of other things. For example, we can now add a nice sequencing operator (printd is defined to print out the specified value and a newline):

```
ready> extern printd(x);
Read extern:
declare double @printd(double)

ready> def binary : 1 (x y) 0;  # Low-precedence operator that ignores operands.
..
ready> printd(123) : printd(456) : printd(789);
123.000000
456.000000
789.000000
Evaluated to 0.000000
```

We can also define a bunch of other "primitive" operations, such as:

```
# Logical unary not.
def unary!(v)
  if v then
    0
  else
    1;

# Unary negate.
def unary-(v)
  0-v;

# Define > with the same precedence as <.
def binary> 10 (LHS RHS)
  RHS < LHS;

# Binary logical or, which does not short circuit.
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;

# Binary logical and, which does not short circuit.
def binary& 6 (LHS RHS)
  if !LHS then
    0
  else
    !!RHS;

# Define = with slightly lower precedence than relationals.
def binary = 9 (LHS RHS)
  !(LHS < RHS | LHS > RHS);

# Define ':' for sequencing: as a low-precedence operator that ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;
```

Given the previous if/then/else support, we can also define interesting functions for I/O. For example, the following prints out a character whose "density" reflects the value passed in: the lower the value, the denser the character:

```
ready>

extern putchard(char)
def printdensity(d)
  if d > 8 then
    putchard(32)  # ' '
  else if d > 4 then
    putchard(46)  # '.'
  else if d > 2 then
    putchard(43)  # '+'
  else
    putchard(42); # '*'
...
ready> printdensity(1): printdensity(2): printdensity(3):
       printdensity(4): printdensity(5): printdensity(9):
       putchard(10);
***+.
Evaluated to 0.000000
```

Based on these simple primitive operations, we can start to define more interesting things. For example, here's a little function that solves for the number of iterations it takes a function in the complex plane to converge:

```
# Determine whether the specific location diverges.
# Solve for z = z^2 + c in the complex plane.
def mandleconverger(real imag iters creal cimag)
  if iters > 255 | (real*real + imag*imag > 4) then
    iters
  else
    mandleconverger(real*real - imag*imag + creal,
                    2*real*imag + cimag,
                    iters+1, creal, cimag);

# Return the number of iterations required for the iteration to escape
def mandleconverge(real imag)
  mandleconverger(real, imag, 0, real, imag);
```

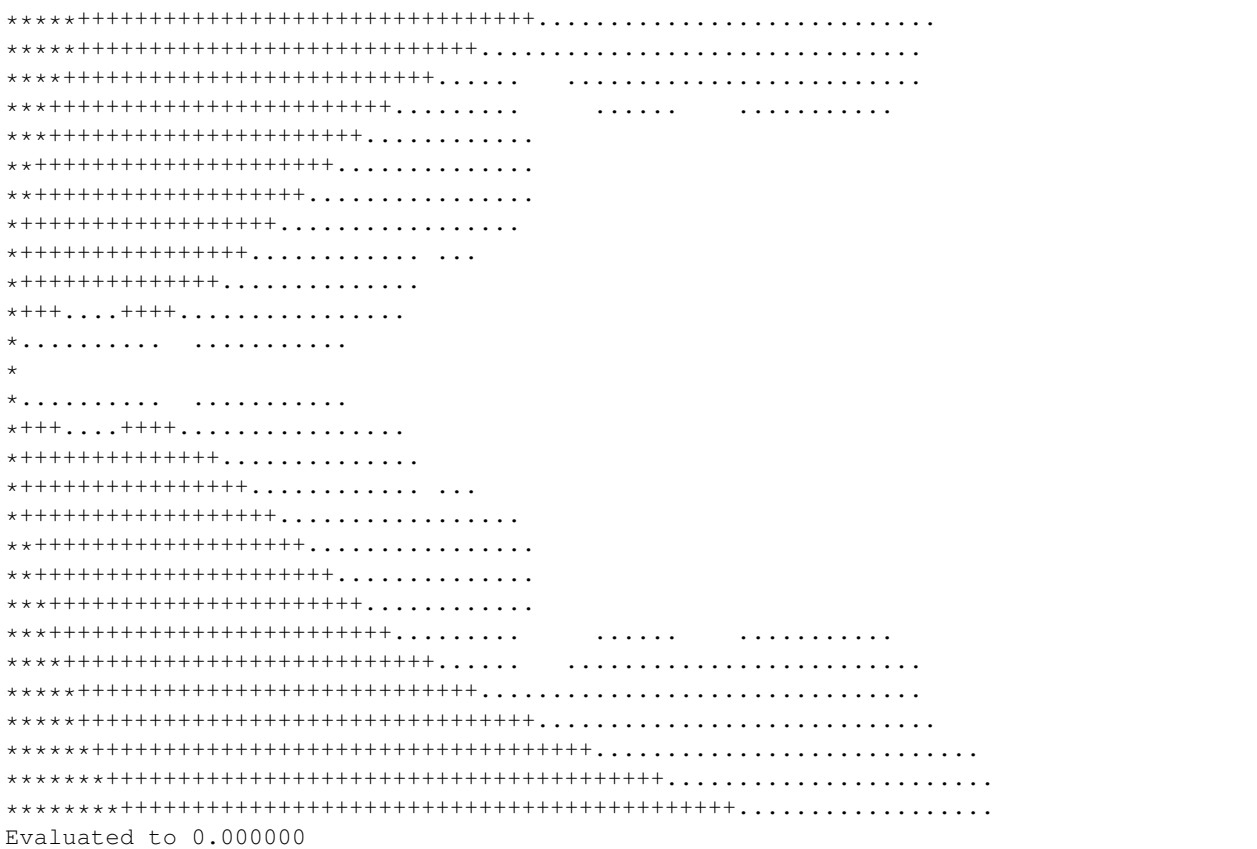
This "`z = z2 + c`" function is a beautiful little creature that is the basis for computation of the Mandelbrot Set. Our `mandelconverge` function returns the number of iterations that it takes for a complex orbit to escape, saturating to 255. This is not a very useful function by itself, but if you plot its value over a two-dimensional plane, you can see the Mandelbrot set. Given that we are limited to using putchard here, our amazing graphical output is limited, but we can whip together something using the density plotter above:

```
# Compute and plot the mandlebrot set with the specified 2 dimensional range
# info.
def mandelhelp(xmin xmax xstep   ymin ymax ystep)
  for y = ymin, y < ymax, ystep in (
    (for x = xmin, x < xmax, xstep in
       printdensity(mandleconverge(x,y)))
    : putchard(10)
  )

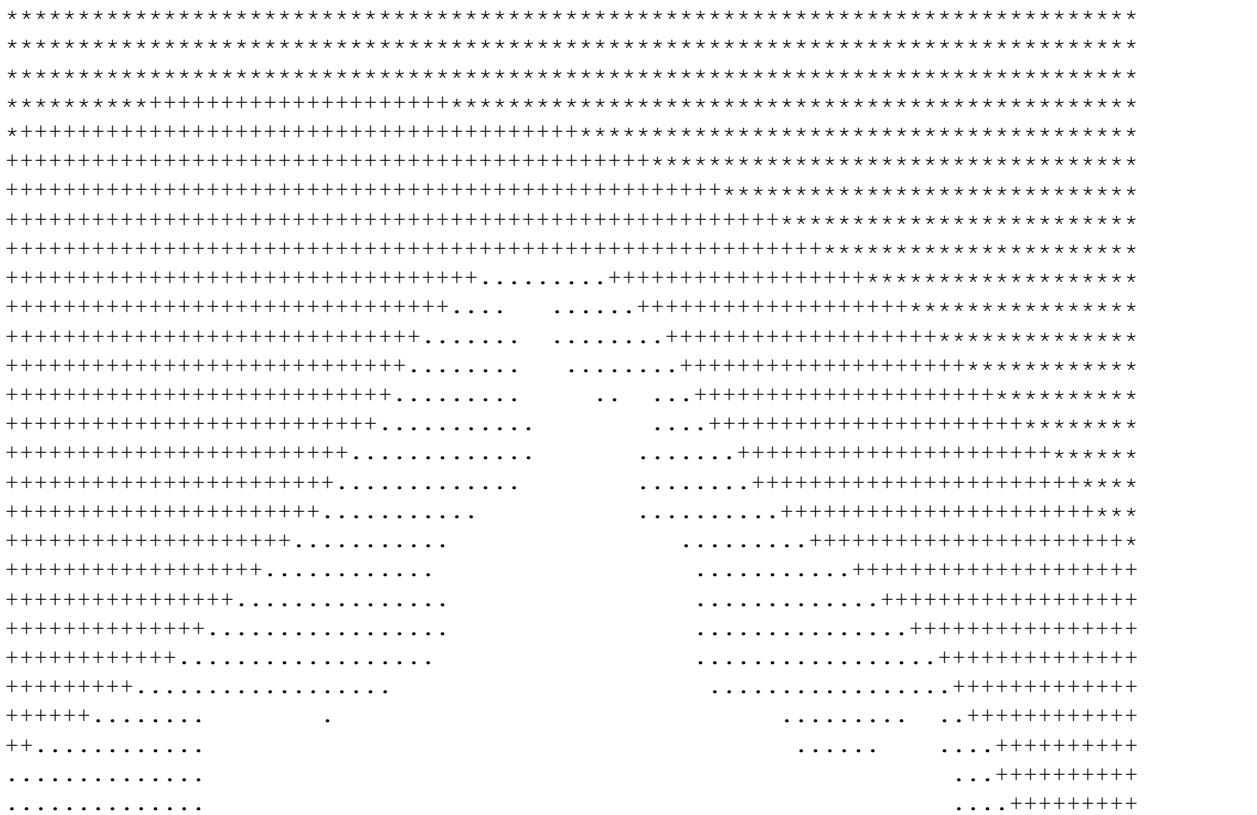
# mandel - This is a convenient helper function for plotting the mandelbrot set
# from the specified position with the specified Magnification.
def mandel(realstart imagstart realmag imagmag)
  mandelhelp(realstart, realstart+realmag*78, realmag,
             imagstart, imagstart+imagmag*40, imagmag);
```

Given this, we can try plotting out the mandlebrot set! Lets try it out:

```
ready> mandel(-2.3, -1.3, 0.05, 0.07);
**************************************************************************************
**************************************************************************************
**************************************************************************************
****************************************.. ...+++++++++++++++++++++++++++++++++++++++
****************************************.... ...+++++++++++++++++++++++++++++++++++++
***************************************.....   ...+++++++++++++++++++++++++++++++++++
*************************************+++++....     ....+++++++++++++++++++++++++++++++
************************************+++++......      .....+++++++++++++++++++++++++++++
***********************************+++++.......       .......+++++++++++++++++++++++++
*********************************+++++....                ... .+++++++++++++++++++++++
********************************+++++.......                    .+++++++++++++++++++++++
******************************+++++...........                   ...+++++++++++++++++++++
*****************************+++++...........                   ...+++++++++++++++++++++
*****************************+++... ..........                     .+++++++++++++++++++++
****************************+++++.....                            .+++++++++++++++++++++
***************************+++++......                           ..+++++++++++++++++++++
*************************+++++.......                           ..+++++++++++++++++++++
************************+++++......                            ..+++++++++++++++++++++
*******+++++.... ....                                          ...+++++++++++++++++++++
*******+++++.... .                                             ...+++++++++++++++++++++
********+++++......                                            ...+++++++++++++++++++++
*************+++++.......                                      ..+++++++++++++++++++++
****************+++++......                                     .+++++++++++++++++++++
*****************+++++.....                                    ..+++++++++++++++++++++
******************+++++... ..........                          .+++++++++++++++++++++
*********************+++++...........                         ...+++++++++++++++++++++
**********************+++++..........                         ...+++++++++++++++++++++
*************************+++++........                          .+++++++++++++++++++++
*******************************+++++....                     ... ..+++++++++++++++++++++
**********************************+++++.......       .......+++++++++++++++++++++++++
***********************************+++++......      .......+++++++++++++++++++++++++++
*************************************+++++....      ....+++++++++++++++++++++++++++++
***************************************+++++.....   ...+++++++++++++++++++++++++++++++
*****************************************+++++....  ...+++++++++++++++++++++++++++++++
*******************************************+++++......+++++++++++++++++++++++++++++++
********************************************+++++.+++++++++++++++++++++++++++++++++
**************************************************************************************
**************************************************************************************
**************************************************************************************
**************************************************************************************
**************************************************************************************
Evaluated to 0.000000
ready> mandel(-2, -1, 0.02, 0.04);
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++...
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++.....
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++........
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++...........
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++.............
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++.        .
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++............
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++...................
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++.....................
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++.........................
```



```
Evaluated to 0.000000
ready> mandel(-0.9, -1.4, 0.02, 0.03);
```



```
.............                                                  .....++++++++
............                                                  ......++++++++
...........                                                  .......++++++++
.........                                                    ........+++++++
.........                                                    ........+++++++
.........                                                    ........+++++++
........                                                        ....+++++++
.......                                                          ...+++++++
                                                                 ...+++++++
                                                                ....+++++++
                                                               .....+++++++
                                                                ....+++++++
                                                                ....+++++++
                                                                ....+++++++
Evaluated to 0.000000
ready> ^D
```

At this point, you may be starting to realize that Kaleidoscope is a real and powerful language. It may not be self-similar :), but it can be used to plot things that are!

With this, we conclude the "adding user-defined operators" chapter of the tutorial. We have successfully augmented our language, adding the ability to extend the language in the library, and we have shown how this can be used to build a simple but interesting end-user application in Kaleidoscope. At this point, Kaleidoscope can build a variety of applications that are functional and can call functions with side-effects, but it can't actually define and mutate a variable itself.

Strikingly, variable mutation is an important feature of some languages, and it is not at all obvious how to add support for mutable variables without having to add an "SSA construction" phase to your front-end. In the next chapter, we will describe how you can add variable mutation without building SSA in your front-end.

Full Code Listing

Here is the complete code listing for our running example, enhanced with the if/then/else and for expressions.. To build this example, use:

```
# Compile
clang++ -g toy.cpp `llvm-config --cppflags --ldflags --libs core jit native` -O3 -o toy
# Run
./toy
```

On some platforms, you will need to specify -rdynamic or -Wl,–export-dynamic when linking. This ensures that symbols defined in the main executable are exported to the dynamic linker and so are available for symbol resolution at run time. This is not needed if you compile your support code into a shared library, although doing that will cause problems on Windows.

Here is the code:

```
#include "llvm/Analysis/Passes.h"
#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include "llvm/PassManager.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Transforms/Scalar.h"
#include <cctype>
```

```
#include <cstdio>
#include <map>
#include <string>
#include <vector>
using namespace llvm;

//===----------------------------------------------------------------------===//
// Lexer
//===----------------------------------------------------------------------===//

// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2, tok_extern = -3,

  // primary
  tok_identifier = -4, tok_number = -5,

  // control
  tok_if = -6, tok_then = -7, tok_else = -8,
  tok_for = -9, tok_in = -10,

  // operators
  tok_binary = -11, tok_unary = -12
};

static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number

/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();

  if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getchar())))
      IdentifierStr += LastChar;

    if (IdentifierStr == "def") return tok_def;
    if (IdentifierStr == "extern") return tok_extern;
    if (IdentifierStr == "if") return tok_if;
    if (IdentifierStr == "then") return tok_then;
    if (IdentifierStr == "else") return tok_else;
    if (IdentifierStr == "for") return tok_for;
    if (IdentifierStr == "in") return tok_in;
    if (IdentifierStr == "binary") return tok_binary;
    if (IdentifierStr == "unary") return tok_unary;
    return tok_identifier;
  }

  if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
```

```
    std::string NumStr;
    do {
      NumStr += LastChar;
      LastChar = getchar();
    } while (isdigit(LastChar) || LastChar == '.');

    NumVal = strtod(NumStr.c_str(), 0);
    return tok_number;
  }

  if (LastChar == '#') {
    // Comment until end of line.
    do LastChar = getchar();
    while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

    if (LastChar != EOF)
      return gettok();
  }

  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}

//===----------------------------------------------------------------------===//
// Abstract Syntax Tree (aka Parse Tree)
//===----------------------------------------------------------------------===//
namespace {
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
  virtual Value *Codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val) : Val(val) {}
  virtual Value *Codegen();
};

/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;
public:
  VariableExprAST(const std::string &name) : Name(name) {}
  virtual Value *Codegen();
};

/// UnaryExprAST - Expression class for a unary operator.
class UnaryExprAST : public ExprAST {
```

```
  char Opcode;
  ExprAST *Operand;
public:
  UnaryExprAST(char opcode, ExprAST *operand)
    : Opcode(opcode), Operand(operand) {}
  virtual Value *Codegen();
};

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  ExprAST *LHS, *RHS;
public:
  BinaryExprAST(char op, ExprAST *lhs, ExprAST *rhs)
    : Op(op), LHS(lhs), RHS(rhs) {}
  virtual Value *Codegen();
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<ExprAST*> Args;
public:
  CallExprAST(const std::string &callee, std::vector<ExprAST*> &args)
    : Callee(callee), Args(args) {}
  virtual Value *Codegen();
};

/// IfExprAST - Expression class for if/then/else.
class IfExprAST : public ExprAST {
  ExprAST *Cond, *Then, *Else;
public:
  IfExprAST(ExprAST *cond, ExprAST *then, ExprAST *_else)
  : Cond(cond), Then(then), Else(_else) {}
  virtual Value *Codegen();
};

/// ForExprAST - Expression class for for/in.
class ForExprAST : public ExprAST {
  std::string VarName;
  ExprAST *Start, *End, *Step, *Body;
public:
  ForExprAST(const std::string &varname, ExprAST *start, ExprAST *end,
             ExprAST *step, ExprAST *body)
    : VarName(varname), Start(start), End(end), Step(step), Body(body) {}
  virtual Value *Codegen();
};

/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its name, and its argument names (thus implicitly the number
/// of arguments the function takes), as well as if it is an operator.
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
  bool isOperator;
  unsigned Precedence;  // Precedence if a binary op.
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args,
```

```
              bool isoperator = false, unsigned prec = 0)
  : Name(name), Args(args), isOperator(isoperator), Precedence(prec) {}

  bool isUnaryOp() const { return isOperator && Args.size() == 1; }
  bool isBinaryOp() const { return isOperator && Args.size() == 2; }

  char getOperatorName() const {
    assert(isUnaryOp() || isBinaryOp());
    return Name[Name.size()-1];
  }

  unsigned getBinaryPrecedence() const { return Precedence; }

  Function *Codegen();
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  PrototypeAST *Proto;
  ExprAST *Body;
public:
  FunctionAST(PrototypeAST *proto, ExprAST *body)
    : Proto(proto), Body(body) {}

  Function *Codegen();
};
} // end anonymous namespace

//===----------------------------------------------------------------------===//
// Parser
//===----------------------------------------------------------------------===//

/// CurTok/getNextToken - Provide a simple token buffer.  CurTok is the current
/// token the parser is looking at.  getNextToken reads another token from the
/// lexer and updates CurTok with its results.
static int CurTok;
static int getNextToken() {
  return CurTok = gettok();
}

/// BinopPrecedence - This holds the precedence for each binary operator that is
/// defined.
static std::map<char, int> BinopPrecedence;

/// GetTokPrecedence - Get the precedence of the pending binary operator token.
static int GetTokPrecedence() {
  if (!isascii(CurTok))
    return -1;

  // Make sure it's a declared binop.
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0) return -1;
  return TokPrec;
}

/// Error* - These are little helper functions for error handling.
ExprAST *Error(const char *Str) { fprintf(stderr, "Error: %s\n", Str);return 0;}
PrototypeAST *ErrorP(const char *Str) { Error(Str); return 0; }
```

```
FunctionAST *ErrorF(const char *Str) { Error(Str); return 0; }

static ExprAST *ParseExpression();

/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static ExprAST *ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return new VariableExprAST(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<ExprAST*> Args;
  if (CurTok != ')') {
    while (1) {
      ExprAST *Arg = ParseExpression();
      if (!Arg) return 0;
      Args.push_back(Arg);

      if (CurTok == ')') break;

      if (CurTok != ',')
        return Error("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return new CallExprAST(IdName, Args);
}

/// numberexpr ::= number
static ExprAST *ParseNumberExpr() {
  ExprAST *Result = new NumberExprAST(NumVal);
  getNextToken(); // consume the number
  return Result;
}

/// parenexpr ::= '(' expression ')'
static ExprAST *ParseParenExpr() {
  getNextToken();  // eat (.
  ExprAST *V = ParseExpression();
  if (!V) return 0;

  if (CurTok != ')')
    return Error("expected ')'");
  getNextToken();  // eat ).
  return V;
}

/// ifexpr ::= 'if' expression 'then' expression 'else' expression
```

```
static ExprAST *ParseIfExpr() {
  getNextToken();  // eat the if.

  // condition.
  ExprAST *Cond = ParseExpression();
  if (!Cond) return 0;

  if (CurTok != tok_then)
    return Error("expected then");
  getNextToken();  // eat the then

  ExprAST *Then = ParseExpression();
  if (Then == 0) return 0;

  if (CurTok != tok_else)
    return Error("expected else");

  getNextToken();

  ExprAST *Else = ParseExpression();
  if (!Else) return 0;

  return new IfExprAST(Cond, Then, Else);
}

/// forexpr ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression
static ExprAST *ParseForExpr() {
  getNextToken();  // eat the for.

  if (CurTok != tok_identifier)
    return Error("expected identifier after for");

  std::string IdName = IdentifierStr;
  getNextToken();  // eat identifier.

  if (CurTok != '=')
    return Error("expected '=' after for");
  getNextToken();  // eat '='.


  ExprAST *Start = ParseExpression();
  if (Start == 0) return 0;
  if (CurTok != ',')
    return Error("expected ',' after for start value");
  getNextToken();

  ExprAST *End = ParseExpression();
  if (End == 0) return 0;

  // The step value is optional.
  ExprAST *Step = 0;
  if (CurTok == ',') {
    getNextToken();
    Step = ParseExpression();
    if (Step == 0) return 0;
  }

  if (CurTok != tok_in)
```

```
    return Error("expected 'in' after for");
  getNextToken();  // eat 'in'.

  ExprAST *Body = ParseExpression();
  if (Body == 0) return 0;

  return new ForExprAST(IdName, Start, End, Step, Body);
}

/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
///   ::= ifexpr
///   ::= forexpr
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  case tok_if:         return ParseIfExpr();
  case tok_for:        return ParseForExpr();
  }
}

/// unary
///   ::= primary
///   ::= '!' unary
static ExprAST *ParseUnary() {
  // If the current token is not an operator, it must be a primary expr.
  if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
    return ParsePrimary();

  // If this is a unary operator, read it.
  int Opc = CurTok;
  getNextToken();
  if (ExprAST *Operand = ParseUnary())
    return new UnaryExprAST(Opc, Operand);
  return 0;
}

/// binoprhs
///   ::= ('+' unary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  // If this is a binop, find its precedence.
  while (1) {
    int TokPrec = GetTokPrecedence();

    // If this is a binop that binds at least as tightly as the current binop,
    // consume it, otherwise we are done.
    if (TokPrec < ExprPrec)
      return LHS;

    // Okay, we know this is a binop.
    int BinOp = CurTok;
    getNextToken();  // eat binop
```

```
    // Parse the unary expression after the binary operator.
    ExprAST *RHS = ParseUnary();
    if (!RHS) return 0;

    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec+1, RHS);
      if (RHS == 0) return 0;
    }

    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }
}

/// expression
///   ::= unary binoprhs
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParseUnary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
}

/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
///   ::= unary LETTER (id)
static PrototypeAST *ParsePrototype() {
  std::string FnName;

  unsigned Kind = 0; // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return ErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_unary:
    getNextToken();
    if (!isascii(CurTok))
      return ErrorP("Expected unary operator");
    FnName = "unary";
    FnName += (char)CurTok;
    Kind = 1;
    getNextToken();
    break;
  case tok_binary:
    getNextToken();
    if (!isascii(CurTok))
      return ErrorP("Expected binary operator");
```

```
    FnName = "binary";
    FnName += (char)CurTok;
    Kind = 2;
    getNextToken();

    // Read the precedence if present.
    if (CurTok == tok_number) {
      if (NumVal < 1 || NumVal > 100)
        return ErrorP("Invalid precedecnce: must be 1..100");
      BinaryPrecedence = (unsigned)NumVal;
      getNextToken();
    }
    break;
  }

  if (CurTok != '(')
    return ErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return ErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  // Verify right number of names for operator.
  if (Kind && ArgNames.size() != Kind)
    return ErrorP("Invalid number of operands for operator");

  return new PrototypeAST(FnName, ArgNames, Kind != 0, BinaryPrecedence);
}

/// definition ::= 'def' prototype expression
static FunctionAST *ParseDefinition() {
  getNextToken();  // eat def.
  PrototypeAST *Proto = ParsePrototype();
  if (Proto == 0) return 0;

  if (ExprAST *E = ParseExpression())
    return new FunctionAST(Proto, E);
  return 0;
}

/// toplevelexpr ::= expression
static FunctionAST *ParseTopLevelExpr() {
  if (ExprAST *E = ParseExpression()) {
    // Make an anonymous proto.
    PrototypeAST *Proto = new PrototypeAST("", std::vector<std::string>());
    return new FunctionAST(Proto, E);
  }
  return 0;
}

/// external ::= 'extern' prototype
static PrototypeAST *ParseExtern() {
  getNextToken();  // eat extern.
```

```
  return ParsePrototype();
}

//===----------------------------------------------------------------------===//
// Code Generation
//===----------------------------------------------------------------------===//

static Module *TheModule;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*> NamedValues;
static FunctionPassManager *TheFPM;

Value *ErrorV(const char *Str) { Error(Str); return 0; }

Value *NumberExprAST::Codegen() {
  return ConstantFP::get(getGlobalContext(), APFloat(Val));
}

Value *VariableExprAST::Codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  return V ? V : ErrorV("Unknown variable name");
}

Value *UnaryExprAST::Codegen() {
  Value *OperandV = Operand->Codegen();
  if (OperandV == 0) return 0;

  Function *F = TheModule->getFunction(std::string("unary")+Opcode);
  if (F == 0)
    return ErrorV("Unknown unary operator");

  return Builder.CreateCall(F, OperandV, "unop");
}

Value *BinaryExprAST::Codegen() {
  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateFAdd(L, R, "addtmp");
  case '-': return Builder.CreateFSub(L, R, "subtmp");
  case '*': return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                "booltmp");
  default: break;
  }

  // If it wasn't a builtin binary operator, it must be a user defined one. Emit
  // a call to it.
  Function *F = TheModule->getFunction(std::string("binary")+Op);
  assert(F && "binary operator not found!");

  Value *Ops[] = { L, R };
```

```
  return Builder.CreateCall(F, Ops, "binop");
}

Value *CallExprAST::Codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (CalleeF == 0)
    return ErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return ErrorV("Incorrect # arguments passed");

  std::vector<Value*> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->Codegen());
    if (ArgsV.back() == 0) return 0;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}

Value *IfExprAST::Codegen() {
  Value *CondV = Cond->Codegen();
  if (CondV == 0) return 0;

  // Convert condition to a bool by comparing equal to 0.0.
  CondV = Builder.CreateFCmpONE(CondV,
                                ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                  "ifcond");

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Create blocks for the then and else cases.  Insert the 'then' block at the
  // end of the function.
  BasicBlock *ThenBB = BasicBlock::Create(getGlobalContext(), "then", TheFunction);
  BasicBlock *ElseBB = BasicBlock::Create(getGlobalContext(), "else");
  BasicBlock *MergeBB = BasicBlock::Create(getGlobalContext(), "ifcont");

  Builder.CreateCondBr(CondV, ThenBB, ElseBB);

  // Emit then value.
  Builder.SetInsertPoint(ThenBB);

  Value *ThenV = Then->Codegen();
  if (ThenV == 0) return 0;

  Builder.CreateBr(MergeBB);
  // Codegen of 'Then' can change the current block, update ThenBB for the PHI.
  ThenBB = Builder.GetInsertBlock();

  // Emit else block.
  TheFunction->getBasicBlockList().push_back(ElseBB);
  Builder.SetInsertPoint(ElseBB);

  Value *ElseV = Else->Codegen();
  if (ElseV == 0) return 0;
```

```
  Builder.CreateBr(MergeBB);
  // Codegen of 'Else' can change the current block, update ElseBB for the PHI.
  ElseBB = Builder.GetInsertBlock();

  // Emit merge block.
  TheFunction->getBasicBlockList().push_back(MergeBB);
  Builder.SetInsertPoint(MergeBB);
  PHINode *PN = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()), 2,
                                  "iftmp");

  PN->addIncoming(ThenV, ThenBB);
  PN->addIncoming(ElseV, ElseBB);
  return PN;
}

Value *ForExprAST::Codegen() {
  // Output this as:
  //   ...
  //   start = startexpr
  //   goto loop
  // loop:
  //   variable = phi [start, loopheader], [nextvariable, loopend]
  //   ...
  //   bodyexpr
  //   ...
  // loopend:
  //   step = stepexpr
  //   nextvariable = variable + step
  //   endcond = endexpr
  //   br endcond, loop, endloop
  // outloop:

  // Emit the start code first, without 'variable' in scope.
  Value *StartVal = Start->Codegen();
  if (StartVal == 0) return 0;

  // Make the new basic block for the loop header, inserting after current
  // block.
  Function *TheFunction = Builder.GetInsertBlock()->getParent();
  BasicBlock *PreheaderBB = Builder.GetInsertBlock();
  BasicBlock *LoopBB = BasicBlock::Create(getGlobalContext(), "loop", TheFunction);

  // Insert an explicit fall through from the current block to the LoopBB.
  Builder.CreateBr(LoopBB);

  // Start insertion in LoopBB.
  Builder.SetInsertPoint(LoopBB);

  // Start the PHI node with an entry for Start.
  PHINode *Variable = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()), 2, VarName.c_str());
  Variable->addIncoming(StartVal, PreheaderBB);

  // Within the loop, the variable is defined equal to the PHI node.  If it
  // shadows an existing variable, we have to restore it, so save it now.
  Value *OldVal = NamedValues[VarName];
  NamedValues[VarName] = Variable;

  // Emit the body of the loop.  This, like any other expr, can change the
```

```
  // current BB.  Note that we ignore the value computed by the body, but don't
  // allow an error.
  if (Body->Codegen() == 0)
    return 0;

  // Emit the step value.
  Value *StepVal;
  if (Step) {
    StepVal = Step->Codegen();
    if (StepVal == 0) return 0;
  } else {
    // If not specified, use 1.0.
    StepVal = ConstantFP::get(getGlobalContext(), APFloat(1.0));
  }

  Value *NextVar = Builder.CreateFAdd(Variable, StepVal, "nextvar");

  // Compute the end condition.
  Value *EndCond = End->Codegen();
  if (EndCond == 0) return EndCond;

  // Convert condition to a bool by comparing equal to 0.0.
  EndCond = Builder.CreateFCmpONE(EndCond,
                              ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                  "loopcond");

  // Create the "after loop" block and insert it.
  BasicBlock *LoopEndBB = Builder.GetInsertBlock();
  BasicBlock *AfterBB = BasicBlock::Create(getGlobalContext(), "afterloop", TheFunction);

  // Insert the conditional branch into the end of LoopEndBB.
  Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

  // Any new code will be inserted in AfterBB.
  Builder.SetInsertPoint(AfterBB);

  // Add a new entry to the PHI node for the backedge.
  Variable->addIncoming(NextVar, LoopEndBB);

  // Restore the unshadowed variable.
  if (OldVal)
    NamedValues[VarName] = OldVal;
  else
    NamedValues.erase(VarName);


  // for expr always returns 0.0.
  return Constant::getNullValue(Type::getDoubleTy(getGlobalContext()));
}

Function *PrototypeAST::Codegen() {
  // Make the function type:  double(double,double) etc.
  std::vector<Type*> Doubles(Args.size(),
                             Type::getDoubleTy(getGlobalContext()));
  FunctionType *FT = FunctionType::get(Type::getDoubleTy(getGlobalContext()),
                                       Doubles, false);

  Function *F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule);
```

```
  // If F conflicted, there was already something named 'Name'.  If it has a
  // body, don't allow redefinition or reextern.
  if (F->getName() != Name) {
    // Delete the one we just made and get the existing one.
    F->eraseFromParent();
    F = TheModule->getFunction(Name);

    // If F already has a body, reject this.
    if (!F->empty()) {
      ErrorF("redefinition of function");
      return 0;
    }

    // If F took a different number of args, reject.
    if (F->arg_size() != Args.size()) {
      ErrorF("redefinition of function with different # args");
      return 0;
    }
  }

  // Set names for all arguments.
  unsigned Idx = 0;
  for (Function::arg_iterator AI = F->arg_begin(); Idx != Args.size();
       ++AI, ++Idx) {
    AI->setName(Args[Idx]);

    // Add arguments to variable symbol table.
    NamedValues[Args[Idx]] = AI;
  }

  return F;
}

Function *FunctionAST::Codegen() {
  NamedValues.clear();

  Function *TheFunction = Proto->Codegen();
  if (TheFunction == 0)
    return 0;

  // If this is an operator, install it.
  if (Proto->isBinaryOp())
    BinopPrecedence[Proto->getOperatorName()] = Proto->getBinaryPrecedence();

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);
  Builder.SetInsertPoint(BB);

  if (Value *RetVal = Body->Codegen()) {
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);

    // Optimize the function.
    TheFPM->run(*TheFunction);
```

```
    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();

  if (Proto->isBinaryOp())
    BinopPrecedence.erase(Proto->getOperatorName());
  return 0;
}

//===----------------------------------------------------------------------===//
// Top-Level parsing and JIT Driver
//===----------------------------------------------------------------------===//

static ExecutionEngine *TheExecutionEngine;

static void HandleDefinition() {
  if (FunctionAST *F = ParseDefinition()) {
    if (Function *LF = F->Codegen()) {
      fprintf(stderr, "Read function definition:");
      LF->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleExtern() {
  if (PrototypeAST *P = ParseExtern()) {
    if (Function *F = P->Codegen()) {
      fprintf(stderr, "Read extern: ");
      F->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (FunctionAST *F = ParseTopLevelExpr()) {
    if (Function *LF = F->Codegen()) {
      // JIT the function, returning a function pointer.
      void *FPtr = TheExecutionEngine->getPointerToFunction(LF);

      // Cast it to the right type (takes no arguments, returns a double) so we
      // can call it as a native function.
      double (*FP)() = (double (*)())(intptr_t)FPtr;
      fprintf(stderr, "Evaluated to %f\n", FP());
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}
```

```
/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (1) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:    return;
    case ';':        getNextToken(); break;  // ignore top-level semicolons.
    case tok_def:    HandleDefinition(); break;
    case tok_extern: HandleExtern(); break;
    default:         HandleTopLevelExpression(); break;
    }
  }
}

//===----------------------------------------------------------------------===//
// "Library" functions that can be "extern'd" from user code.
//===----------------------------------------------------------------------===//

/// putchard - putchar that takes a double and returns 0.
extern "C"
double putchard(double X) {
  putchar((char)X);
  return 0;
}

/// printd - printf that takes a double prints it as "%f\n", returning 0.
extern "C"
double printd(double X) {
  printf("%f\n", X);
  return 0;
}

//===----------------------------------------------------------------------===//
// Main driver code.
//===----------------------------------------------------------------------===//

int main() {
  InitializeNativeTarget();
  LLVMContext &Context = getGlobalContext();

  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.

  // Prime the first token.
  fprintf(stderr, "ready> ");
  getNextToken();

  // Make the module, which holds all the code.
  std::unique_ptr<Module> Owner = make_unique<Module>("my cool jit", Context);
  TheModule = Owner.get();

  // Create the JIT.  This takes ownership of the module.
  std::string ErrStr;
  TheExecutionEngine =
```

```
      EngineBuilder(std::move(Owner)).setErrorStr(&ErrStr).create();
  if (!TheExecutionEngine) {
    fprintf(stderr, "Could not create ExecutionEngine: %s\n", ErrStr.c_str());
    exit(1);
  }

  FunctionPassManager OurFPM(TheModule);

  // Set up the optimizer pipeline.  Start with registering info about how the
  // target lays out data structures.
  TheModule->setDataLayout(TheExecutionEngine->getDataLayout());
  OurFPM.add(new DataLayoutPass());
  // Provide basic AliasAnalysis support for GVN.
  OurFPM.add(createBasicAliasAnalysisPass());
  // Do simple "peephole" optimizations and bit-twiddling optzns.
  OurFPM.add(createInstructionCombiningPass());
  // Reassociate expressions.
  OurFPM.add(createReassociatePass());
  // Eliminate Common SubExpressions.
  OurFPM.add(createGVNPass());
  // Simplify the control flow graph (deleting unreachable blocks, etc).
  OurFPM.add(createCFGSimplificationPass());

  OurFPM.doInitialization();

  // Set the global so the code gen can use this.
  TheFPM = &OurFPM;

  // Run the main "interpreter loop" now.
  MainLoop();

  TheFPM = 0;

  // Print out all of the generated code.
  TheModule->dump();

  return 0;
}
```

Next: Extending the language: mutable variables / SSA construction

Kaleidoscope: Extending the Language: Mutable Variables

Chapter 7 Introduction

Welcome to Chapter 7 of the "Implementing a language with LLVM" tutorial. In chapters 1 through 6, we've built a very respectable, albeit simple, functional programming language. In our journey, we learned some parsing techniques, how to build and represent an AST, how to build LLVM IR, and how to optimize the resultant code as well as JIT compile it.

While Kaleidoscope is interesting as a functional language, the fact that it is functional makes it "too easy" to generate LLVM IR for it. In particular, a functional language makes it very easy to build LLVM IR directly in SSA form. Since LLVM requires that the input code be in SSA form, this is a very nice property and it is often unclear to newcomers how to generate code for an imperative language with mutable variables.

The short (and happy) summary of this chapter is that there is no need for your front-end to build SSA form: LLVM provides highly tuned and well tested support for this, though the way it works is a bit unexpected for some.

Why is this a hard problem?

To understand why mutable variables cause complexities in SSA construction, consider this extremely simple C example:

```
int G, H;
int test(_Bool Condition) {
  int X;
  if (Condition)
    X = G;
  else
    X = H;
  return X;
}
```

In this case, we have the variable "X", whose value depends on the path executed in the program. Because there are two different possible values for X before the return instruction, a PHI node is inserted to merge the two values. The LLVM IR that we want for this example looks like this:

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.2 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.2
}
```

In this example, the loads from the G and H global variables are explicit in the LLVM IR, and they live in the then/else branches of the if statement (cond_true/cond_false). In order to merge the incoming values, the X.2 phi node in the cond_next block selects the right value to use based on where control flow is coming from: if control flow comes from

the cond_false block, X.2 gets the value of X.1. Alternatively, if control flow comes from cond_true, it gets the value of X.0. The intent of this chapter is not to explain the details of SSA form. For more information, see one of the many online references.

The question for this article is "who places the phi nodes when lowering assignments to mutable variables?". The issue here is that LLVM *requires* that its IR be in SSA form: there is no "non-ssa" mode for it. However, SSA construction requires non-trivial algorithms and data structures, so it is inconvenient and wasteful for every front-end to have to reproduce this logic.

Memory in LLVM

The 'trick' here is that while LLVM does require all register values to be in SSA form, it does not require (or permit) memory objects to be in SSA form. In the example above, note that the loads from G and H are direct accesses to G and H: they are not renamed or versioned. This differs from some other compiler systems, which do try to version memory objects. In LLVM, instead of encoding dataflow analysis of memory into the LLVM IR, it is handled with Analysis Passes which are computed on demand.

With this in mind, the high-level idea is that we want to make a stack variable (which lives in memory, because it is on the stack) for each mutable object in a function. To take advantage of this trick, we need to talk about how LLVM represents stack variables.

In LLVM, all memory accesses are explicit with load/store instructions, and it is carefully designed not to have (or need) an "address-of" operator. Notice how the type of the @G/@H global variables is actually "i32*" even though the variable is defined as "i32". What this means is that @G defines *space* for an i32 in the global data area, but its *name* actually refers to the address for that space. Stack variables work the same way, except that instead of being declared with global variable definitions, they are declared with the LLVM alloca instruction:

```
define i32 @example() {
entry:
  %X = alloca i32           ; type of %X is i32*.
  ...
  %tmp = load i32* %X       ; load the stack value %X from the stack.
  %tmp2 = add i32 %tmp, 1   ; increment it
  store i32 %tmp2, i32* %X  ; store it back
  ...
```

This code shows an example of how you can declare and manipulate a stack variable in the LLVM IR. Stack memory allocated with the alloca instruction is fully general: you can pass the address of the stack slot to functions, you can store it in other variables, etc. In our example above, we could rewrite the example to use the alloca technique to avoid using a PHI node:

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  %X = alloca i32           ; type of %X is i32*.
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  store i32 %X.0, i32* %X   ; Update X
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  store i32 %X.1, i32* %X   ; Update X
```

```
  br label %cond_next

cond_next:
  %X.2 = load i32* %X        ; Read X
  ret i32 %X.2
}
```

With this, we have discovered a way to handle arbitrary mutable variables without the need to create Phi nodes at all:

1. Each mutable variable becomes a stack allocation.
2. Each read of the variable becomes a load from the stack.
3. Each update of the variable becomes a store to the stack.
4. Taking the address of a variable just uses the stack address directly.

While this solution has solved our immediate problem, it introduced another one: we have now apparently introduced a lot of stack traffic for very simple and common operations, a major performance problem. Fortunately for us, the LLVM optimizer has a highly-tuned optimization pass named "mem2reg" that handles this case, promoting allocas like this into SSA registers, inserting Phi nodes as appropriate. If you run this example through the pass, for example, you'll get:

```
$ llvm-as < example.ll | opt -mem2reg | llvm-dis
@G = weak global i32 0
@H = weak global i32 0

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.01 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.01
}
```

The mem2reg pass implements the standard "iterated dominance frontier" algorithm for constructing SSA form and has a number of optimizations that speed up (very common) degenerate cases. The mem2reg optimization pass is the answer to dealing with mutable variables, and we highly recommend that you depend on it. Note that mem2reg only works on variables in certain circumstances:

1. mem2reg is alloca-driven: it looks for allocas and if it can handle them, it promotes them. It does not apply to global variables or heap allocations.
2. mem2reg only looks for alloca instructions in the entry block of the function. Being in the entry block guarantees that the alloca is only executed once, which makes analysis simpler.
3. mem2reg only promotes allocas whose uses are direct loads and stores. If the address of the stack object is passed to a function, or if any funny pointer arithmetic is involved, the alloca will not be promoted.
4. mem2reg only works on allocas of first class values (such as pointers, scalars and vectors), and only if the array size of the allocation is 1 (or missing in the .ll file). mem2reg is not capable of promoting structs or arrays

to registers. Note that the "scalarrepl" pass is more powerful and can promote structs, "unions", and arrays in many cases.

All of these properties are easy to satisfy for most imperative languages, and we'll illustrate it below with Kaleidoscope. The final question you may be asking is: should I bother with this nonsense for my front-end? Wouldn't it be better if I just did SSA construction directly, avoiding use of the mem2reg optimization pass? In short, we strongly recommend that you use this technique for building SSA form, unless there is an extremely good reason not to. Using this technique is:

- Proven and well tested: clang uses this technique for local mutable variables. As such, the most common clients of LLVM are using this to handle a bulk of their variables. You can be sure that bugs are found fast and fixed early.
- Extremely Fast: mem2reg has a number of special cases that make it fast in common cases as well as fully general. For example, it has fast-paths for variables that are only used in a single block, variables that only have one assignment point, good heuristics to avoid insertion of unneeded phi nodes, etc.
- Needed for debug info generation: Debug information in LLVM relies on having the address of the variable exposed so that debug info can be attached to it. This technique dovetails very naturally with this style of debug info.

If nothing else, this makes it much easier to get your front-end up and running, and is very simple to implement. Lets extend Kaleidoscope with mutable variables now!

Mutable Variables in Kaleidoscope

Now that we know the sort of problem we want to tackle, lets see what this looks like in the context of our little Kaleidoscope language. We're going to add two features:

1. The ability to mutate variables with the '=' operator.
2. The ability to define new variables.

While the first item is really what this is about, we only have variables for incoming arguments as well as for induction variables, and redefining those only goes so far :). Also, the ability to define new variables is a useful thing regardless of whether you will be mutating them. Here's a motivating example that shows how we could use these:

```
# Define ':' for sequencing: as a low-precedence operator that ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;

# Recursive fib, we could do this before.
def fib(x)
  if (x < 3) then
    1
  else
    fib(x-1)+fib(x-2);

# Iterative fib.
def fibi(x)
  var a = 1, b = 1, c in
  (for i = 3, i < x in
     c = a + b :
     a = b :
     b = c) :
  b;

# Call it.
fibi(10);
```

In order to mutate variables, we have to change our existing variables to use the "alloca trick". Once we have that, we'll add our new operator, then extend Kaleidoscope to support new variable definitions.

Adjusting Existing Variables for Mutation

The symbol table in Kaleidoscope is managed at code generation time by the 'NamedValues' map. This map currently keeps track of the LLVM "Value*" that holds the double value for the named variable. In order to support mutation, we need to change this slightly, so that it NamedValues holds the *memory location* of the variable in question. Note that this change is a refactoring: it changes the structure of the code, but does not (by itself) change the behavior of the compiler. All of these changes are isolated in the Kaleidoscope code generator.

At this point in Kaleidoscope's development, it only supports variables for two things: incoming arguments to functions and the induction variable of 'for' loops. For consistency, we'll allow mutation of these variables in addition to other user-defined variables. This means that these will both need memory locations.

To start our transformation of Kaleidoscope, we'll change the NamedValues map so that it maps to AllocaInst* instead of Value*. Once we do this, the C++ compiler will tell us what parts of the code we need to update:

```
static std::map<std::string, AllocaInst*> NamedValues;
```

Also, since we will need to create these alloca's, we'll use a helper function that ensures that the allocas are created in the entry block of the function:

```
/// CreateEntryBlockAlloca - Create an alloca instruction in the entry block of
/// the function.  This is used for mutable variables etc.
static AllocaInst *CreateEntryBlockAlloca(Function *TheFunction,
                                          const std::string &VarName) {
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(),
                   TheFunction->getEntryBlock().begin());
  return TmpB.CreateAlloca(Type::getDoubleTy(getGlobalContext()), 0,
                           VarName.c_str());
}
```

This funny looking code creates an IRBuilder object that is pointing at the first instruction (.begin()) of the entry block. It then creates an alloca with the expected name and returns it. Because all values in Kaleidoscope are doubles, there is no need to pass in a type to use.

With this in place, the first functionality change we want to make is to variable references. In our new scheme, variables live on the stack, so code generating a reference to them actually needs to produce a load from the stack slot:

```
Value *VariableExprAST::Codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (V == 0) return ErrorV("Unknown variable name");

  // Load the value.
  return Builder.CreateLoad(V, Name.c_str());
}
```

As you can see, this is pretty straightforward. Now we need to update the things that define the variables to set up the alloca. We'll start with ForExprAST::Codegen (see the full code listing for the unabridged code):

```
Function *TheFunction = Builder.GetInsertBlock()->getParent();

// Create an alloca for the variable in the entry block.
AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);

  // Emit the start code first, without 'variable' in scope.
```

```
Value *StartVal = Start->Codegen();
if (StartVal == 0) return 0;

// Store the value into the alloca.
Builder.CreateStore(StartVal, Alloca);
...

// Compute the end condition.
Value *EndCond = End->Codegen();
if (EndCond == 0) return EndCond;

// Reload, increment, and restore the alloca.  This handles the case where
// the body of the loop mutates the variable.
Value *CurVar = Builder.CreateLoad(Alloca);
Value *NextVar = Builder.CreateFAdd(CurVar, StepVal, "nextvar");
Builder.CreateStore(NextVar, Alloca);
...
```

This code is virtually identical to the code before we allowed mutable variables. The big difference is that we no longer have to construct a PHI node, and we use load/store to access the variable as needed.

To support mutable argument variables, we need to also make allocas for them. The code for this is also pretty simple:

```
/// CreateArgumentAllocas - Create an alloca for each argument and register the
/// argument in the symbol table so that references to it will succeed.
void PrototypeAST::CreateArgumentAllocas(Function *F) {
  Function::arg_iterator AI = F->arg_begin();
  for (unsigned Idx = 0, e = Args.size(); Idx != e; ++Idx, ++AI) {
    // Create an alloca for this variable.
    AllocaInst *Alloca = CreateEntryBlockAlloca(F, Args[Idx]);

    // Store the initial value into the alloca.
    Builder.CreateStore(AI, Alloca);

    // Add arguments to variable symbol table.
    NamedValues[Args[Idx]] = Alloca;
  }
}
```

For each argument, we make an alloca, store the input value to the function into the alloca, and register the alloca as the memory location for the argument. This method gets invoked by `FunctionAST::Codegen` right after it sets up the entry block for the function.

The final missing piece is adding the mem2reg pass, which allows us to get good codegen once again:

```
// Set up the optimizer pipeline.  Start with registering info about how the
// target lays out data structures.
OurFPM.add(new DataLayout(*TheExecutionEngine->getDataLayout()));
// Promote allocas to registers.
OurFPM.add(createPromoteMemoryToRegisterPass());
// Do simple "peephole" optimizations and bit-twiddling optzns.
OurFPM.add(createInstructionCombiningPass());
// Reassociate expressions.
OurFPM.add(createReassociatePass());
```

It is interesting to see what the code looks like before and after the mem2reg optimization runs. For example, this is the before/after code for our recursive fib function. Before the optimization:

```
define double @fib(double %x) {
entry:
  %x1 = alloca double
  store double %x, double* %x1
  %x2 = load double* %x1
  %cmptmp = fcmp ult double %x2, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:       ; preds = %entry
  br label %ifcont

else:       ; preds = %entry
  %x3 = load double* %x1
  %subtmp = fsub double %x3, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %x4 = load double* %x1
  %subtmp5 = fsub double %x4, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```

Here there is only one variable (x, the input argument) but you can still see the extremely simple-minded code generation strategy we are using. In the entry block, an alloca is created, and the initial input value is stored into it. Each reference to the variable does a reload from the stack. Also, note that we didn't modify the if/then/else expression, so it still inserts a PHI node. While we could make an alloca for it, it is actually easier to create a PHI node for it, so we still just make the PHI.

Here is the code after the mem2reg pass runs:

```
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:
  br label %ifcont

else:
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp5 = fsub double %x, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```

This is a trivial case for mem2reg, since there are no redefinitions of the variable. The point of showing this is to calm your tension about inserting such blatent inefficiencies :).

After the rest of the optimizers run, we get:

```
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp ueq double %booltmp, 0.000000e+00
  br i1 %ifcond, label %else, label %ifcont

else:
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp5 = fsub double %x, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  ret double %addtmp

ifcont:
  ret double 1.000000e+00
}
```

Here we see that the simplifycfg pass decided to clone the return instruction into the end of the 'else' block. This allowed it to eliminate some branches and the PHI node.

Now that all symbol table references are updated to use stack variables, we'll add the assignment operator.

New Assignment Operator

With our current framework, adding a new assignment operator is really simple. We will parse it just like any other binary operator, but handle it internally (instead of allowing the user to define it). The first step is to set a precedence:

```
int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['='] = 2;
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
```

Now that the parser knows the precedence of the binary operator, it takes care of all the parsing and AST generation. We just need to implement codegen for the assignment operator. This looks like:

```
Value *BinaryExprAST::Codegen() {
  // Special case '=' because we don't want to emit the LHS as an expression.
  if (Op == '=') {
    // Assignment requires the LHS to be an identifier.
    VariableExprAST *LHSE = dynamic_cast<VariableExprAST*>(LHS);
    if (!LHSE)
      return ErrorV("destination of '=' must be a variable");
```

Unlike the rest of the binary operators, our assignment operator doesn't follow the "emit LHS, emit RHS, do computation" model. As such, it is handled as a special case before the other binary operators are handled. The other strange thing is that it requires the LHS to be a variable. It is invalid to have "(x+1) = expr" - only things like "x = expr" are allowed.

```
  // Codegen the RHS.
  Value *Val = RHS->Codegen();
  if (Val == 0) return 0;

  // Look up the name.
  Value *Variable = NamedValues[LHSE->getName()];
  if (Variable == 0) return ErrorV("Unknown variable name");

  Builder.CreateStore(Val, Variable);
  return Val;
}
...
```

Once we have the variable, codegen'ing the assignment is straightforward: we emit the RHS of the assignment, create a store, and return the computed value. Returning a value allows for chained assignments like "X = (Y = Z)".

Now that we have an assignment operator, we can mutate loop variables and arguments. For example, we can now run code like this:

```
# Function to print a double.
extern printd(x);

# Define ':' for sequencing: as a low-precedence operator that ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;

def test(x)
  printd(x) :
  x = 4 :
  printd(x);

test(123);
```

When run, this example prints "123" and then "4", showing that we did actually mutate the value! Okay, we have now officially implemented our goal: getting this to work requires SSA construction in the general case. However, to be really useful, we want the ability to define our own local variables, lets add this next!

User-defined Local Variables

Adding var/in is just like any other other extensions we made to Kaleidoscope: we extend the lexer, the parser, the AST and the code generator. The first step for adding our new 'var/in' construct is to extend the lexer. As before, this is pretty trivial, the code looks like this:

```
enum Token {
  ...
  // var definition
  tok_var = -13
...
}
...
static int gettok() {
...
    if (IdentifierStr == "in") return tok_in;
    if (IdentifierStr == "binary") return tok_binary;
    if (IdentifierStr == "unary") return tok_unary;
    if (IdentifierStr == "var") return tok_var;
    return tok_identifier;
```

```
...
```

The next step is to define the AST node that we will construct. For var/in, it looks like this:

```
/// VarExprAST - Expression class for var/in
class VarExprAST : public ExprAST {
  std::vector<std::pair<std::string, ExprAST*> > VarNames;
  ExprAST *Body;
public:
  VarExprAST(const std::vector<std::pair<std::string, ExprAST*> > &varnames,
             ExprAST *body)
  : VarNames(varnames), Body(body) {}

  virtual Value *Codegen();
};
```

var/in allows a list of names to be defined all at once, and each name can optionally have an initializer value. As such, we capture this information in the VarNames vector. Also, var/in has a body, this body is allowed to access the variables defined by the var/in.

With this in place, we can define the parser pieces. The first thing we do is add it as a primary expression:

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
///   ::= ifexpr
///   ::= forexpr
///   ::= varexpr
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  case tok_if:         return ParseIfExpr();
  case tok_for:        return ParseForExpr();
  case tok_var:        return ParseVarExpr();
  }
}
```

Next we define ParseVarExpr:

```
/// varexpr ::= 'var' identifier ('=' expression)?
//                    (',' identifier ('=' expression)?)* 'in' expression
static ExprAST *ParseVarExpr() {
  getNextToken();  // eat the var.

  std::vector<std::pair<std::string, ExprAST*> > VarNames;

  // At least one variable name is required.
  if (CurTok != tok_identifier)
    return Error("expected identifier after var");
```

The first part of this code parses the list of identifier/expr pairs into the local `VarNames` vector.

```
while (1) {
  std::string Name = IdentifierStr;
  getNextToken();  // eat identifier.
```

```
  // Read the optional initializer.
  ExprAST *Init = 0;
  if (CurTok == '=') {
    getNextToken(); // eat the '='.

    Init = ParseExpression();
    if (Init == 0) return 0;
  }

  VarNames.push_back(std::make_pair(Name, Init));

  // End of var list, exit loop.
  if (CurTok != ',') break;
  getNextToken(); // eat the ','.

  if (CurTok != tok_identifier)
    return Error("expected identifier list after var");
}
```

Once all the variables are parsed, we then parse the body and create the AST node:

```
  // At this point, we have to have 'in'.
  if (CurTok != tok_in)
    return Error("expected 'in' keyword after 'var'");
  getNextToken();  // eat 'in'.

  ExprAST *Body = ParseExpression();
  if (Body == 0) return 0;

  return new VarExprAST(VarNames, Body);
}
```

Now that we can parse and represent the code, we need to support emission of LLVM IR for it. This code starts out with:

```
Value *VarExprAST::Codegen() {
  std::vector<AllocaInst *> OldBindings;

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Register all variables and emit their initializer.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i) {
    const std::string &VarName = VarNames[i].first;
    ExprAST *Init = VarNames[i].second;
```

Basically it loops over all the variables, installing them one at a time. For each variable we put into the symbol table, we remember the previous value that we replace in OldBindings.

```
    // Emit the initializer before adding the variable to scope, this prevents
    // the initializer from referencing the variable itself, and permits stuff
    // like this:
    //  var a = 1 in
    //    var a = a in ...   # refers to outer 'a'.
    Value *InitVal;
    if (Init) {
      InitVal = Init->Codegen();
      if (InitVal == 0) return 0;
    } else { // If not specified, use 0.0.
      InitVal = ConstantFP::get(getGlobalContext(), APFloat(0.0));
```

```
  }

  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
  Builder.CreateStore(InitVal, Alloca);

  // Remember the old variable binding so that we can restore the binding when
  // we unrecurse.
  OldBindings.push_back(NamedValues[VarName]);

  // Remember this binding.
  NamedValues[VarName] = Alloca;
}
```

There are more comments here than code. The basic idea is that we emit the initializer, create the alloca, then update the symbol table to point to it. Once all the variables are installed in the symbol table, we evaluate the body of the var/in expression:

```
// Codegen the body, now that all vars are in scope.
Value *BodyVal = Body->Codegen();
if (BodyVal == 0) return 0;
```

Finally, before returning, we restore the previous variable bindings:

```
  // Pop all our variables from scope.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i)
    NamedValues[VarNames[i].first] = OldBindings[i];

  // Return the body computation.
  return BodyVal;
}
```

The end result of all of this is that we get properly scoped variable definitions, and we even (trivially) allow mutation of them :).

With this, we completed what we set out to do. Our nice iterative fib example from the intro compiles and runs just fine. The mem2reg pass optimizes all of our stack variables into SSA registers, inserting PHI nodes where needed, and our front-end remains simple: no "iterated dominance frontier" computation anywhere in sight.

Full Code Listing

Here is the complete code listing for our running example, enhanced with mutable variables and var/in support. To build this example, use:

```
# Compile
clang++ -g toy.cpp `llvm-config --cppflags --ldflags --libs core jit native` -O3 -o toy
# Run
./toy
```

Here is the code:

```
#include "llvm/Analysis/Passes.h"
#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
```

```
#include "llvm/PassManager.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Transforms/Scalar.h"
#include <cctype>
#include <cstdio>
#include <map>
#include <string>
#include <vector>
using namespace llvm;

//===----------------------------------------------------------------------===//
// Lexer
//===----------------------------------------------------------------------===//

// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2, tok_extern = -3,

  // primary
  tok_identifier = -4, tok_number = -5,

  // control
  tok_if = -6, tok_then = -7, tok_else = -8,
  tok_for = -9, tok_in = -10,

  // operators
  tok_binary = -11, tok_unary = -12,

  // var definition
  tok_var = -13
};

static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number

/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();

  if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getchar())))
      IdentifierStr += LastChar;

    if (IdentifierStr == "def") return tok_def;
    if (IdentifierStr == "extern") return tok_extern;
    if (IdentifierStr == "if") return tok_if;
    if (IdentifierStr == "then") return tok_then;
    if (IdentifierStr == "else") return tok_else;
    if (IdentifierStr == "for") return tok_for;
```

```
    if (IdentifierStr == "in") return tok_in;
    if (IdentifierStr == "binary") return tok_binary;
    if (IdentifierStr == "unary") return tok_unary;
    if (IdentifierStr == "var") return tok_var;
    return tok_identifier;
  }

  if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
    std::string NumStr;
    do {
      NumStr += LastChar;
      LastChar = getchar();
    } while (isdigit(LastChar) || LastChar == '.');

    NumVal = strtod(NumStr.c_str(), 0);
    return tok_number;
  }

  if (LastChar == '#') {
    // Comment until end of line.
    do LastChar = getchar();
    while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

    if (LastChar != EOF)
      return gettok();
  }

  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}

//===----------------------------------------------------------------------===//
// Abstract Syntax Tree (aka Parse Tree)
//===----------------------------------------------------------------------===//
namespace {
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
  virtual Value *Codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val) : Val(val) {}
  virtual Value *Codegen();
};

/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
```

```
  std::string Name;
public:
  VariableExprAST(const std::string &name) : Name(name) {}
  const std::string &getName() const { return Name; }
  virtual Value *Codegen();
};

/// UnaryExprAST - Expression class for a unary operator.
class UnaryExprAST : public ExprAST {
  char Opcode;
  ExprAST *Operand;
public:
  UnaryExprAST(char opcode, ExprAST *operand)
    : Opcode(opcode), Operand(operand) {}
  virtual Value *Codegen();
};

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  ExprAST *LHS, *RHS;
public:
  BinaryExprAST(char op, ExprAST *lhs, ExprAST *rhs)
    : Op(op), LHS(lhs), RHS(rhs) {}
  virtual Value *Codegen();
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<ExprAST*> Args;
public:
  CallExprAST(const std::string &callee, std::vector<ExprAST*> &args)
    : Callee(callee), Args(args) {}
  virtual Value *Codegen();
};

/// IfExprAST - Expression class for if/then/else.
class IfExprAST : public ExprAST {
  ExprAST *Cond, *Then, *Else;
public:
  IfExprAST(ExprAST *cond, ExprAST *then, ExprAST *_else)
  : Cond(cond), Then(then), Else(_else) {}
  virtual Value *Codegen();
};

/// ForExprAST - Expression class for for/in.
class ForExprAST : public ExprAST {
  std::string VarName;
  ExprAST *Start, *End, *Step, *Body;
public:
  ForExprAST(const std::string &varname, ExprAST *start, ExprAST *end,
             ExprAST *step, ExprAST *body)
    : VarName(varname), Start(start), End(end), Step(step), Body(body) {}
  virtual Value *Codegen();
};

/// VarExprAST - Expression class for var/in
```

```
class VarExprAST : public ExprAST {
  std::vector<std::pair<std::string, ExprAST*> > VarNames;
  ExprAST *Body;
public:
  VarExprAST(const std::vector<std::pair<std::string, ExprAST*> > &varnames,
             ExprAST *body)
  : VarNames(varnames), Body(body) {}

  virtual Value *Codegen();
};

/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its argument names as well as if it is an operator.
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
  bool isOperator;
  unsigned Precedence;  // Precedence if a binary op.
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args,
               bool isoperator = false, unsigned prec = 0)
  : Name(name), Args(args), isOperator(isoperator), Precedence(prec) {}

  bool isUnaryOp() const { return isOperator && Args.size() == 1; }
  bool isBinaryOp() const { return isOperator && Args.size() == 2; }

  char getOperatorName() const {
    assert(isUnaryOp() || isBinaryOp());
    return Name[Name.size()-1];
  }

  unsigned getBinaryPrecedence() const { return Precedence; }

  Function *Codegen();

  void CreateArgumentAllocas(Function *F);
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  PrototypeAST *Proto;
  ExprAST *Body;
public:
  FunctionAST(PrototypeAST *proto, ExprAST *body)
    : Proto(proto), Body(body) {}

  Function *Codegen();
};
} // end anonymous namespace

//===----------------------------------------------------------------------===//
// Parser
//===----------------------------------------------------------------------===//

/// CurTok/getNextToken - Provide a simple token buffer.  CurTok is the current
/// token the parser is looking at.  getNextToken reads another token from the
/// lexer and updates CurTok with its results.
static int CurTok;
```

```
static int getNextToken() {
  return CurTok = gettok();
}

/// BinopPrecedence - This holds the precedence for each binary operator that is
/// defined.
static std::map<char, int> BinopPrecedence;

/// GetTokPrecedence - Get the precedence of the pending binary operator token.
static int GetTokPrecedence() {
  if (!isascii(CurTok))
    return -1;

  // Make sure it's a declared binop.
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0) return -1;
  return TokPrec;
}

/// Error* - These are little helper functions for error handling.
ExprAST *Error(const char *Str) { fprintf(stderr, "Error: %s\n", Str);return 0;}
PrototypeAST *ErrorP(const char *Str) { Error(Str); return 0; }
FunctionAST *ErrorF(const char *Str) { Error(Str); return 0; }

static ExprAST *ParseExpression();

/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static ExprAST *ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return new VariableExprAST(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<ExprAST*> Args;
  if (CurTok != ')') {
    while (1) {
      ExprAST *Arg = ParseExpression();
      if (!Arg) return 0;
      Args.push_back(Arg);

      if (CurTok == ')') break;

      if (CurTok != ',')
        return Error("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return new CallExprAST(IdName, Args);
```

```
}

/// numberexpr ::= number
static ExprAST *ParseNumberExpr() {
  ExprAST *Result = new NumberExprAST(NumVal);
  getNextToken(); // consume the number
  return Result;
}

/// parenexpr ::= '(' expression ')'
static ExprAST *ParseParenExpr() {
  getNextToken();  // eat (.
  ExprAST *V = ParseExpression();
  if (!V) return 0;

  if (CurTok != ')')
    return Error("expected ')'");
  getNextToken();  // eat ).
  return V;
}

/// ifexpr ::= 'if' expression 'then' expression 'else' expression
static ExprAST *ParseIfExpr() {
  getNextToken();  // eat the if.

  // condition.
  ExprAST *Cond = ParseExpression();
  if (!Cond) return 0;

  if (CurTok != tok_then)
    return Error("expected then");
  getNextToken();  // eat the then

  ExprAST *Then = ParseExpression();
  if (Then == 0) return 0;

  if (CurTok != tok_else)
    return Error("expected else");

  getNextToken();

  ExprAST *Else = ParseExpression();
  if (!Else) return 0;

  return new IfExprAST(Cond, Then, Else);
}

/// forexpr ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression
static ExprAST *ParseForExpr() {
  getNextToken();  // eat the for.

  if (CurTok != tok_identifier)
    return Error("expected identifier after for");

  std::string IdName = IdentifierStr;
  getNextToken();  // eat identifier.

  if (CurTok != '=')
```

```
    return Error("expected '=' after for");
  getNextToken();  // eat '='.


  ExprAST *Start = ParseExpression();
  if (Start == 0) return 0;
  if (CurTok != ',')
    return Error("expected ',' after for start value");
  getNextToken();

  ExprAST *End = ParseExpression();
  if (End == 0) return 0;

  // The step value is optional.
  ExprAST *Step = 0;
  if (CurTok == ',') {
    getNextToken();
    Step = ParseExpression();
    if (Step == 0) return 0;
  }

  if (CurTok != tok_in)
    return Error("expected 'in' after for");
  getNextToken();  // eat 'in'.

  ExprAST *Body = ParseExpression();
  if (Body == 0) return 0;

  return new ForExprAST(IdName, Start, End, Step, Body);
}

/// varexpr ::= 'var' identifier ('=' expression)?
//                    (',' identifier ('=' expression)?)* 'in' expression
static ExprAST *ParseVarExpr() {
  getNextToken();  // eat the var.

  std::vector<std::pair<std::string, ExprAST*> > VarNames;

  // At least one variable name is required.
  if (CurTok != tok_identifier)
    return Error("expected identifier after var");

  while (1) {
    std::string Name = IdentifierStr;
    getNextToken();  // eat identifier.

    // Read the optional initializer.
    ExprAST *Init = 0;
    if (CurTok == '=') {
      getNextToken(); // eat the '='.

      Init = ParseExpression();
      if (Init == 0) return 0;
    }

    VarNames.push_back(std::make_pair(Name, Init));

    // End of var list, exit loop.
```

```
    if (CurTok != ',') break;
    getNextToken(); // eat the ','.

    if (CurTok != tok_identifier)
      return Error("expected identifier list after var");
  }

  // At this point, we have to have 'in'.
  if (CurTok != tok_in)
    return Error("expected 'in' keyword after 'var'");
  getNextToken();  // eat 'in'.

  ExprAST *Body = ParseExpression();
  if (Body == 0) return 0;

  return new VarExprAST(VarNames, Body);
}

/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
///   ::= ifexpr
///   ::= forexpr
///   ::= varexpr
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  case tok_if:         return ParseIfExpr();
  case tok_for:        return ParseForExpr();
  case tok_var:        return ParseVarExpr();
  }
}

/// unary
///   ::= primary
///   ::= '!' unary
static ExprAST *ParseUnary() {
  // If the current token is not an operator, it must be a primary expr.
  if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
    return ParsePrimary();

  // If this is a unary operator, read it.
  int Opc = CurTok;
  getNextToken();
  if (ExprAST *Operand = ParseUnary())
    return new UnaryExprAST(Opc, Operand);
  return 0;
}

/// binoprhs
///   ::= ('+' unary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  // If this is a binop, find its precedence.
  while (1) {
```

```
    int TokPrec = GetTokPrecedence();

    // If this is a binop that binds at least as tightly as the current binop,
    // consume it, otherwise we are done.
    if (TokPrec < ExprPrec)
      return LHS;

    // Okay, we know this is a binop.
    int BinOp = CurTok;
    getNextToken();  // eat binop

    // Parse the unary expression after the binary operator.
    ExprAST *RHS = ParseUnary();
    if (!RHS) return 0;

    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec+1, RHS);
      if (RHS == 0) return 0;
    }

    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }
}

/// expression
///   ::= unary binoprhs
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParseUnary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
}

/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
///   ::= unary LETTER (id)
static PrototypeAST *ParsePrototype() {
  std::string FnName;

  unsigned Kind = 0; // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return ErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_unary:
    getNextToken();
```

```
    if (!isascii(CurTok))
      return ErrorP("Expected unary operator");
    FnName = "unary";
    FnName += (char)CurTok;
    Kind = 1;
    getNextToken();
    break;
  case tok_binary:
    getNextToken();
    if (!isascii(CurTok))
      return ErrorP("Expected binary operator");
    FnName = "binary";
    FnName += (char)CurTok;
    Kind = 2;
    getNextToken();

    // Read the precedence if present.
    if (CurTok == tok_number) {
      if (NumVal < 1 || NumVal > 100)
        return ErrorP("Invalid precedecnce: must be 1..100");
      BinaryPrecedence = (unsigned)NumVal;
      getNextToken();
    }
    break;
  }

  if (CurTok != '(')
    return ErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return ErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  // Verify right number of names for operator.
  if (Kind && ArgNames.size() != Kind)
    return ErrorP("Invalid number of operands for operator");

  return new PrototypeAST(FnName, ArgNames, Kind != 0, BinaryPrecedence);
}

/// definition ::= 'def' prototype expression
static FunctionAST *ParseDefinition() {
  getNextToken();  // eat def.
  PrototypeAST *Proto = ParsePrototype();
  if (Proto == 0) return 0;

  if (ExprAST *E = ParseExpression())
    return new FunctionAST(Proto, E);
  return 0;
}

/// toplevelexpr ::= expression
static FunctionAST *ParseTopLevelExpr() {
```

```
  if (ExprAST *E = ParseExpression()) {
    // Make an anonymous proto.
    PrototypeAST *Proto = new PrototypeAST("", std::vector<std::string>());
    return new FunctionAST(Proto, E);
  }
  return 0;
}

/// external ::= 'extern' prototype
static PrototypeAST *ParseExtern() {
  getNextToken();  // eat extern.
  return ParsePrototype();
}

//===----------------------------------------------------------------------===//
// Code Generation
//===----------------------------------------------------------------------===//

static Module *TheModule;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, AllocaInst*> NamedValues;
static FunctionPassManager *TheFPM;

Value *ErrorV(const char *Str) { Error(Str); return 0; }

/// CreateEntryBlockAlloca - Create an alloca instruction in the entry block of
/// the function.  This is used for mutable variables etc.
static AllocaInst *CreateEntryBlockAlloca(Function *TheFunction,
                                          const std::string &VarName) {
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(),
                   TheFunction->getEntryBlock().begin());
  return TmpB.CreateAlloca(Type::getDoubleTy(getGlobalContext()), 0,
                           VarName.c_str());
}

Value *NumberExprAST::Codegen() {
  return ConstantFP::get(getGlobalContext(), APFloat(Val));
}

Value *VariableExprAST::Codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (V == 0) return ErrorV("Unknown variable name");

  // Load the value.
  return Builder.CreateLoad(V, Name.c_str());
}

Value *UnaryExprAST::Codegen() {
  Value *OperandV = Operand->Codegen();
  if (OperandV == 0) return 0;

  Function *F = TheModule->getFunction(std::string("unary")+Opcode);
  if (F == 0)
    return ErrorV("Unknown unary operator");

  return Builder.CreateCall(F, OperandV, "unop");
}
```

```
Value *BinaryExprAST::Codegen() {
  // Special case '=' because we don't want to emit the LHS as an expression.
  if (Op == '=') {
    // Assignment requires the LHS to be an identifier.
    VariableExprAST *LHSE = dynamic_cast<VariableExprAST*>(LHS);
    if (!LHSE)
      return ErrorV("destination of '=' must be a variable");
    // Codegen the RHS.
    Value *Val = RHS->Codegen();
    if (Val == 0) return 0;

    // Look up the name.
    Value *Variable = NamedValues[LHSE->getName()];
    if (Variable == 0) return ErrorV("Unknown variable name");

    Builder.CreateStore(Val, Variable);
    return Val;
  }

  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateFAdd(L, R, "addtmp");
  case '-': return Builder.CreateFSub(L, R, "subtmp");
  case '*': return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                "booltmp");
  default: break;
  }

  // If it wasn't a builtin binary operator, it must be a user defined one. Emit
  // a call to it.
  Function *F = TheModule->getFunction(std::string("binary")+Op);
  assert(F && "binary operator not found!");

  Value *Ops[] = { L, R };
  return Builder.CreateCall(F, Ops, "binop");
}

Value *CallExprAST::Codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (CalleeF == 0)
    return ErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return ErrorV("Incorrect # arguments passed");

  std::vector<Value*> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->Codegen());
    if (ArgsV.back() == 0) return 0;
```

```
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}

Value *IfExprAST::Codegen() {
  Value *CondV = Cond->Codegen();
  if (CondV == 0) return 0;

  // Convert condition to a bool by comparing equal to 0.0.
  CondV = Builder.CreateFCmpONE(CondV,
                                ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                  "ifcond");

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Create blocks for the then and else cases.  Insert the 'then' block at the
  // end of the function.
  BasicBlock *ThenBB = BasicBlock::Create(getGlobalContext(), "then", TheFunction);
  BasicBlock *ElseBB = BasicBlock::Create(getGlobalContext(), "else");
  BasicBlock *MergeBB = BasicBlock::Create(getGlobalContext(), "ifcont");

  Builder.CreateCondBr(CondV, ThenBB, ElseBB);

  // Emit then value.
  Builder.SetInsertPoint(ThenBB);

  Value *ThenV = Then->Codegen();
  if (ThenV == 0) return 0;

  Builder.CreateBr(MergeBB);
  // Codegen of 'Then' can change the current block, update ThenBB for the PHI.
  ThenBB = Builder.GetInsertBlock();

  // Emit else block.
  TheFunction->getBasicBlockList().push_back(ElseBB);
  Builder.SetInsertPoint(ElseBB);

  Value *ElseV = Else->Codegen();
  if (ElseV == 0) return 0;

  Builder.CreateBr(MergeBB);
  // Codegen of 'Else' can change the current block, update ElseBB for the PHI.
  ElseBB = Builder.GetInsertBlock();

  // Emit merge block.
  TheFunction->getBasicBlockList().push_back(MergeBB);
  Builder.SetInsertPoint(MergeBB);
  PHINode *PN = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()), 2,
                                  "iftmp");

  PN->addIncoming(ThenV, ThenBB);
  PN->addIncoming(ElseV, ElseBB);
  return PN;
}

Value *ForExprAST::Codegen() {
  // Output this as:
```

```
//   var = alloca double
//   ...
//   start = startexpr
//   store start -> var
//   goto loop
// loop:
//   ...
//   bodyexpr
//   ...
// loopend:
//   step = stepexpr
//   endcond = endexpr
//
//   curvar = load var
//   nextvar = curvar + step
//   store nextvar -> var
//   br endcond, loop, endloop
// outloop:

Function *TheFunction = Builder.GetInsertBlock()->getParent();

// Create an alloca for the variable in the entry block.
AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);

// Emit the start code first, without 'variable' in scope.
Value *StartVal = Start->Codegen();
if (StartVal == 0) return 0;

// Store the value into the alloca.
Builder.CreateStore(StartVal, Alloca);

// Make the new basic block for the loop header, inserting after current
// block.
BasicBlock *LoopBB = BasicBlock::Create(getGlobalContext(), "loop", TheFunction);

// Insert an explicit fall through from the current block to the LoopBB.
Builder.CreateBr(LoopBB);

// Start insertion in LoopBB.
Builder.SetInsertPoint(LoopBB);

// Within the loop, the variable is defined equal to the PHI node.  If it
// shadows an existing variable, we have to restore it, so save it now.
AllocaInst *OldVal = NamedValues[VarName];
NamedValues[VarName] = Alloca;

// Emit the body of the loop.  This, like any other expr, can change the
// current BB.  Note that we ignore the value computed by the body, but don't
// allow an error.
if (Body->Codegen() == 0)
  return 0;

// Emit the step value.
Value *StepVal;
if (Step) {
  StepVal = Step->Codegen();
  if (StepVal == 0) return 0;
} else {
```

```
    // If not specified, use 1.0.
    StepVal = ConstantFP::get(getGlobalContext(), APFloat(1.0));
  }

  // Compute the end condition.
  Value *EndCond = End->Codegen();
  if (EndCond == 0) return EndCond;

  // Reload, increment, and restore the alloca.  This handles the case where
  // the body of the loop mutates the variable.
  Value *CurVar = Builder.CreateLoad(Alloca, VarName.c_str());
  Value *NextVar = Builder.CreateFAdd(CurVar, StepVal, "nextvar");
  Builder.CreateStore(NextVar, Alloca);

  // Convert condition to a bool by comparing equal to 0.0.
  EndCond = Builder.CreateFCmpONE(EndCond,
                              ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                  "loopcond");

  // Create the "after loop" block and insert it.
  BasicBlock *AfterBB = BasicBlock::Create(getGlobalContext(), "afterloop", TheFunction);

  // Insert the conditional branch into the end of LoopEndBB.
  Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

  // Any new code will be inserted in AfterBB.
  Builder.SetInsertPoint(AfterBB);

  // Restore the unshadowed variable.
  if (OldVal)
    NamedValues[VarName] = OldVal;
  else
    NamedValues.erase(VarName);


  // for expr always returns 0.0.
  return Constant::getNullValue(Type::getDoubleTy(getGlobalContext()));
}

Value *VarExprAST::Codegen() {
  std::vector<AllocaInst *> OldBindings;

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Register all variables and emit their initializer.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i) {
    const std::string &VarName = VarNames[i].first;
    ExprAST *Init = VarNames[i].second;

    // Emit the initializer before adding the variable to scope, this prevents
    // the initializer from referencing the variable itself, and permits stuff
    // like this:
    //  var a = 1 in
    //    var a = a in ...   # refers to outer 'a'.
    Value *InitVal;
    if (Init) {
      InitVal = Init->Codegen();
      if (InitVal == 0) return 0;
```

```
    } else { // If not specified, use 0.0.
      InitVal = ConstantFP::get(getGlobalContext(), APFloat(0.0));
    }

    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
    Builder.CreateStore(InitVal, Alloca);

    // Remember the old variable binding so that we can restore the binding when
    // we unrecurse.
    OldBindings.push_back(NamedValues[VarName]);

    // Remember this binding.
    NamedValues[VarName] = Alloca;
  }

  // Codegen the body, now that all vars are in scope.
  Value *BodyVal = Body->Codegen();
  if (BodyVal == 0) return 0;

  // Pop all our variables from scope.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i)
    NamedValues[VarNames[i].first] = OldBindings[i];

  // Return the body computation.
  return BodyVal;
}

Function *PrototypeAST::Codegen() {
  // Make the function type:  double(double,double) etc.
  std::vector<Type*> Doubles(Args.size(),
                             Type::getDoubleTy(getGlobalContext()));
  FunctionType *FT = FunctionType::get(Type::getDoubleTy(getGlobalContext()),
                                       Doubles, false);

  Function *F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule);

  // If F conflicted, there was already something named 'Name'.  If it has a
  // body, don't allow redefinition or reextern.
  if (F->getName() != Name) {
    // Delete the one we just made and get the existing one.
    F->eraseFromParent();
    F = TheModule->getFunction(Name);

    // If F already has a body, reject this.
    if (!F->empty()) {
      ErrorF("redefinition of function");
      return 0;
    }

    // If F took a different number of args, reject.
    if (F->arg_size() != Args.size()) {
      ErrorF("redefinition of function with different # args");
      return 0;
    }
  }

  // Set names for all arguments.
  unsigned Idx = 0;
```

```
  for (Function::arg_iterator AI = F->arg_begin(); Idx != Args.size();
       ++AI, ++Idx)
    AI->setName(Args[Idx]);

  return F;
}

/// CreateArgumentAllocas - Create an alloca for each argument and register the
/// argument in the symbol table so that references to it will succeed.
void PrototypeAST::CreateArgumentAllocas(Function *F) {
  Function::arg_iterator AI = F->arg_begin();
  for (unsigned Idx = 0, e = Args.size(); Idx != e; ++Idx, ++AI) {
    // Create an alloca for this variable.
    AllocaInst *Alloca = CreateEntryBlockAlloca(F, Args[Idx]);

    // Store the initial value into the alloca.
    Builder.CreateStore(AI, Alloca);

    // Add arguments to variable symbol table.
    NamedValues[Args[Idx]] = Alloca;
  }
}

Function *FunctionAST::Codegen() {
  NamedValues.clear();

  Function *TheFunction = Proto->Codegen();
  if (TheFunction == 0)
    return 0;

  // If this is an operator, install it.
  if (Proto->isBinaryOp())
    BinopPrecedence[Proto->getOperatorName()] = Proto->getBinaryPrecedence();

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);
  Builder.SetInsertPoint(BB);

  // Add all arguments to the symbol table and create their allocas.
  Proto->CreateArgumentAllocas(TheFunction);

  if (Value *RetVal = Body->Codegen()) {
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);

    // Optimize the function.
    TheFPM->run(*TheFunction);

    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();

  if (Proto->isBinaryOp())
```

```
    BinopPrecedence.erase(Proto->getOperatorName());
  return 0;
}

//===----------------------------------------------------------------------===//
// Top-Level parsing and JIT Driver
//===----------------------------------------------------------------------===//

static ExecutionEngine *TheExecutionEngine;

static void HandleDefinition() {
  if (FunctionAST *F = ParseDefinition()) {
    if (Function *LF = F->Codegen()) {
      fprintf(stderr, "Read function definition:");
      LF->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleExtern() {
  if (PrototypeAST *P = ParseExtern()) {
    if (Function *F = P->Codegen()) {
      fprintf(stderr, "Read extern: ");
      F->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (FunctionAST *F = ParseTopLevelExpr()) {
    if (Function *LF = F->Codegen()) {
      // JIT the function, returning a function pointer.
      void *FPtr = TheExecutionEngine->getPointerToFunction(LF);

      // Cast it to the right type (takes no arguments, returns a double) so we
      // can call it as a native function.
      double (*FP)() = (double (*)())(intptr_t)FPtr;
      fprintf(stderr, "Evaluated to %f\n", FP());
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (1) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:    return;
```

```
    case ';':        getNextToken(); break;  // ignore top-level semicolons.
    case tok_def:    HandleDefinition(); break;
    case tok_extern: HandleExtern(); break;
    default:         HandleTopLevelExpression(); break;
    }
  }
}

//===----------------------------------------------------------------------===//
// "Library" functions that can be "extern'd" from user code.
//===----------------------------------------------------------------------===//

/// putchard - putchar that takes a double and returns 0.
extern "C"
double putchard(double X) {
  putchar((char)X);
  return 0;
}

/// printd - printf that takes a double prints it as "%f\n", returning 0.
extern "C"
double printd(double X) {
  printf("%f\n", X);
  return 0;
}

//===----------------------------------------------------------------------===//
// Main driver code.
//===----------------------------------------------------------------------===//

int main() {
  InitializeNativeTarget();
  LLVMContext &Context = getGlobalContext();

  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['='] = 2;
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.

  // Prime the first token.
  fprintf(stderr, "ready> ");
  getNextToken();

  // Make the module, which holds all the code.
  std::unique_ptr<Module> Owner = make_unique<Module>("my cool jit", Context);
  TheModule = Owner.get();

  // Create the JIT.  This takes ownership of the module.
  std::string ErrStr;
  TheExecutionEngine =
      EngineBuilder(std::move(Owner)).setErrorStr(&ErrStr).create();
  if (!TheExecutionEngine) {
    fprintf(stderr, "Could not create ExecutionEngine: %s\n", ErrStr.c_str());
    exit(1);
  }
```

```
  FunctionPassManager OurFPM(TheModule);

  // Set up the optimizer pipeline.  Start with registering info about how the
  // target lays out data structures.
  TheModule->setDataLayout(TheExecutionEngine->getDataLayout());
  OurFPM.add(new DataLayoutPass());
  // Provide basic AliasAnalysis support for GVN.
  OurFPM.add(createBasicAliasAnalysisPass());
  // Promote allocas to registers.
  OurFPM.add(createPromoteMemoryToRegisterPass());
  // Do simple "peephole" optimizations and bit-twiddling optzns.
  OurFPM.add(createInstructionCombiningPass());
  // Reassociate expressions.
  OurFPM.add(createReassociatePass());
  // Eliminate Common SubExpressions.
  OurFPM.add(createGVNPass());
  // Simplify the control flow graph (deleting unreachable blocks, etc).
  OurFPM.add(createCFGSimplificationPass());

  OurFPM.doInitialization();

  // Set the global so the code gen can use this.
  TheFPM = &OurFPM;

  // Run the main "interpreter loop" now.
  MainLoop();

  TheFPM = 0;

  // Print out all of the generated code.
  TheModule->dump();

  return 0;
}
```

Next: Conclusion and other useful LLVM tidbits

Kaleidoscope: Conclusion and other useful LLVM tidbits

Tutorial Conclusion

Welcome to the final chapter of the "Implementing a language with LLVM" tutorial. In the course of this tutorial, we have grown our little Kaleidoscope language from being a useless toy, to being a semi-interesting (but probably still useless) toy. :)

It is interesting to see how far we've come, and how little code it has taken. We built the entire lexer, parser, AST, code generator, and an interactive run-loop (with a JIT!) by-hand in under 700 lines of (non-comment/non-blank) code.

Our little language supports a couple of interesting features: it supports user defined binary and unary operators, it uses JIT compilation for immediate evaluation, and it supports a few control flow constructs with SSA construction.

Part of the idea of this tutorial was to show you how easy and fun it can be to define, build, and play with languages. Building a compiler need not be a scary or mystical process! Now that you've seen some of the basics, I strongly encourage you to take the code and hack on it. For example, try adding:

- **global variables** - While global variables have questional value in modern software engineering, they are often useful when putting together quick little hacks like the Kaleidoscope compiler itself. Fortunately, our current setup makes it very easy to add global variables: just have value lookup check to see if an unresolved variable is in the global variable symbol table before rejecting it. To create a new global variable, make an instance of the LLVM `GlobalVariable` class.
- **typed variables** - Kaleidoscope currently only supports variables of type double. This gives the language a very nice elegance, because only supporting one type means that you never have to specify types. Different languages have different ways of handling this. The easiest way is to require the user to specify types for every variable definition, and record the type of the variable in the symbol table along with its Value*.
- **arrays, structs, vectors, etc** - Once you add types, you can start extending the type system in all sorts of interesting ways. Simple arrays are very easy and are quite useful for many different applications. Adding them is mostly an exercise in learning how the LLVM getelementptr instruction works: it is so nifty/unconventional, it has its own FAQ! If you add support for recursive types (e.g. linked lists), make sure to read the section in the LLVM Programmer's Manual that describes how to construct them.
- **standard runtime** - Our current language allows the user to access arbitrary external functions, and we use it for things like "printd" and "putchard". As you extend the language to add higher-level constructs, often these constructs make the most sense if they are lowered to calls into a language-supplied runtime. For example, if you add hash tables to the language, it would probably make sense to add the routines to a runtime, instead of inlining them all the way.
- **memory management** - Currently we can only access the stack in Kaleidoscope. It would also be useful to be able to allocate heap memory, either with calls to the standard libc malloc/free interface or with a garbage collector. If you would like to use garbage collection, note that LLVM fully supports Accurate Garbage Collection including algorithms that move objects and need to scan/update the stack.
- **debugger support** - LLVM supports generation of DWARF Debug info which is understood by common debuggers like GDB. Adding support for debug info is fairly straightforward. The best way to understand it is to compile some C/C++ code with "`clang -g -O0`" and taking a look at what it produces.
- **exception handling support** - LLVM supports generation of zero cost exceptions which interoperate with code compiled in other languages. You could also generate code by implicitly making every function return an error value and checking it. You could also make explicit use of setjmp/longjmp. There are many different ways to go here.
- **object orientation, generics, database access, complex numbers, geometric programming, ...** - Really, there is no end of crazy features that you can add to the language.
- **unusual domains** - We've been talking about applying LLVM to a domain that many people are interested in: building a compiler for a specific language. However, there are many other domains that can use compiler technology that are not typically considered. For example, LLVM has been used to implement OpenGL graphics acceleration, translate C++ code to ActionScript, and many other cute and clever things. Maybe you will be the first to JIT compile a regular expression interpreter into native code with LLVM?

Have fun - try doing something crazy and unusual. Building a language like everyone else always has, is much less fun than trying something a little crazy or off the wall and seeing how it turns out. If you get stuck or want to talk about it, feel free to email the llvmdev mailing list: it has lots of people who are interested in languages and are often willing to help out.

Before we end this tutorial, I want to talk about some "tips and tricks" for generating LLVM IR. These are some of the more subtle things that may not be obvious, but are very useful if you want to take advantage of LLVM's capabilities.

Properties of the LLVM IR

We have a couple common questions about code in the LLVM IR form - lets just get these out of the way right now, shall we?

Target Independence Kaleidoscope is an example of a "portable language": any program written in Kaleidoscope will work the same way on any target that it runs on. Many other languages have this property, e.g. lisp, java, haskell, javascript, python, etc (note that while these languages are portable, not all their libraries are).

One nice aspect of LLVM is that it is often capable of preserving target independence in the IR: you can take the LLVM IR for a Kaleidoscope-compiled program and run it on any target that LLVM supports, even emitting C code and compiling that on targets that LLVM doesn't support natively. You can trivially tell that the Kaleidoscope compiler generates target-independent code because it never queries for any target-specific information when generating code.

The fact that LLVM provides a compact, target-independent, representation for code gets a lot of people excited. Unfortunately, these people are usually thinking about C or a language from the C family when they are asking questions about language portability. I say "unfortunately", because there is really no way to make (fully general) C code portable, other than shipping the source code around (and of course, C source code is not actually portable in general either - ever port a really old application from 32- to 64-bits?).

The problem with C (again, in its full generality) is that it is heavily laden with target specific assumptions. As one simple example, the preprocessor often destructively removes target-independence from the code when it processes the input text:

```
#ifdef __i386__
  int X = 1;
#else
  int X = 42;
#endif
```

While it is possible to engineer more and more complex solutions to problems like this, it cannot be solved in full generality in a way that is better than shipping the actual source code.

That said, there are interesting subsets of C that can be made portable. If you are willing to fix primitive types to a fixed size (say int = 32-bits, and long = 64-bits), don't care about ABI compatibility with existing binaries, and are willing to give up some other minor features, you can have portable code. This can make sense for specialized domains such as an in-kernel language.

Safety Guarantees Many of the languages above are also "safe" languages: it is impossible for a program written in Java to corrupt its address space and crash the process (assuming the JVM has no bugs). Safety is an interesting property that requires a combination of language design, runtime support, and often operating system support.

It is certainly possible to implement a safe language in LLVM, but LLVM IR does not itself guarantee safety. The LLVM IR allows unsafe pointer casts, use after free bugs, buffer over-runs, and a variety of other problems. Safety needs to be implemented as a layer on top of LLVM and, conveniently, several groups have investigated this. Ask on the llvmdev mailing list if you are interested in more details.

Language-Specific Optimizations One thing about LLVM that turns off many people is that it does not solve all the world's problems in one system (sorry 'world hunger', someone else will have to solve you some other day). One specific complaint is that people perceive LLVM as being incapable of performing high-level language-specific optimization: LLVM "loses too much information".

Unfortunately, this is really not the place to give you a full and unified version of "Chris Lattner's theory of compiler design". Instead, I'll make a few observations:

First, you're right that LLVM does lose information. For example, as of this writing, there is no way to distinguish in the LLVM IR whether an SSA-value came from a C "int" or a C "long" on an ILP32 machine (other than debug info). Both get compiled down to an 'i32' value and the information about what it came from is lost. The more general issue here, is that the LLVM type system uses "structural equivalence" instead of "name equivalence". Another place this surprises people is if you have two types in a high-level language that have the same structure (e.g. two different structs that have a single int field): these types will compile down into a single LLVM type and it will be impossible to tell what it came from.

Second, while LLVM does lose information, LLVM is not a fixed target: we continue to enhance and improve it in many different ways. In addition to adding new features (LLVM did not always support exceptions or debug info), we also extend the IR to capture important information for optimization (e.g. whether an argument is sign or zero extended, information about pointers aliasing, etc). Many of the enhancements are user-driven: people want LLVM to include some specific feature, so they go ahead and extend it.

Third, it is *possible and easy* to add language-specific optimizations, and you have a number of choices in how to do it. As one trivial example, it is easy to add language-specific optimization passes that "know" things about code compiled for a language. In the case of the C family, there is an optimization pass that "knows" about the standard C library functions. If you call "exit(0)" in main(), it knows that it is safe to optimize that into "return 0;" because C specifies what the 'exit' function does.

In addition to simple library knowledge, it is possible to embed a variety of other language-specific information into the LLVM IR. If you have a specific need and run into a wall, please bring the topic up on the llvmdev list. At the very worst, you can always treat LLVM as if it were a "dumb code generator" and implement the high-level optimizations you desire in your front-end, on the language-specific AST.

Tips and Tricks

There is a variety of useful tips and tricks that you come to know after working on/with LLVM that aren't obvious at first glance. Instead of letting everyone rediscover them, this section talks about some of these issues.

Implementing portable offsetof/sizeof One interesting thing that comes up, if you are trying to keep the code generated by your compiler "target independent", is that you often need to know the size of some LLVM type or the offset of some field in an llvm structure. For example, you might need to pass the size of a type into a function that allocates memory.

Unfortunately, this can vary widely across targets: for example the width of a pointer is trivially target-specific. However, there is a clever way to use the getelementptr instruction that allows you to compute this in a portable way.

Garbage Collected Stack Frames Some languages want to explicitly manage their stack frames, often so that they are garbage collected or to allow easy implementation of closures. There are often better ways to implement these features than explicit stack frames, but LLVM does support them, if you want. It requires your front-end to convert the code into Continuation Passing Style and the use of tail calls (which LLVM also supports).

2.15.2 Kaleidoscope: Implementing a Language with LLVM in Objective Caml

Kaleidoscope: Tutorial Introduction and the Lexer

Tutorial Introduction

Welcome to the "Implementing a language with LLVM" tutorial. This tutorial runs through the implementation of a simple language, showing how fun and easy it can be. This tutorial will get you up and started as well as help to build a framework you can extend to other languages. The code in this tutorial can also be used as a playground to hack on other LLVM specific things.

The goal of this tutorial is to progressively unveil our language, describing how it is built up over time. This will let us cover a fairly broad range of language design and LLVM-specific usage issues, showing and explaining the code for it all along the way, without overwhelming you with tons of details up front.

It is useful to point out ahead of time that this tutorial is really about teaching compiler techniques and LLVM specifically, *not* about teaching modern and sane software engineering principles. In practice, this means that we'll take a number of shortcuts to simplify the exposition. For example, the code leaks memory, uses global variables all over the place, doesn't use nice design patterns like visitors, etc... but it is very simple. If you dig in and use the code as a basis for future projects, fixing these deficiencies shouldn't be hard.

I've tried to put this tutorial together in a way that makes chapters easy to skip over if you are already familiar with or are uninterested in the various pieces. The structure of the tutorial is:

- Chapter #1: Introduction to the Kaleidoscope language, and the definition of its Lexer - This shows where we are going and the basic functionality that we want it to do. In order to make this tutorial maximally understandable and hackable, we choose to implement everything in Objective Caml instead of using lexer and parser generators. LLVM obviously works just fine with such tools, feel free to use one if you prefer.
- Chapter #2: Implementing a Parser and AST - With the lexer in place, we can talk about parsing techniques and basic AST construction. This tutorial describes recursive descent parsing and operator precedence parsing. Nothing in Chapters 1 or 2 is LLVM-specific, the code doesn't even link in LLVM at this point. :)
- Chapter #3: Code generation to LLVM IR - With the AST ready, we can show off how easy generation of LLVM IR really is.
- Chapter #4: Adding JIT and Optimizer Support - Because a lot of people are interested in using LLVM as a JIT, we'll dive right into it and show you the 3 lines it takes to add JIT support. LLVM is also useful in many other ways, but this is one simple and "sexy" way to shows off its power. :)
- Chapter #5: Extending the Language: Control Flow - With the language up and running, we show how to extend it with control flow operations (if/then/else and a 'for' loop). This gives us a chance to talk about simple SSA construction and control flow.
- Chapter #6: Extending the Language: User-defined Operators - This is a silly but fun chapter that talks about extending the language to let the user program define their own arbitrary unary and binary operators (with assignable precedence!). This lets us build a significant piece of the "language" as library routines.
- Chapter #7: Extending the Language: Mutable Variables - This chapter talks about adding user-defined local variables along with an assignment operator. The interesting part about this is how easy and trivial it is to construct SSA form in LLVM: no, LLVM does *not* require your front-end to construct SSA form!
- Chapter #8: Conclusion and other useful LLVM tidbits - This chapter wraps up the series by talking about potential ways to extend the language, but also includes a bunch of pointers to info about "special topics" like adding garbage collection support, exceptions, debugging, support for "spaghetti stacks", and a bunch of other tips and tricks.

By the end of the tutorial, we'll have written a bit less than 700 lines of non-comment, non-blank, lines of code. With this small amount of code, we'll have built up a very reasonable compiler for a non-trivial language including a hand-written lexer, parser, AST, as well as code generation support with a JIT compiler. While other systems may have interesting "hello world" tutorials, I think the breadth of this tutorial is a great testament to the strengths of LLVM and why you should consider it if you're interested in language or compiler design.

A note about this tutorial: we expect you to extend the language and play with it on your own. Take the code and go crazy hacking away at it, compilers don't need to be scary creatures - it can be a lot of fun to play with languages!

The Basic Language

This tutorial will be illustrated with a toy language that we'll call "Kaleidoscope" (derived from "meaning beautiful, form, and view"). Kaleidoscope is a procedural language that allows you to define functions, use conditionals, math, etc. Over the course of the tutorial, we'll extend Kaleidoscope to support the if/then/else construct, a for loop, user defined operators, JIT compilation with a simple command line interface, etc.

Because we want to keep things simple, the only datatype in Kaleidoscope is a 64-bit floating point type (aka 'float' in O'Caml parlance). As such, all values are implicitly double precision and the language doesn't require type declarations. This gives the language a very nice and simple syntax. For example, the following simple example computes Fibonacci numbers:

```
# Compute the x'th fibonacci number.
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2)

# This expression will compute the 40th number.
fib(40)
```

We also allow Kaleidoscope to call into standard library functions (the LLVM JIT makes this completely trivial). This means that you can use the 'extern' keyword to define a function before you use it (this is also useful for mutually recursive functions). For example:

```
extern sin(arg);
extern cos(arg);
extern atan2(arg1 arg2);

atan2(sin(.4), cos(42))
```

A more interesting example is included in Chapter 6 where we write a little Kaleidoscope application that displays a Mandelbrot Set at various levels of magnification.

Lets dive into the implementation of this language!

The Lexer

When it comes to implementing a language, the first thing needed is the ability to process a text file and recognize what it says. The traditional way to do this is to use a "lexer" (aka 'scanner') to break the input up into "tokens". Each token returned by the lexer includes a token code and potentially some metadata (e.g. the numeric value of a number). First, we define the possibilities:

```
(* The lexer returns these 'Kwd' if it is an unknown character, otherwise one of
 * these others for known things. *)
type token =
```

```
  (* commands *)
  | Def | Extern

  (* primary *)
  | Ident of string | Number of float

  (* unknown *)
  | Kwd of char
```

Each token returned by our lexer will be one of the token variant values. An unknown character like '+' will be returned as `Token.Kwd '+'`. If the curr token is an identifier, the value will be `Token.Ident s`. If the current token is a numeric literal (like 1.0), the value will be `Token.Number 1.0`.

The actual implementation of the lexer is a collection of functions driven by a function named `Lexer.lex`. The `Lexer.lex` function is called to return the next token from standard input. We will use Camlp4 to simplify the tokenization of the standard input. Its definition starts as:

```
(*===----------------------------------------------------------------------===
 * Lexer
 *===----------------------------------------------------------------------===*)

let rec lex = parser
  (* Skip any whitespace. *)
  | [< ' (' ' | '\n' | '\r' | '\t'); stream >] -> lex stream
```

`Lexer.lex` works by recursing over a `char Stream.t` to read characters one at a time from the standard input. It eats them as it recognizes them and stores them in in a `Token.token` variant. The first thing that it has to do is ignore whitespace between tokens. This is accomplished with the recursive call above.

The next thing `Lexer.lex` needs to do is recognize identifiers and specific keywords like "def". Kaleidoscope does this with a pattern match and a helper function.

```
  (* identifier: [a-zA-Z][a-zA-Z0-9] *)
  | [< ' ('A' .. 'Z' | 'a' .. 'z' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_ident buffer stream

...

and lex_ident buffer = parser
  | [< ' ('A' .. 'Z' | 'a' .. 'z' | '0' .. '9' as c); stream >] ->
      Buffer.add_char buffer c;
      lex_ident buffer stream
  | [< stream=lex >] ->
      match Buffer.contents buffer with
      | "def" -> [< 'Token.Def; stream >]
      | "extern" -> [< 'Token.Extern; stream >]
      | id -> [< 'Token.Ident id; stream >]
```

Numeric values are similar:

```
  (* number: [0-9.]+ *)
  | [< ' ('0' .. '9' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_number buffer stream

...
```

```
and lex_number buffer = parser
  | [< ' ('0' .. '9' | '.' as c); stream >] ->
       Buffer.add_char buffer c;
       lex_number buffer stream
  | [< stream=lex >] ->
       [< 'Token.Number (float_of_string (Buffer.contents buffer)); stream >]
```

This is all pretty straight-forward code for processing input. When reading a numeric value from input, we use the ocaml `float_of_string` function to convert it to a numeric value that we store in `Token.Number`. Note that this isn't doing sufficient error checking: it will raise `Failure` if the string "1.23.45.67". Feel free to extend it :). Next we handle comments:

```
  (* Comment until end of line. *)
  | [< ' ('#'); stream >] ->
       lex_comment stream

...

and lex_comment = parser
  | [< ' ('\n'); stream=lex >] -> stream
  | [< 'c; e=lex_comment >] -> e
  | [< >] -> [< >]
```

We handle comments by skipping to the end of the line and then return the next token. Finally, if the input doesn't match one of the above cases, it is either an operator character like '+' or the end of the file. These are handled with this code:

```
(* Otherwise, just return the character as its ascii value. *)
| [< 'c; stream >] ->
    [< 'Token.Kwd c; lex stream >]

(* end of stream. *)
| [< >] -> [< >]
```

With this, we have the complete lexer for the basic Kaleidoscope language (the full code listing for the Lexer is available in the next chapter of the tutorial). Next we'll build a simple parser that uses this to build an Abstract Syntax Tree. When we have that, we'll include a driver so that you can use the lexer and parser together.

Next: Implementing a Parser and AST

Kaleidoscope: Implementing a Parser and AST

Chapter 2 Introduction

Welcome to Chapter 2 of the "Implementing a language with LLVM in Objective Caml" tutorial. This chapter shows you how to use the lexer, built in Chapter 1, to build a full parser for our Kaleidoscope language. Once we have a parser, we'll define and build an Abstract Syntax Tree (AST).

The parser we will build uses a combination of Recursive Descent Parsing and Operator-Precedence Parsing to parse the Kaleidoscope language (the latter for binary expressions and the former for everything else). Before we get to parsing though, lets talk about the output of the parser: the Abstract Syntax Tree.

The Abstract Syntax Tree (AST)

The AST for a program captures its behavior in such a way that it is easy for later stages of the compiler (e.g. code generation) to interpret. We basically want one object for each construct in the language, and the AST should closely model the language. In Kaleidoscope, we have expressions, a prototype, and a function object. We'll start with expressions first:

```
(* expr - Base type for all expression nodes. *)
type expr =
  (* variant for numeric literals like "1.0". *)
  | Number of float
```

The code above shows the definition of the base ExprAST class and one subclass which we use for numeric literals. The important thing to note about this code is that the Number variant captures the numeric value of the literal as an instance variable. This allows later phases of the compiler to know what the stored numeric value is.

Right now we only create the AST, so there are no useful functions on them. It would be very easy to add a function to pretty print the code, for example. Here are the other expression AST node definitions that we'll use in the basic form of the Kaleidoscope language:

```
(* variant for referencing a variable, like "a". *)
| Variable of string

(* variant for a binary operator. *)
| Binary of char * expr * expr

(* variant for function calls. *)
| Call of string * expr array
```

This is all (intentionally) rather straight-forward: variables capture the variable name, binary operators capture their opcode (e.g. '+'), and calls capture a function name as well as a list of any argument expressions. One thing that is nice about our AST is that it captures the language features without talking about the syntax of the language. Note that there is no discussion about precedence of binary operators, lexical structure, etc.

For our basic language, these are all of the expression nodes we'll define. Because it doesn't have conditional control flow, it isn't Turing-complete; we'll fix that in a later installment. The two things we need next are a way to talk about the interface to a function, and a way to talk about functions themselves:

```
(* proto - This type represents the "prototype" for a function, which captures
 * its name, and its argument names (thus implicitly the number of arguments the
 * function takes). *)
type proto = Prototype of string * string array

(* func - This type represents a function definition itself. *)
type func = Function of proto * expr
```

In Kaleidoscope, functions are typed with just a count of their arguments. Since all values are double precision floating point, the type of each argument doesn't need to be stored anywhere. In a more aggressive and realistic language, the "expr" variants would probably have a type field.

With this scaffolding, we can now talk about parsing expressions and function bodies in Kaleidoscope.

Parser Basics

Now that we have an AST to build, we need to define the parser code to build it. The idea here is that we want to parse something like "x+y" (which is returned as three tokens by the lexer) into an AST that could be generated with calls like this:

```
let x = Variable "x" in
let y = Variable "y" in
let result = Binary ('+', x, y) in
...
```

The error handling routines make use of the builtin `Stream.Failure` and `Stream.Error`'s. ``Stream.Failure`` is raised when the parser is unable to find any matching token in the first position of a pattern. `Stream.Error` is raised when the first token matches, but the rest do not. The error recovery in our parser will not be the best and is not particular user-friendly, but it will be enough for our tutorial. These exceptions make it easier to handle errors in routines that have various return types.

With these basic types and exceptions, we can implement the first piece of our grammar: numeric literals.

Basic Expression Parsing

We start with numeric literals, because they are the simplest to process. For each production in our grammar, we'll define a function which parses that production. We call this class of expressions "primary" expressions, for reasons that will become more clear later in the tutorial. In order to parse an arbitrary primary expression, we need to determine what sort of expression it is. For numeric literals, we have:

```
(* primary
 *   ::= identifier
 *   ::= numberexpr
 *   ::= parenexpr *)
parse_primary = parser
  (* numberexpr ::= number *)
  | [< 'Token.Number n >] -> Ast.Number n
```

This routine is very simple: it expects to be called when the current token is a `Token.Number` token. It takes the current number value, creates a `Ast.Number` node, advances the lexer to the next token, and finally returns.

There are some interesting aspects to this. The most important one is that this routine eats all of the tokens that correspond to the production and returns the lexer buffer with the next token (which is not part of the grammar production) ready to go. This is a fairly standard way to go for recursive descent parsers. For a better example, the parenthesis operator is defined like this:

```
(* parenexpr ::= '(' expression ')' *)
| [< 'Token.Kwd '('; e=parse_expr; 'Token.Kwd ')' ?? "expected ')'" >] -> e
```

This function illustrates a number of interesting things about the parser:

1) It shows how we use the `Stream.Error` exception. When called, this function expects that the current token is a '(' token, but after parsing the subexpression, it is possible that there is no ')' waiting. For example, if the user types in "(4 x" instead of "(4)", the parser should emit an error. Because errors can occur, the parser needs a way to indicate

that they happened. In our parser, we use the camlp4 shortcut syntax `token ?? "parse error"`, where if the token before the `??` does not match, then `Stream.Error "parse error"` will be raised.

2) Another interesting aspect of this function is that it uses recursion by calling `Parser.parse_primary` (we will soon see that `Parser.parse_primary` can call `Parser.parse_primary`). This is powerful because it allows us to handle recursive grammars, and keeps each production very simple. Note that parentheses do not cause construction of AST nodes themselves. While we could do it this way, the most important role of parentheses are to guide the parser and provide grouping. Once the parser constructs the AST, parentheses are not needed.

The next simple production is for handling variable references and function calls:

```
(* identifierexpr
 *   ::= identifier
 *   ::= identifier '(' argumentexpr ')' *)
| [< 'Token.Ident id; stream >] ->
    let rec parse_args accumulator = parser
      | [< e=parse_expr; stream >] ->
          begin parser
            | [< 'Token.Kwd ','; e=parse_args (e :: accumulator) >] -> e
            | [< >] -> e :: accumulator
          end stream
      | [< >] -> accumulator
    in
    let rec parse_ident id = parser
      (* Call. *)
      | [< 'Token.Kwd '(';
           args=parse_args [];
           'Token.Kwd ')' ?? "expected ')'">] ->
          Ast.Call (id, Array.of_list (List.rev args))

      (* Simple variable ref. *)
      | [< >] -> Ast.Variable id
    in
    parse_ident id stream
```

This routine follows the same style as the other routines. (It expects to be called if the current token is a `Token.Ident` token). It also has recursion and error handling. One interesting aspect of this is that it uses *look-ahead* to determine if the current identifier is a stand alone variable reference or if it is a function call expression. It handles this by checking to see if the token after the identifier is a '(' token, constructing either a `Ast.Variable` or `Ast.Call` node as appropriate.

We finish up by raising an exception if we received a token we didn't expect:

```
| [< >] -> raise (Stream.Error "unknown token when expecting an expression.")
```

Now that basic expressions are handled, we need to handle binary expressions. They are a bit more complex.

Binary Expression Parsing

Binary expressions are significantly harder to parse because they are often ambiguous. For example, when given the string "x+y*z", the parser can choose to parse it as either "(x+y)*z" or "x+(y*z)". With common definitions from mathematics, we expect the later parse, because "*" (multiplication) has higher *precedence* than "+" (addition).

There are many ways to handle this, but an elegant and efficient way is to use Operator-Precedence Parsing. This parsing technique uses the precedence of binary operators to guide recursion. To start with, we need a table of precedences:

```
(* binop_precedence - This holds the precedence for each binary operator that is
 * defined *)
let binop_precedence:(char, int) Hashtbl.t = Hashtbl.create 10

(* precedence - Get the precedence of the pending binary operator token. *)
let precedence c = try Hashtbl.find binop_precedence c with Not_found -> -1

...

let main () =
  (* Install standard binary operators.
   * 1 is the lowest precedence. *)
  Hashtbl.add Parser.binop_precedence '<' 10;
  Hashtbl.add Parser.binop_precedence '+' 20;
  Hashtbl.add Parser.binop_precedence '-' 20;
  Hashtbl.add Parser.binop_precedence '*' 40;    (* highest. *)
  ...
```

For the basic form of Kaleidoscope, we will only support 4 binary operators (this can obviously be extended by you, our brave and intrepid reader). The `Parser.precedence` function returns the precedence for the current token, or -1 if the token is not a binary operator. Having a `Hashtbl.t` makes it easy to add new operators and makes it clear that the algorithm doesn't depend on the specific operators involved, but it would be easy enough to eliminate the `Hashtbl.t` and do the comparisons in the `Parser.precedence` function. (Or just use a fixed-size array).

With the helper above defined, we can now start parsing binary expressions. The basic idea of operator precedence parsing is to break down an expression with potentially ambiguous binary operators into pieces. Consider ,for example, the expression "a+b+(c+d)*e*f+g". Operator precedence parsing considers this as a stream of primary expressions separated by binary operators. As such, it will first parse the leading primary expression "a", then it will see the pairs [+, b] [+, (c+d)] [*, e] [*, f] and [+, g]. Note that because parentheses are primary expressions, the binary expression parser doesn't need to worry about nested subexpressions like (c+d) at all.

To start, an expression is a primary expression potentially followed by a sequence of [binop,primaryexpr] pairs:

```
(* expression
 *   ::= primary binoprhs *)
and parse_expr = parser
  | [< lhs=parse_primary; stream >] -> parse_bin_rhs 0 lhs stream
```

`Parser.parse_bin_rhs` is the function that parses the sequence of pairs for us. It takes a precedence and a pointer to an expression for the part that has been parsed so far. Note that "x" is a perfectly valid expression: As such, "binoprhs" is allowed to be empty, in which case it returns the expression that is passed into it. In our example above, the code passes the expression for "a" into `Parser.parse_bin_rhs` and the current token is "+".

The precedence value passed into `Parser.parse_bin_rhs` indicates the *minimal operator precedence* that the function is allowed to eat. For example, if the current pair stream is [+, x] and `Parser.parse_bin_rhs` is passed in a precedence of 40, it will not consume any tokens (because the precedence of '+' is only 20). With this in mind, `Parser.parse_bin_rhs` starts with:

```
(* binoprhs
 *   ::= ('+' primary)* *)
and parse_bin_rhs expr_prec lhs stream =
  match Stream.peek stream with
  (* If this is a binop, find its precedence. *)
  | Some (Token.Kwd c) when Hashtbl.mem binop_precedence c ->
      let token_prec = precedence c in

      (* If this is a binop that binds at least as tightly as the current binop,
       * consume it, otherwise we are done. *)
      if token_prec < expr_prec then lhs else begin
```

This code gets the precedence of the current token and checks to see if if is too low. Because we defined invalid tokens to have a precedence of -1, this check implicitly knows that the pair-stream ends when the token stream runs out of binary operators. If this check succeeds, we know that the token is a binary operator and that it will be included in this expression:

```
(* Eat the binop. *)
Stream.junk stream;

(* Parse the primary expression after the binary operator *)
let rhs = parse_primary stream in

(* Okay, we know this is a binop. *)
let rhs =
  match Stream.peek stream with
  | Some (Token.Kwd c2) ->
```

As such, this code eats (and remembers) the binary operator and then parses the primary expression that follows. This builds up the whole pair, the first of which is [+, b] for the running example.

Now that we parsed the left-hand side of an expression and one pair of the RHS sequence, we have to decide which way the expression associates. In particular, we could have "(a+b) binop unparsed" or "a + (b binop unparsed)". To determine this, we look ahead at "binop" to determine its precedence and compare it to BinOp's precedence (which is '+' in this case):

```
(* If BinOp binds less tightly with rhs than the operator after
 * rhs, let the pending operator take rhs as its lhs. *)
let next_prec = precedence c2 in
if token_prec < next_prec
```

If the precedence of the binop to the right of "RHS" is lower or equal to the precedence of our current operator, then we know that the parentheses associate as "(a+b) binop ...". In our example, the current operator is "+" and the next operator is "+", we know that they have the same precedence. In this case we'll create the AST node for "a+b", and then continue parsing:

```
    ... if body omitted ...
  in

  (* Merge lhs/rhs. *)
  let lhs = Ast.Binary (c, lhs, rhs) in
  parse_bin_rhs expr_prec lhs stream
end
```

In our example above, this will turn "a+b+" into "(a+b)" and execute the next iteration of the loop, with "+" as the current token. The code above will eat, remember, and parse "(c+d)" as the primary expression, which makes the current pair equal to [+, (c+d)]. It will then evaluate the 'if' conditional above with "*" as the binop to the right of the primary. In this case, the precedence of "*" is higher than the precedence of "+" so the if condition will be entered.

The critical question left here is "how can the if condition parse the right hand side in full"? In particular, to build the AST correctly for our example, it needs to get all of "(c+d)*e*f" as the RHS expression variable. The code to do this is surprisingly simple (code from the above two blocks duplicated for context):

```
    match Stream.peek stream with
    | Some (Token.Kwd c2) ->
        (* If BinOp binds less tightly with rhs than the operator after
         * rhs, let the pending operator take rhs as its lhs. *)
        if token_prec < precedence c2
        then parse_bin_rhs (token_prec + 1) rhs stream
```

```
        else rhs
    | _ -> rhs
  in

  (* Merge lhs/rhs. *)
  let lhs = Ast.Binary (c, lhs, rhs) in
  parse_bin_rhs expr_prec lhs stream
end
```

At this point, we know that the binary operator to the RHS of our primary has higher precedence than the binop we are currently parsing. As such, we know that any sequence of pairs whose operators are all higher precedence than "+" should be parsed together and returned as "RHS". To do this, we recursively invoke the `Parser.parse_bin_rhs` function specifying "token_prec+1" as the minimum precedence required for it to continue. In our example above, this will cause it to return the AST node for "(c+d)*e*f" as RHS, which is then set as the RHS of the '+' expression.

Finally, on the next iteration of the while loop, the "+g" piece is parsed and added to the AST. With this little bit of code (14 non-trivial lines), we correctly handle fully general binary expression parsing in a very elegant way. This was a whirlwind tour of this code, and it is somewhat subtle. I recommend running through it with a few tough examples to see how it works.

This wraps up handling of expressions. At this point, we can point the parser at an arbitrary token stream and build an expression from it, stopping at the first token that is not part of the expression. Next up we need to handle function definitions, etc.

Parsing the Rest

The next thing missing is handling of function prototypes. In Kaleidoscope, these are used both for 'extern' function declarations as well as function body definitions. The code to do this is straight-forward and not very interesting (once you've survived expressions):

```
(* prototype
 *   ::= id '(' id* ')' *)
let parse_prototype =
  let rec parse_args accumulator = parser
    | [< 'Token.Ident id; e=parse_args (id::accumulator) >] -> e
    | [< >] -> accumulator
  in

  parser
  | [< 'Token.Ident id;
       'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      (* success. *)
      Ast.Prototype (id, Array.of_list (List.rev args))

  | [< >] ->
      raise (Stream.Error "expected function name in prototype")
```

Given this, a function definition is very simple, just a prototype plus an expression to implement the body:

```
(* definition ::= 'def' prototype expression *)
let parse_definition = parser
  | [< 'Token.Def; p=parse_prototype; e=parse_expr >] ->
      Ast.Function (p, e)
```

In addition, we support 'extern' to declare functions like 'sin' and 'cos' as well as to support forward declaration of user functions. These 'extern's are just prototypes with no body:

```
(*  external ::= 'extern' prototype *)
let parse_extern = parser
  | [< 'Token.Extern; e=parse_prototype >] -> e
```

Finally, we'll also let the user type in arbitrary top-level expressions and evaluate them on the fly. We will handle this by defining anonymous nullary (zero argument) functions for them:

```
(* toplevelexpr ::= expression *)
let parse_toplevel = parser
  | [< e=parse_expr >] ->
      (* Make an anonymous proto. *)
      Ast.Function (Ast.Prototype ("", [||]), e)
```

Now that we have all the pieces, let's build a little driver that will let us actually *execute* this code we've built!

The Driver

The driver for this simply invokes all of the parsing pieces with a top-level dispatch loop. There isn't much interesting here, so I'll just include the top-level loop. See below for full code in the "Top-Level Parsing" section.

```
(* top ::= definition | external | expression | ';' *)
let rec main_loop stream =
  match Stream.peek stream with
  | None -> ()

  (* ignore top-level semicolons. *)
  | Some (Token.Kwd ';') ->
      Stream.junk stream;
      main_loop stream

  | Some token ->
      begin
        try match token with
        | Token.Def ->
            ignore(Parser.parse_definition stream);
            print_endline "parsed a function definition.";
        | Token.Extern ->
            ignore(Parser.parse_extern stream);
            print_endline "parsed an extern.";
        | _ ->
            (* Evaluate a top-level expression into an anonymous function. *)
            ignore(Parser.parse_toplevel stream);
            print_endline "parsed a top-level expr";
        with Stream.Error s ->
          (* Skip token for error recovery. *)
          Stream.junk stream;
          print_endline s;
      end;
      print_string "ready> "; flush stdout;
      main_loop stream
```

The most interesting part of this is that we ignore top-level semicolons. Why is this, you ask? The basic reason is that if you type "4 + 5" at the command line, the parser doesn't know whether that is the end of what you will type or not. For example, on the next line you could type "def foo..." in which case 4+5 is the end of a top-level expression. Alternatively you could type "* 6", which would continue the expression. Having top-level semicolons allows you to type "4+5;", and the parser will know you are done.

Conclusions

With just under 300 lines of commented code (240 lines of non-comment, non-blank code), we fully defined our minimal language, including a lexer, parser, and AST builder. With this done, the executable will validate Kaleidoscope code and tell us if it is grammatically invalid. For example, here is a sample interaction:

```
$ ./toy.byte
ready> def foo(x y) x+foo(y, 4.0);
Parsed a function definition.
ready> def foo(x y) x+y y;
Parsed a function definition.
Parsed a top-level expr
ready> def foo(x y) x+y );
Parsed a function definition.
Error: unknown token when expecting an expression
ready> extern sin(a);
ready> Parsed an extern
ready> ^D
$
```

There is a lot of room for extension here. You can define new AST nodes, extend the language in many ways, etc. In the next installment, we will describe how to generate LLVM Intermediate Representation (IR) from the AST.

Full Code Listing

Here is the complete code listing for this and the previous chapter. Note that it is fully self-contained: you don't need LLVM or any external libraries at all for this. (Besides the ocaml standard libraries, of course.) To build this, just compile with:

```
# Compile
ocamlbuild toy.byte
# Run
./toy.byte
```

Here is the code:

_tags:

```
<{lexer,parser}.ml>: use_camlp4, pp(camlp4of)
```

token.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer Tokens
 *===----------------------------------------------------------------------===*)

(* The lexer returns these 'Kwd' if it is an unknown character, otherwise one of
 * these others for known things. *)
type token =
  (* commands *)
  | Def | Extern

  (* primary *)
  | Ident of string | Number of float

  (* unknown *)
  | Kwd of char
```

lexer.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer
 *===----------------------------------------------------------------------===*)

let rec lex = parser
  (* Skip any whitespace. *)
  | [< ' (' ' | '\n' | '\r' | '\t'); stream >] -> lex stream

  (* identifier: [a-zA-Z][a-zA-Z0-9] *)
  | [< ' ('A' .. 'Z' | 'a' .. 'z' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_ident buffer stream

  (* number: [0-9.]+ *)
  | [< ' ('0' .. '9' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_number buffer stream

  (* Comment until end of line. *)
  | [< ' ('#'); stream >] ->
      lex_comment stream

  (* Otherwise, just return the character as its ascii value. *)
  | [< 'c; stream >] ->
      [< 'Token.Kwd c; lex stream >]

  (* end of stream. *)
  | [< >] -> [< >]

and lex_number buffer = parser
  | [< ' ('0' .. '9' | '.' as c); stream >] ->
      Buffer.add_char buffer c;
      lex_number buffer stream
  | [< stream=lex >] ->
      [< 'Token.Number (float_of_string (Buffer.contents buffer)); stream >]

and lex_ident buffer = parser
  | [< ' ('A' .. 'Z' | 'a' .. 'z' | '0' .. '9' as c); stream >] ->
      Buffer.add_char buffer c;
      lex_ident buffer stream
  | [< stream=lex >] ->
      match Buffer.contents buffer with
      | "def" -> [< 'Token.Def; stream >]
      | "extern" -> [< 'Token.Extern; stream >]
      | id -> [< 'Token.Ident id; stream >]

and lex_comment = parser
  | [< ' ('\n'); stream=lex >] -> stream
  | [< 'c; e=lex_comment >] -> e
  | [< >] -> [< >]
```

ast.ml:

```
(*===----------------------------------------------------------------------===
 * Abstract Syntax Tree (aka Parse Tree)
```

```
 *===----------------------------------------------------------------------===*)

(* expr - Base type for all expression nodes. *)
type expr =
  (* variant for numeric literals like "1.0". *)
  | Number of float

  (* variant for referencing a variable, like "a". *)
  | Variable of string

  (* variant for a binary operator. *)
  | Binary of char * expr * expr

  (* variant for function calls. *)
  | Call of string * expr array

(* proto - This type represents the "prototype" for a function, which captures
 * its name, and its argument names (thus implicitly the number of arguments the
 * function takes). *)
type proto = Prototype of string * string array

(* func - This type represents a function definition itself. *)
type func = Function of proto * expr
```

parser.ml:

```
(*===----------------------------------------------------------------------===
 * Parser
 *===----------------------------------------------------------------------===*)

(* binop_precedence - This holds the precedence for each binary operator that is
 * defined *)
let binop_precedence:(char, int) Hashtbl.t = Hashtbl.create 10

(* precedence - Get the precedence of the pending binary operator token. *)
let precedence c = try Hashtbl.find binop_precedence c with Not_found -> -1

(* primary
 *   ::= identifier
 *   ::= numberexpr
 *   ::= parenexpr *)
let rec parse_primary = parser
  (* numberexpr ::= number *)
  | [< 'Token.Number n >] -> Ast.Number n

  (* parenexpr ::= '(' expression ')' *)
  | [< 'Token.Kwd '('; e=parse_expr; 'Token.Kwd ')' ?? "expected ')'" >] -> e

  (* identifierexpr
   *   ::= identifier
   *   ::= identifier '(' argumentexpr ')' *)
  | [< 'Token.Ident id; stream >] ->
      let rec parse_args accumulator = parser
        | [< e=parse_expr; stream >] ->
            begin parser
              | [< 'Token.Kwd ','; e=parse_args (e :: accumulator) >] -> e
              | [< >] -> e :: accumulator
            end stream
```

```
        | [< >] -> accumulator
      in
      let rec parse_ident id = parser
        (* Call. *)
        | [< 'Token.Kwd '('; 
             args=parse_args [];
             'Token.Kwd ')' ?? "expected ')'">] ->
           Ast.Call (id, Array.of_list (List.rev args))

        (* Simple variable ref. *)
        | [< >] -> Ast.Variable id
      in
      parse_ident id stream

  | [< >] -> raise (Stream.Error "unknown token when expecting an expression.")

(* binoprhs
 *   ::= ('+' primary)* *)
and parse_bin_rhs expr_prec lhs stream =
  match Stream.peek stream with
  (* If this is a binop, find its precedence. *)
  | Some (Token.Kwd c) when Hashtbl.mem binop_precedence c ->
      let token_prec = precedence c in

      (* If this is a binop that binds at least as tightly as the current binop,
       * consume it, otherwise we are done. *)
      if token_prec < expr_prec then lhs else begin
        (* Eat the binop. *)
        Stream.junk stream;

        (* Parse the primary expression after the binary operator. *)
        let rhs = parse_primary stream in

        (* Okay, we know this is a binop. *)
        let rhs =
          match Stream.peek stream with
          | Some (Token.Kwd c2) ->
              (* If BinOp binds less tightly with rhs than the operator after
               * rhs, let the pending operator take rhs as its lhs. *)
              let next_prec = precedence c2 in
              if token_prec < next_prec
              then parse_bin_rhs (token_prec + 1) rhs stream
              else rhs
          | _ -> rhs
        in

        (* Merge lhs/rhs. *)
        let lhs = Ast.Binary (c, lhs, rhs) in
        parse_bin_rhs expr_prec lhs stream
      end
  | _ -> lhs

(* expression
 *   ::= primary binoprhs *)
and parse_expr = parser
  | [< lhs=parse_primary; stream >] -> parse_bin_rhs 0 lhs stream

(* prototype
```

```
 *   ::= id '(' id* ')' *)
let parse_prototype =
  let rec parse_args accumulator = parser
    | [< 'Token.Ident id; e=parse_args (id::accumulator) >] -> e
    | [< >] -> accumulator
  in

  parser
  | [< 'Token.Ident id;
       'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      (* success. *)
      Ast.Prototype (id, Array.of_list (List.rev args))

  | [< >] ->
      raise (Stream.Error "expected function name in prototype")

(* definition ::= 'def' prototype expression *)
let parse_definition = parser
  | [< 'Token.Def; p=parse_prototype; e=parse_expr >] ->
      Ast.Function (p, e)

(* toplevelexpr ::= expression *)
let parse_toplevel = parser
  | [< e=parse_expr >] ->
      (* Make an anonymous proto. *)
      Ast.Function (Ast.Prototype ("", [||]), e)

(*  external ::= 'extern' prototype *)
let parse_extern = parser
  | [< 'Token.Extern; e=parse_prototype >] -> e
```

toplevel.ml:

```
(*===----------------------------------------------------------------------===
 * Top-Level parsing and JIT Driver
 *===----------------------------------------------------------------------===*)

(* top ::= definition | external | expression | ';' *)
let rec main_loop stream =
  match Stream.peek stream with
  | None -> ()

  (* ignore top-level semicolons. *)
  | Some (Token.Kwd ';') ->
      Stream.junk stream;
      main_loop stream

  | Some token ->
      begin
        try match token with
        | Token.Def ->
            ignore(Parser.parse_definition stream);
            print_endline "parsed a function definition.";
        | Token.Extern ->
            ignore(Parser.parse_extern stream);
            print_endline "parsed an extern.";
```

```
          | _ ->
              (* Evaluate a top-level expression into an anonymous function. *)
              ignore(Parser.parse_toplevel stream);
              print_endline "parsed a top-level expr";
          with Stream.Error s ->
            (* Skip token for error recovery. *)
            Stream.junk stream;
            print_endline s;
        end;
        print_string "ready> "; flush stdout;
        main_loop stream
```

toy.ml:

```
(*===----------------------------------------------------------------------===
 * Main driver code.
 *===----------------------------------------------------------------------===*)

let main () =
  (* Install standard binary operators.
   * 1 is the lowest precedence. *)
  Hashtbl.add Parser.binop_precedence '<' 10;
  Hashtbl.add Parser.binop_precedence '+' 20;
  Hashtbl.add Parser.binop_precedence '-' 20;
  Hashtbl.add Parser.binop_precedence '*' 40;    (* highest. *)

  (* Prime the first token. *)
  print_string "ready> "; flush stdout;
  let stream = Lexer.lex (Stream.of_channel stdin) in

  (* Run the main "interpreter loop" now. *)
  Toplevel.main_loop stream;
;;

main ()
```

Next: Implementing Code Generation to LLVM IR

Kaleidoscope: Code generation to LLVM IR

Chapter 3 Introduction

Welcome to Chapter 3 of the "Implementing a language with LLVM" tutorial. This chapter shows you how to transform the Abstract Syntax Tree, built in Chapter 2, into LLVM IR. This will teach you a little bit about how LLVM does things, as well as demonstrate how easy it is to use. It's much more work to build a lexer and parser than it is to generate LLVM IR code. :)

Please note: the code in this chapter and later require LLVM 2.3 or LLVM SVN to work. LLVM 2.2 and before will not work with it.

Code Generation Setup

In order to generate LLVM IR, we want some simple setup to get started. First we define virtual code generation (codegen) methods in each AST class:

```
let rec codegen_expr = function
  | Ast.Number n -> ...
  | Ast.Variable name -> ...
```

The `Codegen.codegen_expr` function says to emit IR for that AST node along with all the things it depends on, and they all return an LLVM Value object. "Value" is the class used to represent a "Static Single Assignment (SSA) register" or "SSA value" in LLVM. The most distinct aspect of SSA values is that their value is computed as the related instruction executes, and it does not get a new value until (and if) the instruction re-executes. In other words, there is no way to "change" an SSA value. For more information, please read up on Static Single Assignment - the concepts are really quite natural once you grok them.

The second thing we want is an "Error" exception like we used for the parser, which will be used to report errors found during code generation (for example, use of an undeclared parameter):

```
exception Error of string

let context = global_context ()
let the_module = create_module context "my cool jit"
let builder = builder context
let named_values:(string, llvalue) Hashtbl.t = Hashtbl.create 10
let double_type = double_type context
```

The static variables will be used during code generation. `Codgen.the_module` is the LLVM construct that contains all of the functions and global variables in a chunk of code. In many ways, it is the top-level structure that the LLVM IR uses to contain code.

The `Codegen.builder` object is a helper object that makes it easy to generate LLVM instructions. Instances of the `'IRBuilder <http://llvm.org/doxygen/IRBuilder_8h-source.html>'_` class keep track of the current place to insert instructions and has methods to create new instructions.

The `Codegen.named_values` map keeps track of which values are defined in the current scope and what their LLVM representation is. (In other words, it is a symbol table for the code). In this form of Kaleidoscope, the only things that can be referenced are function parameters. As such, function parameters will be in this map when generating code for their function body.

With these basics in place, we can start talking about how to generate code for each expression. Note that this assumes that the `Codgen.builder` has been set up to generate code *into* something. For now, we'll assume that this has already been done, and we'll just use it to emit code.

Expression Code Generation

Generating LLVM code for expression nodes is very straightforward: less than 30 lines of commented code for all four of our expression nodes. First we'll do numeric literals:

```
| Ast.Number n -> const_float double_type n
```

In the LLVM IR, numeric constants are represented with the `ConstantFP` class, which holds the numeric value in an `APFloat` internally (`APFloat` has the capability of holding floating point constants of Arbitrary Precision).

This code basically just creates and returns a `ConstantFP`. Note that in the LLVM IR that constants are all uniqued together and shared. For this reason, the API uses "the foo::get(..)" idiom instead of "new foo(..)" or "foo::Create(..)".

```
| Ast.Variable name ->
    (try Hashtbl.find named_values name with
      | Not_found -> raise (Error "unknown variable name"))
```

References to variables are also quite simple using LLVM. In the simple version of Kaleidoscope, we assume that the variable has already been emitted somewhere and its value is available. In practice, the only values that can be in the `Codegen.named_values` map are function arguments. This code simply checks to see that the specified name is in the map (if not, an unknown variable is being referenced) and returns the value for it. In future chapters, we'll add support for loop induction variables in the symbol table, and for local variables.

```
| Ast.Binary (op, lhs, rhs) ->
    let lhs_val = codegen_expr lhs in
    let rhs_val = codegen_expr rhs in
    begin
      match op with
      | '+' -> build_fadd lhs_val rhs_val "addtmp" builder
      | '-' -> build_fsub lhs_val rhs_val "subtmp" builder
      | '*' -> build_fmul lhs_val rhs_val "multmp" builder
      | '<' ->
          (* Convert bool 0/1 to double 0.0 or 1.0 *)
          let i = build_fcmp Fcmp.Ult lhs_val rhs_val "cmptmp" builder in
          build_uitofp i double_type "booltmp" builder
      | _ -> raise (Error "invalid binary operator")
    end
```

Binary operators start to get more interesting. The basic idea here is that we recursively emit code for the left-hand side of the expression, then the right-hand side, then we compute the result of the binary expression. In this code, we do a simple switch on the opcode to create the right LLVM instruction.

In the example above, the LLVM builder class is starting to show its value. IRBuilder knows where to insert the newly created instruction, all you have to do is specify what instruction to create (e.g. with `Llvm.create_add`), which operands to use (`lhs` and `rhs` here) and optionally provide a name for the generated instruction.

One nice thing about LLVM is that the name is just a hint. For instance, if the code above emits multiple "addtmp" variables, LLVM will automatically provide each one with an increasing, unique numeric suffix. Local value names for instructions are purely optional, but it makes it much easier to read the IR dumps.

LLVM instructions are constrained by strict rules: for example, the Left and Right operators of an add instruction must have the same type, and the result type of the add must match the operand types. Because all values in Kaleidoscope are doubles, this makes for very simple code for add, sub and mul.

On the other hand, LLVM specifies that the fcmp instruction always returns an 'i1' value (a one bit integer). The problem with this is that Kaleidoscope wants the value to be a 0.0 or 1.0 value. In order to get these semantics, we combine the fcmp instruction with a uitofp instruction. This instruction converts its input integer into a floating point value by treating the input as an unsigned value. In contrast, if we used the sitofp instruction, the Kaleidoscope '<' operator would return 0.0 and -1.0, depending on the input value.

```
| Ast.Call (callee, args) ->
    (* Look up the name in the module table. *)
    let callee =
      match lookup_function callee the_module with
      | Some callee -> callee
      | None -> raise (Error "unknown function referenced")
    in
    let params = params callee in
```

```
    (* If argument mismatch error. *)
    if Array.length params == Array.length args then () else
      raise (Error "incorrect # arguments passed");
    let args = Array.map codegen_expr args in
    build_call callee args "calltmp" builder
```

Code generation for function calls is quite straightforward with LLVM. The code above initially does a function name lookup in the LLVM Module's symbol table. Recall that the LLVM Module is the container that holds all of the functions we are JIT'ing. By giving each function the same name as what the user specifies, we can use the LLVM symbol table to resolve function names for us.

Once we have the function to call, we recursively codegen each argument that is to be passed in, and create an LLVM call instruction. Note that LLVM uses the native C calling conventions by default, allowing these calls to also call into standard library functions like "sin" and "cos", with no additional effort.

This wraps up our handling of the four basic expressions that we have so far in Kaleidoscope. Feel free to go in and add some more. For example, by browsing the LLVM language reference you'll find several other interesting instructions that are really easy to plug into our basic framework.

Function Code Generation

Code generation for prototypes and functions must handle a number of details, which make their code less beautiful than expression code generation, but allows us to illustrate some important points. First, lets talk about code generation for prototypes: they are used both for function bodies and external function declarations. The code starts with:

```
let codegen_proto = function
  | Ast.Prototype (name, args) ->
      (* Make the function type: double(double,double) etc. *)
      let doubles = Array.make (Array.length args) double_type in
      let ft = function_type double_type doubles in
      let f =
        match lookup_function name the_module with
```

This code packs a lot of power into a few lines. Note first that this function returns a "Function*" instead of a "Value*" (although at the moment they both are modeled by `llvalue` in ocaml). Because a "prototype" really talks about the external interface for a function (not the value computed by an expression), it makes sense for it to return the LLVM Function it corresponds to when codegen'd.

The call to `Llvm.function_type` creates the `Llvm.llvalue` that should be used for a given Prototype. Since all function arguments in Kaleidoscope are of type double, the first line creates a vector of "N" LLVM double types. It then uses the `Llvm.function_type` method to create a function type that takes "N" doubles as arguments, returns one double as a result, and that is not vararg (that uses the function `Llvm.var_arg_function_type`). Note that Types in LLVM are uniqued just like `Constant`'s are, so you don't "new" a type, you "get" it.

The final line above checks if the function has already been defined in `Codegen.the_module`. If not, we will create it.

```
| None -> declare_function name ft the_module
```

This indicates the type and name to use, as well as which module to insert into. By default we assume a function has `Llvm.Linkage.ExternalLinkage`. "external linkage" means that the function may be defined outside the current module and/or that it is callable by functions outside the module. The "`name`" passed in is the name the user specified: this name is registered in "`Codegen.the_module`"s symbol table, which is used by the function call code above.

In Kaleidoscope, I choose to allow redefinitions of functions in two cases: first, we want to allow 'extern'ing a function more than once, as long as the prototypes for the externs match (since all arguments have the same type, we just have

to check that the number of arguments match). Second, we want to allow 'extern'ing a function and then defining a body for it. This is useful when defining mutually recursive functions.

```
  (* If 'f' conflicted, there was already something named 'name'. If it
   * has a body, don't allow redefinition or reextern. *)
  | Some f ->
      (* If 'f' already has a body, reject this. *)
      if Array.length (basic_blocks f) == 0 then () else
        raise (Error "redefinition of function");

      (* If 'f' took a different number of arguments, reject. *)
      if Array.length (params f) == Array.length args then () else
        raise (Error "redefinition of function with different # args");
      f
in
```

In order to verify the logic above, we first check to see if the pre-existing function is "empty". In this case, empty means that it has no basic blocks in it, which means it has no body. If it has no body, it is a forward declaration. Since we don't allow anything after a full definition of the function, the code rejects this case. If the previous reference to a function was an 'extern', we simply verify that the number of arguments for that definition and this one match up. If not, we emit an error.

```
(* Set names for all arguments. *)
Array.iteri (fun i a ->
  let n = args.(i) in
  set_value_name n a;
  Hashtbl.add named_values n a;
) (params f);
f
```

The last bit of code for prototypes loops over all of the arguments in the function, setting the name of the LLVM Argument objects to match, and registering the arguments in the `Codegen.named_values` map for future use by the `Ast.Variable` variant. Once this is set up, it returns the Function object to the caller. Note that we don't check for conflicting argument names here (e.g. "extern foo(a b a)"). Doing so would be very straight-forward with the mechanics we have already used above.

```
let codegen_func = function
  | Ast.Function (proto, body) ->
      Hashtbl.clear named_values;
      let the_function = codegen_proto proto in
```

Code generation for function definitions starts out simply enough: we just codegen the prototype (Proto) and verify that it is ok. We then clear out the `Codegen.named_values` map to make sure that there isn't anything in it from the last function we compiled. Code generation of the prototype ensures that there is an LLVM Function object that is ready to go for us.

```
(* Create a new basic block to start insertion into. *)
let bb = append_block context "entry" the_function in
position_at_end bb builder;

try
  let ret_val = codegen_expr body in
```

Now we get to the point where the `Codegen.builder` is set up. The first line creates a new basic block (named "entry"), which is inserted into `the_function`. The second line then tells the builder that new instructions should be inserted into the end of the new basic block. Basic blocks in LLVM are an important part of functions that define the Control Flow Graph. Since we don't have any control flow, our functions will only contain one block at this point. We'll fix this in Chapter 5 :).

```
let ret_val = codegen_expr body in

(* Finish off the function. *)
let _ = build_ret ret_val builder in

(* Validate the generated code, checking for consistency. *)
Llvm_analysis.assert_valid_function the_function;

the_function
```

Once the insertion point is set up, we call the `Codegen.codegen_func` method for the root expression of the function. If no error happens, this emits code to compute the expression into the entry block and returns the value that was computed. Assuming no error, we then create an LLVM ret instruction, which completes the function. Once the function is built, we call `Llvm_analysis.assert_valid_function`, which is provided by LLVM. This function does a variety of consistency checks on the generated code, to determine if our compiler is doing everything right. Using this is important: it can catch a lot of bugs. Once the function is finished and validated, we return it.

```
with e ->
  delete_function the_function;
  raise e
```

The only piece left here is handling of the error case. For simplicity, we handle this by merely deleting the function we produced with the `Llvm.delete_function` method. This allows the user to redefine a function that they incorrectly typed in before: if we didn't delete it, it would live in the symbol table, with a body, preventing future redefinition.

This code does have a bug, though. Since the `Codegen.codegen_proto` can return a previously defined forward declaration, our code can actually delete a forward declaration. There are a number of ways to fix this bug, see what you can come up with! Here is a testcase:

```
extern foo(a b);     # ok, defines foo.
def foo(a b) c;      # error, 'c' is invalid.
def bar() foo(1, 2); # error, unknown function "foo"
```

Driver Changes and Closing Thoughts

For now, code generation to LLVM doesn't really get us much, except that we can look at the pretty IR calls. The sample code inserts calls to Codegen into the "`Toplevel.main_loop`", and then dumps out the LLVM IR. This gives a nice way to look at the LLVM IR for simple functions. For example:

```
ready> 4+5;
Read top-level expression:
define double @""() {
entry:
        %addtmp = fadd double 4.000000e+00, 5.000000e+00
        ret double %addtmp
}
```

Note how the parser turns the top-level expression into anonymous functions for us. This will be handy when we add JIT support in the next chapter. Also note that the code is very literally transcribed, no optimizations are being performed. We will add optimizations explicitly in the next chapter.

```
ready> def foo(a b) a*a + 2*a*b + b*b;
Read function definition:
define double @foo(double %a, double %b) {
entry:
        %multmp = fmul double %a, %a
```

```
        %multmp1 = fmul double 2.000000e+00, %a
        %multmp2 = fmul double %multmp1, %b
        %addtmp = fadd double %multmp, %multmp2
        %multmp3 = fmul double %b, %b
        %addtmp4 = fadd double %addtmp, %multmp3
        ret double %addtmp4
}
```

This shows some simple arithmetic. Notice the striking similarity to the LLVM builder calls that we use to create the instructions.

```
ready> def bar(a) foo(a, 4.0) + bar(31337);
Read function definition:
define double @bar(double %a) {
entry:
        %calltmp = call double @foo(double %a, double 4.000000e+00)
        %calltmp1 = call double @bar(double 3.133700e+04)
        %addtmp = fadd double %calltmp, %calltmp1
        ret double %addtmp
}
```

This shows some function calls. Note that this function will take a long time to execute if you call it. In the future we'll add conditional control flow to actually make recursion useful :).

```
ready> extern cos(x);
Read extern:
declare double @cos(double)

ready> cos(1.234);
Read top-level expression:
define double @""() {
entry:
        %calltmp = call double @cos(double 1.234000e+00)
        ret double %calltmp
}
```

This shows an extern for the libm "cos" function, and a call to it.

```
ready> ^D
; ModuleID = 'my cool jit'

define double @""() {
entry:
        %addtmp = fadd double 4.000000e+00, 5.000000e+00
        ret double %addtmp
}

define double @foo(double %a, double %b) {
entry:
        %multmp = fmul double %a, %a
        %multmp1 = fmul double 2.000000e+00, %a
        %multmp2 = fmul double %multmp1, %b
        %addtmp = fadd double %multmp, %multmp2
        %multmp3 = fmul double %b, %b
        %addtmp4 = fadd double %addtmp, %multmp3
        ret double %addtmp4
}

define double @bar(double %a) {
```

```
entry:
        %calltmp = call double @foo(double %a, double 4.000000e+00)
        %calltmp1 = call double @bar(double 3.133700e+04)
        %addtmp = fadd double %calltmp, %calltmp1
        ret double %addtmp
}

declare double @cos(double)

define double @""() {
entry:
        %calltmp = call double @cos(double 1.234000e+00)
        ret double %calltmp
}
```

When you quit the current demo, it dumps out the IR for the entire module generated. Here you can see the big picture with all the functions referencing each other.

This wraps up the third chapter of the Kaleidoscope tutorial. Up next, we'll describe how to add JIT codegen and optimizer support to this so we can actually start running code!

Full Code Listing

Here is the complete code listing for our running example, enhanced with the LLVM code generator. Because this uses the LLVM libraries, we need to link them in. To do this, we use the llvm-config tool to inform our makefile/command line about which options to use:

```
# Compile
ocamlbuild toy.byte
# Run
./toy.byte
```

Here is the code:

_tags:

```
<{lexer,parser}.ml>: use_camlp4, pp(camlp4of)
<*.{byte,native}>: g++, use_llvm, use_llvm_analysis
```

myocamlbuild.ml:

```
open Ocamlbuild_plugin;;

ocaml_lib ~extern:true "llvm";;
ocaml_lib ~extern:true "llvm_analysis";;

flag ["link"; "ocaml"; "g++"] (S[A"-cc"; A"g++"]);;
```

token.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer Tokens
 *===----------------------------------------------------------------------===*)

(* The lexer returns these 'Kwd' if it is an unknown character, otherwise one of
 * these others for known things. *)
type token =
  (* commands *)
```

```
  | Def | Extern

  (* primary *)
  | Ident of string | Number of float

  (* unknown *)
  | Kwd of char
```

lexer.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer
 *===----------------------------------------------------------------------===*)

let rec lex = parser
  (* Skip any whitespace. *)
  | [< ' (' ' | '\n' | '\r' | '\t'); stream >] -> lex stream

  (* identifier: [a-zA-Z][a-zA-Z0-9] *)
  | [< ' ('A' .. 'Z' | 'a' .. 'z' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_ident buffer stream

  (* number: [0-9.]+ *)
  | [< ' ('0' .. '9' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_number buffer stream

  (* Comment until end of line. *)
  | [< ' ('#'); stream >] ->
      lex_comment stream

  (* Otherwise, just return the character as its ascii value. *)
  | [< 'c; stream >] ->
      [< 'Token.Kwd c; lex stream >]

  (* end of stream. *)
  | [< >] -> [< >]

and lex_number buffer = parser
  | [< ' ('0' .. '9' | '.' as c); stream >] ->
      Buffer.add_char buffer c;
      lex_number buffer stream
  | [< stream=lex >] ->
      [< 'Token.Number (float_of_string (Buffer.contents buffer)); stream >]

and lex_ident buffer = parser
  | [< ' ('A' .. 'Z' | 'a' .. 'z' | '0' .. '9' as c); stream >] ->
      Buffer.add_char buffer c;
      lex_ident buffer stream
  | [< stream=lex >] ->
      match Buffer.contents buffer with
      | "def" -> [< 'Token.Def; stream >]
      | "extern" -> [< 'Token.Extern; stream >]
      | id -> [< 'Token.Ident id; stream >]
```

```
and lex_comment = parser
  | [< ' ('\n'); stream=lex >] -> stream
  | [< 'c; e=lex_comment >] -> e
  | [< >] -> [< >]
```

ast.ml:

```
(*===----------------------------------------------------------------------===
 * Abstract Syntax Tree (aka Parse Tree)
 *===----------------------------------------------------------------------===*)

(* expr - Base type for all expression nodes. *)
type expr =
  (* variant for numeric literals like "1.0". *)
  | Number of float

  (* variant for referencing a variable, like "a". *)
  | Variable of string

  (* variant for a binary operator. *)
  | Binary of char * expr * expr

  (* variant for function calls. *)
  | Call of string * expr array

(* proto - This type represents the "prototype" for a function, which captures
 * its name, and its argument names (thus implicitly the number of arguments the
 * function takes). *)
type proto = Prototype of string * string array

(* func - This type represents a function definition itself. *)
type func = Function of proto * expr
```

parser.ml:

```
(*===----------------------------------------------------------------------===
 * Parser
 *===----------------------------------------------------------------------===*)

(* binop_precedence - This holds the precedence for each binary operator that is
 * defined *)
let binop_precedence:(char, int) Hashtbl.t = Hashtbl.create 10

(* precedence - Get the precedence of the pending binary operator token. *)
let precedence c = try Hashtbl.find binop_precedence c with Not_found -> -1

(* primary
 *   ::= identifier
 *   ::= numberexpr
 *   ::= parenexpr *)
let rec parse_primary = parser
  (* numberexpr ::= number *)
  | [< 'Token.Number n >] -> Ast.Number n

  (* parenexpr ::= '(' expression ')' *)
  | [< 'Token.Kwd '('; e=parse_expr; 'Token.Kwd ')' ?? "expected ')'" >] -> e

  (* identifierexpr
   *   ::= identifier
```

```
     *   ::= identifier '(' argumentexpr ')' *)
    | [< 'Token.Ident id; stream >] ->
        let rec parse_args accumulator = parser
          | [< e=parse_expr; stream >] ->
              begin parser
                | [< 'Token.Kwd ','; e=parse_args (e :: accumulator) >] -> e
                | [< >] -> e :: accumulator
              end stream
          | [< >] -> accumulator
        in
        let rec parse_ident id = parser
          (* Call. *)
          | [< 'Token.Kwd '(';
               args=parse_args [];
               'Token.Kwd ')' ?? "expected ')'">] ->
              Ast.Call (id, Array.of_list (List.rev args))

          (* Simple variable ref. *)
          | [< >] -> Ast.Variable id
        in
        parse_ident id stream

    | [< >] -> raise (Stream.Error "unknown token when expecting an expression.")

  (* binoprhs
   *   ::= ('+' primary)* *)
  and parse_bin_rhs expr_prec lhs stream =
    match Stream.peek stream with
    (* If this is a binop, find its precedence. *)
    | Some (Token.Kwd c) when Hashtbl.mem binop_precedence c ->
        let token_prec = precedence c in

        (* If this is a binop that binds at least as tightly as the current binop,
         * consume it, otherwise we are done. *)
        if token_prec < expr_prec then lhs else begin
          (* Eat the binop. *)
          Stream.junk stream;

          (* Parse the primary expression after the binary operator. *)
          let rhs = parse_primary stream in

          (* Okay, we know this is a binop. *)
          let rhs =
            match Stream.peek stream with
            | Some (Token.Kwd c2) ->
                (* If BinOp binds less tightly with rhs than the operator after
                 * rhs, let the pending operator take rhs as its lhs. *)
                let next_prec = precedence c2 in
                if token_prec < next_prec
                then parse_bin_rhs (token_prec + 1) rhs stream
                else rhs
            | _ -> rhs
          in

          (* Merge lhs/rhs. *)
          let lhs = Ast.Binary (c, lhs, rhs) in
          parse_bin_rhs expr_prec lhs stream
        end
```

```
    | _ -> lhs

(* expression
 *   ::= primary binoprhs *)
and parse_expr = parser
  | [< lhs=parse_primary; stream >] -> parse_bin_rhs 0 lhs stream

(* prototype
 *   ::= id '(' id* ')' *)
let parse_prototype =
  let rec parse_args accumulator = parser
    | [< 'Token.Ident id; e=parse_args (id::accumulator) >] -> e
    | [< >] -> accumulator
  in

  parser
  | [< 'Token.Ident id;
       'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      (* success. *)
      Ast.Prototype (id, Array.of_list (List.rev args))

  | [< >] ->
      raise (Stream.Error "expected function name in prototype")

(* definition ::= 'def' prototype expression *)
let parse_definition = parser
  | [< 'Token.Def; p=parse_prototype; e=parse_expr >] ->
      Ast.Function (p, e)

(* toplevelexpr ::= expression *)
let parse_toplevel = parser
  | [< e=parse_expr >] ->
      (* Make an anonymous proto. *)
      Ast.Function (Ast.Prototype ("", [||]), e)

(*  external ::= 'extern' prototype *)
let parse_extern = parser
  | [< 'Token.Extern; e=parse_prototype >] -> e
```

codegen.ml:

```
(*===----------------------------------------------------------------------===
 * Code Generation
 *===----------------------------------------------------------------------===*)

open Llvm

exception Error of string

let context = global_context ()
let the_module = create_module context "my cool jit"
let builder = builder context
let named_values:(string, llvalue) Hashtbl.t = Hashtbl.create 10
let double_type = double_type context

let rec codegen_expr = function
```

```
  | Ast.Number n -> const_float double_type n
  | Ast.Variable name ->
      (try Hashtbl.find named_values name with
        | Not_found -> raise (Error "unknown variable name"))
  | Ast.Binary (op, lhs, rhs) ->
      let lhs_val = codegen_expr lhs in
      let rhs_val = codegen_expr rhs in
      begin
        match op with
        | '+' -> build_add lhs_val rhs_val "addtmp" builder
        | '-' -> build_sub lhs_val rhs_val "subtmp" builder
        | '*' -> build_mul lhs_val rhs_val "multmp" builder
        | '<' ->
            (* Convert bool 0/1 to double 0.0 or 1.0 *)
            let i = build_fcmp Fcmp.Ult lhs_val rhs_val "cmptmp" builder in
            build_uitofp i double_type "booltmp" builder
        | _ -> raise (Error "invalid binary operator")
      end
  | Ast.Call (callee, args) ->
      (* Look up the name in the module table. *)
      let callee =
        match lookup_function callee the_module with
        | Some callee -> callee
        | None -> raise (Error "unknown function referenced")
      in
      let params = params callee in

      (* If argument mismatch error. *)
      if Array.length params == Array.length args then () else
        raise (Error "incorrect # arguments passed");
      let args = Array.map codegen_expr args in
      build_call callee args "calltmp" builder

let codegen_proto = function
  | Ast.Prototype (name, args) ->
      (* Make the function type: double(double,double) etc. *)
      let doubles = Array.make (Array.length args) double_type in
      let ft = function_type double_type doubles in
      let f =
        match lookup_function name the_module with
        | None -> declare_function name ft the_module

        (* If 'f' conflicted, there was already something named 'name'. If it
         * has a body, don't allow redefinition or reextern. *)
        | Some f ->
            (* If 'f' already has a body, reject this. *)
            if block_begin f <> At_end f then
              raise (Error "redefinition of function");

            (* If 'f' took a different number of arguments, reject. *)
            if element_type (type_of f) <> ft then
              raise (Error "redefinition of function with different # args");
            f
      in

      (* Set names for all arguments. *)
      Array.iteri (fun i a ->
        let n = args.(i) in
```

```
        set_value_name n a;
        Hashtbl.add named_values n a;
      ) (params f);
      f

let codegen_func = function
  | Ast.Function (proto, body) ->
      Hashtbl.clear named_values;
      let the_function = codegen_proto proto in

      (* Create a new basic block to start insertion into. *)
      let bb = append_block context "entry" the_function in
      position_at_end bb builder;

      try
        let ret_val = codegen_expr body in

        (* Finish off the function. *)
        let _ = build_ret ret_val builder in

        (* Validate the generated code, checking for consistency. *)
        Llvm_analysis.assert_valid_function the_function;

        the_function
      with e ->
        delete_function the_function;
        raise e
```

toplevel.ml:

```
(*===----------------------------------------------------------------------===
 * Top-Level parsing and JIT Driver
 *===----------------------------------------------------------------------===*)

open Llvm

(* top ::= definition | external | expression | ';' *)
let rec main_loop stream =
  match Stream.peek stream with
  | None -> ()

  (* ignore top-level semicolons. *)
  | Some (Token.Kwd ';') ->
      Stream.junk stream;
      main_loop stream

  | Some token ->
      begin
        try match token with
        | Token.Def ->
            let e = Parser.parse_definition stream in
            print_endline "parsed a function definition.";
            dump_value (Codegen.codegen_func e);
        | Token.Extern ->
            let e = Parser.parse_extern stream in
            print_endline "parsed an extern.";
            dump_value (Codegen.codegen_proto e);
        | _ ->
            (* Evaluate a top-level expression into an anonymous function. *)
```

```
            let e = Parser.parse_toplevel stream in
            print_endline "parsed a top-level expr";
            dump_value (Codegen.codegen_func e);
          with Stream.Error s | Codegen.Error s ->
            (* Skip token for error recovery. *)
            Stream.junk stream;
            print_endline s;
        end;
        print_string "ready> "; flush stdout;
        main_loop stream
```

toy.ml:

```
(*===----------------------------------------------------------------------===
 * Main driver code.
 *===----------------------------------------------------------------------===*)

open Llvm

let main () =
  (* Install standard binary operators.
   * 1 is the lowest precedence. *)
  Hashtbl.add Parser.binop_precedence '<' 10;
  Hashtbl.add Parser.binop_precedence '+' 20;
  Hashtbl.add Parser.binop_precedence '-' 20;
  Hashtbl.add Parser.binop_precedence '*' 40;    (* highest. *)

  (* Prime the first token. *)
  print_string "ready> "; flush stdout;
  let stream = Lexer.lex (Stream.of_channel stdin) in

  (* Run the main "interpreter loop" now. *)
  Toplevel.main_loop stream;

  (* Print out all the generated code. *)
  dump_module Codegen.the_module
;;

main ()
```

Next: Adding JIT and Optimizer Support

Kaleidoscope: Adding JIT and Optimizer Support

Chapter 4 Introduction

Welcome to Chapter 4 of the "Implementing a language with LLVM" tutorial. Chapters 1-3 described the implementation of a simple language and added support for generating LLVM IR. This chapter describes two new techniques:

adding optimizer support to your language, and adding JIT compiler support. These additions will demonstrate how to get nice, efficient code for the Kaleidoscope language.

Trivial Constant Folding

Note: the default `IRBuilder` now always includes the constant folding optimisations below.

Our demonstration for Chapter 3 is elegant and easy to extend. Unfortunately, it does not produce wonderful code. For example, when compiling simple code, we don't get obvious optimizations:

```
ready> def test(x) 1+2+x;
Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double 1.000000e+00, 2.000000e+00
        %addtmp1 = fadd double %addtmp, %x
        ret double %addtmp1
}
```

This code is a very, very literal transcription of the AST built by parsing the input. As such, this transcription lacks optimizations like constant folding (we'd like to get "`add x, 3.0`" in the example above) as well as other more important optimizations. Constant folding, in particular, is a very common and very important optimization: so much so that many language implementors implement constant folding support in their AST representation.

With LLVM, you don't need this support in the AST. Since all calls to build LLVM IR go through the LLVM builder, it would be nice if the builder itself checked to see if there was a constant folding opportunity when you call it. If so, it could just do the constant fold and return the constant instead of creating an instruction. This is exactly what the `LLVMFoldingBuilder` class does.

All we did was switch from `LLVMBuilder` to `LLVMFoldingBuilder`. Though we change no other code, we now have all of our instructions implicitly constant folded without us having to do anything about it. For example, the input above now compiles to:

```
ready> def test(x) 1+2+x;
Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double 3.000000e+00, %x
        ret double %addtmp
}
```

Well, that was easy :). In practice, we recommend always using `LLVMFoldingBuilder` when generating code like this. It has no "syntactic overhead" for its use (you don't have to uglify your compiler with constant checks everywhere) and it can dramatically reduce the amount of LLVM IR that is generated in some cases (particular for languages with a macro preprocessor or that use a lot of constants).

On the other hand, the `LLVMFoldingBuilder` is limited by the fact that it does all of its analysis inline with the code as it is built. If you take a slightly more complex example:

```
ready> def test(x) (1+2+x)*(x+(1+2));
ready> Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double 3.000000e+00, %x
        %addtmp1 = fadd double %x, 3.000000e+00
        %multmp = fmul double %addtmp, %addtmp1
        ret double %multmp
}
```

In this case, the LHS and RHS of the multiplication are the same value. We'd really like to see this generate "`tmp = x+3; result = tmp*tmp;`" instead of computing "`x*3`" twice.

Unfortunately, no amount of local analysis will be able to detect and correct this. This requires two transformations: reassociation of expressions (to make the add's lexically identical) and Common Subexpression Elimination (CSE) to delete the redundant add instruction. Fortunately, LLVM provides a broad range of optimizations that you can use, in the form of "passes".

LLVM Optimization Passes

LLVM provides many optimization passes, which do many different sorts of things and have different tradeoffs. Unlike other systems, LLVM doesn't hold to the mistaken notion that one set of optimizations is right for all languages and for all situations. LLVM allows a compiler implementor to make complete decisions about what optimizations to use, in which order, and in what situation.

As a concrete example, LLVM supports both "whole module" passes, which look across as large of body of code as they can (often a whole file, but if run at link time, this can be a substantial portion of the whole program). It also supports and includes "per-function" passes which just operate on a single function at a time, without looking at other functions. For more information on passes and how they are run, see the How to Write a Pass document and the List of LLVM Passes.

For Kaleidoscope, we are currently generating functions on the fly, one at a time, as the user types them in. We aren't shooting for the ultimate optimization experience in this setting, but we also want to catch the easy and quick stuff where possible. As such, we will choose to run a few per-function optimizations as the user types the function in. If we wanted to make a "static Kaleidoscope compiler", we would use exactly the code we have now, except that we would defer running the optimizer until the entire file has been parsed.

In order to get per-function optimizations going, we need to set up a Llvm.PassManager to hold and organize the LLVM optimizations that we want to run. Once we have that, we can add a set of optimizations to run. The code looks like this:

```
(* Create the JIT. *)
let the_execution_engine = ExecutionEngine.create Codegen.the_module in
let the_fpm = PassManager.create_function Codegen.the_module in

(* Set up the optimizer pipeline.  Start with registering info about how the
 * target lays out data structures. *)
DataLayout.add (ExecutionEngine.target_data the_execution_engine) the_fpm;

(* Do simple "peephole" optimizations and bit-twiddling optzn. *)
add_instruction_combining the_fpm;

(* reassociate expressions. *)
add_reassociation the_fpm;

(* Eliminate Common SubExpressions. *)
add_gvn the_fpm;

(* Simplify the control flow graph (deleting unreachable blocks, etc). *)
add_cfg_simplification the_fpm;

ignore (PassManager.initialize the_fpm);

(* Run the main "interpreter loop" now. *)
Toplevel.main_loop the_fpm the_execution_engine stream;
```

The meat of the matter here, is the definition of "`the_fpm`". It requires a pointer to the `the_module` to construct itself. Once it is set up, we use a series of "add" calls to add a bunch of LLVM passes. The first pass is basically

boilerplate, it adds a pass so that later optimizations know how the data structures in the program are laid out. The "the_execution_engine" variable is related to the JIT, which we will get to in the next section.

In this case, we choose to add 4 optimization passes. The passes we chose here are a pretty standard set of "cleanup" optimizations that are useful for a wide variety of code. I won't delve into what they do but, believe me, they are a good starting place :).

Once the Llvm.PassManager. is set up, we need to make use of it. We do this by running it after our newly created function is constructed (in Codegen.codegen_func), but before it is returned to the client:

```
let codegen_func the_fpm = function
      ...
      try
        let ret_val = codegen_expr body in

        (* Finish off the function. *)
        let _ = build_ret ret_val builder in

        (* Validate the generated code, checking for consistency. *)
        Llvm_analysis.assert_valid_function the_function;

        (* Optimize the function. *)
        let _ = PassManager.run_function the_function the_fpm in

        the_function
```

As you can see, this is pretty straightforward. The the_fpm optimizes and updates the LLVM Function* in place, improving (hopefully) its body. With this in place, we can try our test above again:

```
ready> def test(x) (1+2+x)*(x+(1+2));
ready> Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double %x, 3.000000e+00
        %multmp = fmul double %addtmp, %addtmp
        ret double %multmp
}
```

As expected, we now get our nicely optimized code, saving a floating point add instruction from every execution of this function.

LLVM provides a wide variety of optimizations that can be used in certain circumstances. Some documentation about the various passes is available, but it isn't very complete. Another good source of ideas can come from looking at the passes that Clang runs to get started. The "opt" tool allows you to experiment with passes from the command line, so you can see if they do anything.

Now that we have reasonable code coming out of our front-end, lets talk about executing it!

Adding a JIT Compiler

Code that is available in LLVM IR can have a wide variety of tools applied to it. For example, you can run optimizations on it (as we did above), you can dump it out in textual or binary forms, you can compile the code to an assembly file (.s) for some target, or you can JIT compile it. The nice thing about the LLVM IR representation is that it is the "common currency" between many different parts of the compiler.

In this section, we'll add JIT compiler support to our interpreter. The basic idea that we want for Kaleidoscope is to have the user enter function bodies as they do now, but immediately evaluate the top-level expressions they type in. For example, if they type in "1 + 2;", we should evaluate and print out 3. If they define a function, they should be able to call it from the command line.

In order to do this, we first declare and initialize the JIT. This is done by adding a global variable and a call in `main`:

```
...
let main () =
  ...
  (* Create the JIT. *)
  let the_execution_engine = ExecutionEngine.create Codegen.the_module in
  ...
```

This creates an abstract "Execution Engine" which can be either a JIT compiler or the LLVM interpreter. LLVM will automatically pick a JIT compiler for you if one is available for your platform, otherwise it will fall back to the interpreter.

Once the `Llvm_executionengine.ExecutionEngine.t` is created, the JIT is ready to be used. There are a variety of APIs that are useful, but the simplest one is the "`Llvm_executionengine.ExecutionEngine.run_function`" function. This method JIT compiles the specified LLVM Function and returns a function pointer to the generated machine code. In our case, this means that we can change the code that parses a top-level expression to look like this:

```
(* Evaluate a top-level expression into an anonymous function. *)
let e = Parser.parse_toplevel stream in
print_endline "parsed a top-level expr";
let the_function = Codegen.codegen_func the_fpm e in
dump_value the_function;

(* JIT the function, returning a function pointer. *)
let result = ExecutionEngine.run_function the_function [||]
  the_execution_engine in

print_string "Evaluated to ";
print_float (GenericValue.as_float Codegen.double_type result);
print_newline ();
```

Recall that we compile top-level expressions into a self-contained LLVM function that takes no arguments and returns the computed double. Because the LLVM JIT compiler matches the native platform ABI, this means that you can just cast the result pointer to a function pointer of that type and call it directly. This means, there is no difference between JIT compiled code and native machine code that is statically linked into your application.

With just these two changes, lets see how Kaleidoscope works now!

```
ready> 4+5;
define double @""() {
entry:
        ret double 9.000000e+00
}

Evaluated to 9.000000
```

Well this looks like it is basically working. The dump of the function shows the "no argument function that always returns double" that we synthesize for each top level expression that is typed in. This demonstrates very basic functionality, but can we do more?

```
ready> def testfunc(x y) x + y*2;
Read function definition:
define double @testfunc(double %x, double %y) {
entry:
        %multmp = fmul double %y, 2.000000e+00
        %addtmp = fadd double %multmp, %x
        ret double %addtmp
}
```

```
ready> testfunc(4, 10);
define double @""() {
entry:
        %calltmp = call double @testfunc(double 4.000000e+00, double 1.000000e+01)
        ret double %calltmp
}

Evaluated to 24.000000
```

This illustrates that we can now call user code, but there is something a bit subtle going on here. Note that we only invoke the JIT on the anonymous functions that *call testfunc*, but we never invoked it on *testfunc* itself. What actually happened here is that the JIT scanned for all non-JIT'd functions transitively called from the anonymous function and compiled all of them before returning from `run_function`.

The JIT provides a number of other more advanced interfaces for things like freeing allocated machine code, rejit'ing functions to update them, etc. However, even with this simple code, we get some surprisingly powerful capabilities - check this out (I removed the dump of the anonymous functions, you should get the idea by now :) :

```
ready> extern sin(x);
Read extern:
declare double @sin(double)

ready> extern cos(x);
Read extern:
declare double @cos(double)

ready> sin(1.0);
Evaluated to 0.841471

ready> def foo(x) sin(x)*sin(x) + cos(x)*cos(x);
Read function definition:
define double @foo(double %x) {
entry:
        %calltmp = call double @sin(double %x)
        %multmp = fmul double %calltmp, %calltmp
        %calltmp2 = call double @cos(double %x)
        %multmp4 = fmul double %calltmp2, %calltmp2
        %addtmp = fadd double %multmp, %multmp4
        ret double %addtmp
}

ready> foo(4.0);
Evaluated to 1.000000
```

Whoa, how does the JIT know about sin and cos? The answer is surprisingly simple: in this example, the JIT started execution of a function and got to a function call. It realized that the function was not yet JIT compiled and invoked the standard set of routines to resolve the function. In this case, there is no body defined for the function, so the JIT ended up calling "`dlsym("sin")`" on the Kaleidoscope process itself. Since "`sin`" is defined within the JIT's address space, it simply patches up calls in the module to call the libm version of `sin` directly.

The LLVM JIT provides a number of interfaces (look in the `llvm_executionengine.mli` file) for controlling how unknown functions get resolved. It allows you to establish explicit mappings between IR objects and addresses (useful for LLVM global variables that you want to map to static tables, for example), allows you to dynamically decide on the fly based on the function name, and even allows you to have the JIT compile functions lazily the first time they're called.

One interesting application of this is that we can now extend the language by writing arbitrary C code to implement operations. For example, if we add:

```
/* putchard - putchar that takes a double and returns 0. */
extern "C"
double putchard(double X) {
  putchar((char)X);
  return 0;
}
```

Now we can produce simple output to the console by using things like: "`extern putchard(x); putchard(120);`", which prints a lowercase 'x' on the console (120 is the ASCII code for 'x'). Similar code could be used to implement file I/O, console input, and many other capabilities in Kaleidoscope.

This completes the JIT and optimizer chapter of the Kaleidoscope tutorial. At this point, we can compile a non-Turing-complete programming language, optimize and JIT compile it in a user-driven way. Next up we'll look into extending the language with control flow constructs, tackling some interesting LLVM IR issues along the way.

Full Code Listing

Here is the complete code listing for our running example, enhanced with the LLVM JIT and optimizer. To build this example, use:

```
# Compile
ocamlbuild toy.byte
# Run
./toy.byte
```

Here is the code:

_tags:

```
<{lexer,parser}.ml>: use_camlp4, pp(camlp4of)
<*.{byte,native}>: g++, use_llvm, use_llvm_analysis
<*.{byte,native}>: use_llvm_executionengine, use_llvm_target
<*.{byte,native}>: use_llvm_scalar_opts, use_bindings
```

myocamlbuild.ml:

```
open Ocamlbuild_plugin;;

ocaml_lib ~extern:true "llvm";;
ocaml_lib ~extern:true "llvm_analysis";;
ocaml_lib ~extern:true "llvm_executionengine";;
ocaml_lib ~extern:true "llvm_target";;
ocaml_lib ~extern:true "llvm_scalar_opts";;

flag ["link"; "ocaml"; "g++"] (S[A"-cc"; A"g++"]);;
dep ["link"; "ocaml"; "use_bindings"] ["bindings.o"];;
```

token.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer Tokens
 *===----------------------------------------------------------------------===*)

(* The lexer returns these 'Kwd' if it is an unknown character, otherwise one of
 * these others for known things. *)
type token =
  (* commands *)
  | Def | Extern
```

```
  (* primary *)
  | Ident of string | Number of float

  (* unknown *)
  | Kwd of char
```

lexer.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer
 *===----------------------------------------------------------------------===*)

let rec lex = parser
  (* Skip any whitespace. *)
  | [< ' (' ' | '\n' | '\r' | '\t'); stream >] -> lex stream

  (* identifier: [a-zA-Z][a-zA-Z0-9] *)
  | [< ' ('A' .. 'Z' | 'a' .. 'z' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_ident buffer stream

  (* number: [0-9.]+ *)
  | [< ' ('0' .. '9' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_number buffer stream

  (* Comment until end of line. *)
  | [< ' ('#'); stream >] ->
      lex_comment stream

  (* Otherwise, just return the character as its ascii value. *)
  | [< 'c; stream >] ->
      [< 'Token.Kwd c; lex stream >]

  (* end of stream. *)
  | [< >] -> [< >]

and lex_number buffer = parser
  | [< ' ('0' .. '9' | '.' as c); stream >] ->
      Buffer.add_char buffer c;
      lex_number buffer stream
  | [< stream=lex >] ->
      [< 'Token.Number (float_of_string (Buffer.contents buffer)); stream >]

and lex_ident buffer = parser
  | [< ' ('A' .. 'Z' | 'a' .. 'z' | '0' .. '9' as c); stream >] ->
      Buffer.add_char buffer c;
      lex_ident buffer stream
  | [< stream=lex >] ->
      match Buffer.contents buffer with
      | "def" -> [< 'Token.Def; stream >]
      | "extern" -> [< 'Token.Extern; stream >]
      | id -> [< 'Token.Ident id; stream >]

and lex_comment = parser
  | [< ' ('\n'); stream=lex >] -> stream
```

```
  | [< 'c; e=lex_comment >] -> e
  | [< >] -> [< >]
```

ast.ml:

```
(*===----------------------------------------------------------------------===
 * Abstract Syntax Tree (aka Parse Tree)
 *===----------------------------------------------------------------------===*)

(* expr - Base type for all expression nodes. *)
type expr =
  (* variant for numeric literals like "1.0". *)
  | Number of float

  (* variant for referencing a variable, like "a". *)
  | Variable of string

  (* variant for a binary operator. *)
  | Binary of char * expr * expr

  (* variant for function calls. *)
  | Call of string * expr array

(* proto - This type represents the "prototype" for a function, which captures
 * its name, and its argument names (thus implicitly the number of arguments the
 * function takes). *)
type proto = Prototype of string * string array

(* func - This type represents a function definition itself. *)
type func = Function of proto * expr
```

parser.ml:

```
(*===----------------------------------------------------------------------===
 * Parser
 *===----------------------------------------------------------------------===*)

(* binop_precedence - This holds the precedence for each binary operator that is
 * defined *)
let binop_precedence:(char, int) Hashtbl.t = Hashtbl.create 10

(* precedence - Get the precedence of the pending binary operator token. *)
let precedence c = try Hashtbl.find binop_precedence c with Not_found -> -1

(* primary
 *   ::= identifier
 *   ::= numberexpr
 *   ::= parenexpr *)
let rec parse_primary = parser
  (* numberexpr ::= number *)
  | [< 'Token.Number n >] -> Ast.Number n

  (* parenexpr ::= '(' expression ')' *)
  | [< 'Token.Kwd '('; e=parse_expr; 'Token.Kwd ')' ?? "expected ')'" >] -> e

  (* identifierexpr
   *   ::= identifier
   *   ::= identifier '(' argumentexpr ')' *)
  | [< 'Token.Ident id; stream >] ->
```

```
      let rec parse_args accumulator = parser
        | [< e=parse_expr; stream >] ->
            begin parser
              | [< 'Token.Kwd ','; e=parse_args (e :: accumulator) >] -> e
              | [< >] -> e :: accumulator
            end stream
        | [< >] -> accumulator
      in
      let rec parse_ident id = parser
        (* Call. *)
        | [< 'Token.Kwd '(';
             args=parse_args [];
             'Token.Kwd ')' ?? "expected ')'">] ->
            Ast.Call (id, Array.of_list (List.rev args))

        (* Simple variable ref. *)
        | [< >] -> Ast.Variable id
      in
      parse_ident id stream

  | [< >] -> raise (Stream.Error "unknown token when expecting an expression.")

(* binoprhs
 *   ::= ('+' primary)* *)
and parse_bin_rhs expr_prec lhs stream =
  match Stream.peek stream with
  (* If this is a binop, find its precedence. *)
  | Some (Token.Kwd c) when Hashtbl.mem binop_precedence c ->
      let token_prec = precedence c in

      (* If this is a binop that binds at least as tightly as the current binop,
       * consume it, otherwise we are done. *)
      if token_prec < expr_prec then lhs else begin
        (* Eat the binop. *)
        Stream.junk stream;

        (* Parse the primary expression after the binary operator. *)
        let rhs = parse_primary stream in

        (* Okay, we know this is a binop. *)
        let rhs =
          match Stream.peek stream with
          | Some (Token.Kwd c2) ->
              (* If BinOp binds less tightly with rhs than the operator after
               * rhs, let the pending operator take rhs as its lhs. *)
              let next_prec = precedence c2 in
              if token_prec < next_prec
              then parse_bin_rhs (token_prec + 1) rhs stream
              else rhs
          | _ -> rhs
        in

        (* Merge lhs/rhs. *)
        let lhs = Ast.Binary (c, lhs, rhs) in
        parse_bin_rhs expr_prec lhs stream
      end
  | _ -> lhs
```

```
(* expression
 *   ::= primary binoprhs *)
and parse_expr = parser
  | [< lhs=parse_primary; stream >] -> parse_bin_rhs 0 lhs stream

(* prototype
 *   ::= id '(' id* ')' *)
let parse_prototype =
  let rec parse_args accumulator = parser
    | [< 'Token.Ident id; e=parse_args (id::accumulator) >] -> e
    | [< >] -> accumulator
  in

  parser
  | [< 'Token.Ident id;
       'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      (* success. *)
      Ast.Prototype (id, Array.of_list (List.rev args))

  | [< >] ->
      raise (Stream.Error "expected function name in prototype")

(* definition ::= 'def' prototype expression *)
let parse_definition = parser
  | [< 'Token.Def; p=parse_prototype; e=parse_expr >] ->
      Ast.Function (p, e)

(* toplevelexpr ::= expression *)
let parse_toplevel = parser
  | [< e=parse_expr >] ->
      (* Make an anonymous proto. *)
      Ast.Function (Ast.Prototype ("", [||]), e)

(*  external ::= 'extern' prototype *)
let parse_extern = parser
  | [< 'Token.Extern; e=parse_prototype >] -> e
```

codegen.ml:

```
(*===----------------------------------------------------------------------===
 * Code Generation
 *===----------------------------------------------------------------------===*)

open Llvm

exception Error of string

let context = global_context ()
let the_module = create_module context "my cool jit"
let builder = builder context
let named_values:(string, llvalue) Hashtbl.t = Hashtbl.create 10
let double_type = double_type context

let rec codegen_expr = function
  | Ast.Number n -> const_float double_type n
  | Ast.Variable name ->
```

```
      (try Hashtbl.find named_values name with
        | Not_found -> raise (Error "unknown variable name"))
  | Ast.Binary (op, lhs, rhs) ->
      let lhs_val = codegen_expr lhs in
      let rhs_val = codegen_expr rhs in
      begin
        match op with
        | '+' -> build_add lhs_val rhs_val "addtmp" builder
        | '-' -> build_sub lhs_val rhs_val "subtmp" builder
        | '*' -> build_mul lhs_val rhs_val "multmp" builder
        | '<' ->
            (* Convert bool 0/1 to double 0.0 or 1.0 *)
            let i = build_fcmp Fcmp.Ult lhs_val rhs_val "cmptmp" builder in
            build_uitofp i double_type "booltmp" builder
        | _ -> raise (Error "invalid binary operator")
      end
  | Ast.Call (callee, args) ->
      (* Look up the name in the module table. *)
      let callee =
        match lookup_function callee the_module with
        | Some callee -> callee
        | None -> raise (Error "unknown function referenced")
      in
      let params = params callee in

      (* If argument mismatch error. *)
      if Array.length params == Array.length args then () else
        raise (Error "incorrect # arguments passed");
      let args = Array.map codegen_expr args in
      build_call callee args "calltmp" builder

let codegen_proto = function
  | Ast.Prototype (name, args) ->
      (* Make the function type: double(double,double) etc. *)
      let doubles = Array.make (Array.length args) double_type in
      let ft = function_type double_type doubles in
      let f =
        match lookup_function name the_module with
        | None -> declare_function name ft the_module

        (* If 'f' conflicted, there was already something named 'name'. If it
         * has a body, don't allow redefinition or reextern. *)
        | Some f ->
            (* If 'f' already has a body, reject this. *)
            if block_begin f <> At_end f then
              raise (Error "redefinition of function");

            (* If 'f' took a different number of arguments, reject. *)
            if element_type (type_of f) <> ft then
              raise (Error "redefinition of function with different # args");
            f
      in

      (* Set names for all arguments. *)
      Array.iteri (fun i a ->
        let n = args.(i) in
        set_value_name n a;
        Hashtbl.add named_values n a;
```

```
      ) (params f);
      f

let codegen_func the_fpm = function
  | Ast.Function (proto, body) ->
      Hashtbl.clear named_values;
      let the_function = codegen_proto proto in

      (* Create a new basic block to start insertion into. *)
      let bb = append_block context "entry" the_function in
      position_at_end bb builder;

      try
        let ret_val = codegen_expr body in

        (* Finish off the function. *)
        let _ = build_ret ret_val builder in

        (* Validate the generated code, checking for consistency. *)
        Llvm_analysis.assert_valid_function the_function;

        (* Optimize the function. *)
        let _ = PassManager.run_function the_function the_fpm in

        the_function
      with e ->
        delete_function the_function;
        raise e
```

toplevel.ml:

```
(*===----------------------------------------------------------------------===
 * Top-Level parsing and JIT Driver
 *===----------------------------------------------------------------------===*)

open Llvm
open Llvm_executionengine

(* top ::= definition | external | expression | ';' *)
let rec main_loop the_fpm the_execution_engine stream =
  match Stream.peek stream with
  | None -> ()

  (* ignore top-level semicolons. *)
  | Some (Token.Kwd ';') ->
      Stream.junk stream;
      main_loop the_fpm the_execution_engine stream

  | Some token ->
      begin
        try match token with
        | Token.Def ->
            let e = Parser.parse_definition stream in
            print_endline "parsed a function definition.";
            dump_value (Codegen.codegen_func the_fpm e);
        | Token.Extern ->
            let e = Parser.parse_extern stream in
            print_endline "parsed an extern.";
            dump_value (Codegen.codegen_proto e);
```

```
          | _ ->
              (* Evaluate a top-level expression into an anonymous function. *)
              let e = Parser.parse_toplevel stream in
              print_endline "parsed a top-level expr";
              let the_function = Codegen.codegen_func the_fpm e in
              dump_value the_function;

              (* JIT the function, returning a function pointer. *)
              let result = ExecutionEngine.run_function the_function [||]
                the_execution_engine in

              print_string "Evaluated to ";
              print_float (GenericValue.as_float Codegen.double_type result);
              print_newline ();
            with Stream.Error s | Codegen.Error s ->
              (* Skip token for error recovery. *)
              Stream.junk stream;
              print_endline s;
        end;
        print_string "ready> "; flush stdout;
        main_loop the_fpm the_execution_engine stream
```

toy.ml:

```
    (*===----------------------------------------------------------------------===
     * Main driver code.
     *===----------------------------------------------------------------------===*)

    open Llvm
    open Llvm_executionengine
    open Llvm_target
    open Llvm_scalar_opts

    let main () =
      ignore (initialize_native_target ());

      (* Install standard binary operators.
       * 1 is the lowest precedence. *)
      Hashtbl.add Parser.binop_precedence '<' 10;
      Hashtbl.add Parser.binop_precedence '+' 20;
      Hashtbl.add Parser.binop_precedence '-' 20;
      Hashtbl.add Parser.binop_precedence '*' 40;    (* highest. *)

      (* Prime the first token. *)
      print_string "ready> "; flush stdout;
      let stream = Lexer.lex (Stream.of_channel stdin) in

      (* Create the JIT. *)
      let the_execution_engine = ExecutionEngine.create Codegen.the_module in
      let the_fpm = PassManager.create_function Codegen.the_module in

      (* Set up the optimizer pipeline.  Start with registering info about how the
       * target lays out data structures. *)
      DataLayout.add (ExecutionEngine.target_data the_execution_engine) the_fpm;

      (* Do simple "peephole" optimizations and bit-twiddling optzn. *)
      add_instruction_combination the_fpm;
```

```
  (* reassociate expressions. *)
  add_reassociation the_fpm;

  (* Eliminate Common SubExpressions. *)
  add_gvn the_fpm;

  (* Simplify the control flow graph (deleting unreachable blocks, etc). *)
  add_cfg_simplification the_fpm;

  ignore (PassManager.initialize the_fpm);

  (* Run the main "interpreter loop" now. *)
  Toplevel.main_loop the_fpm the_execution_engine stream;

  (* Print out all the generated code. *)
  dump_module Codegen.the_module
;;

main ()
```

bindings.c

```
#include <stdio.h>

/* putchard - putchar that takes a double and returns 0. */
extern double putchard(double X) {
  putchar((char)X);
  return 0;
}
```

Next: Extending the language: control flow

Kaleidoscope: Extending the Language: Control Flow

Chapter 5 Introduction

Welcome to Chapter 5 of the "Implementing a language with LLVM" tutorial. Parts 1-4 described the implementation of the simple Kaleidoscope language and included support for generating LLVM IR, followed by optimizations and

a JIT compiler. Unfortunately, as presented, Kaleidoscope is mostly useless: it has no control flow other than call and return. This means that you can't have conditional branches in the code, significantly limiting its power. In this episode of "build that compiler", we'll extend Kaleidoscope to have an if/then/else expression plus a simple 'for' loop.

If/Then/Else

Extending Kaleidoscope to support if/then/else is quite straightforward. It basically requires adding lexer support for this "new" concept to the lexer, parser, AST, and LLVM code emitter. This example is nice, because it shows how easy it is to "grow" a language over time, incrementally extending it as new ideas are discovered.

Before we get going on "how" we add this extension, lets talk about "what" we want. The basic idea is that we want to be able to write this sort of thing:

```
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2);
```

In Kaleidoscope, every construct is an expression: there are no statements. As such, the if/then/else expression needs to return a value like any other. Since we're using a mostly functional form, we'll have it evaluate its conditional, then return the 'then' or 'else' value based on how the condition was resolved. This is very similar to the C "?:" expression.

The semantics of the if/then/else expression is that it evaluates the condition to a boolean equality value: 0.0 is considered to be false and everything else is considered to be true. If the condition is true, the first subexpression is evaluated and returned, if the condition is false, the second subexpression is evaluated and returned. Since Kaleidoscope allows side-effects, this behavior is important to nail down.

Now that we know what we "want", lets break this down into its constituent pieces.

Lexer Extensions for If/Then/Else The lexer extensions are straightforward. First we add new variants for the relevant tokens:

```
(* control *)
| If | Then | Else | For | In
```

Once we have that, we recognize the new keywords in the lexer. This is pretty simple stuff:

```
...
match Buffer.contents buffer with
| "def" -> [< 'Token.Def; stream >]
| "extern" -> [< 'Token.Extern; stream >]
| "if" -> [< 'Token.If; stream >]
| "then" -> [< 'Token.Then; stream >]
| "else" -> [< 'Token.Else; stream >]
| "for" -> [< 'Token.For; stream >]
| "in" -> [< 'Token.In; stream >]
| id -> [< 'Token.Ident id; stream >]
```

AST Extensions for If/Then/Else To represent the new expression we add a new AST variant for it:

```
type expr =
  ...
  (* variant for if/then/else. *)
  | If of expr * expr * expr
```

The AST variant just has pointers to the various subexpressions.

Parser Extensions for If/Then/Else Now that we have the relevant tokens coming from the lexer and we have the AST node to build, our parsing logic is relatively straightforward. First we define a new parsing function:

```
let rec parse_primary = parser
  ...
  (* ifexpr ::= 'if' expr 'then' expr 'else' expr *)
  | [< 'Token.If; c=parse_expr;
       'Token.Then ?? "expected 'then'"; t=parse_expr;
       'Token.Else ?? "expected 'else'"; e=parse_expr >] ->
      Ast.If (c, t, e)
```

Next we hook it up as a primary expression:

```
let rec parse_primary = parser
  ...
  (* ifexpr ::= 'if' expr 'then' expr 'else' expr *)
  | [< 'Token.If; c=parse_expr;
       'Token.Then ?? "expected 'then'"; t=parse_expr;
       'Token.Else ?? "expected 'else'"; e=parse_expr >] ->
      Ast.If (c, t, e)
```

LLVM IR for If/Then/Else Now that we have it parsing and building the AST, the final piece is adding LLVM code generation support. This is the most interesting part of the if/then/else example, because this is where it starts to introduce new concepts. All of the code above has been thoroughly described in previous chapters.

To motivate the code we want to produce, lets take a look at a simple example. Consider:

```
extern foo();
extern bar();
def baz(x) if x then foo() else bar();
```

If you disable optimizations, the code you'll (soon) get from Kaleidoscope looks like this:

```
declare double @foo()

declare double @bar()

define double @baz(double %x) {
entry:
  %ifcond = fcmp one double %x, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:     ; preds = %entry
  %calltmp = call double @foo()
  br label %ifcont

else:     ; preds = %entry
  %calltmp1 = call double @bar()
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ %calltmp, %then ], [ %calltmp1, %else ]
  ret double %iftmp
}
```

To visualize the control flow graph, you can use a nifty feature of the LLVM 'opt' tool. If you put this LLVM IR into "t.ll" and run "`llvm-as < t.ll | opt -analyze -view-cfg`", a window will pop up and you'll see this graph:

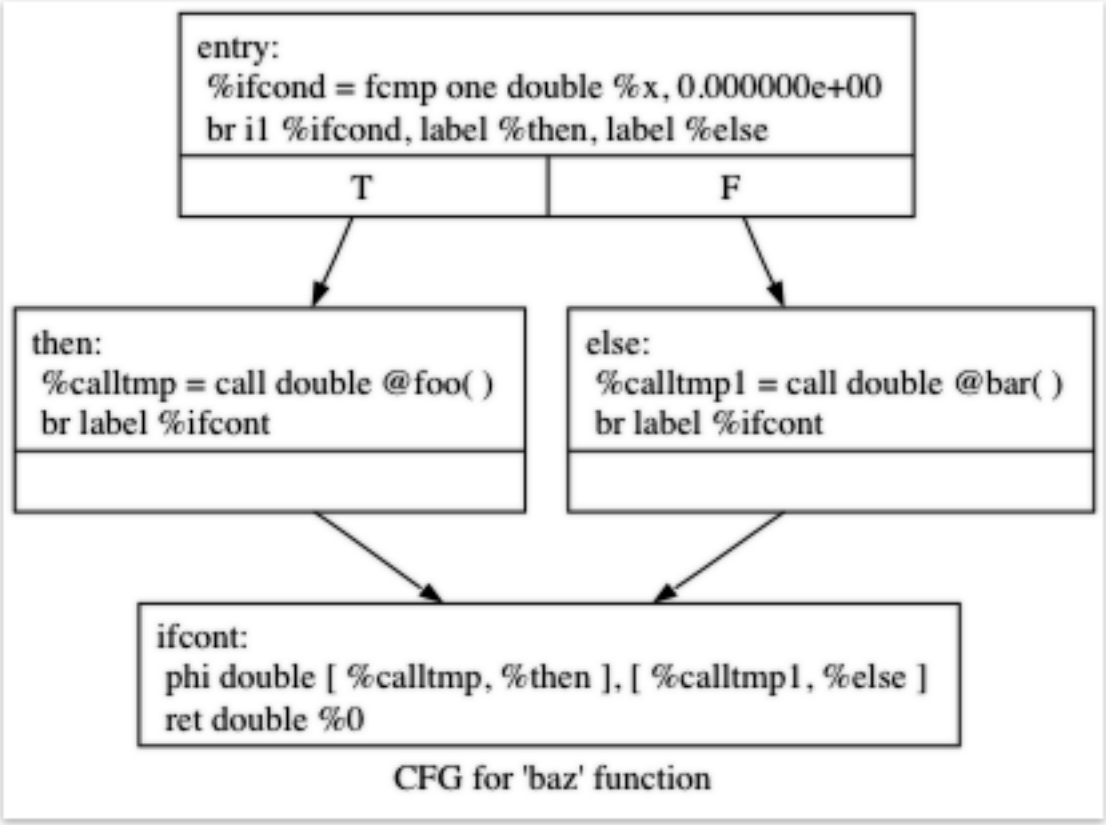


Figure 2.2: Example CFG

Another way to get this is to call "`Llvm_analysis.view_function_cfg f`" or "`Llvm_analysis.view_function_cfg_only f`" (where `f` is a "`Function`") either by inserting actual calls into the code and recompiling or by calling these in the debugger. LLVM has many nice features for visualizing various graphs.

Getting back to the generated code, it is fairly simple: the entry block evaluates the conditional expression ("x" in our case here) and compares the result to 0.0 with the "`fcmp one`" instruction ('one' is "Ordered and Not Equal"). Based on the result of this expression, the code jumps to either the "then" or "else" blocks, which contain the expressions for the true/false cases.

Once the then/else blocks are finished executing, they both branch back to the 'ifcont' block to execute the code that happens after the if/then/else. In this case the only thing left to do is to return to the caller of the function. The question then becomes: how does the code know which expression to return?

The answer to this question involves an important SSA operation: the Phi operation. If you're not familiar with SSA, the wikipedia article is a good introduction and there are various other introductions to it available on your favorite search engine. The short version is that "execution" of the Phi operation requires "remembering" which block control came from. The Phi operation takes on the value corresponding to the input control block. In this case, if control comes in from the "then" block, it gets the value of "calltmp". If control comes from the "else" block, it gets the value of "calltmp1".

At this point, you are probably starting to think "Oh no! This means my simple and elegant front-end will have to start generating SSA form in order to use LLVM!". Fortunately, this is not the case, and we strongly advise *not* implementing an SSA construction algorithm in your front-end unless there is an amazingly good reason to do so. In practice, there are two sorts of values that float around in code written for your average imperative programming language that might need Phi nodes:

1. Code that involves user variables: `x = 1; x = x + 1;`
2. Values that are implicit in the structure of your AST, such as the Phi node in this case.

In Chapter 7 of this tutorial ("mutable variables"), we'll talk about #1 in depth. For now, just believe me that you don't need SSA construction to handle this case. For #2, you have the choice of using the techniques that we will describe for #1, or you can insert Phi nodes directly, if convenient. In this case, it is really really easy to generate the Phi node, so we choose to do it directly.

Okay, enough of the motivation and overview, lets generate code!

Code Generation for If/Then/Else In order to generate code for this, we implement the `Codegen` method for `IfExprAST`:

```
let rec codegen_expr = function
  ...
  | Ast.If (cond, then_, else_) ->
      let cond = codegen_expr cond in

      (* Convert condition to a bool by comparing equal to 0.0 *)
      let zero = const_float double_type 0.0 in
      let cond_val = build_fcmp Fcmp.One cond zero "ifcond" builder in
```

This code is straightforward and similar to what we saw before. We emit the expression for the condition, then compare that value to zero to get a truth value as a 1-bit (bool) value.

```
(* Grab the first block so that we might later add the conditional branch
 * to it at the end of the function. *)
let start_bb = insertion_block builder in
let the_function = block_parent start_bb in
```

```
let then_bb = append_block context "then" the_function in
position_at_end then_bb builder;
```

As opposed to the C++ tutorial, we have to build our basic blocks bottom up since we can't have dangling BasicBlocks. We start off by saving a pointer to the first block (which might not be the entry block), which we'll need to build a conditional branch later. We do this by asking the `builder` for the current BasicBlock. The fourth line gets the current Function object that is being built. It gets this by the `start_bb` for its "parent" (the function it is currently embedded into).

Once it has that, it creates one block. It is automatically appended into the function's list of blocks.

```
(* Emit 'then' value. *)
position_at_end then_bb builder;
let then_val = codegen_expr then_ in

(* Codegen of 'then' can change the current block, update then_bb for the
 * phi. We create a new name because one is used for the phi node, and the
 * other is used for the conditional branch. *)
let new_then_bb = insertion_block builder in
```

We move the builder to start inserting into the "then" block. Strictly speaking, this call moves the insertion point to be at the end of the specified block. However, since the "then" block is empty, it also starts out by inserting at the beginning of the block. :)

Once the insertion point is set, we recursively codegen the "then" expression from the AST.

The final line here is quite subtle, but is very important. The basic issue is that when we create the Phi node in the merge block, we need to set up the block/value pairs that indicate how the Phi will work. Importantly, the Phi node expects to have an entry for each predecessor of the block in the CFG. Why then, are we getting the current block when we just set it to ThenBB 5 lines above? The problem is that the "Then" expression may actually itself change the block that the Builder is emitting into if, for example, it contains a nested "if/then/else" expression. Because calling Codegen recursively could arbitrarily change the notion of the current block, we are required to get an up-to-date value for code that will set up the Phi node.

```
(* Emit 'else' value. *)
let else_bb = append_block context "else" the_function in
position_at_end else_bb builder;
let else_val = codegen_expr else_ in

(* Codegen of 'else' can change the current block, update else_bb for the
 * phi. *)
let new_else_bb = insertion_block builder in
```

Code generation for the 'else' block is basically identical to codegen for the 'then' block.

```
(* Emit merge block. *)
let merge_bb = append_block context "ifcont" the_function in
position_at_end merge_bb builder;
let incoming = [(then_val, new_then_bb); (else_val, new_else_bb)] in
let phi = build_phi incoming "iftmp" builder in
```

The first two lines here are now familiar: the first adds the "merge" block to the Function object. The second block changes the insertion point so that newly created code will go into the "merge" block. Once that is done, we need to create the PHI node and set up the block/value pairs for the PHI.

```
(* Return to the start block to add the conditional branch. *)
position_at_end start_bb builder;
ignore (build_cond_br cond_val then_bb else_bb builder);
```

Once the blocks are created, we can emit the conditional branch that chooses between them. Note that creating new blocks does not implicitly affect the IRBuilder, so it is still inserting into the block that the condition went into. This is why we needed to save the "start" block.

```
(* Set a unconditional branch at the end of the 'then' block and the
 * 'else' block to the 'merge' block. *)
position_at_end new_then_bb builder; ignore (build_br merge_bb builder);
position_at_end new_else_bb builder; ignore (build_br merge_bb builder);

(* Finally, set the builder to the end of the merge block. *)
position_at_end merge_bb builder;

phi
```

To finish off the blocks, we create an unconditional branch to the merge block. One interesting (and very important) aspect of the LLVM IR is that it requires all basic blocks to be "terminated" with a control flow instruction such as return or branch. This means that all control flow, *including fall throughs* must be made explicit in the LLVM IR. If you violate this rule, the verifier will emit an error.

Finally, the CodeGen function returns the phi node as the value computed by the if/then/else expression. In our example above, this returned value will feed into the code for the top-level function, which will create the return instruction.

Overall, we now have the ability to execute conditional code in Kaleidoscope. With this extension, Kaleidoscope is a fairly complete language that can calculate a wide variety of numeric functions. Next up we'll add another useful expression that is familiar from non-functional languages...

'for' Loop Expression

Now that we know how to add basic control flow constructs to the language, we have the tools to add more powerful things. Lets add something more aggressive, a 'for' expression:

```
extern putchard(char);
def printstar(n)
  for i = 1, i < n, 1.0 in
    putchard(42);  # ascii 42 = '*'

# print 100 '*' characters
printstar(100);
```

This expression defines a new variable ("i" in this case) which iterates from a starting value, while the condition ("i < n" in this case) is true, incrementing by an optional step value ("1.0" in this case). If the step value is omitted, it defaults to 1.0. While the loop is true, it executes its body expression. Because we don't have anything better to return, we'll just define the loop as always returning 0.0. In the future when we have mutable variables, it will get more useful.

As before, lets talk about the changes that we need to Kaleidoscope to support this.

Lexer Extensions for the 'for' Loop The lexer extensions are the same sort of thing as for if/then/else:

```
... in Token.token ...
(* control *)
| If | Then | Else
| For | In

... in Lexer.lex_ident...
    match Buffer.contents buffer with
```

```
    | "def" -> [< 'Token.Def; stream >]
    | "extern" -> [< 'Token.Extern; stream >]
    | "if" -> [< 'Token.If; stream >]
    | "then" -> [< 'Token.Then; stream >]
    | "else" -> [< 'Token.Else; stream >]
    | "for" -> [< 'Token.For; stream >]
    | "in" -> [< 'Token.In; stream >]
    | id -> [< 'Token.Ident id; stream >]
```

AST Extensions for the 'for' Loop The AST variant is just as simple. It basically boils down to capturing the variable name and the constituent expressions in the node.

```
type expr =
  ...
  (* variant for for/in. *)
  | For of string * expr * expr * expr option * expr
```

Parser Extensions for the 'for' Loop The parser code is also fairly standard. The only interesting thing here is handling of the optional step value. The parser code handles it by checking to see if the second comma is present. If not, it sets the step value to null in the AST node:

```
let rec parse_primary = parser
  ...
  (* forexpr
        ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression *)
  | [< 'Token.For;
       'Token.Ident id ?? "expected identifier after for";
       'Token.Kwd '=' ?? "expected '=' after for";
       stream >] ->
      begin parser
        | [<
             start=parse_expr;
             'Token.Kwd ',' ?? "expected ',' after for";
             end_=parse_expr;
             stream >] ->
            let step =
              begin parser
              | [< 'Token.Kwd ','; step=parse_expr >] -> Some step
              | [< >] -> None
              end stream
            in
            begin parser
            | [< 'Token.In; body=parse_expr >] ->
                Ast.For (id, start, end_, step, body)
            | [< >] ->
                raise (Stream.Error "expected 'in' after for")
            end stream
        | [< >] ->
            raise (Stream.Error "expected '=' after for")
      end stream
```

LLVM IR for the 'for' Loop Now we get to the good part: the LLVM IR we want to generate for this thing. With the simple example above, we get this LLVM IR (note that this dump is generated with optimizations disabled for clarity):

```
declare double @putchard(double)

define double @printstar(double %n) {
entry:
        ; initial value = 1.0 (inlined into phi)
  br label %loop

loop:     ; preds = %loop, %entry
  %i = phi double [ 1.000000e+00, %entry ], [ %nextvar, %loop ]
        ; body
  %calltmp = call double @putchard(double 4.200000e+01)
        ; increment
  %nextvar = fadd double %i, 1.000000e+00

        ; termination test
  %cmptmp = fcmp ult double %i, %n
  %booltmp = uitofp i1 %cmptmp to double
  %loopcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %loopcond, label %loop, label %afterloop

afterloop:     ; preds = %loop
        ; loop always returns 0.0
  ret double 0.000000e+00
}
```

This loop contains all the same constructs we saw before: a phi node, several expressions, and some basic blocks. Lets see how this fits together.

Code Generation for the 'for' Loop The first part of Codegen is very simple: we just output the start expression for the loop value:

```
let rec codegen_expr = function
  ...
  | Ast.For (var_name, start, end_, step, body) ->
      (* Emit the start code first, without 'variable' in scope. *)
      let start_val = codegen_expr start in
```

With this out of the way, the next step is to set up the LLVM basic block for the start of the loop body. In the case above, the whole loop body is one block, but remember that the body code itself could consist of multiple blocks (e.g. if it contains an if/then/else or a for/in expression).

```
(* Make the new basic block for the loop header, inserting after current
 * block. *)
let preheader_bb = insertion_block builder in
let the_function = block_parent preheader_bb in
let loop_bb = append_block context "loop" the_function in

(* Insert an explicit fall through from the current block to the
 * loop_bb. *)
ignore (build_br loop_bb builder);
```

This code is similar to what we saw for if/then/else. Because we will need it to create the Phi node, we remember the block that falls through into the loop. Once we have that, we create the actual block that starts the loop and create an unconditional branch for the fall-through between the two blocks.

```
(* Start insertion in loop_bb. *)
position_at_end loop_bb builder;
```

```
(* Start the PHI node with an entry for start. *)
let variable = build_phi [(start_val, preheader_bb)] var_name builder in
```

Now that the "preheader" for the loop is set up, we switch to emitting code for the loop body. To begin with, we move the insertion point and create the PHI node for the loop induction variable. Since we already know the incoming value for the starting value, we add it to the Phi node. Note that the Phi will eventually get a second value for the backedge, but we can't set it up yet (because it doesn't exist!).

```
(* Within the loop, the variable is defined equal to the PHI node. If it
 * shadows an existing variable, we have to restore it, so save it
 * now. *)
let old_val =
  try Some (Hashtbl.find named_values var_name) with Not_found -> None
in
Hashtbl.add named_values var_name variable;

(* Emit the body of the loop.  This, like any other expr, can change the
 * current BB.  Note that we ignore the value computed by the body, but
 * don't allow an error *)
ignore (codegen_expr body);
```

Now the code starts to get more interesting. Our 'for' loop introduces a new variable to the symbol table. This means that our symbol table can now contain either function arguments or loop variables. To handle this, before we codegen the body of the loop, we add the loop variable as the current value for its name. Note that it is possible that there is a variable of the same name in the outer scope. It would be easy to make this an error (emit an error and return null if there is already an entry for VarName) but we choose to allow shadowing of variables. In order to handle this correctly, we remember the Value that we are potentially shadowing in `old_val` (which will be None if there is no shadowed variable).

Once the loop variable is set into the symbol table, the code recursively codegen's the body. This allows the body to use the loop variable: any references to it will naturally find it in the symbol table.

```
(* Emit the step value. *)
let step_val =
  match step with
  | Some step -> codegen_expr step
  (* If not specified, use 1.0. *)
  | None -> const_float double_type 1.0
in

let next_var = build_add variable step_val "nextvar" builder in
```

Now that the body is emitted, we compute the next value of the iteration variable by adding the step value, or 1.0 if it isn't present. '`next_var`' will be the value of the loop variable on the next iteration of the loop.

```
(* Compute the end condition. *)
let end_cond = codegen_expr end_ in

(* Convert condition to a bool by comparing equal to 0.0. *)
let zero = const_float double_type 0.0 in
let end_cond = build_fcmp Fcmp.One end_cond zero "loopcond" builder in
```

Finally, we evaluate the exit value of the loop, to determine whether the loop should exit. This mirrors the condition evaluation for the if/then/else statement.

```
(* Create the "after loop" block and insert it. *)
let loop_end_bb = insertion_block builder in
let after_bb = append_block context "afterloop" the_function in
```

```
(* Insert the conditional branch into the end of loop_end_bb. *)
ignore (build_cond_br end_cond loop_bb after_bb builder);

(* Any new code will be inserted in after_bb. *)
position_at_end after_bb builder;
```

With the code for the body of the loop complete, we just need to finish up the control flow for it. This code remembers the end block (for the phi node), then creates the block for the loop exit ("afterloop"). Based on the value of the exit condition, it creates a conditional branch that chooses between executing the loop again and exiting the loop. Any future code is emitted in the "afterloop" block, so it sets the insertion position to it.

```
(* Add a new entry to the PHI node for the backedge. *)
add_incoming (next_var, loop_end_bb) variable;

(* Restore the unshadowed variable. *)
begin match old_val with
| Some old_val -> Hashtbl.add named_values var_name old_val
| None -> ()
end;

(* for expr always returns 0.0. *)
const_null double_type
```

The final code handles various cleanups: now that we have the "`next_var`" value, we can add the incoming value to the loop PHI node. After that, we remove the loop variable from the symbol table, so that it isn't in scope after the for loop. Finally, code generation of the for loop always returns 0.0, so that is what we return from `Codegen.codegen_expr`.

With this, we conclude the "adding control flow to Kaleidoscope" chapter of the tutorial. In this chapter we added two control flow constructs, and used them to motivate a couple of aspects of the LLVM IR that are important for front-end implementors to know. In the next chapter of our saga, we will get a bit crazier and add user-defined operators to our poor innocent language.

Full Code Listing

Here is the complete code listing for our running example, enhanced with the if/then/else and for expressions.. To build this example, use:

```
# Compile
ocamlbuild toy.byte
# Run
./toy.byte
```

Here is the code:

_tags:

```
<{lexer,parser}.ml>: use_camlp4, pp(camlp4of)
<*.{byte,native}>: g++, use_llvm, use_llvm_analysis
<*.{byte,native}>: use_llvm_executionengine, use_llvm_target
<*.{byte,native}>: use_llvm_scalar_opts, use_bindings
```

myocamlbuild.ml:

```
open Ocamlbuild_plugin;;

ocaml_lib ~extern:true "llvm";;
ocaml_lib ~extern:true "llvm_analysis";;
```

```
ocaml_lib ~extern:true "llvm_executionengine";;
ocaml_lib ~extern:true "llvm_target";;
ocaml_lib ~extern:true "llvm_scalar_opts";;

flag ["link"; "ocaml"; "g++"] (S[A"-cc"; A"g++"]);;
dep ["link"; "ocaml"; "use_bindings"] ["bindings.o"];;
```

token.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer Tokens
 *===----------------------------------------------------------------------===*)

(* The lexer returns these 'Kwd' if it is an unknown character, otherwise one of
 * these others for known things. *)
type token =
  (* commands *)
  | Def | Extern

  (* primary *)
  | Ident of string | Number of float

  (* unknown *)
  | Kwd of char

  (* control *)
  | If | Then | Else
  | For | In
```

lexer.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer
 *===----------------------------------------------------------------------===*)

let rec lex = parser
  (* Skip any whitespace. *)
  | [< ' (' ' | '\n' | '\r' | '\t'); stream >] -> lex stream

  (* identifier: [a-zA-Z][a-zA-Z0-9] *)
  | [< ' ('A' .. 'Z' | 'a' .. 'z' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_ident buffer stream

  (* number: [0-9.]+ *)
  | [< ' ('0' .. '9' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_number buffer stream

  (* Comment until end of line. *)
  | [< ' ('#'); stream >] ->
      lex_comment stream

  (* Otherwise, just return the character as its ascii value. *)
  | [< 'c; stream >] ->
      [< 'Token.Kwd c; lex stream >]
```

```
    (* end of stream. *)
    | [< >] -> [< >]

and lex_number buffer = parser
    | [< ' ('0' .. '9' | '.' as c); stream >] ->
        Buffer.add_char buffer c;
        lex_number buffer stream
    | [< stream=lex >] ->
        [< 'Token.Number (float_of_string (Buffer.contents buffer)); stream >]

and lex_ident buffer = parser
    | [< ' ('A' .. 'Z' | 'a' .. 'z' | '0' .. '9' as c); stream >] ->
        Buffer.add_char buffer c;
        lex_ident buffer stream
    | [< stream=lex >] ->
        match Buffer.contents buffer with
        | "def" -> [< 'Token.Def; stream >]
        | "extern" -> [< 'Token.Extern; stream >]
        | "if" -> [< 'Token.If; stream >]
        | "then" -> [< 'Token.Then; stream >]
        | "else" -> [< 'Token.Else; stream >]
        | "for" -> [< 'Token.For; stream >]
        | "in" -> [< 'Token.In; stream >]
        | id -> [< 'Token.Ident id; stream >]

and lex_comment = parser
    | [< ' ('\n'); stream=lex >] -> stream
    | [< 'c; e=lex_comment >] -> e
    | [< >] -> [< >]
```

ast.ml:

```
(*===----------------------------------------------------------------------===
 * Abstract Syntax Tree (aka Parse Tree)
 *===----------------------------------------------------------------------===*)

(* expr - Base type for all expression nodes. *)
type expr =
    (* variant for numeric literals like "1.0". *)
    | Number of float

    (* variant for referencing a variable, like "a". *)
    | Variable of string

    (* variant for a binary operator. *)
    | Binary of char * expr * expr

    (* variant for function calls. *)
    | Call of string * expr array

    (* variant for if/then/else. *)
    | If of expr * expr * expr

    (* variant for for/in. *)
    | For of string * expr * expr * expr option * expr

(* proto - This type represents the "prototype" for a function, which captures
 * its name, and its argument names (thus implicitly the number of arguments the
```

```
 * function takes). *)
type proto = Prototype of string * string array

(* func - This type represents a function definition itself. *)
type func = Function of proto * expr
```

parser.ml:

```
(*===---------------------------------------------------------------------===
 * Parser
 *===---------------------------------------------------------------------===*)

(* binop_precedence - This holds the precedence for each binary operator that is
 * defined *)
let binop_precedence:(char, int) Hashtbl.t = Hashtbl.create 10

(* precedence - Get the precedence of the pending binary operator token. *)
let precedence c = try Hashtbl.find binop_precedence c with Not_found -> -1

(* primary
 *   ::= identifier
 *   ::= numberexpr
 *   ::= parenexpr
 *   ::= ifexpr
 *   ::= forexpr *)
let rec parse_primary = parser
  (* numberexpr ::= number *)
  | [< 'Token.Number n >] -> Ast.Number n

  (* parenexpr ::= '(' expression ')' *)
  | [< 'Token.Kwd '('; e=parse_expr; 'Token.Kwd ')' ?? "expected ')'" >] -> e

  (* identifierexpr
   *   ::= identifier
   *   ::= identifier '(' argumentexpr ')' *)
  | [< 'Token.Ident id; stream >] ->
      let rec parse_args accumulator = parser
        | [< e=parse_expr; stream >] ->
            begin parser
              | [< 'Token.Kwd ','; e=parse_args (e :: accumulator) >] -> e
              | [< >] -> e :: accumulator
            end stream
        | [< >] -> accumulator
      in
      let rec parse_ident id = parser
        (* Call. *)
        | [< 'Token.Kwd '(';
             args=parse_args [];
             'Token.Kwd ')' ?? "expected ')'">] ->
            Ast.Call (id, Array.of_list (List.rev args))

        (* Simple variable ref. *)
        | [< >] -> Ast.Variable id
      in
      parse_ident id stream

  (* ifexpr ::= 'if' expr 'then' expr 'else' expr *)
  | [< 'Token.If; c=parse_expr;
```

```
        'Token.Then ?? "expected 'then'"; t=parse_expr;
        'Token.Else ?? "expected 'else'"; e=parse_expr >] ->
      Ast.If (c, t, e)

  (* forexpr
       ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression *)
  | [< 'Token.For;
       'Token.Ident id ?? "expected identifier after for";
       'Token.Kwd '=' ?? "expected '=' after for";
       stream >] ->
      begin parser
        | [<
             start=parse_expr;
             'Token.Kwd ',' ?? "expected ',' after for";
             end_=parse_expr;
             stream >] ->
            let step =
              begin parser
              | [< 'Token.Kwd ','; step=parse_expr >] -> Some step
              | [< >] -> None
              end stream
            in
            begin parser
            | [< 'Token.In; body=parse_expr >] ->
                Ast.For (id, start, end_, step, body)
            | [< >] ->
                raise (Stream.Error "expected 'in' after for")
            end stream
        | [< >] ->
            raise (Stream.Error "expected '=' after for")
      end stream

  | [< >] -> raise (Stream.Error "unknown token when expecting an expression.")

(* binoprhs
 *   ::= ('+' primary)* *)
and parse_bin_rhs expr_prec lhs stream =
  match Stream.peek stream with
  (* If this is a binop, find its precedence. *)
  | Some (Token.Kwd c) when Hashtbl.mem binop_precedence c ->
      let token_prec = precedence c in

      (* If this is a binop that binds at least as tightly as the current binop,
       * consume it, otherwise we are done. *)
      if token_prec < expr_prec then lhs else begin
        (* Eat the binop. *)
        Stream.junk stream;

        (* Parse the primary expression after the binary operator. *)
        let rhs = parse_primary stream in

        (* Okay, we know this is a binop. *)
        let rhs =
          match Stream.peek stream with
          | Some (Token.Kwd c2) ->
              (* If BinOp binds less tightly with rhs than the operator after
               * rhs, let the pending operator take rhs as its lhs. *)
              let next_prec = precedence c2 in
```

```
                if token_prec < next_prec
                then parse_bin_rhs (token_prec + 1) rhs stream
                else rhs
            | _ -> rhs
          in

          (* Merge lhs/rhs. *)
          let lhs = Ast.Binary (c, lhs, rhs) in
          parse_bin_rhs expr_prec lhs stream
        end
    | _ -> lhs

(* expression
 *   ::= primary binoprhs *)
and parse_expr = parser
  | [< lhs=parse_primary; stream >] -> parse_bin_rhs 0 lhs stream

(* prototype
 *   ::= id '(' id* ')' *)
let parse_prototype =
  let rec parse_args accumulator = parser
    | [< 'Token.Ident id; e=parse_args (id::accumulator) >] -> e
    | [< >] -> accumulator
  in

  parser
  | [< 'Token.Ident id;
       'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      (* success. *)
      Ast.Prototype (id, Array.of_list (List.rev args))

  | [< >] ->
      raise (Stream.Error "expected function name in prototype")

(* definition ::= 'def' prototype expression *)
let parse_definition = parser
  | [< 'Token.Def; p=parse_prototype; e=parse_expr >] ->
      Ast.Function (p, e)

(* toplevelexpr ::= expression *)
let parse_toplevel = parser
  | [< e=parse_expr >] ->
      (* Make an anonymous proto. *)
      Ast.Function (Ast.Prototype ("", [||]), e)

(*  external ::= 'extern' prototype *)
let parse_extern = parser
  | [< 'Token.Extern; e=parse_prototype >] -> e
```

codegen.ml:

```
(*===----------------------------------------------------------------------===
 * Code Generation
 *===----------------------------------------------------------------------===*)

open Llvm
```

```
exception Error of string

let context = global_context ()
let the_module = create_module context "my cool jit"
let builder = builder context
let named_values:(string, llvalue) Hashtbl.t = Hashtbl.create 10
let double_type = double_type context

let rec codegen_expr = function
  | Ast.Number n -> const_float double_type n
  | Ast.Variable name ->
      (try Hashtbl.find named_values name with
        | Not_found -> raise (Error "unknown variable name"))
  | Ast.Binary (op, lhs, rhs) ->
      let lhs_val = codegen_expr lhs in
      let rhs_val = codegen_expr rhs in
      begin
        match op with
        | '+' -> build_add lhs_val rhs_val "addtmp" builder
        | '-' -> build_sub lhs_val rhs_val "subtmp" builder
        | '*' -> build_mul lhs_val rhs_val "multmp" builder
        | '<' ->
            (* Convert bool 0/1 to double 0.0 or 1.0 *)
            let i = build_fcmp Fcmp.Ult lhs_val rhs_val "cmptmp" builder in
            build_uitofp i double_type "booltmp" builder
        | _ -> raise (Error "invalid binary operator")
      end
  | Ast.Call (callee, args) ->
      (* Look up the name in the module table. *)
      let callee =
        match lookup_function callee the_module with
        | Some callee -> callee
        | None -> raise (Error "unknown function referenced")
      in
      let params = params callee in

      (* If argument mismatch error. *)
      if Array.length params == Array.length args then () else
        raise (Error "incorrect # arguments passed");
      let args = Array.map codegen_expr args in
      build_call callee args "calltmp" builder
  | Ast.If (cond, then_, else_) ->
      let cond = codegen_expr cond in

      (* Convert condition to a bool by comparing equal to 0.0 *)
      let zero = const_float double_type 0.0 in
      let cond_val = build_fcmp Fcmp.One cond zero "ifcond" builder in

      (* Grab the first block so that we might later add the conditional branch
       * to it at the end of the function. *)
      let start_bb = insertion_block builder in
      let the_function = block_parent start_bb in

      let then_bb = append_block context "then" the_function in

      (* Emit 'then' value. *)
      position_at_end then_bb builder;
      let then_val = codegen_expr then_ in
```

```
      (* Codegen of 'then' can change the current block, update then_bb for the
       * phi. We create a new name because one is used for the phi node, and the
       * other is used for the conditional branch. *)
      let new_then_bb = insertion_block builder in

      (* Emit 'else' value. *)
      let else_bb = append_block context "else" the_function in
      position_at_end else_bb builder;
      let else_val = codegen_expr else_ in

      (* Codegen of 'else' can change the current block, update else_bb for the
       * phi. *)
      let new_else_bb = insertion_block builder in

      (* Emit merge block. *)
      let merge_bb = append_block context "ifcont" the_function in
      position_at_end merge_bb builder;
      let incoming = [(then_val, new_then_bb); (else_val, new_else_bb)] in
      let phi = build_phi incoming "iftmp" builder in

      (* Return to the start block to add the conditional branch. *)
      position_at_end start_bb builder;
      ignore (build_cond_br cond_val then_bb else_bb builder);

      (* Set a unconditional branch at the end of the 'then' block and the
       * 'else' block to the 'merge' block. *)
      position_at_end new_then_bb builder; ignore (build_br merge_bb builder);
      position_at_end new_else_bb builder; ignore (build_br merge_bb builder);

      (* Finally, set the builder to the end of the merge block. *)
      position_at_end merge_bb builder;

      phi
  | Ast.For (var_name, start, end_, step, body) ->
      (* Emit the start code first, without 'variable' in scope. *)
      let start_val = codegen_expr start in

      (* Make the new basic block for the loop header, inserting after current
       * block. *)
      let preheader_bb = insertion_block builder in
      let the_function = block_parent preheader_bb in
      let loop_bb = append_block context "loop" the_function in

      (* Insert an explicit fall through from the current block to the
       * loop_bb. *)
      ignore (build_br loop_bb builder);

      (* Start insertion in loop_bb. *)
      position_at_end loop_bb builder;

      (* Start the PHI node with an entry for start. *)
      let variable = build_phi [(start_val, preheader_bb)] var_name builder in

      (* Within the loop, the variable is defined equal to the PHI node. If it
       * shadows an existing variable, we have to restore it, so save it
       * now. *)
      let old_val =
        try Some (Hashtbl.find named_values var_name) with Not_found -> None
```

```
        in
        Hashtbl.add named_values var_name variable;

        (* Emit the body of the loop.  This, like any other expr, can change the
         * current BB.  Note that we ignore the value computed by the body, but
         * don't allow an error *)
        ignore (codegen_expr body);

        (* Emit the step value. *)
        let step_val =
          match step with
            | Some step -> codegen_expr step
            (* If not specified, use 1.0. *)
            | None -> const_float double_type 1.0
        in

        let next_var = build_add variable step_val "nextvar" builder in

        (* Compute the end condition. *)
        let end_cond = codegen_expr end_ in

        (* Convert condition to a bool by comparing equal to 0.0. *)
        let zero = const_float double_type 0.0 in
        let end_cond = build_fcmp Fcmp.One end_cond zero "loopcond" builder in

        (* Create the "after loop" block and insert it. *)
        let loop_end_bb = insertion_block builder in
        let after_bb = append_block context "afterloop" the_function in

        (* Insert the conditional branch into the end of loop_end_bb. *)
        ignore (build_cond_br end_cond loop_bb after_bb builder);

        (* Any new code will be inserted in after_bb. *)
        position_at_end after_bb builder;

        (* Add a new entry to the PHI node for the backedge. *)
        add_incoming (next_var, loop_end_bb) variable;

        (* Restore the unshadowed variable. *)
        begin match old_val with
        | Some old_val -> Hashtbl.add named_values var_name old_val
        | None -> ()
        end;

        (* for expr always returns 0.0. *)
        const_null double_type

    let codegen_proto = function
      | Ast.Prototype (name, args) ->
        (* Make the function type: double(double,double) etc. *)
        let doubles = Array.make (Array.length args) double_type in
        let ft = function_type double_type doubles in
        let f =
          match lookup_function name the_module with
            | None -> declare_function name ft the_module

          (* If 'f' conflicted, there was already something named 'name'. If it
           * has a body, don't allow redefinition or reextern. *)
```

```
      | Some f ->
          (* If 'f' already has a body, reject this. *)
          if block_begin f <> At_end f then
            raise (Error "redefinition of function");

          (* If 'f' took a different number of arguments, reject. *)
          if element_type (type_of f) <> ft then
            raise (Error "redefinition of function with different # args");
          f
    in

    (* Set names for all arguments. *)
    Array.iteri (fun i a ->
      let n = args.(i) in
      set_value_name n a;
      Hashtbl.add named_values n a;
    ) (params f);
    f

let codegen_func the_fpm = function
  | Ast.Function (proto, body) ->
      Hashtbl.clear named_values;
      let the_function = codegen_proto proto in

      (* Create a new basic block to start insertion into. *)
      let bb = append_block context "entry" the_function in
      position_at_end bb builder;

      try
        let ret_val = codegen_expr body in

        (* Finish off the function. *)
        let _ = build_ret ret_val builder in

        (* Validate the generated code, checking for consistency. *)
        Llvm_analysis.assert_valid_function the_function;

        (* Optimize the function. *)
        let _ = PassManager.run_function the_function the_fpm in

        the_function
      with e ->
        delete_function the_function;
        raise e
```

toplevel.ml:

```
(*===----------------------------------------------------------------------===
 * Top-Level parsing and JIT Driver
 *===----------------------------------------------------------------------===*)

open Llvm
open Llvm_executionengine

(* top ::= definition | external | expression | ';' *)
let rec main_loop the_fpm the_execution_engine stream =
  match Stream.peek stream with
  | None -> ()
```

```
      (* ignore top-level semicolons. *)
      | Some (Token.Kwd ';') ->
          Stream.junk stream;
          main_loop the_fpm the_execution_engine stream

      | Some token ->
          begin
            try match token with
            | Token.Def ->
                let e = Parser.parse_definition stream in
                print_endline "parsed a function definition.";
                dump_value (Codegen.codegen_func the_fpm e);
            | Token.Extern ->
                let e = Parser.parse_extern stream in
                print_endline "parsed an extern.";
                dump_value (Codegen.codegen_proto e);
            | _ ->
                (* Evaluate a top-level expression into an anonymous function. *)
                let e = Parser.parse_toplevel stream in
                print_endline "parsed a top-level expr";
                let the_function = Codegen.codegen_func the_fpm e in
                dump_value the_function;

                (* JIT the function, returning a function pointer. *)
                let result = ExecutionEngine.run_function the_function [||]
                  the_execution_engine in

                print_string "Evaluated to ";
                print_float (GenericValue.as_float Codegen.double_type result);
                print_newline ();
            with Stream.Error s | Codegen.Error s ->
              (* Skip token for error recovery. *)
              Stream.junk stream;
              print_endline s;
          end;
          print_string "ready> "; flush stdout;
          main_loop the_fpm the_execution_engine stream
```

toy.ml:

```
    (*===----------------------------------------------------------------------===
     * Main driver code.
     *===----------------------------------------------------------------------===*)

    open Llvm
    open Llvm_executionengine
    open Llvm_target
    open Llvm_scalar_opts

    let main () =
      ignore (initialize_native_target ());

      (* Install standard binary operators.
       * 1 is the lowest precedence. *)
      Hashtbl.add Parser.binop_precedence '<' 10;
      Hashtbl.add Parser.binop_precedence '+' 20;
      Hashtbl.add Parser.binop_precedence '-' 20;
      Hashtbl.add Parser.binop_precedence '*' 40;    (* highest. *)
```

```
  (* Prime the first token. *)
  print_string "ready> "; flush stdout;
  let stream = Lexer.lex (Stream.of_channel stdin) in

  (* Create the JIT. *)
  let the_execution_engine = ExecutionEngine.create Codegen.the_module in
  let the_fpm = PassManager.create_function Codegen.the_module in

  (* Set up the optimizer pipeline.  Start with registering info about how the
   * target lays out data structures. *)
  DataLayout.add (ExecutionEngine.target_data the_execution_engine) the_fpm;

  (* Do simple "peephole" optimizations and bit-twiddling optzn. *)
  add_instruction_combination the_fpm;

  (* reassociate expressions. *)
  add_reassociation the_fpm;

  (* Eliminate Common SubExpressions. *)
  add_gvn the_fpm;

  (* Simplify the control flow graph (deleting unreachable blocks, etc). *)
  add_cfg_simplification the_fpm;

  ignore (PassManager.initialize the_fpm);

  (* Run the main "interpreter loop" now. *)
  Toplevel.main_loop the_fpm the_execution_engine stream;

  (* Print out all the generated code. *)
  dump_module Codegen.the_module
;;

main ()
```

bindings.c

```
#include <stdio.h>

/* putchard - putchar that takes a double and returns 0. */
extern double putchard(double X) {
  putchar((char)X);
  return 0;
}
```

Next: Extending the language: user-defined operators

Kaleidoscope: Extending the Language: User-defined Operators

Chapter 6 Introduction

Welcome to Chapter 6 of the "Implementing a language with LLVM" tutorial. At this point in our tutorial, we now have a fully functional language that is fairly minimal, but also useful. There is still one big problem with it, however. Our language doesn't have many useful operators (like division, logical negation, or even any comparisons besides less-than).

This chapter of the tutorial takes a wild digression into adding user-defined operators to the simple and beautiful Kaleidoscope language. This digression now gives us a simple and ugly language in some ways, but also a powerful one at the same time. One of the great things about creating your own language is that you get to decide what is good or bad. In this tutorial we'll assume that it is okay to use this as a way to show some interesting parsing techniques.

At the end of this tutorial, we'll run through an example Kaleidoscope application that renders the Mandelbrot set. This gives an example of what you can build with Kaleidoscope and its feature set.

User-defined Operators: the Idea

The "operator overloading" that we will add to Kaleidoscope is more general than languages like C++. In C++, you are only allowed to redefine existing operators: you can't programatically change the grammar, introduce new operators, change precedence levels, etc. In this chapter, we will add this capability to Kaleidoscope, which will let the user round out the set of operators that are supported.

The point of going into user-defined operators in a tutorial like this is to show the power and flexibility of using a hand-written parser. Thus far, the parser we have been implementing uses recursive descent for most parts of the grammar and operator precedence parsing for the expressions. See Chapter 2 for details. Without using operator precedence parsing, it would be very difficult to allow the programmer to introduce new operators into the grammar: the grammar is dynamically extensible as the JIT runs.

The two specific features we'll add are programmable unary operators (right now, Kaleidoscope has no unary operators at all) as well as binary operators. An example of this is:

```
# Logical unary not.
def unary!(v)
  if v then
    0
  else
    1;

# Define > with the same precedence as <.
def binary> 10 (LHS RHS)
  RHS < LHS;

# Binary "logical or", (note that it does not "short circuit")
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;

# Define = with slightly lower precedence than relationals.
def binary= 9 (LHS RHS)
  !(LHS < RHS | LHS > RHS);
```

Many languages aspire to being able to implement their standard runtime library in the language itself. In Kaleidoscope, we can implement significant parts of the language in the library!

We will break down implementation of these features into two parts: implementing support for user-defined binary operators and adding unary operators.

User-defined Binary Operators

Adding support for user-defined binary operators is pretty simple with our current framework. We'll first add support for the unary/binary keywords:

```
type token =
  ...
  (* operators *)
  | Binary | Unary

...

and lex_ident buffer = parser
  ...
      | "for" -> [< 'Token.For; stream >]
      | "in" -> [< 'Token.In; stream >]
      | "binary" -> [< 'Token.Binary; stream >]
      | "unary" -> [< 'Token.Unary; stream >]
```

This just adds lexer support for the unary and binary keywords, like we did in previous chapters. One nice thing about our current AST, is that we represent binary operators with full generalisation by using their ASCII code as the opcode. For our extended operators, we'll use this same representation, so we don't need any new AST or parser support.

On the other hand, we have to be able to represent the definitions of these new operators, in the "def binary| 5" part of the function definition. In our grammar so far, the "name" for the function definition is parsed as the "prototype" production and into the `Ast.Prototype` AST node. To represent our new user-defined operators as prototypes, we have to extend the `Ast.Prototype` AST node like this:

```
(* proto - This type represents the "prototype" for a function, which captures
 * its name, and its argument names (thus implicitly the number of arguments the
 * function takes). *)
type proto =
  | Prototype of string * string array
  | BinOpPrototype of string * string array * int
```

Basically, in addition to knowing a name for the prototype, we now keep track of whether it was an operator, and if it was, what precedence level the operator is at. The precedence is only used for binary operators (as you'll see below, it just doesn't apply for unary operators). Now that we have a way to represent the prototype for a user-defined operator, we need to parse it:

```
(* prototype
 *   ::= id '(' id* ')'
 *   ::= binary LETTER number? (id, id)
 *   ::= unary LETTER number? (id) *)
let parse_prototype =
  let rec parse_args accumulator = parser
    | [< 'Token.Ident id; e=parse_args (id::accumulator) >] -> e
    | [< >] -> accumulator
  in
  let parse_operator = parser
    | [< 'Token.Unary >] -> "unary", 1
    | [< 'Token.Binary >] -> "binary", 2
  in
  let parse_binary_precedence = parser
```

```
    | [< 'Token.Number n >] -> int_of_float n
    | [< >] -> 30
  in
  parser
  | [< 'Token.Ident id;
       'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      (* success. *)
      Ast.Prototype (id, Array.of_list (List.rev args))
  | [< (prefix, kind)=parse_operator;
       'Token.Kwd op ?? "expected an operator";
       (* Read the precedence if present. *)
       binary_precedence=parse_binary_precedence;
       'Token.Kwd '(' ?? "expected '(' in prototype";
        args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      let name = prefix ^ (String.make 1 op) in
      let args = Array.of_list (List.rev args) in

      (* Verify right number of arguments for operator. *)
      if Array.length args != kind
      then raise (Stream.Error "invalid number of operands for operator")
      else
        if kind == 1 then
          Ast.Prototype (name, args)
        else
          Ast.BinOpPrototype (name, args, binary_precedence)
  | [< >] ->
      raise (Stream.Error "expected function name in prototype")
```

This is all fairly straightforward parsing code, and we have already seen a lot of similar code in the past. One interesting part about the code above is the couple lines that set up `name` for binary operators. This builds names like "binary@" for a newly defined "@" operator. This then takes advantage of the fact that symbol names in the LLVM symbol table are allowed to have any character in them, including embedded nul characters.

The next interesting thing to add, is codegen support for these binary operators. Given our current structure, this is a simple addition of a default case for our existing binary operator node:

```
let codegen_expr = function
  ...
  | Ast.Binary (op, lhs, rhs) ->
      let lhs_val = codegen_expr lhs in
      let rhs_val = codegen_expr rhs in
      begin
        match op with
        | '+' -> build_add lhs_val rhs_val "addtmp" builder
        | '-' -> build_sub lhs_val rhs_val "subtmp" builder
        | '*' -> build_mul lhs_val rhs_val "multmp" builder
        | '<' ->
            (* Convert bool 0/1 to double 0.0 or 1.0 *)
            let i = build_fcmp Fcmp.Ult lhs_val rhs_val "cmptmp" builder in
            build_uitofp i double_type "booltmp" builder
        | _ ->
            (* If it wasn't a builtin binary operator, it must be a user defined
             * one. Emit a call to it. *)
            let callee = "binary" ^ (String.make 1 op) in
            let callee =
```

```
            match lookup_function callee the_module with
            | Some callee -> callee
            | None -> raise (Error "binary operator not found!")
          in
          build_call callee [|lhs_val; rhs_val|] "binop" builder
      end
```

As you can see above, the new code is actually really simple. It just does a lookup for the appropriate operator in the symbol table and generates a function call to it. Since user-defined operators are just built as normal functions (because the "prototype" boils down to a function with the right name) everything falls into place.

The final piece of code we are missing, is a bit of top level magic:

```
let codegen_func the_fpm = function
  | Ast.Function (proto, body) ->
      Hashtbl.clear named_values;
      let the_function = codegen_proto proto in

      (* If this is an operator, install it. *)
      begin match proto with
      | Ast.BinOpPrototype (name, args, prec) ->
          let op = name.[String.length name - 1] in
          Hashtbl.add Parser.binop_precedence op prec;
      | _ -> ()
      end;

      (* Create a new basic block to start insertion into. *)
      let bb = append_block context "entry" the_function in
      position_at_end bb builder;
      ...
```

Basically, before codegening a function, if it is a user-defined operator, we register it in the precedence table. This allows the binary operator parsing logic we already have in place to handle it. Since we are working on a fully-general operator precedence parser, this is all we need to do to "extend the grammar".

Now we have useful user-defined binary operators. This builds a lot on the previous framework we built for other operators. Adding unary operators is a bit more challenging, because we don't have any framework for it yet - lets see what it takes.

User-defined Unary Operators

Since we don't currently support unary operators in the Kaleidoscope language, we'll need to add everything to support them. Above, we added simple support for the 'unary' keyword to the lexer. In addition to that, we need an AST node:

```
type expr =
  ...
  (* variant for a unary operator. *)
  | Unary of char * expr
  ...
```

This AST node is very simple and obvious by now. It directly mirrors the binary operator AST node, except that it only has one child. With this, we need to add the parsing logic. Parsing a unary operator is pretty simple: we'll add a new function to do it:

```
(* unary
 *   ::= primary
 *   ::= '!' unary *)
and parse_unary = parser
```

```
  (* If this is a unary operator, read it. *)
  | [< 'Token.Kwd op when op != '(' && op != ')'; operand=parse_expr >] ->
      Ast.Unary (op, operand)

  (* If the current token is not an operator, it must be a primary expr. *)
  | [< stream >] -> parse_primary stream
```

The grammar we add is pretty straightforward here. If we see a unary operator when parsing a primary operator, we eat the operator as a prefix and parse the remaining piece as another unary operator. This allows us to handle multiple unary operators (e.g. "!!x"). Note that unary operators can't have ambiguous parses like binary operators can, so there is no need for precedence information.

The problem with this function, is that we need to call ParseUnary from somewhere. To do this, we change previous callers of ParsePrimary to call `parse_unary` instead:

```
(* binoprhs
 *   ::= ('+' primary)* *)
and parse_bin_rhs expr_prec lhs stream =
        ...
        (* Parse the unary expression after the binary operator. *)
        let rhs = parse_unary stream in
        ...

...

(* expression
 *   ::= primary binoprhs *)
and parse_expr = parser
  | [< lhs=parse_unary; stream >] -> parse_bin_rhs 0 lhs stream
```

With these two simple changes, we are now able to parse unary operators and build the AST for them. Next up, we need to add parser support for prototypes, to parse the unary operator prototype. We extend the binary operator code above with:

```
(* prototype
 *   ::= id '(' id* ')'
 *   ::= binary LETTER number? (id, id)
 *   ::= unary LETTER number? (id) *)
let parse_prototype =
  let rec parse_args accumulator = parser
    | [< 'Token.Ident id; e=parse_args (id::accumulator) >] -> e
    | [< >] -> accumulator
  in
  let parse_operator = parser
    | [< 'Token.Unary >] -> "unary", 1
    | [< 'Token.Binary >] -> "binary", 2
  in
  let parse_binary_precedence = parser
    | [< 'Token.Number n >] -> int_of_float n
    | [< >] -> 30
  in
  parser
  | [< 'Token.Ident id;
       'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      (* success. *)
      Ast.Prototype (id, Array.of_list (List.rev args))
```

```
  | [< (prefix, kind)=parse_operator;
      'Token.Kwd op ?? "expected an operator";
      (* Read the precedence if present. *)
      binary_precedence=parse_binary_precedence;
      'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
      'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      let name = prefix ^ (String.make 1 op) in
      let args = Array.of_list (List.rev args) in

      (* Verify right number of arguments for operator. *)
      if Array.length args != kind
      then raise (Stream.Error "invalid number of operands for operator")
      else
        if kind == 1 then
          Ast.Prototype (name, args)
        else
          Ast.BinOpPrototype (name, args, binary_precedence)
  | [< >] ->
      raise (Stream.Error "expected function name in prototype")
```

As with binary operators, we name unary operators with a name that includes the operator character. This assists us at code generation time. Speaking of, the final piece we need to add is codegen support for unary operators. It looks like this:

```
let rec codegen_expr = function
  ...
  | Ast.Unary (op, operand) ->
      let operand = codegen_expr operand in
      let callee = "unary" ^ (String.make 1 op) in
      let callee =
        match lookup_function callee the_module with
        | Some callee -> callee
        | None -> raise (Error "unknown unary operator")
      in
      build_call callee [|operand|] "unop" builder
```

This code is similar to, but simpler than, the code for binary operators. It is simpler primarily because it doesn't need to handle any predefined operators.

Kicking the Tires

It is somewhat hard to believe, but with a few simple extensions we've covered in the last chapters, we have grown a real-ish language. With this, we can do a lot of interesting things, including I/O, math, and a bunch of other things. For example, we can now add a nice sequencing operator (printd is defined to print out the specified value and a newline):

```
ready> extern printd(x);
Read extern: declare double @printd(double)
ready> def binary : 1 (x y) 0;  # Low-precedence operator that ignores operands.
..
ready> printd(123) : printd(456) : printd(789);
123.000000
456.000000
789.000000
Evaluated to 0.000000
```

We can also define a bunch of other "primitive" operations, such as:

```
# Logical unary not.
def unary!(v)
  if v then
    0
  else
    1;

# Unary negate.
def unary-(v)
  0-v;

# Define > with the same precedence as <.
def binary> 10 (LHS RHS)
  RHS < LHS;

# Binary logical or, which does not short circuit.
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;

# Binary logical and, which does not short circuit.
def binary& 6 (LHS RHS)
  if !LHS then
    0
  else
    !!RHS;

# Define = with slightly lower precedence than relationals.
def binary = 9 (LHS RHS)
  !(LHS < RHS | LHS > RHS);
```

Given the previous if/then/else support, we can also define interesting functions for I/O. For example, the following prints out a character whose "density" reflects the value passed in: the lower the value, the denser the character:

```
ready>

extern putchard(char)
def printdensity(d)
  if d > 8 then
    putchard(32)  # ' '
  else if d > 4 then
    putchard(46)  # '.'
  else if d > 2 then
    putchard(43)  # '+'
  else
    putchard(42); # '*'
...
ready> printdensity(1): printdensity(2): printdensity(3) :
          printdensity(4): printdensity(5): printdensity(9): putchard(10);
*++..
Evaluated to 0.000000
```

Based on these simple primitive operations, we can start to define more interesting things. For example, here's a little function that solves for the number of iterations it takes a function in the complex plane to converge:

```
# determine whether the specific location diverges.
# Solve for z = z^2 + c in the complex plane.
def mandleconverger(real imag iters creal cimag)
  if iters > 255 | (real*real + imag*imag > 4) then
    iters
  else
    mandleconverger(real*real - imag*imag + creal,
                    2*real*imag + cimag,
                    iters+1, creal, cimag);

# return the number of iterations required for the iteration to escape
def mandleconverge(real imag)
  mandleconverger(real, imag, 0, real, imag);
```

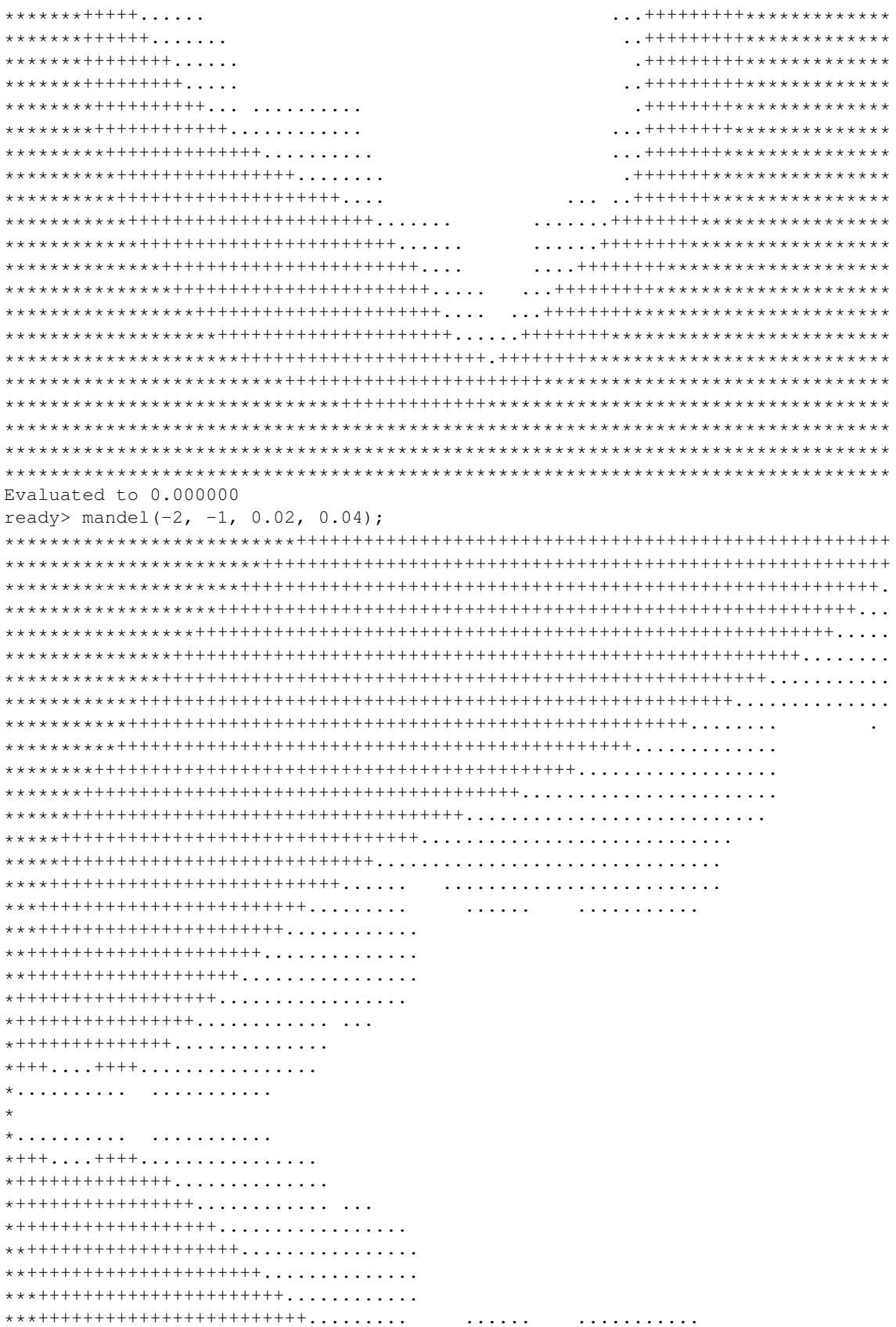
This "z = z^2 + c" function is a beautiful little creature that is the basis for computation of the Mandelbrot Set. Our `mandelconverge` function returns the number of iterations that it takes for a complex orbit to escape, saturating to 255. This is not a very useful function by itself, but if you plot its value over a two-dimensional plane, you can see the Mandelbrot set. Given that we are limited to using putchard here, our amazing graphical output is limited, but we can whip together something using the density plotter above:

```
# compute and plot the mandlebrot set with the specified 2 dimensional range
# info.
def mandelhelp(xmin xmax xstep   ymin ymax ystep)
  for y = ymin, y < ymax, ystep in (
    (for x = xmin, x < xmax, xstep in
       printdensity(mandleconverge(x,y)))
    : putchard(10)
  )

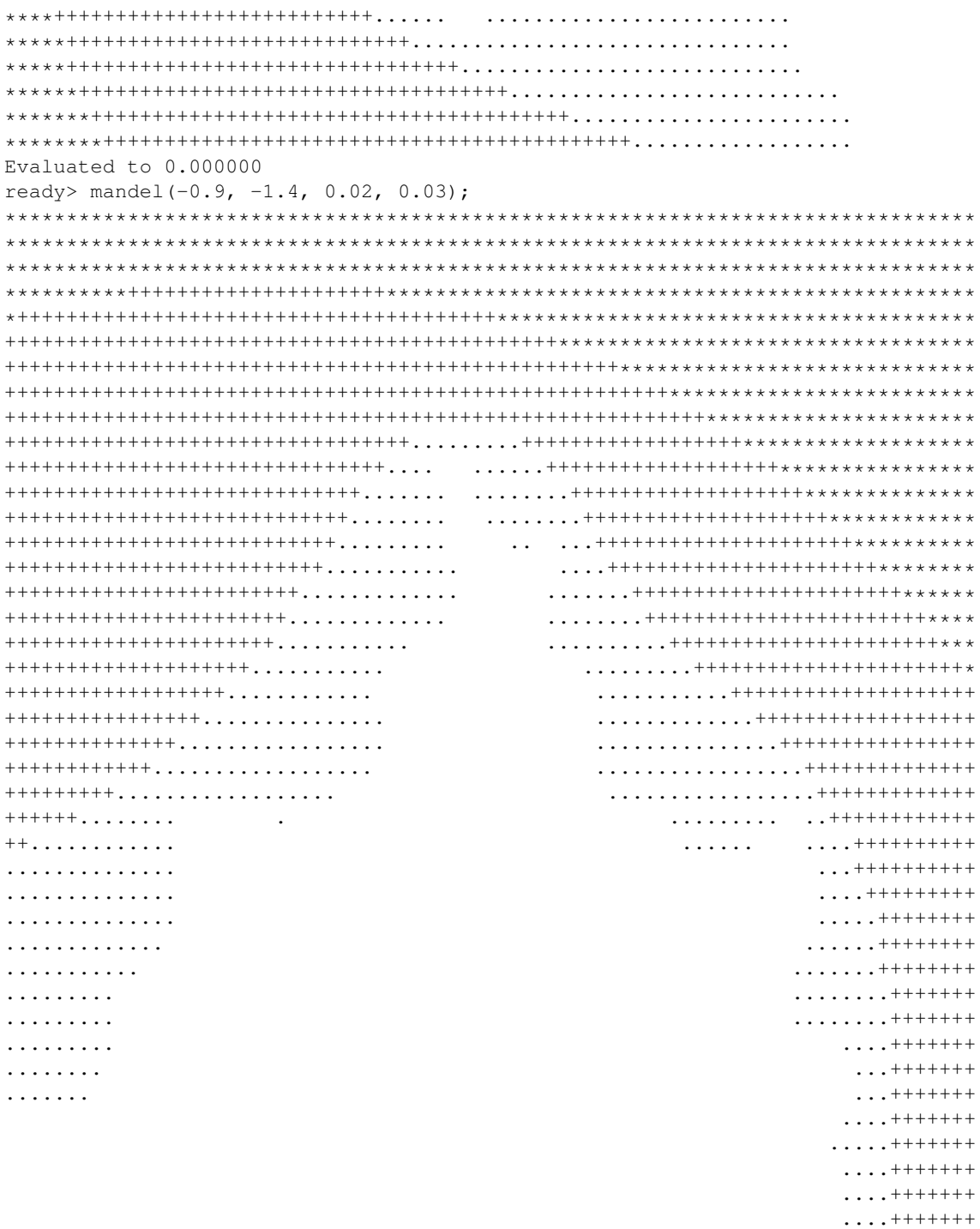
# mandel - This is a convenient helper function for plotting the mandelbrot set
# from the specified position with the specified Magnification.
def mandel(realstart imagstart realmag imagmag)
  mandelhelp(realstart, realstart+realmag*78, realmag,
             imagstart, imagstart+imagmag*40, imagmag);
```

Given this, we can try plotting out the mandlebrot set! Lets try it out:

```
ready> mandel(-2.3, -1.3, 0.05, 0.07);
**************************************************************************************
**************************************************************************************
**************************************************************************************
**********************+++++++++++++++++++++.. ...++++++++++++++++++++++++++++++++++
*****************************************+++.... ...++++++++++++++++++++++++++++++++
*************************************++++++.....   ...++++++++++++++++++++++++++++++
************************************++++++....     ....++++++++++++++++++++++++++++
**********************++++++++++++++++++++++......      .....++++++++++++++++++++++++
*********************+++++++++++++++++++++++.......       .......++++++++++++++++++++++
*******************************+++++++++++++++++....               ... .++++++++++++++++++++++
****************************+++++++++++++++++++.......                    .++++++++++++++++++++
************************+++++++++++++++++++++...........                  ...++++++++++++++++++++
********************+++++++++++++++++++++...........                     ...++++++++++++++++++++
******************+++++++++++++++++++... ..........                       .++++++++++++++++++++
****************+++++++++++++++++++.....                                  .++++++++++++++++++++
**************+++++++++++++++++++......                                  ..++++++++++++++++++++
************++++++++++++++++++.......                                   ..++++++++++++++++++++
**********++++++++++++++++++......                                      ..++++++++++++++++++++
*******.... ....                                                        ...++++++++++++++++++++
*******.... .                                                           ...++++++++++++++++++++
```



```
Evaluated to 0.000000
ready> mandel(-2, -1, 0.02, 0.04);
```



```
Evaluated to 0.000000
ready> ^D
```

At this point, you may be starting to realize that Kaleidoscope is a real and powerful language. It may not be self-similar :), but it can be used to plot things that are!

With this, we conclude the "adding user-defined operators" chapter of the tutorial. We have successfully augmented our language, adding the ability to extend the language in the library, and we have shown how this can be used to build a simple but interesting end-user application in Kaleidoscope. At this point, Kaleidoscope can build a variety

of applications that are functional and can call functions with side-effects, but it can't actually define and mutate a variable itself.

Strikingly, variable mutation is an important feature of some languages, and it is not at all obvious how to add support for mutable variables without having to add an "SSA construction" phase to your front-end. In the next chapter, we will describe how you can add variable mutation without building SSA in your front-end.

Full Code Listing

Here is the complete code listing for our running example, enhanced with the if/then/else and for expressions.. To build this example, use:

```
# Compile
ocamlbuild toy.byte
# Run
./toy.byte
```

Here is the code:

_tags:

```
<{lexer,parser}.ml>: use_camlp4, pp(camlp4of)
<*.{byte,native}>: g++, use_llvm, use_llvm_analysis
<*.{byte,native}>: use_llvm_executionengine, use_llvm_target
<*.{byte,native}>: use_llvm_scalar_opts, use_bindings
```

myocamlbuild.ml:

```
open Ocamlbuild_plugin;;

ocaml_lib ~extern:true "llvm";;
ocaml_lib ~extern:true "llvm_analysis";;
ocaml_lib ~extern:true "llvm_executionengine";;
ocaml_lib ~extern:true "llvm_target";;
ocaml_lib ~extern:true "llvm_scalar_opts";;

flag ["link"; "ocaml"; "g++"] (S[A"-cc"; A"g++"; A"-cclib"; A"-rdynamic"]);;
dep ["link"; "ocaml"; "use_bindings"] ["bindings.o"];;
```

token.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer Tokens
 *===----------------------------------------------------------------------===*)

(* The lexer returns these 'Kwd' if it is an unknown character, otherwise one of
 * these others for known things. *)
type token =
  (* commands *)
  | Def | Extern

  (* primary *)
  | Ident of string | Number of float

  (* unknown *)
  | Kwd of char

  (* control *)
  | If | Then | Else
```

```
  | For | In

  (* operators *)
  | Binary | Unary
```

lexer.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer
 *===----------------------------------------------------------------------===*)

let rec lex = parser
  (* Skip any whitespace. *)
  | [< ' (' ' | '\n' | '\r' | '\t'); stream >] -> lex stream

  (* identifier: [a-zA-Z][a-zA-Z0-9] *)
  | [< ' ('A' .. 'Z' | 'a' .. 'z' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_ident buffer stream

  (* number: [0-9.]+ *)
  | [< ' ('0' .. '9' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_number buffer stream

  (* Comment until end of line. *)
  | [< ' ('#'); stream >] ->
      lex_comment stream

  (* Otherwise, just return the character as its ascii value. *)
  | [< 'c; stream >] ->
      [< 'Token.Kwd c; lex stream >]

  (* end of stream. *)
  | [< >] -> [< >]

and lex_number buffer = parser
  | [< ' ('0' .. '9' | '.' as c); stream >] ->
      Buffer.add_char buffer c;
      lex_number buffer stream
  | [< stream=lex >] ->
      [< 'Token.Number (float_of_string (Buffer.contents buffer)); stream >]

and lex_ident buffer = parser
  | [< ' ('A' .. 'Z' | 'a' .. 'z' | '0' .. '9' as c); stream >] ->
      Buffer.add_char buffer c;
      lex_ident buffer stream
  | [< stream=lex >] ->
      match Buffer.contents buffer with
      | "def" -> [< 'Token.Def; stream >]
      | "extern" -> [< 'Token.Extern; stream >]
      | "if" -> [< 'Token.If; stream >]
      | "then" -> [< 'Token.Then; stream >]
      | "else" -> [< 'Token.Else; stream >]
      | "for" -> [< 'Token.For; stream >]
      | "in" -> [< 'Token.In; stream >]
```

```
        | "binary" -> [< 'Token.Binary; stream >]
        | "unary" -> [< 'Token.Unary; stream >]
        | id -> [< 'Token.Ident id; stream >]

and lex_comment = parser
  | [< ' ('\n'); stream=lex >] -> stream
  | [< 'c; e=lex_comment >] -> e
  | [< >] -> [< >]
```

ast.ml:

```
(*===----------------------------------------------------------------------===
 * Abstract Syntax Tree (aka Parse Tree)
 *===----------------------------------------------------------------------===*)

(* expr - Base type for all expression nodes. *)
type expr =
  (* variant for numeric literals like "1.0". *)
  | Number of float

  (* variant for referencing a variable, like "a". *)
  | Variable of string

  (* variant for a unary operator. *)
  | Unary of char * expr

  (* variant for a binary operator. *)
  | Binary of char * expr * expr

  (* variant for function calls. *)
  | Call of string * expr array

  (* variant for if/then/else. *)
  | If of expr * expr * expr

  (* variant for for/in. *)
  | For of string * expr * expr * expr option * expr

(* proto - This type represents the "prototype" for a function, which captures
 * its name, and its argument names (thus implicitly the number of arguments the
 * function takes). *)
type proto =
  | Prototype of string * string array
  | BinOpPrototype of string * string array * int

(* func - This type represents a function definition itself. *)
type func = Function of proto * expr
```

parser.ml:

```
(*===----------------------------------------------------------------------===
 * Parser
 *===----------------------------------------------------------------------===*)

(* binop_precedence - This holds the precedence for each binary operator that is
 * defined *)
let binop_precedence:(char, int) Hashtbl.t = Hashtbl.create 10

(* precedence - Get the precedence of the pending binary operator token. *)
```

```
let precedence c = try Hashtbl.find binop_precedence c with Not_found -> -1

(* primary
 *   ::= identifier
 *   ::= numberexpr
 *   ::= parenexpr
 *   ::= ifexpr
 *   ::= forexpr *)
let rec parse_primary = parser
  (* numberexpr ::= number *)
  | [< 'Token.Number n >] -> Ast.Number n

  (* parenexpr ::= '(' expression ')' *)
  | [< 'Token.Kwd '('; e=parse_expr; 'Token.Kwd ')' ?? "expected ')'" >] -> e

  (* identifierexpr
   *   ::= identifier
   *   ::= identifier '(' argumentexpr ')' *)
  | [< 'Token.Ident id; stream >] ->
      let rec parse_args accumulator = parser
        | [< e=parse_expr; stream >] ->
            begin parser
              | [< 'Token.Kwd ','; e=parse_args (e :: accumulator) >] -> e
              | [< >] -> e :: accumulator
            end stream
        | [< >] -> accumulator
      in
      let rec parse_ident id = parser
        (* Call. *)
        | [< 'Token.Kwd '(';
             args=parse_args [];
             'Token.Kwd ')' ?? "expected ')'">] ->
            Ast.Call (id, Array.of_list (List.rev args))

        (* Simple variable ref. *)
        | [< >] -> Ast.Variable id
      in
      parse_ident id stream

  (* ifexpr ::= 'if' expr 'then' expr 'else' expr *)
  | [< 'Token.If; c=parse_expr;
       'Token.Then ?? "expected 'then'"; t=parse_expr;
       'Token.Else ?? "expected 'else'"; e=parse_expr >] ->
      Ast.If (c, t, e)

  (* forexpr
        ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression *)
  | [< 'Token.For;
       'Token.Ident id ?? "expected identifier after for";
       'Token.Kwd '=' ?? "expected '=' after for";
       stream >] ->
      begin parser
        | [<
             start=parse_expr;
             'Token.Kwd ',' ?? "expected ',' after for";
             end_=parse_expr;
             stream >] ->
            let step =
```

```
              begin parser
              | [< 'Token.Kwd ','; step=parse_expr >] -> Some step
              | [< >] -> None
              end stream
            in
            begin parser
            | [< 'Token.In; body=parse_expr >] ->
                Ast.For (id, start, end_, step, body)
            | [< >] ->
                raise (Stream.Error "expected 'in' after for")
            end stream
        | [< >] ->
            raise (Stream.Error "expected '=' after for")
      end stream

  | [< >] -> raise (Stream.Error "unknown token when expecting an expression.")

(* unary
 *   ::= primary
 *   ::= '!' unary *)
and parse_unary = parser
  (* If this is a unary operator, read it. *)
  | [< 'Token.Kwd op when op != '(' && op != ')'; operand=parse_expr >] ->
      Ast.Unary (op, operand)

  (* If the current token is not an operator, it must be a primary expr. *)
  | [< stream >] -> parse_primary stream

(* binoprhs
 *   ::= ('+' primary)* *)
and parse_bin_rhs expr_prec lhs stream =
  match Stream.peek stream with
  (* If this is a binop, find its precedence. *)
  | Some (Token.Kwd c) when Hashtbl.mem binop_precedence c ->
      let token_prec = precedence c in

      (* If this is a binop that binds at least as tightly as the current binop,
       * consume it, otherwise we are done. *)
      if token_prec < expr_prec then lhs else begin
        (* Eat the binop. *)
        Stream.junk stream;

        (* Parse the unary expression after the binary operator. *)
        let rhs = parse_unary stream in

        (* Okay, we know this is a binop. *)
        let rhs =
          match Stream.peek stream with
          | Some (Token.Kwd c2) ->
              (* If BinOp binds less tightly with rhs than the operator after
               * rhs, let the pending operator take rhs as its lhs. *)
              let next_prec = precedence c2 in
              if token_prec < next_prec
              then parse_bin_rhs (token_prec + 1) rhs stream
              else rhs
          | _ -> rhs
        in
```

```
          (* Merge lhs/rhs. *)
          let lhs = Ast.Binary (c, lhs, rhs) in
          parse_bin_rhs expr_prec lhs stream
        end
    | _ -> lhs

(* expression
 *   ::= primary binoprhs *)
and parse_expr = parser
    | [< lhs=parse_unary; stream >] -> parse_bin_rhs 0 lhs stream

(* prototype
 *   ::= id '(' id* ')'
 *   ::= binary LETTER number? (id, id)
 *   ::= unary LETTER number? (id) *)
let parse_prototype =
  let rec parse_args accumulator = parser
    | [< 'Token.Ident id; e=parse_args (id::accumulator) >] -> e
    | [< >] -> accumulator
  in
  let parse_operator = parser
    | [< 'Token.Unary >] -> "unary", 1
    | [< 'Token.Binary >] -> "binary", 2
  in
  let parse_binary_precedence = parser
    | [< 'Token.Number n >] -> int_of_float n
    | [< >] -> 30
  in
  parser
  | [< 'Token.Ident id;
       'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      (* success. *)
      Ast.Prototype (id, Array.of_list (List.rev args))
  | [< (prefix, kind)=parse_operator;
       'Token.Kwd op ?? "expected an operator";
       (* Read the precedence if present. *)
       binary_precedence=parse_binary_precedence;
       'Token.Kwd '(' ?? "expected '(' in prototype";
        args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      let name = prefix ^ (String.make 1 op) in
      let args = Array.of_list (List.rev args) in

      (* Verify right number of arguments for operator. *)
      if Array.length args != kind
      then raise (Stream.Error "invalid number of operands for operator")
      else
        if kind == 1 then
          Ast.Prototype (name, args)
        else
          Ast.BinOpPrototype (name, args, binary_precedence)
  | [< >] ->
      raise (Stream.Error "expected function name in prototype")

(* definition ::= 'def' prototype expression *)
let parse_definition = parser
```

```
  | [< 'Token.Def; p=parse_prototype; e=parse_expr >] ->
      Ast.Function (p, e)

(* toplevelexpr ::= expression *)
let parse_toplevel = parser
  | [< e=parse_expr >] ->
      (* Make an anonymous proto. *)
      Ast.Function (Ast.Prototype ("", [||]), e)

(*  external ::= 'extern' prototype *)
let parse_extern = parser
  | [< 'Token.Extern; e=parse_prototype >] -> e
```

codegen.ml:

```
(*===----------------------------------------------------------------------===
 * Code Generation
 *===----------------------------------------------------------------------===*)

open Llvm

exception Error of string

let context = global_context ()
let the_module = create_module context "my cool jit"
let builder = builder context
let named_values:(string, llvalue) Hashtbl.t = Hashtbl.create 10
let double_type = double_type context

let rec codegen_expr = function
  | Ast.Number n -> const_float double_type n
  | Ast.Variable name ->
      (try Hashtbl.find named_values name with
        | Not_found -> raise (Error "unknown variable name"))
  | Ast.Unary (op, operand) ->
      let operand = codegen_expr operand in
      let callee = "unary" ^ (String.make 1 op) in
      let callee =
        match lookup_function callee the_module with
        | Some callee -> callee
        | None -> raise (Error "unknown unary operator")
      in
      build_call callee [|operand|] "unop" builder
  | Ast.Binary (op, lhs, rhs) ->
      let lhs_val = codegen_expr lhs in
      let rhs_val = codegen_expr rhs in
      begin
        match op with
        | '+' -> build_add lhs_val rhs_val "addtmp" builder
        | '-' -> build_sub lhs_val rhs_val "subtmp" builder
        | '*' -> build_mul lhs_val rhs_val "multtmp" builder
        | '<' ->
            (* Convert bool 0/1 to double 0.0 or 1.0 *)
            let i = build_fcmp Fcmp.Ult lhs_val rhs_val "cmptmp" builder in
            build_uitofp i double_type "booltmp" builder
        | _ ->
            (* If it wasn't a builtin binary operator, it must be a user defined
             * one. Emit a call to it. *)
```

```
            let callee = "binary" ^ (String.make 1 op) in
            let callee =
              match lookup_function callee the_module with
              | Some callee -> callee
              | None -> raise (Error "binary operator not found!")
            in
            build_call callee [|lhs_val; rhs_val|] "binop" builder
      end
  | Ast.Call (callee, args) ->
      (* Look up the name in the module table. *)
      let callee =
        match lookup_function callee the_module with
        | Some callee -> callee
        | None -> raise (Error "unknown function referenced")
      in
      let params = params callee in

      (* If argument mismatch error. *)
      if Array.length params == Array.length args then () else
        raise (Error "incorrect # arguments passed");
      let args = Array.map codegen_expr args in
      build_call callee args "calltmp" builder
  | Ast.If (cond, then_, else_) ->
      let cond = codegen_expr cond in

      (* Convert condition to a bool by comparing equal to 0.0 *)
      let zero = const_float double_type 0.0 in
      let cond_val = build_fcmp Fcmp.One cond zero "ifcond" builder in

      (* Grab the first block so that we might later add the conditional branch
       * to it at the end of the function. *)
      let start_bb = insertion_block builder in
      let the_function = block_parent start_bb in

      let then_bb = append_block context "then" the_function in

      (* Emit 'then' value. *)
      position_at_end then_bb builder;
      let then_val = codegen_expr then_ in

      (* Codegen of 'then' can change the current block, update then_bb for the
       * phi. We create a new name because one is used for the phi node, and the
       * other is used for the conditional branch. *)
      let new_then_bb = insertion_block builder in

      (* Emit 'else' value. *)
      let else_bb = append_block context "else" the_function in
      position_at_end else_bb builder;
      let else_val = codegen_expr else_ in

      (* Codegen of 'else' can change the current block, update else_bb for the
       * phi. *)
      let new_else_bb = insertion_block builder in

      (* Emit merge block. *)
      let merge_bb = append_block context "ifcont" the_function in
      position_at_end merge_bb builder;
      let incoming = [(then_val, new_then_bb); (else_val, new_else_bb)] in
```

```
    let phi = build_phi incoming "iftmp" builder in

    (* Return to the start block to add the conditional branch. *)
    position_at_end start_bb builder;
    ignore (build_cond_br cond_val then_bb else_bb builder);

    (* Set a unconditional branch at the end of the 'then' block and the
     * 'else' block to the 'merge' block. *)
    position_at_end new_then_bb builder; ignore (build_br merge_bb builder);
    position_at_end new_else_bb builder; ignore (build_br merge_bb builder);

    (* Finally, set the builder to the end of the merge block. *)
    position_at_end merge_bb builder;

    phi
| Ast.For (var_name, start, end_, step, body) ->
    (* Emit the start code first, without 'variable' in scope. *)
    let start_val = codegen_expr start in

    (* Make the new basic block for the loop header, inserting after current
     * block. *)
    let preheader_bb = insertion_block builder in
    let the_function = block_parent preheader_bb in
    let loop_bb = append_block context "loop" the_function in

    (* Insert an explicit fall through from the current block to the
     * loop_bb. *)
    ignore (build_br loop_bb builder);

    (* Start insertion in loop_bb. *)
    position_at_end loop_bb builder;

    (* Start the PHI node with an entry for start. *)
    let variable = build_phi [(start_val, preheader_bb)] var_name builder in

    (* Within the loop, the variable is defined equal to the PHI node. If it
     * shadows an existing variable, we have to restore it, so save it
     * now. *)
    let old_val =
      try Some (Hashtbl.find named_values var_name) with Not_found -> None
    in
    Hashtbl.add named_values var_name variable;

    (* Emit the body of the loop.  This, like any other expr, can change the
     * current BB.  Note that we ignore the value computed by the body, but
     * don't allow an error *)
    ignore (codegen_expr body);

    (* Emit the step value. *)
    let step_val =
      match step with
      | Some step -> codegen_expr step
      (* If not specified, use 1.0. *)
      | None -> const_float double_type 1.0
    in

    let next_var = build_add variable step_val "nextvar" builder in
```

```
        (* Compute the end condition. *)
        let end_cond = codegen_expr end_ in

        (* Convert condition to a bool by comparing equal to 0.0. *)
        let zero = const_float double_type 0.0 in
        let end_cond = build_fcmp Fcmp.One end_cond zero "loopcond" builder in

        (* Create the "after loop" block and insert it. *)
        let loop_end_bb = insertion_block builder in
        let after_bb = append_block context "afterloop" the_function in

        (* Insert the conditional branch into the end of loop_end_bb. *)
        ignore (build_cond_br end_cond loop_bb after_bb builder);

        (* Any new code will be inserted in after_bb. *)
        position_at_end after_bb builder;

        (* Add a new entry to the PHI node for the backedge. *)
        add_incoming (next_var, loop_end_bb) variable;

        (* Restore the unshadowed variable. *)
        begin match old_val with
        | Some old_val -> Hashtbl.add named_values var_name old_val
        | None -> ()
        end;

        (* for expr always returns 0.0. *)
        const_null double_type

let codegen_proto = function
  | Ast.Prototype (name, args) | Ast.BinOpPrototype (name, args, _) ->
        (* Make the function type: double(double,double) etc. *)
        let doubles = Array.make (Array.length args) double_type in
        let ft = function_type double_type doubles in
        let f =
          match lookup_function name the_module with
          | None -> declare_function name ft the_module

          (* If 'f' conflicted, there was already something named 'name'. If it
           * has a body, don't allow redefinition or reextern. *)
          | Some f ->
              (* If 'f' already has a body, reject this. *)
              if block_begin f <> At_end f then
                raise (Error "redefinition of function");

              (* If 'f' took a different number of arguments, reject. *)
              if element_type (type_of f) <> ft then
                raise (Error "redefinition of function with different # args");
              f
        in

        (* Set names for all arguments. *)
        Array.iteri (fun i a ->
          let n = args.(i) in
          set_value_name n a;
          Hashtbl.add named_values n a;
        ) (params f);
        f
```

```
let codegen_func the_fpm = function
  | Ast.Function (proto, body) ->
      Hashtbl.clear named_values;
      let the_function = codegen_proto proto in

      (* If this is an operator, install it. *)
      begin match proto with
      | Ast.BinOpPrototype (name, args, prec) ->
          let op = name.[String.length name - 1] in
          Hashtbl.add Parser.binop_precedence op prec;
      | _ -> ()
      end;

      (* Create a new basic block to start insertion into. *)
      let bb = append_block context "entry" the_function in
      position_at_end bb builder;

      try
        let ret_val = codegen_expr body in

        (* Finish off the function. *)
        let _ = build_ret ret_val builder in

        (* Validate the generated code, checking for consistency. *)
        Llvm_analysis.assert_valid_function the_function;

        (* Optimize the function. *)
        let _ = PassManager.run_function the_function the_fpm in

        the_function
      with e ->
        delete_function the_function;
        raise e
```

toplevel.ml:

```
(*===----------------------------------------------------------------------===
 * Top-Level parsing and JIT Driver
 *===----------------------------------------------------------------------===*)

open Llvm
open Llvm_executionengine

(* top ::= definition | external | expression | ';' *)
let rec main_loop the_fpm the_execution_engine stream =
  match Stream.peek stream with
  | None -> ()

  (* ignore top-level semicolons. *)
  | Some (Token.Kwd ';') ->
      Stream.junk stream;
      main_loop the_fpm the_execution_engine stream

  | Some token ->
      begin
        try match token with
        | Token.Def ->
            let e = Parser.parse_definition stream in
```

```
            print_endline "parsed a function definition.";
            dump_value (Codegen.codegen_func the_fpm e);
          | Token.Extern ->
            let e = Parser.parse_extern stream in
            print_endline "parsed an extern.";
            dump_value (Codegen.codegen_proto e);
          | _ ->
            (* Evaluate a top-level expression into an anonymous function. *)
            let e = Parser.parse_toplevel stream in
            print_endline "parsed a top-level expr";
            let the_function = Codegen.codegen_func the_fpm e in
            dump_value the_function;

            (* JIT the function, returning a function pointer. *)
            let result = ExecutionEngine.run_function the_function [||]
              the_execution_engine in

            print_string "Evaluated to ";
            print_float (GenericValue.as_float Codegen.double_type result);
            print_newline ();
        with Stream.Error s | Codegen.Error s ->
          (* Skip token for error recovery. *)
          Stream.junk stream;
          print_endline s;
      end;
      print_string "ready> "; flush stdout;
      main_loop the_fpm the_execution_engine stream
```

toy.ml:

```
(*===----------------------------------------------------------------------===
 * Main driver code.
 *===----------------------------------------------------------------------===*)

open Llvm
open Llvm_executionengine
open Llvm_target
open Llvm_scalar_opts

let main () =
  ignore (initialize_native_target ());

  (* Install standard binary operators.
   * 1 is the lowest precedence. *)
  Hashtbl.add Parser.binop_precedence '<' 10;
  Hashtbl.add Parser.binop_precedence '+' 20;
  Hashtbl.add Parser.binop_precedence '-' 20;
  Hashtbl.add Parser.binop_precedence '*' 40;    (* highest. *)

  (* Prime the first token. *)
  print_string "ready> "; flush stdout;
  let stream = Lexer.lex (Stream.of_channel stdin) in

  (* Create the JIT. *)
  let the_execution_engine = ExecutionEngine.create Codegen.the_module in
  let the_fpm = PassManager.create_function Codegen.the_module in

  (* Set up the optimizer pipeline.  Start with registering info about how the
```

```
     * target lays out data structures. *)
    DataLayout.add (ExecutionEngine.target_data the_execution_engine) the_fpm;

    (* Do simple "peephole" optimizations and bit-twiddling optzn. *)
    add_instruction_combination the_fpm;

    (* reassociate expressions. *)
    add_reassociation the_fpm;

    (* Eliminate Common SubExpressions. *)
    add_gvn the_fpm;

    (* Simplify the control flow graph (deleting unreachable blocks, etc). *)
    add_cfg_simplification the_fpm;

    ignore (PassManager.initialize the_fpm);

    (* Run the main "interpreter loop" now. *)
    Toplevel.main_loop the_fpm the_execution_engine stream;

    (* Print out all the generated code. *)
    dump_module Codegen.the_module
  ;;

  main ()
```

bindings.c

```
#include <stdio.h>

/* putchard - putchar that takes a double and returns 0. */
extern double putchard(double X) {
  putchar((char)X);
  return 0;
}

/* printd - printf that takes a double prints it as "%f\n", returning 0. */
extern double printd(double X) {
  printf("%f\n", X);
  return 0;
}
```

Next: Extending the language: mutable variables / SSA construction

Kaleidoscope: Extending the Language: Mutable Variables

Chapter 7 Introduction

Welcome to Chapter 7 of the "Implementing a language with LLVM" tutorial. In chapters 1 through 6, we've built a very respectable, albeit simple, functional programming language. In our journey, we learned some parsing techniques, how to build and represent an AST, how to build LLVM IR, and how to optimize the resultant code as well as JIT compile it.

While Kaleidoscope is interesting as a functional language, the fact that it is functional makes it "too easy" to generate LLVM IR for it. In particular, a functional language makes it very easy to build LLVM IR directly in SSA form. Since LLVM requires that the input code be in SSA form, this is a very nice property and it is often unclear to newcomers how to generate code for an imperative language with mutable variables.

The short (and happy) summary of this chapter is that there is no need for your front-end to build SSA form: LLVM provides highly tuned and well tested support for this, though the way it works is a bit unexpected for some.

Why is this a hard problem?

To understand why mutable variables cause complexities in SSA construction, consider this extremely simple C example:

```
int G, H;
int test(_Bool Condition) {
  int X;
  if (Condition)
    X = G;
  else
    X = H;
  return X;
}
```

In this case, we have the variable "X", whose value depends on the path executed in the program. Because there are two different possible values for X before the return instruction, a PHI node is inserted to merge the two values. The LLVM IR that we want for this example looks like this:

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.2 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.2
}
```

In this example, the loads from the G and H global variables are explicit in the LLVM IR, and they live in the then/else branches of the if statement (cond_true/cond_false). In order to merge the incoming values, the X.2 phi node in the cond_next block selects the right value to use based on where control flow is coming from: if control flow comes from

the cond_false block, X.2 gets the value of X.1. Alternatively, if control flow comes from cond_true, it gets the value of X.0. The intent of this chapter is not to explain the details of SSA form. For more information, see one of the many online references.

The question for this article is "who places the phi nodes when lowering assignments to mutable variables?". The issue here is that LLVM *requires* that its IR be in SSA form: there is no "non-ssa" mode for it. However, SSA construction requires non-trivial algorithms and data structures, so it is inconvenient and wasteful for every front-end to have to reproduce this logic.

Memory in LLVM

The 'trick' here is that while LLVM does require all register values to be in SSA form, it does not require (or permit) memory objects to be in SSA form. In the example above, note that the loads from G and H are direct accesses to G and H: they are not renamed or versioned. This differs from some other compiler systems, which do try to version memory objects. In LLVM, instead of encoding dataflow analysis of memory into the LLVM IR, it is handled with Analysis Passes which are computed on demand.

With this in mind, the high-level idea is that we want to make a stack variable (which lives in memory, because it is on the stack) for each mutable object in a function. To take advantage of this trick, we need to talk about how LLVM represents stack variables.

In LLVM, all memory accesses are explicit with load/store instructions, and it is carefully designed not to have (or need) an "address-of" operator. Notice how the type of the @G/@H global variables is actually "i32*" even though the variable is defined as "i32". What this means is that @G defines *space* for an i32 in the global data area, but its *name* actually refers to the address for that space. Stack variables work the same way, except that instead of being declared with global variable definitions, they are declared with the LLVM alloca instruction:

```
define i32 @example() {
entry:
  %X = alloca i32           ; type of %X is i32*.
  ...
  %tmp = load i32* %X       ; load the stack value %X from the stack.
  %tmp2 = add i32 %tmp, 1   ; increment it
  store i32 %tmp2, i32* %X  ; store it back
  ...
```

This code shows an example of how you can declare and manipulate a stack variable in the LLVM IR. Stack memory allocated with the alloca instruction is fully general: you can pass the address of the stack slot to functions, you can store it in other variables, etc. In our example above, we could rewrite the example to use the alloca technique to avoid using a PHI node:

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  %X = alloca i32           ; type of %X is i32*.
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
        store i32 %X.0, i32* %X   ; Update X
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
        store i32 %X.1, i32* %X   ; Update X
```

```
  br label %cond_next

cond_next:
  %X.2 = load i32* %X        ; Read X
  ret i32 %X.2
}
```

With this, we have discovered a way to handle arbitrary mutable variables without the need to create Phi nodes at all:

1. Each mutable variable becomes a stack allocation.
2. Each read of the variable becomes a load from the stack.
3. Each update of the variable becomes a store to the stack.
4. Taking the address of a variable just uses the stack address directly.

While this solution has solved our immediate problem, it introduced another one: we have now apparently introduced a lot of stack traffic for very simple and common operations, a major performance problem. Fortunately for us, the LLVM optimizer has a highly-tuned optimization pass named "mem2reg" that handles this case, promoting allocas like this into SSA registers, inserting Phi nodes as appropriate. If you run this example through the pass, for example, you'll get:

```
$ llvm-as < example.ll | opt -mem2reg | llvm-dis
@G = weak global i32 0
@H = weak global i32 0

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.01 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.01
}
```

The mem2reg pass implements the standard "iterated dominance frontier" algorithm for constructing SSA form and has a number of optimizations that speed up (very common) degenerate cases. The mem2reg optimization pass is the answer to dealing with mutable variables, and we highly recommend that you depend on it. Note that mem2reg only works on variables in certain circumstances:

1. mem2reg is alloca-driven: it looks for allocas and if it can handle them, it promotes them. It does not apply to global variables or heap allocations.
2. mem2reg only looks for alloca instructions in the entry block of the function. Being in the entry block guarantees that the alloca is only executed once, which makes analysis simpler.
3. mem2reg only promotes allocas whose uses are direct loads and stores. If the address of the stack object is passed to a function, or if any funny pointer arithmetic is involved, the alloca will not be promoted.
4. mem2reg only works on allocas of first class values (such as pointers, scalars and vectors), and only if the array size of the allocation is 1 (or missing in the .ll file). mem2reg is not capable of promoting structs or arrays

to registers. Note that the "scalarrepl" pass is more powerful and can promote structs, "unions", and arrays in many cases.

All of these properties are easy to satisfy for most imperative languages, and we'll illustrate it below with Kaleidoscope. The final question you may be asking is: should I bother with this nonsense for my front-end? Wouldn't it be better if I just did SSA construction directly, avoiding use of the mem2reg optimization pass? In short, we strongly recommend that you use this technique for building SSA form, unless there is an extremely good reason not to. Using this technique is:

- Proven and well tested: clang uses this technique for local mutable variables. As such, the most common clients of LLVM are using this to handle a bulk of their variables. You can be sure that bugs are found fast and fixed early.
- Extremely Fast: mem2reg has a number of special cases that make it fast in common cases as well as fully general. For example, it has fast-paths for variables that are only used in a single block, variables that only have one assignment point, good heuristics to avoid insertion of unneeded phi nodes, etc.
- Needed for debug info generation: Debug information in LLVM relies on having the address of the variable exposed so that debug info can be attached to it. This technique dovetails very naturally with this style of debug info.

If nothing else, this makes it much easier to get your front-end up and running, and is very simple to implement. Lets extend Kaleidoscope with mutable variables now!

Mutable Variables in Kaleidoscope

Now that we know the sort of problem we want to tackle, lets see what this looks like in the context of our little Kaleidoscope language. We're going to add two features:

1. The ability to mutate variables with the '=' operator.
2. The ability to define new variables.

While the first item is really what this is about, we only have variables for incoming arguments as well as for induction variables, and redefining those only goes so far :). Also, the ability to define new variables is a useful thing regardless of whether you will be mutating them. Here's a motivating example that shows how we could use these:

```
# Define ':' for sequencing: as a low-precedence operator that ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;

# Recursive fib, we could do this before.
def fib(x)
  if (x < 3) then
    1
  else
    fib(x-1)+fib(x-2);

# Iterative fib.
def fibi(x)
  var a = 1, b = 1, c in
  (for i = 3, i < x in
     c = a + b :
     a = b :
     b = c) :
  b;

# Call it.
fibi(10);
```

In order to mutate variables, we have to change our existing variables to use the "alloca trick". Once we have that, we'll add our new operator, then extend Kaleidoscope to support new variable definitions.

Adjusting Existing Variables for Mutation

The symbol table in Kaleidoscope is managed at code generation time by the 'named_values' map. This map currently keeps track of the LLVM "Value*" that holds the double value for the named variable. In order to support mutation, we need to change this slightly, so that it named_values holds the *memory location* of the variable in question. Note that this change is a refactoring: it changes the structure of the code, but does not (by itself) change the behavior of the compiler. All of these changes are isolated in the Kaleidoscope code generator.

At this point in Kaleidoscope's development, it only supports variables for two things: incoming arguments to functions and the induction variable of 'for' loops. For consistency, we'll allow mutation of these variables in addition to other user-defined variables. This means that these will both need memory locations.

To start our transformation of Kaleidoscope, we'll change the named_values map so that it maps to AllocaInst* instead of Value*. Once we do this, the C++ compiler will tell us what parts of the code we need to update:

Note: the ocaml bindings currently model both Value*'s and AllocInst*'s as Llvm.llvalue's, but this may change in the future to be more type safe.

```
let named_values:(string, llvalue) Hashtbl.t = Hashtbl.create 10
```

Also, since we will need to create these alloca's, we'll use a helper function that ensures that the allocas are created in the entry block of the function:

```
(* Create an alloca instruction in the entry block of the function. This
 * is used for mutable variables etc. *)
let create_entry_block_alloca the_function var_name =
  let builder = builder_at (instr_begin (entry_block the_function)) in
  build_alloca double_type var_name builder
```

This funny looking code creates an Llvm.llbuilder object that is pointing at the first instruction of the entry block. It then creates an alloca with the expected name and returns it. Because all values in Kaleidoscope are doubles, there is no need to pass in a type to use.

With this in place, the first functionality change we want to make is to variable references. In our new scheme, variables live on the stack, so code generating a reference to them actually needs to produce a load from the stack slot:

```
let rec codegen_expr = function
  ...
  | Ast.Variable name ->
      let v = try Hashtbl.find named_values name with
        | Not_found -> raise (Error "unknown variable name")
      in
      (* Load the value. *)
      build_load v name builder
```

As you can see, this is pretty straightforward. Now we need to update the things that define the variables to set up the alloca. We'll start with codegen_expr Ast.For ... (see the full code listing for the unabridged code):

```
| Ast.For (var_name, start, end_, step, body) ->
    let the_function = block_parent (insertion_block builder) in

    (* Create an alloca for the variable in the entry block. *)
    let alloca = create_entry_block_alloca the_function var_name in

    (* Emit the start code first, without 'variable' in scope. *)
```

```
    let start_val = codegen_expr start in

    (* Store the value into the alloca. *)
    ignore(build_store start_val alloca builder);

    ...

    (* Within the loop, the variable is defined equal to the PHI node. If it
     * shadows an existing variable, we have to restore it, so save it
     * now. *)
    let old_val =
      try Some (Hashtbl.find named_values var_name) with Not_found -> None
    in
    Hashtbl.add named_values var_name alloca;

    ...

    (* Compute the end condition. *)
    let end_cond = codegen_expr end_ in

    (* Reload, increment, and restore the alloca. This handles the case where
     * the body of the loop mutates the variable. *)
    let cur_var = build_load alloca var_name builder in
    let next_var = build_add cur_var step_val "nextvar" builder in
    ignore(build_store next_var alloca builder);
    ...
```

This code is virtually identical to the code before we allowed mutable variables. The big difference is that we no longer have to construct a PHI node, and we use load/store to access the variable as needed.

To support mutable argument variables, we need to also make allocas for them. The code for this is also pretty simple:

```
(* Create an alloca for each argument and register the argument in the symbol
 * table so that references to it will succeed. *)
let create_argument_allocas the_function proto =
  let args = match proto with
    | Ast.Prototype (_, args) | Ast.BinOpPrototype (_, args, _) -> args
  in
  Array.iteri (fun i ai ->
    let var_name = args.(i) in
    (* Create an alloca for this variable. *)
    let alloca = create_entry_block_alloca the_function var_name in

    (* Store the initial value into the alloca. *)
    ignore(build_store ai alloca builder);

    (* Add arguments to variable symbol table. *)
    Hashtbl.add named_values var_name alloca;
  ) (params the_function)
```

For each argument, we make an alloca, store the input value to the function into the alloca, and register the alloca as the memory location for the argument. This method gets invoked by `Codegen.codegen_func` right after it sets up the entry block for the function.

The final missing piece is adding the mem2reg pass, which allows us to get good codegen once again:

```
let main () =
  ...
  let the_fpm = PassManager.create_function Codegen.the_module in
```

```
  (* Set up the optimizer pipeline.  Start with registering info about how the
   * target lays out data structures. *)
  DataLayout.add (ExecutionEngine.target_data the_execution_engine) the_fpm;

  (* Promote allocas to registers. *)
  add_memory_to_register_promotion the_fpm;

  (* Do simple "peephole" optimizations and bit-twiddling optzn. *)
  add_instruction_combining the_fpm;

  (* reassociate expressions. *)
  add_reassociation the_fpm;
```

It is interesting to see what the code looks like before and after the mem2reg optimization runs. For example, this is the before/after code for our recursive fib function. Before the optimization:

```
define double @fib(double %x) {
entry:
  %x1 = alloca double
  store double %x, double* %x1
  %x2 = load double* %x1
  %cmptmp = fcmp ult double %x2, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:     ; preds = %entry
  br label %ifcont

else:     ; preds = %entry
  %x3 = load double* %x1
  %subtmp = fsub double %x3, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %x4 = load double* %x1
  %subtmp5 = fsub double %x4, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```

Here there is only one variable (x, the input argument) but you can still see the extremely simple-minded code generation strategy we are using. In the entry block, an alloca is created, and the initial input value is stored into it. Each reference to the variable does a reload from the stack. Also, note that we didn't modify the if/then/else expression, so it still inserts a PHI node. While we could make an alloca for it, it is actually easier to create a PHI node for it, so we still just make the PHI.

Here is the code after the mem2reg pass runs:

```
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else
```

```
then:
  br label %ifcont

else:
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp5 = fsub double %x, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```

This is a trivial case for mem2reg, since there are no redefinitions of the variable. The point of showing this is to calm your tension about inserting such blatent inefficiencies :).

After the rest of the optimizers run, we get:

```
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp ueq double %booltmp, 0.000000e+00
  br i1 %ifcond, label %else, label %ifcont

else:
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp5 = fsub double %x, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  ret double %addtmp

ifcont:
  ret double 1.000000e+00
}
```

Here we see that the simplifycfg pass decided to clone the return instruction into the end of the 'else' block. This allowed it to eliminate some branches and the PHI node.

Now that all symbol table references are updated to use stack variables, we'll add the assignment operator.

New Assignment Operator

With our current framework, adding a new assignment operator is really simple. We will parse it just like any other binary operator, but handle it internally (instead of allowing the user to define it). The first step is to set a precedence:

```
let main () =
  (* Install standard binary operators.
   * 1 is the lowest precedence. *)
  Hashtbl.add Parser.binop_precedence '=' 2;
  Hashtbl.add Parser.binop_precedence '<' 10;
  Hashtbl.add Parser.binop_precedence '+' 20;
  Hashtbl.add Parser.binop_precedence '-' 20;
  ...
```

Now that the parser knows the precedence of the binary operator, it takes care of all the parsing and AST generation. We just need to implement codegen for the assignment operator. This looks like:

```
let rec codegen_expr = function
      begin match op with
      | '=' ->
          (* Special case '=' because we don't want to emit the LHS as an
           * expression. *)
          let name =
            match lhs with
            | Ast.Variable name -> name
            | _ -> raise (Error "destination of '=' must be a variable")
          in
```

Unlike the rest of the binary operators, our assignment operator doesn't follow the "emit LHS, emit RHS, do computation" model. As such, it is handled as a special case before the other binary operators are handled. The other strange thing is that it requires the LHS to be a variable. It is invalid to have "(x+1) = expr" - only things like "x = expr" are allowed.

```
    (* Codegen the rhs. *)
    let val_ = codegen_expr rhs in

    (* Lookup the name. *)
    let variable = try Hashtbl.find named_values name with
    | Not_found -> raise (Error "unknown variable name")
    in
    ignore(build_store val_ variable builder);
    val_
| _ ->
      ...
```

Once we have the variable, codegen'ing the assignment is straightforward: we emit the RHS of the assignment, create a store, and return the computed value. Returning a value allows for chained assignments like "X = (Y = Z)".

Now that we have an assignment operator, we can mutate loop variables and arguments. For example, we can now run code like this:

```
# Function to print a double.
extern printd(x);

# Define ':' for sequencing: as a low-precedence operator that ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;

def test(x)
  printd(x) :
  x = 4 :
  printd(x);

test(123);
```

When run, this example prints "123" and then "4", showing that we did actually mutate the value! Okay, we have now officially implemented our goal: getting this to work requires SSA construction in the general case. However, to be really useful, we want the ability to define our own local variables, lets add this next!

User-defined Local Variables

Adding var/in is just like any other other extensions we made to Kaleidoscope: we extend the lexer, the parser, the AST and the code generator. The first step for adding our new 'var/in' construct is to extend the lexer. As before, this is pretty trivial, the code looks like this:

```
type token =
  ...
  (* var definition *)
  | Var

...

and lex_ident buffer = parser
      ...
      | "in" -> [< 'Token.In; stream >]
      | "binary" -> [< 'Token.Binary; stream >]
      | "unary" -> [< 'Token.Unary; stream >]
      | "var" -> [< 'Token.Var; stream >]
      ...
```

The next step is to define the AST node that we will construct. For var/in, it looks like this:

```
type expr =
  ...
  (* variant for var/in. *)
  | Var of (string * expr option) array * expr
  ...
```

var/in allows a list of names to be defined all at once, and each name can optionally have an initializer value. As such, we capture this information in the VarNames vector. Also, var/in has a body, this body is allowed to access the variables defined by the var/in.

With this in place, we can define the parser pieces. The first thing we do is add it as a primary expression:

```
(* primary
 *   ::= identifier
 *   ::= numberexpr
 *   ::= parenexpr
 *   ::= ifexpr
 *   ::= forexpr
 *   ::= varexpr *)
let rec parse_primary = parser
  ...
  (* varexpr
   *   ::= 'var' identifier ('=' expression?
   *             (',' identifier ('=' expression)?)* 'in' expression *)
  | [< 'Token.Var;
       (* At least one variable name is required. *)
       'Token.Ident id ?? "expected identifier after var";
       init=parse_var_init;
       var_names=parse_var_names [(id, init)];
       (* At this point, we have to have 'in'. *)
       'Token.In ?? "expected 'in' keyword after 'var'";
       body=parse_expr >] ->
      Ast.Var (Array.of_list (List.rev var_names), body)

...
```

```
and parse_var_init = parser
  (* read in the optional initializer. *)
  | [< 'Token.Kwd '='; e=parse_expr >] -> Some e
  | [< >] -> None

and parse_var_names accumulator = parser
  | [< 'Token.Kwd ',';
       'Token.Ident id ?? "expected identifier list after var";
       init=parse_var_init;
       e=parse_var_names ((id, init) :: accumulator) >] -> e
  | [< >] -> accumulator
```

Now that we can parse and represent the code, we need to support emission of LLVM IR for it. This code starts out with:

```
let rec codegen_expr = function
  ...
  | Ast.Var (var_names, body)
      let old_bindings = ref [] in

      let the_function = block_parent (insertion_block builder) in

      (* Register all variables and emit their initializer. *)
      Array.iter (fun (var_name, init) ->
```

Basically it loops over all the variables, installing them one at a time. For each variable we put into the symbol table, we remember the previous value that we replace in OldBindings.

```
  (* Emit the initializer before adding the variable to scope, this
   * prevents the initializer from referencing the variable itself, and
   * permits stuff like this:
   *   var a = 1 in
   *     var a = a in ...   # refers to outer 'a'. *)
  let init_val =
    match init with
    | Some init -> codegen_expr init
    (* If not specified, use 0.0. *)
    | None -> const_float double_type 0.0
  in

  let alloca = create_entry_block_alloca the_function var_name in
  ignore(build_store init_val alloca builder);

  (* Remember the old variable binding so that we can restore the binding
   * when we unrecurse. *)

  begin
    try
      let old_value = Hashtbl.find named_values var_name in
      old_bindings := (var_name, old_value) :: !old_bindings;
    with Not_found > ()
  end;

  (* Remember this binding. *)
  Hashtbl.add named_values var_name alloca;
) var_names;
```

There are more comments here than code. The basic idea is that we emit the initializer, create the alloca, then update the symbol table to point to it. Once all the variables are installed in the symbol table, we evaluate the body of the

var/in expression:

```
(* Codegen the body, now that all vars are in scope. *)
let body_val = codegen_expr body in
```

Finally, before returning, we restore the previous variable bindings:

```
(* Pop all our variables from scope. *)
List.iter (fun (var_name, old_value) ->
  Hashtbl.add named_values var_name old_value
) !old_bindings;

(* Return the body computation. *)
body_val
```

The end result of all of this is that we get properly scoped variable definitions, and we even (trivially) allow mutation of them :).

With this, we completed what we set out to do. Our nice iterative fib example from the intro compiles and runs just fine. The mem2reg pass optimizes all of our stack variables into SSA registers, inserting PHI nodes where needed, and our front-end remains simple: no "iterated dominance frontier" computation anywhere in sight.

Full Code Listing

Here is the complete code listing for our running example, enhanced with mutable variables and var/in support. To build this example, use:

```
# Compile
ocamlbuild toy.byte
# Run
./toy.byte
```

Here is the code:

_tags:

```
<{lexer,parser}.ml>: use_camlp4, pp(camlp4of)
<*.{byte,native}>: g++, use_llvm, use_llvm_analysis
<*.{byte,native}>: use_llvm_executionengine, use_llvm_target
<*.{byte,native}>: use_llvm_scalar_opts, use_bindings
```

myocamlbuild.ml:

```
open Ocamlbuild_plugin;;

ocaml_lib ~extern:true "llvm";;
ocaml_lib ~extern:true "llvm_analysis";;
ocaml_lib ~extern:true "llvm_executionengine";;
ocaml_lib ~extern:true "llvm_target";;
ocaml_lib ~extern:true "llvm_scalar_opts";;

flag ["link"; "ocaml"; "g++"] (S[A"-cc"; A"g++"; A"-cclib"; A"-rdynamic"]);;
dep ["link"; "ocaml"; "use_bindings"] ["bindings.o"];;
```

token.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer Tokens
 *===----------------------------------------------------------------------===*)
```

```
(* The lexer returns these 'Kwd' if it is an unknown character, otherwise one of
 * these others for known things. *)
type token =
  (* commands *)
  | Def | Extern

  (* primary *)
  | Ident of string | Number of float

  (* unknown *)
  | Kwd of char

  (* control *)
  | If | Then | Else
  | For | In

  (* operators *)
  | Binary | Unary

  (* var definition *)
  | Var
```

lexer.ml:

```
(*===----------------------------------------------------------------------===
 * Lexer
 *===----------------------------------------------------------------------===*)

let rec lex = parser
  (* Skip any whitespace. *)
  | [< ' (' ' | '\n' | '\r' | '\t'); stream >] -> lex stream

  (* identifier: [a-zA-Z][a-zA-Z0-9] *)
  | [< ' ('A' .. 'Z' | 'a' .. 'z' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_ident buffer stream

  (* number: [0-9.]+ *)
  | [< ' ('0' .. '9' as c); stream >] ->
      let buffer = Buffer.create 1 in
      Buffer.add_char buffer c;
      lex_number buffer stream

  (* Comment until end of line. *)
  | [< ' ('#'); stream >] ->
      lex_comment stream

  (* Otherwise, just return the character as its ascii value. *)
  | [< 'c; stream >] ->
      [< 'Token.Kwd c; lex stream >]

  (* end of stream. *)
  | [< >] -> [< >]

and lex_number buffer = parser
  | [< ' ('0' .. '9' | '.' as c); stream >] ->
      Buffer.add_char buffer c;
```

```
        lex_number buffer stream
    | [< stream=lex >] ->
        [< 'Token.Number (float_of_string (Buffer.contents buffer)); stream >]

and lex_ident buffer = parser
    | [< ' ('A' .. 'Z' | 'a' .. 'z' | '0' .. '9' as c); stream >] ->
        Buffer.add_char buffer c;
        lex_ident buffer stream
    | [< stream=lex >] ->
        match Buffer.contents buffer with
        | "def" -> [< 'Token.Def; stream >]
        | "extern" -> [< 'Token.Extern; stream >]
        | "if" -> [< 'Token.If; stream >]
        | "then" -> [< 'Token.Then; stream >]
        | "else" -> [< 'Token.Else; stream >]
        | "for" -> [< 'Token.For; stream >]
        | "in" -> [< 'Token.In; stream >]
        | "binary" -> [< 'Token.Binary; stream >]
        | "unary" -> [< 'Token.Unary; stream >]
        | "var" -> [< 'Token.Var; stream >]
        | id -> [< 'Token.Ident id; stream >]

and lex_comment = parser
    | [< ' ('\n'); stream=lex >] -> stream
    | [< 'c; e=lex_comment >] -> e
    | [< >] -> [< >]
```

ast.ml:

```
(*===----------------------------------------------------------------------===
 * Abstract Syntax Tree (aka Parse Tree)
 *===----------------------------------------------------------------------===*)

(* expr - Base type for all expression nodes. *)
type expr =
  (* variant for numeric literals like "1.0". *)
  | Number of float

  (* variant for referencing a variable, like "a". *)
  | Variable of string

  (* variant for a unary operator. *)
  | Unary of char * expr

  (* variant for a binary operator. *)
  | Binary of char * expr * expr

  (* variant for function calls. *)
  | Call of string * expr array

  (* variant for if/then/else. *)
  | If of expr * expr * expr

  (* variant for for/in. *)
  | For of string * expr * expr * expr option * expr

  (* variant for var/in. *)
  | Var of (string * expr option) array * expr
```

```
(* proto - This type represents the "prototype" for a function, which captures
 * its name, and its argument names (thus implicitly the number of arguments the
 * function takes). *)
type proto =
  | Prototype of string * string array
  | BinOpPrototype of string * string array * int

(* func - This type represents a function definition itself. *)
type func = Function of proto * expr
```

parser.ml:

```
(*===----------------------------------------------------------------------===
 * Parser
 *===----------------------------------------------------------------------===*)

(* binop_precedence - This holds the precedence for each binary operator that is
 * defined *)
let binop_precedence:(char, int) Hashtbl.t = Hashtbl.create 10

(* precedence - Get the precedence of the pending binary operator token. *)
let precedence c = try Hashtbl.find binop_precedence c with Not_found -> -1

(* primary
 *   ::= identifier
 *   ::= numberexpr
 *   ::= parenexpr
 *   ::= ifexpr
 *   ::= forexpr
 *   ::= varexpr *)
let rec parse_primary = parser
  (* numberexpr ::= number *)
  | [< 'Token.Number n >] -> Ast.Number n

  (* parenexpr ::= '(' expression ')' *)
  | [< 'Token.Kwd '('; e=parse_expr; 'Token.Kwd ')' ?? "expected ')'" >] -> e

  (* identifierexpr
   *   ::= identifier
   *   ::= identifier '(' argumentexpr ')' *)
  | [< 'Token.Ident id; stream >] ->
      let rec parse_args accumulator = parser
        | [< e=parse_expr; stream >] ->
            begin parser
              | [< 'Token.Kwd ','; e=parse_args (e :: accumulator) >] -> e
              | [< >] -> e :: accumulator
            end stream
        | [< >] -> accumulator
      in
      let rec parse_ident id = parser
        (* Call. *)
        | [< 'Token.Kwd '(';
             args=parse_args [];
             'Token.Kwd ')' ?? "expected ')'">] ->
            Ast.Call (id, Array.of_list (List.rev args))

        (* Simple variable ref. *)
        | [< >] -> Ast.Variable id
```

```
      in
      parse_ident id stream

  (* ifexpr ::= 'if' expr 'then' expr 'else' expr *)
  | [< 'Token.If; c=parse_expr;
       'Token.Then ?? "expected 'then'"; t=parse_expr;
       'Token.Else ?? "expected 'else'"; e=parse_expr >] ->
      Ast.If (c, t, e)

  (* forexpr
        ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression *)
  | [< 'Token.For;
       'Token.Ident id ?? "expected identifier after for";
       'Token.Kwd '=' ?? "expected '=' after for";
       stream >] ->
      begin parser
        | [<
             start=parse_expr;
             'Token.Kwd ',' ?? "expected ',' after for";
             end_=parse_expr;
             stream >] ->
            let step =
              begin parser
              | [< 'Token.Kwd ','; step=parse_expr >] -> Some step
              | [< >] -> None
              end stream
            in
            begin parser
            | [< 'Token.In; body=parse_expr >] ->
                Ast.For (id, start, end_, step, body)
            | [< >] ->
                raise (Stream.Error "expected 'in' after for")
            end stream
        | [< >] ->
            raise (Stream.Error "expected '=' after for")
      end stream

  (* varexpr
   *   ::= 'var' identifier ('=' expression?
   *             (',' identifier ('=' expression)?)* 'in' expression *)
  | [< 'Token.Var;
       (* At least one variable name is required. *)
       'Token.Ident id ?? "expected identifier after var";
       init=parse_var_init;
       var_names=parse_var_names [(id, init)];
       (* At this point, we have to have 'in'. *)
       'Token.In ?? "expected 'in' keyword after 'var'";
       body=parse_expr >] ->
      Ast.Var (Array.of_list (List.rev var_names), body)

  | [< >] -> raise (Stream.Error "unknown token when expecting an expression.")

(* unary
 *   ::= primary
 *   ::= '!' unary *)
and parse_unary = parser
  (* If this is a unary operator, read it. *)
  | [< 'Token.Kwd op when op != '(' && op != ')'; operand=parse_expr >] ->
```

```
      Ast.Unary (op, operand)

  (* If the current token is not an operator, it must be a primary expr. *)
  | [< stream >] -> parse_primary stream

(* binoprhs
 *   ::= ('+' primary)* *)
and parse_bin_rhs expr_prec lhs stream =
  match Stream.peek stream with
  (* If this is a binop, find its precedence. *)
  | Some (Token.Kwd c) when Hashtbl.mem binop_precedence c ->
      let token_prec = precedence c in

      (* If this is a binop that binds at least as tightly as the current binop,
       * consume it, otherwise we are done. *)
      if token_prec < expr_prec then lhs else begin
        (* Eat the binop. *)
        Stream.junk stream;

        (* Parse the primary expression after the binary operator. *)
        let rhs = parse_unary stream in

        (* Okay, we know this is a binop. *)
        let rhs =
          match Stream.peek stream with
          | Some (Token.Kwd c2) ->
              (* If BinOp binds less tightly with rhs than the operator after
               * rhs, let the pending operator take rhs as its lhs. *)
              let next_prec = precedence c2 in
              if token_prec < next_prec
              then parse_bin_rhs (token_prec + 1) rhs stream
              else rhs
          | _ -> rhs
        in

        (* Merge lhs/rhs. *)
        let lhs = Ast.Binary (c, lhs, rhs) in
        parse_bin_rhs expr_prec lhs stream
      end
  | _ -> lhs

and parse_var_init = parser
  (* read in the optional initializer. *)
  | [< 'Token.Kwd '='; e=parse_expr >] -> Some e
  | [< >] -> None

and parse_var_names accumulator = parser
  | [< 'Token.Kwd ',';
       'Token.Ident id ?? "expected identifier list after var";
       init=parse_var_init;
       e=parse_var_names ((id, init) :: accumulator) >] -> e
  | [< >] -> accumulator

(* expression
 *   ::= primary binoprhs *)
and parse_expr = parser
  | [< lhs=parse_unary; stream >] -> parse_bin_rhs 0 lhs stream
```

```
(* prototype
 *   ::= id '(' id* ')'
 *   ::= binary LETTER number? (id, id)
 *   ::= unary LETTER number? (id) *)
let parse_prototype =
  let rec parse_args accumulator = parser
    | [< 'Token.Ident id; e=parse_args (id::accumulator) >] -> e
    | [< >] -> accumulator
  in
  let parse_operator = parser
    | [< 'Token.Unary >] -> "unary", 1
    | [< 'Token.Binary >] -> "binary", 2
  in
  let parse_binary_precedence = parser
    | [< 'Token.Number n >] -> int_of_float n
    | [< >] -> 30
  in
  parser
  | [< 'Token.Ident id;
       'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      (* success. *)
      Ast.Prototype (id, Array.of_list (List.rev args))
  | [< (prefix, kind)=parse_operator;
       'Token.Kwd op ?? "expected an operator";
       (* Read the precedence if present. *)
       binary_precedence=parse_binary_precedence;
       'Token.Kwd '(' ?? "expected '(' in prototype";
       args=parse_args [];
       'Token.Kwd ')' ?? "expected ')' in prototype" >] ->
      let name = prefix ^ (String.make 1 op) in
      let args = Array.of_list (List.rev args) in

      (* Verify right number of arguments for operator. *)
      if Array.length args != kind
      then raise (Stream.Error "invalid number of operands for operator")
      else
        if kind == 1 then
          Ast.Prototype (name, args)
        else
          Ast.BinOpPrototype (name, args, binary_precedence)
  | [< >] ->
      raise (Stream.Error "expected function name in prototype")

(* definition ::= 'def' prototype expression *)
let parse_definition = parser
  | [< 'Token.Def; p=parse_prototype; e=parse_expr >] ->
      Ast.Function (p, e)

(* toplevelexpr ::= expression *)
let parse_toplevel = parser
  | [< e=parse_expr >] ->
      (* Make an anonymous proto. *)
      Ast.Function (Ast.Prototype ("", [||]), e)

(*  external ::= 'extern' prototype *)
let parse_extern = parser
```

```
    | [< 'Token.Extern; e=parse_prototype >] -> e
```

codegen.ml:

```
(*===----------------------------------------------------------------------===
 * Code Generation
 *===----------------------------------------------------------------------===*)

open Llvm

exception Error of string

let context = global_context ()
let the_module = create_module context "my cool jit"
let builder = builder context
let named_values:(string, llvalue) Hashtbl.t = Hashtbl.create 10
let double_type = double_type context

(* Create an alloca instruction in the entry block of the function. This
 * is used for mutable variables etc. *)
let create_entry_block_alloca the_function var_name =
  let builder = builder_at context (instr_begin (entry_block the_function)) in
  build_alloca double_type var_name builder

let rec codegen_expr = function
  | Ast.Number n -> const_float double_type n
  | Ast.Variable name ->
      let v = try Hashtbl.find named_values name with
        | Not_found -> raise (Error "unknown variable name")
      in
      (* Load the value. *)
      build_load v name builder
  | Ast.Unary (op, operand) ->
      let operand = codegen_expr operand in
      let callee = "unary" ^ (String.make 1 op) in
      let callee =
        match lookup_function callee the_module with
        | Some callee -> callee
        | None -> raise (Error "unknown unary operator")
      in
      build_call callee [|operand|] "unop" builder
  | Ast.Binary (op, lhs, rhs) ->
      begin match op with
      | '=' ->
          (* Special case '=' because we don't want to emit the LHS as an
           * expression. *)
          let name =
            match lhs with
            | Ast.Variable name -> name
            | _ -> raise (Error "destination of '=' must be a variable")
          in

          (* Codegen the rhs. *)
          let val_ = codegen_expr rhs in

          (* Lookup the name. *)
          let variable = try Hashtbl.find named_values name with
          | Not_found -> raise (Error "unknown variable name")
```

```
        in
        ignore(build_store val_ variable builder);
        val_
    | _ ->
        let lhs_val = codegen_expr lhs in
        let rhs_val = codegen_expr rhs in
        begin
          match op with
          | '+' -> build_add lhs_val rhs_val "addtmp" builder
          | '-' -> build_sub lhs_val rhs_val "subtmp" builder
          | '*' -> build_mul lhs_val rhs_val "multmp" builder
          | '<' ->
              (* Convert bool 0/1 to double 0.0 or 1.0 *)
              let i = build_fcmp Fcmp.Ult lhs_val rhs_val "cmptmp" builder in
              build_uitofp i double_type "booltmp" builder
          | _ ->
              (* If it wasn't a builtin binary operator, it must be a user defined
               * one. Emit a call to it. *)
              let callee = "binary" ^ (String.make 1 op) in
              let callee =
                match lookup_function callee the_module with
                | Some callee -> callee
                | None -> raise (Error "binary operator not found!")
              in
              build_call callee [|lhs_val; rhs_val|] "binop" builder
        end
  end
| Ast.Call (callee, args) ->
    (* Look up the name in the module table. *)
    let callee =
      match lookup_function callee the_module with
      | Some callee -> callee
      | None -> raise (Error "unknown function referenced")
    in
    let params = params callee in

    (* If argument mismatch error. *)
    if Array.length params == Array.length args then () else
      raise (Error "incorrect # arguments passed");
    let args = Array.map codegen_expr args in
    build_call callee args "calltmp" builder
| Ast.If (cond, then_, else_) ->
    let cond = codegen_expr cond in

    (* Convert condition to a bool by comparing equal to 0.0 *)
    let zero = const_float double_type 0.0 in
    let cond_val = build_fcmp Fcmp.One cond zero "ifcond" builder in

    (* Grab the first block so that we might later add the conditional branch
     * to it at the end of the function. *)
    let start_bb = insertion_block builder in
    let the_function = block_parent start_bb in

    let then_bb = append_block context "then" the_function in

    (* Emit 'then' value. *)
    position_at_end then_bb builder;
    let then_val = codegen_expr then_ in
```

```
      (* Codegen of 'then' can change the current block, update then_bb for the
       * phi. We create a new name because one is used for the phi node, and the
       * other is used for the conditional branch. *)
      let new_then_bb = insertion_block builder in

      (* Emit 'else' value. *)
      let else_bb = append_block context "else" the_function in
      position_at_end else_bb builder;
      let else_val = codegen_expr else_ in

      (* Codegen of 'else' can change the current block, update else_bb for the
       * phi. *)
      let new_else_bb = insertion_block builder in

      (* Emit merge block. *)
      let merge_bb = append_block context "ifcont" the_function in
      position_at_end merge_bb builder;
      let incoming = [(then_val, new_then_bb); (else_val, new_else_bb)] in
      let phi = build_phi incoming "iftmp" builder in

      (* Return to the start block to add the conditional branch. *)
      position_at_end start_bb builder;
      ignore (build_cond_br cond_val then_bb else_bb builder);

      (* Set a unconditional branch at the end of the 'then' block and the
       * 'else' block to the 'merge' block. *)
      position_at_end new_then_bb builder; ignore (build_br merge_bb builder);
      position_at_end new_else_bb builder; ignore (build_br merge_bb builder);

      (* Finally, set the builder to the end of the merge block. *)
      position_at_end merge_bb builder;

      phi
  | Ast.For (var_name, start, end_, step, body) ->
      (* Output this as:
       *   var = alloca double
       *   ...
       *   start = startexpr
       *   store start -> var
       *   goto loop
       * loop:
       *   ...
       *   bodyexpr
       *   ...
       * loopend:
       *   step = stepexpr
       *   endcond = endexpr
       *
       *   curvar = load var
       *   nextvar = curvar + step
       *   store nextvar -> var
       *   br endcond, loop, endloop
       * outloop: *)

      let the_function = block_parent (insertion_block builder) in

      (* Create an alloca for the variable in the entry block. *)
      let alloca = create_entry_block_alloca the_function var_name in
```

```
(* Emit the start code first, without 'variable' in scope. *)
let start_val = codegen_expr start in

(* Store the value into the alloca. *)
ignore(build_store start_val alloca builder);

(* Make the new basic block for the loop header, inserting after current
 * block. *)
let loop_bb = append_block context "loop" the_function in

(* Insert an explicit fall through from the current block to the
 * loop_bb. *)
ignore (build_br loop_bb builder);

(* Start insertion in loop_bb. *)
position_at_end loop_bb builder;

(* Within the loop, the variable is defined equal to the PHI node. If it
 * shadows an existing variable, we have to restore it, so save it
 * now. *)
let old_val =
  try Some (Hashtbl.find named_values var_name) with Not_found -> None
in
Hashtbl.add named_values var_name alloca;

(* Emit the body of the loop.  This, like any other expr, can change the
 * current BB.  Note that we ignore the value computed by the body, but
 * don't allow an error *)
ignore (codegen_expr body);

(* Emit the step value. *)
let step_val =
  match step with
  | Some step -> codegen_expr step
  (* If not specified, use 1.0. *)
  | None -> const_float double_type 1.0
in

(* Compute the end condition. *)
let end_cond = codegen_expr end_ in

(* Reload, increment, and restore the alloca. This handles the case where
 * the body of the loop mutates the variable. *)
let cur_var = build_load alloca var_name builder in
let next_var = build_add cur_var step_val "nextvar" builder in
ignore(build_store next_var alloca builder);

(* Convert condition to a bool by comparing equal to 0.0. *)
let zero = const_float double_type 0.0 in
let end_cond = build_fcmp Fcmp.One end_cond zero "loopcond" builder in

(* Create the "after loop" block and insert it. *)
let after_bb = append_block context "afterloop" the_function in

(* Insert the conditional branch into the end of loop_end_bb. *)
ignore (build_cond_br end_cond loop_bb after_bb builder);

(* Any new code will be inserted in after_bb. *)
```

```
      position_at_end after_bb builder;

      (* Restore the unshadowed variable. *)
      begin match old_val with
      | Some old_val -> Hashtbl.add named_values var_name old_val
      | None -> ()
      end;

      (* for expr always returns 0.0. *)
      const_null double_type
  | Ast.Var (var_names, body) ->
      let old_bindings = ref [] in

      let the_function = block_parent (insertion_block builder) in

      (* Register all variables and emit their initializer. *)
      Array.iter (fun (var_name, init) ->
        (* Emit the initializer before adding the variable to scope, this
         * prevents the initializer from referencing the variable itself, and
         * permits stuff like this:
         *   var a = 1 in
         *     var a = a in ...   # refers to outer 'a'. *)
        let init_val =
          match init with
          | Some init -> codegen_expr init
          (* If not specified, use 0.0. *)
          | None -> const_float double_type 0.0
        in

        let alloca = create_entry_block_alloca the_function var_name in
        ignore(build_store init_val alloca builder);

        (* Remember the old variable binding so that we can restore the binding
         * when we unrecurse. *)
        begin
          try
            let old_value = Hashtbl.find named_values var_name in
            old_bindings := (var_name, old_value) :: !old_bindings;
          with Not_found -> ()
        end;

        (* Remember this binding. *)
        Hashtbl.add named_values var_name alloca;
      ) var_names;

      (* Codegen the body, now that all vars are in scope. *)
      let body_val = codegen_expr body in

      (* Pop all our variables from scope. *)
      List.iter (fun (var_name, old_value) ->
        Hashtbl.add named_values var_name old_value
      ) !old_bindings;

      (* Return the body computation. *)
      body_val

let codegen_proto = function
  | Ast.Prototype (name, args) | Ast.BinOpPrototype (name, args, _) ->
```

```
      (* Make the function type: double(double,double) etc. *)
      let doubles = Array.make (Array.length args) double_type in
      let ft = function_type double_type doubles in
      let f =
        match lookup_function name the_module with
        | None -> declare_function name ft the_module

        (* If 'f' conflicted, there was already something named 'name'. If it
         * has a body, don't allow redefinition or reextern. *)
        | Some f ->
            (* If 'f' already has a body, reject this. *)
            if block_begin f <> At_end f then
              raise (Error "redefinition of function");

            (* If 'f' took a different number of arguments, reject. *)
            if element_type (type_of f) <> ft then
              raise (Error "redefinition of function with different # args");
            f
      in

      (* Set names for all arguments. *)
      Array.iteri (fun i a ->
        let n = args.(i) in
        set_value_name n a;
        Hashtbl.add named_values n a;
      ) (params f);
      f

(* Create an alloca for each argument and register the argument in the symbol
 * table so that references to it will succeed. *)
let create_argument_allocas the_function proto =
  let args = match proto with
    | Ast.Prototype (_, args) | Ast.BinOpPrototype (_, args, _) -> args
  in
  Array.iteri (fun i ai ->
    let var_name = args.(i) in
    (* Create an alloca for this variable. *)
    let alloca = create_entry_block_alloca the_function var_name in

    (* Store the initial value into the alloca. *)
    ignore(build_store ai alloca builder);

    (* Add arguments to variable symbol table. *)
    Hashtbl.add named_values var_name alloca;
  ) (params the_function)

let codegen_func the_fpm = function
  | Ast.Function (proto, body) ->
      Hashtbl.clear named_values;
      let the_function = codegen_proto proto in

      (* If this is an operator, install it. *)
      begin match proto with
      | Ast.BinOpPrototype (name, args, prec) ->
          let op = name.[String.length name - 1] in
          Hashtbl.add Parser.binop_precedence op prec;
      | _ -> ()
      end;
```

```
      (* Create a new basic block to start insertion into. *)
      let bb = append_block context "entry" the_function in
      position_at_end bb builder;

      try
        (* Add all arguments to the symbol table and create their allocas. *)
        create_argument_allocas the_function proto;

        let ret_val = codegen_expr body in

        (* Finish off the function. *)
        let _ = build_ret ret_val builder in

        (* Validate the generated code, checking for consistency. *)
        Llvm_analysis.assert_valid_function the_function;

        (* Optimize the function. *)
        let _ = PassManager.run_function the_function the_fpm in

        the_function
      with e ->
        delete_function the_function;
        raise e
```

toplevel.ml:

```
(*===----------------------------------------------------------------------===
 * Top-Level parsing and JIT Driver
 *===----------------------------------------------------------------------===*)

open Llvm
open Llvm_executionengine

(* top ::= definition | external | expression | ';' *)
let rec main_loop the_fpm the_execution_engine stream =
  match Stream.peek stream with
  | None -> ()

  (* ignore top-level semicolons. *)
  | Some (Token.Kwd ';') ->
      Stream.junk stream;
      main_loop the_fpm the_execution_engine stream

  | Some token ->
      begin
        try match token with
        | Token.Def ->
            let e = Parser.parse_definition stream in
            print_endline "parsed a function definition.";
            dump_value (Codegen.codegen_func the_fpm e);
        | Token.Extern ->
            let e = Parser.parse_extern stream in
            print_endline "parsed an extern.";
            dump_value (Codegen.codegen_proto e);
        | _ ->
            (* Evaluate a top-level expression into an anonymous function. *)
            let e = Parser.parse_toplevel stream in
            print_endline "parsed a top-level expr";
```

```
            let the_function = Codegen.codegen_func the_fpm e in
            dump_value the_function;

            (* JIT the function, returning a function pointer. *)
            let result = ExecutionEngine.run_function the_function [||]
              the_execution_engine in

            print_string "Evaluated to ";
            print_float (GenericValue.as_float Codegen.double_type result);
            print_newline ();
          with Stream.Error s | Codegen.Error s ->
            (* Skip token for error recovery. *)
            Stream.junk stream;
            print_endline s;
        end;
        print_string "ready> "; flush stdout;
        main_loop the_fpm the_execution_engine stream
```

toy.ml:

```
    (*===----------------------------------------------------------------------===
     * Main driver code.
     *===----------------------------------------------------------------------===*)

    open Llvm
    open Llvm_executionengine
    open Llvm_target
    open Llvm_scalar_opts

    let main () =
      ignore (initialize_native_target ());

      (* Install standard binary operators.
       * 1 is the lowest precedence. *)
      Hashtbl.add Parser.binop_precedence '=' 2;
      Hashtbl.add Parser.binop_precedence '<' 10;
      Hashtbl.add Parser.binop_precedence '+' 20;
      Hashtbl.add Parser.binop_precedence '-' 20;
      Hashtbl.add Parser.binop_precedence '*' 40;    (* highest. *)

      (* Prime the first token. *)
      print_string "ready> "; flush stdout;
      let stream = Lexer.lex (Stream.of_channel stdin) in

      (* Create the JIT. *)
      let the_execution_engine = ExecutionEngine.create Codegen.the_module in
      let the_fpm = PassManager.create_function Codegen.the_module in

      (* Set up the optimizer pipeline.  Start with registering info about how the
       * target lays out data structures. *)
      DataLayout.add (ExecutionEngine.target_data the_execution_engine) the_fpm;

      (* Promote allocas to registers. *)
      add_memory_to_register_promotion the_fpm;

      (* Do simple "peephole" optimizations and bit-twiddling optzn. *)
      add_instruction_combination the_fpm;
```

```
  (* reassociate expressions. *)
  add_reassociation the_fpm;

  (* Eliminate Common SubExpressions. *)
  add_gvn the_fpm;

  (* Simplify the control flow graph (deleting unreachable blocks, etc). *)
  add_cfg_simplification the_fpm;

  ignore (PassManager.initialize the_fpm);

  (* Run the main "interpreter loop" now. *)
  Toplevel.main_loop the_fpm the_execution_engine stream;

  (* Print out all the generated code. *)
  dump_module Codegen.the_module
;;

main ()
```

bindings.c

```
#include <stdio.h>

/* putchard - putchar that takes a double and returns 0. */
extern double putchard(double X) {
  putchar((char)X);
  return 0;
}

/* printd - printf that takes a double prints it as "%f\n", returning 0. */
extern double printd(double X) {
  printf("%f\n", X);
  return 0;
}
```

Next: Conclusion and other useful LLVM tidbits

Kaleidoscope: Conclusion and other useful LLVM tidbits

Tutorial Conclusion

Welcome to the final chapter of the "Implementing a language with LLVM" tutorial. In the course of this tutorial, we have grown our little Kaleidoscope language from being a useless toy, to being a semi-interesting (but probably still

useless) toy. :)

It is interesting to see how far we've come, and how little code it has taken. We built the entire lexer, parser, AST, code generator, and an interactive run-loop (with a JIT!) by-hand in under 700 lines of (non-comment/non-blank) code.

Our little language supports a couple of interesting features: it supports user defined binary and unary operators, it uses JIT compilation for immediate evaluation, and it supports a few control flow constructs with SSA construction.

Part of the idea of this tutorial was to show you how easy and fun it can be to define, build, and play with languages. Building a compiler need not be a scary or mystical process! Now that you've seen some of the basics, I strongly encourage you to take the code and hack on it. For example, try adding:

- **global variables** - While global variables have questional value in modern software engineering, they are often useful when putting together quick little hacks like the Kaleidoscope compiler itself. Fortunately, our current setup makes it very easy to add global variables: just have value lookup check to see if an unresolved variable is in the global variable symbol table before rejecting it. To create a new global variable, make an instance of the LLVM `GlobalVariable` class.
- **typed variables** - Kaleidoscope currently only supports variables of type double. This gives the language a very nice elegance, because only supporting one type means that you never have to specify types. Different languages have different ways of handling this. The easiest way is to require the user to specify types for every variable definition, and record the type of the variable in the symbol table along with its Value*.
- **arrays, structs, vectors, etc** - Once you add types, you can start extending the type system in all sorts of interesting ways. Simple arrays are very easy and are quite useful for many different applications. Adding them is mostly an exercise in learning how the LLVM getelementptr instruction works: it is so nifty/unconventional, it has its own FAQ! If you add support for recursive types (e.g. linked lists), make sure to read the section in the LLVM Programmer's Manual that describes how to construct them.
- **standard runtime** - Our current language allows the user to access arbitrary external functions, and we use it for things like "printd" and "putchard". As you extend the language to add higher-level constructs, often these constructs make the most sense if they are lowered to calls into a language-supplied runtime. For example, if you add hash tables to the language, it would probably make sense to add the routines to a runtime, instead of inlining them all the way.
- **memory management** - Currently we can only access the stack in Kaleidoscope. It would also be useful to be able to allocate heap memory, either with calls to the standard libc malloc/free interface or with a garbage collector. If you would like to use garbage collection, note that LLVM fully supports Accurate Garbage Collection including algorithms that move objects and need to scan/update the stack.
- **debugger support** - LLVM supports generation of DWARF Debug info which is understood by common debuggers like GDB. Adding support for debug info is fairly straightforward. The best way to understand it is to compile some C/C++ code with "`clang -g -O0`" and taking a look at what it produces.
- **exception handling support** - LLVM supports generation of zero cost exceptions which interoperate with code compiled in other languages. You could also generate code by implicitly making every function return an error value and checking it. You could also make explicit use of setjmp/longjmp. There are many different ways to go here.
- **object orientation, generics, database access, complex numbers, geometric programming, ...** - Really, there is no end of crazy features that you can add to the language.
- **unusual domains** - We've been talking about applying LLVM to a domain that many people are interested in: building a compiler for a specific language. However, there are many other domains that can use compiler technology that are not typically considered. For example, LLVM has been used to implement OpenGL graphics acceleration, translate C++ code to ActionScript, and many other cute and clever things. Maybe you will be the first to JIT compile a regular expression interpreter into native code with LLVM?

Have fun - try doing something crazy and unusual. Building a language like everyone else always has, is much less fun than trying something a little crazy or off the wall and seeing how it turns out. If you get stuck or want to talk

about it, feel free to email the llvmdev mailing list: it has lots of people who are interested in languages and are often willing to help out.

Before we end this tutorial, I want to talk about some "tips and tricks" for generating LLVM IR. These are some of the more subtle things that may not be obvious, but are very useful if you want to take advantage of LLVM's capabilities.

Properties of the LLVM IR

We have a couple common questions about code in the LLVM IR form - lets just get these out of the way right now, shall we?

Target Independence Kaleidoscope is an example of a "portable language": any program written in Kaleidoscope will work the same way on any target that it runs on. Many other languages have this property, e.g. lisp, java, haskell, javascript, python, etc (note that while these languages are portable, not all their libraries are).

One nice aspect of LLVM is that it is often capable of preserving target independence in the IR: you can take the LLVM IR for a Kaleidoscope-compiled program and run it on any target that LLVM supports, even emitting C code and compiling that on targets that LLVM doesn't support natively. You can trivially tell that the Kaleidoscope compiler generates target-independent code because it never queries for any target-specific information when generating code.

The fact that LLVM provides a compact, target-independent, representation for code gets a lot of people excited. Unfortunately, these people are usually thinking about C or a language from the C family when they are asking questions about language portability. I say "unfortunately", because there is really no way to make (fully general) C code portable, other than shipping the source code around (and of course, C source code is not actually portable in general either - ever port a really old application from 32- to 64-bits?).

The problem with C (again, in its full generality) is that it is heavily laden with target specific assumptions. As one simple example, the preprocessor often destructively removes target-independence from the code when it processes the input text:

```
#ifdef __i386__
  int X = 1;
#else
  int X = 42;
#endif
```

While it is possible to engineer more and more complex solutions to problems like this, it cannot be solved in full generality in a way that is better than shipping the actual source code.

That said, there are interesting subsets of C that can be made portable. If you are willing to fix primitive types to a fixed size (say int = 32-bits, and long = 64-bits), don't care about ABI compatibility with existing binaries, and are willing to give up some other minor features, you can have portable code. This can make sense for specialized domains such as an in-kernel language.

Safety Guarantees Many of the languages above are also "safe" languages: it is impossible for a program written in Java to corrupt its address space and crash the process (assuming the JVM has no bugs). Safety is an interesting property that requires a combination of language design, runtime support, and often operating system support.

It is certainly possible to implement a safe language in LLVM, but LLVM IR does not itself guarantee safety. The LLVM IR allows unsafe pointer casts, use after free bugs, buffer over-runs, and a variety of other problems. Safety needs to be implemented as a layer on top of LLVM and, conveniently, several groups have investigated this. Ask on the llvmdev mailing list if you are interested in more details.

Language-Specific Optimizations One thing about LLVM that turns off many people is that it does not solve all the world's problems in one system (sorry 'world hunger', someone else will have to solve you some other day). One specific complaint is that people perceive LLVM as being incapable of performing high-level language-specific optimization: LLVM "loses too much information".

Unfortunately, this is really not the place to give you a full and unified version of "Chris Lattner's theory of compiler design". Instead, I'll make a few observations:

First, you're right that LLVM does lose information. For example, as of this writing, there is no way to distinguish in the LLVM IR whether an SSA-value came from a C "int" or a C "long" on an ILP32 machine (other than debug info). Both get compiled down to an 'i32' value and the information about what it came from is lost. The more general issue here, is that the LLVM type system uses "structural equivalence" instead of "name equivalence". Another place this surprises people is if you have two types in a high-level language that have the same structure (e.g. two different structs that have a single int field): these types will compile down into a single LLVM type and it will be impossible to tell what it came from.

Second, while LLVM does lose information, LLVM is not a fixed target: we continue to enhance and improve it in many different ways. In addition to adding new features (LLVM did not always support exceptions or debug info), we also extend the IR to capture important information for optimization (e.g. whether an argument is sign or zero extended, information about pointers aliasing, etc). Many of the enhancements are user-driven: people want LLVM to include some specific feature, so they go ahead and extend it.

Third, it is *possible and easy* to add language-specific optimizations, and you have a number of choices in how to do it. As one trivial example, it is easy to add language-specific optimization passes that "know" things about code compiled for a language. In the case of the C family, there is an optimization pass that "knows" about the standard C library functions. If you call "exit(0)" in main(), it knows that it is safe to optimize that into "return 0;" because C specifies what the 'exit' function does.

In addition to simple library knowledge, it is possible to embed a variety of other language-specific information into the LLVM IR. If you have a specific need and run into a wall, please bring the topic up on the llvmdev list. At the very worst, you can always treat LLVM as if it were a "dumb code generator" and implement the high-level optimizations you desire in your front-end, on the language-specific AST.

Tips and Tricks

There is a variety of useful tips and tricks that you come to know after working on/with LLVM that aren't obvious at first glance. Instead of letting everyone rediscover them, this section talks about some of these issues.

Implementing portable offsetof/sizeof One interesting thing that comes up, if you are trying to keep the code generated by your compiler "target independent", is that you often need to know the size of some LLVM type or the offset of some field in an llvm structure. For example, you might need to pass the size of a type into a function that allocates memory.

Unfortunately, this can vary widely across targets: for example the width of a pointer is trivially target-specific. However, there is a clever way to use the getelementptr instruction that allows you to compute this in a portable way.

Garbage Collected Stack Frames Some languages want to explicitly manage their stack frames, often so that they are garbage collected or to allow easy implementation of closures. There are often better ways to implement these features than explicit stack frames, but LLVM does support them, if you want. It requires your front-end to convert the code into Continuation Passing Style and the use of tail calls (which LLVM also supports).

2.15.3 External Tutorials

Tutorial: Creating an LLVM Backend for the Cpu0 Architecture A step-by-step tutorial for developing an LLVM backend. Under active development at https://github.com/Jonathan2251/lbd (please contribute!).

Howto: Implementing LLVM Integrated Assembler A simple guide for how to implement an LLVM integrated assembler for an architecture.

2.15.4 Advanced Topics

1. Writing an Optimization for LLVM

2.16 LLVM 3.5 Release Notes

Warning: These are in-progress notes for the upcoming LLVM 3.6 release. You may prefer the LLVM 3.5 Release Notes.

2.16.1 Introduction

This document contains the release notes for the LLVM Compiler Infrastructure, release 3.6. Here we describe the status of LLVM, including major improvements from the previous release, improvements in various subprojects of LLVM, and some of the current users of the code. All LLVM releases may be downloaded from the LLVM releases web site.

For more information about LLVM, including information about the latest release, please check out the main LLVM web site. If you have questions or comments, the LLVM Developer's Mailing List is a good place to send them.

Note that if you are reading this file from a Subversion checkout or the main LLVM web page, this document applies to the *next* release, not the current one. To see the release notes for a specific release, please see the releases page.

2.16.2 Non-comprehensive list of changes in this release

- Support for AuroraUX has been removed.
- Added support for a native object file-based bitcode wrapper format.
- ... next change ...

Changes to the ARM Backend

During this release ...

Changes to the MIPS Target

During this release ...

Changes to the PowerPC Target

During this release ...

2.16.3 External Open Source Projects Using LLVM 3.6

An exciting aspect of LLVM is that it is used as an enabling technology for a lot of other language and tools projects. This section lists some of the projects that have already been updated to work with LLVM 3.6.

- A project

2.16.4 Additional Information

A wide variety of additional information is available on the LLVM web page, in particular in the documentation section. The web page also contains versions of the API documentation which is up-to-date with the Subversion version of the source code. You can access versions of these documents specific to this release by going into the `llvm/docs/` directory in the LLVM tree.

If you have any questions or comments about LLVM, please feel free to contact us via the mailing lists.

2.17 LLVM's Analysis and Transform Passes

2.17.1 Introduction

This document serves as a high level summary of the optimization features that LLVM provides. Optimizations are implemented as Passes that traverse some portion of a program to either collect information or transform the program. The table below divides the passes that LLVM provides into three categories. Analysis passes compute information that other passes can use or for debugging or program visualization purposes. Transform passes can use (or invalidate) the analysis passes. Transform passes all mutate the program in some way. Utility passes provides some utility but don't otherwise fit categorization. For example passes to extract functions to bitcode or write a module to bitcode are neither analysis nor transform passes. The table of contents above provides a quick summary of each pass and links to the more complete pass description later in the document.

2.17.2 Analysis Passes

This section describes the LLVM Analysis Passes.

`-aa-eval`: Exhaustive Alias Analysis Precision Evaluator

This is a simple N^2 alias analysis accuracy evaluator. Basically, for each function in the program, it simply queries to see how the alias analysis implementation answers alias queries between each pair of pointers in the function.

This is inspired and adapted from code by: Naveen Neelakantam, Francesco Spadini, and Wojciech Stryjewski.

`-basicaa`: Basic Alias Analysis (stateless AA impl)

A basic alias analysis pass that implements identities (two different globals cannot alias, etc), but does no stateful analysis.

`-basiccg`: Basic CallGraph Construction

Yet to be written.

`-count-aa`: Count Alias Analysis Query Responses

A pass which can be used to count how many alias queries are being made and how the alias analysis implementation being used responds.

`-da`: Dependence Analysis

Dependence analysis framework, which is used to detect dependences in memory accesses.

`-debug-aa`: AA use debugger

This simple pass checks alias analysis users to ensure that if they create a new value, they do not query AA without informing it of the value. It acts as a shim over any other AA pass you want.

Yes keeping track of every value in the program is expensive, but this is a debugging pass.

`-domfrontier`: Dominance Frontier Construction

This pass is a simple dominator construction algorithm for finding forward dominator frontiers.

`-domtree`: Dominator Tree Construction

This pass is a simple dominator construction algorithm for finding forward dominators.

`-dot-callgraph`: Print Call Graph to "dot" file

This pass, only available in `opt`, prints the call graph into a `.dot` graph. This graph can then be processed with the "dot" tool to convert it to postscript or some other suitable format.

`-dot-cfg`: Print CFG of function to "dot" file

This pass, only available in `opt`, prints the control flow graph into a `.dot` graph. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

`-dot-cfg-only`: Print CFG of function to "dot" file (with no function bodies)

This pass, only available in `opt`, prints the control flow graph into a `.dot` graph, omitting the function bodies. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

`-dot-dom`: Print dominance tree of function to "dot" file

This pass, only available in `opt`, prints the dominator tree into a `.dot` graph. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

`-dot-dom-only`: Print dominance tree of function to "dot" file (with no function bodies)

This pass, only available in `opt`, prints the dominator tree into a `.dot` graph, omitting the function bodies. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

`-dot-postdom`: Print postdominance tree of function to "dot" file

This pass, only available in `opt`, prints the post dominator tree into a `.dot` graph. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

`-dot-postdom-only`: Print postdominance tree of function to "dot" file (with no function bodies)

This pass, only available in `opt`, prints the post dominator tree into a `.dot` graph, omitting the function bodies. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

`-globalsmodref-aa`: Simple mod/ref analysis for globals

This simple pass provides alias and mod/ref information for global values that do not have their address taken, and keeps track of whether functions read or write memory (are "pure"). For this simple (but very common) case, we can provide pretty accurate and useful information.

`-instcount`: Counts the various types of `Instruction`s

This pass collects the count of all instructions and reports them.

`-intervals`: Interval Partition Construction

This analysis calculates and represents the interval partition of a function, or a preexisting interval partition.

In this way, the interval partition may be used to reduce a flow graph down to its degenerate single node interval partition (unless it is irreducible).

`-iv-users`: Induction Variable Users

Bookkeeping for "interesting" users of expressions computed from induction variables.

`-lazy-value-info`: Lazy Value Information Analysis

Interface for lazy computation of value constraint information.

`-libcall-aa`: LibCall Alias Analysis

LibCall Alias Analysis.

`-lint`: Statically lint-checks LLVM IR

This pass statically checks for common and easily-identified constructs which produce undefined or likely unintended behavior in LLVM IR.

It is not a guarantee of correctness, in two ways. First, it isn't comprehensive. There are checks which could be done statically which are not yet implemented. Some of these are indicated by TODO comments, but those aren't comprehensive either. Second, many conditions cannot be checked statically. This pass does no dynamic instrumentation, so it can't check for all possible problems.

Another limitation is that it assumes all code will be executed. A store through a null pointer in a basic block which is never reached is harmless, but this pass will warn about it anyway.

Optimization passes may make conditions that this pass checks for more or less obvious. If an optimization pass appears to be introducing a warning, it may be that the optimization pass is merely exposing an existing condition in the code.

This code may be run before *instcombine*. In many cases, instcombine checks for the same kinds of things and turns instructions with undefined behavior into unreachable (or equivalent). Because of this, this pass makes some effort to look through bitcasts and so on.

`-loops`: Natural Loop Information

This analysis is used to identify natural loops and determine the loop depth of various nodes of the CFG. Note that the loops identified may actually be several natural loops that share the same header node... not just a single natural loop.

`-memdep`: Memory Dependence Analysis

An analysis that determines, for a given memory operation, what preceding memory operations it depends on. It builds on alias analysis information, and tries to provide a lazy, caching interface to a common kind of alias information query.

`-module-debuginfo`: Decodes module-level debug info

This pass decodes the debug info metadata in a module and prints in a (sufficiently-prepared-) human-readable form.

For example, run this pass from `opt` along with the `-analyze` option, and it'll print to standard output.

`-no-aa`: No Alias Analysis (always returns 'may' alias)

This is the default implementation of the Alias Analysis interface. It always returns "I don't know" for alias queries. NoAA is unlike other alias analysis implementations, in that it does not chain to a previous analysis. As such it doesn't follow many of the rules that other alias analyses must.

`-postdomfrontier`: Post-Dominance Frontier Construction

This pass is a simple post-dominator construction algorithm for finding post-dominator frontiers.

`-postdomtree`: Post-Dominator Tree Construction

This pass is a simple post-dominator construction algorithm for finding post-dominators.

`-print-alias-sets`: Alias Set Printer

Yet to be written.

`-print-callgraph`: Print a call graph

This pass, only available in `opt`, prints the call graph to standard error in a human-readable form.

`-print-callgraph-sccs`: Print SCCs of the Call Graph

This pass, only available in `opt`, prints the SCCs of the call graph to standard error in a human-readable form.

`-print-cfg-sccs`: Print SCCs of each function CFG

This pass, only available in `opt`, printsthe SCCs of each function CFG to standard error in a human-readable fom.

`-print-dom-info`: Dominator Info Printer

Dominator Info Printer.

`-print-externalfnconstants`: Print external fn callsites passed constants

This pass, only available in `opt`, prints out call sites to external functions that are called with constant arguments. This can be useful when looking for standard library functions we should constant fold or handle in alias analyses.

`-print-function`: Print function to stderr

The `PrintFunctionPass` class is designed to be pipelined with other `FunctionPasses`, and prints out the functions of the module as they are processed.

`-print-module`: Print module to stderr

This pass simply prints out the entire module when it is executed.

`-print-used-types`: Find Used Types

This pass is used to seek out all of the types in use by the program. Note that this analysis explicitly does not include types only used by the symbol table.

`-regions`: Detect single entry single exit regions

The `RegionInfo` pass detects single entry single exit regions in a function, where a region is defined as any subgraph that is connected to the remaining graph at only two spots. Furthermore, an hierarchical region tree is built.

`-scalar-evolution`: Scalar Evolution Analysis

The `ScalarEvolution` analysis can be used to analyze and catagorize scalar expressions in loops. It specializes in recognizing general induction variables, representing them with the abstract and opaque `SCEV` class. Given this analysis, trip counts of loops and other important properties can be obtained.

This analysis is primarily useful for induction variable substitution and strength reduction.

`-scev-aa`: ScalarEvolution-based Alias Analysis

Simple alias analysis implemented in terms of `ScalarEvolution` queries.

This differs from traditional loop dependence analysis in that it tests for dependencies within a single iteration of a loop, rather than dependencies between different iterations.

`ScalarEvolution` has a more complete understanding of pointer arithmetic than `BasicAliasAnalysis`‘ collection of ad-hoc analyses.

`-targetdata`: Target Data Layout

Provides other passes access to information on how the size and alignment required by the target ABI for various data types.

2.17.3 Transform Passes

This section describes the LLVM Transform Passes.

`-adce`: Aggressive Dead Code Elimination

ADCE aggressively tries to eliminate code. This pass is similar to *DCE* but it assumes that values are dead until proven otherwise. This is similar to *SCCP*, except applied to the liveness of values.

`-always-inline`: Inliner for `always_inline` functions

A custom inliner that handles only functions that are marked as "always inline".

`-argpromotion`: Promote 'by reference' arguments to scalars

This pass promotes "by reference" arguments to be "by value" arguments. In practice, this means looking for internal functions that have pointer arguments. If it can prove, through the use of alias analysis, that an argument is *only* loaded, then it can pass the value into the function instead of the address of the value. This can cause recursive simplification of code and lead to the elimination of allocas (especially in C++ template code like the STL).

This pass also handles aggregate arguments that are passed into a function, scalarizing them if the elements of the aggregate are only loaded. Note that it refuses to scalarize aggregates which would require passing in more than three operands to the function, because passing thousands of operands for a large array or structure is unprofitable!

Note that this transformation could also be done for arguments that are only stored to (returning the value instead), but does not currently. This case would be best handled when and if LLVM starts supporting multiple return values from functions.

`-bb-vectorize`: Basic-Block Vectorization

This pass combines instructions inside basic blocks to form vector instructions. It iterates over each basic block, attempting to pair compatible instructions, repeating this process until no additional pairs are selected for vectorization. When the outputs of some pair of compatible instructions are used as inputs by some other pair of compatible instructions, those pairs are part of a potential vectorization chain. Instruction pairs are only fused into vector instructions when they are part of a chain longer than some threshold length. Moreover, the pass attempts to find the best possible chain for each pair of compatible instructions. These heuristics are intended to prevent vectorization in cases where it would not yield a performance increase of the resulting code.

`-block-placement`: Profile Guided Basic Block Placement

This pass is a very simple profile guided basic block placement algorithm. The idea is to put frequently executed blocks together at the start of the function and hopefully increase the number of fall-through conditional branches. If there is no profile information for a particular function, this pass basically orders blocks in depth-first order.

`-break-crit-edges`: Break critical edges in CFG

Break all of the critical edges in the CFG by inserting a dummy basic block. It may be "required" by passes that cannot deal with critical edges. This transformation obviously invalidates the CFG, but can update forward dominator (set, immediate dominators, tree, and frontier) information.

`-codegenprepare`: Optimize for code generation

This pass munges the code in the input function to better prepare it for SelectionDAG-based code generation. This works around limitations in its basic-block-at-a-time approach. It should eventually be removed.

`-constmerge`: Merge Duplicate Global Constants

Merges duplicate global constants together into a single constant that is shared. This is useful because some passes (i.e., TraceValues) insert a lot of string constants into the program, regardless of whether or not an existing string is available.

`-constprop`: Simple constant propagation

This pass implements constant propagation and merging. It looks for instructions involving only constant operands and replaces them with a constant value instead of an instruction. For example:

```
add i32 1, 2
```

becomes

```
i32 3
```

NOTE: this pass has a habit of making definitions be dead. It is a good idea to run a *Dead Instruction Elimination* pass sometime after running this pass.

`-dce`: Dead Code Elimination

Dead code elimination is similar to *dead instruction elimination*, but it rechecks instructions that were used by removed instructions to see if they are newly dead.

`-deadargelim`: Dead Argument Elimination

This pass deletes dead arguments from internal functions. Dead argument elimination removes arguments which are directly dead, as well as arguments only passed into function calls as dead arguments of other functions. This pass also deletes dead arguments in a similar way.

This pass is often useful as a cleanup pass to run after aggressive interprocedural passes, which add possibly-dead arguments.

`-deadtypeelim`: Dead Type Elimination

This pass is used to cleanup the output of GCC. It eliminate names for types that are unused in the entire translation unit, using the *find used types* pass.

`-die`: Dead Instruction Elimination

Dead instruction elimination performs a single pass over the function, removing instructions that are obviously dead.

`-dse`: Dead Store Elimination

A trivial dead store elimination that only considers basic-block local redundant stores.

`-functionattrs`: Deduce function attributes

A simple interprocedural pass which walks the call-graph, looking for functions which do not access or only read non-local memory, and marking them `readnone`/`readonly`. In addition, it marks function arguments (of pointer type) "`nocapture`" if a call to the function does not create any copies of the pointer value that outlive the call. This more or less means that the pointer is only dereferenced, and not returned from the function or stored in a global. This pass is implemented as a bottom-up traversal of the call-graph.

`-globaldce`: Dead Global Elimination

This transform is designed to eliminate unreachable internal globals from the program. It uses an aggressive algorithm, searching out globals that are known to be alive. After it finds all of the globals which are needed, it deletes whatever is left over. This allows it to delete recursive chunks of the program which are unreachable.

`-globalopt`: Global Variable Optimizer

This pass transforms simple global variables that never have their address taken. If obviously true, it marks read/write globals as constant, deletes variables only stored to, etc.

`-gvn`: Global Value Numbering

This pass performs global value numbering to eliminate fully and partially redundant instructions. It also performs redundant load elimination.

`-indvars`: Canonicalize Induction Variables

This transformation analyzes and transforms the induction variables (and computations derived from them) into simpler forms suitable for subsequent analysis and transformation.

This transformation makes the following changes to each loop with an identifiable induction variable:

- All loops are transformed to have a *single* canonical induction variable which starts at zero and steps by one.
- The canonical induction variable is guaranteed to be the first PHI node in the loop header block.
- Any pointer arithmetic recurrences are raised to use array subscripts.

If the trip count of a loop is computable, this pass also makes the following changes:

- The exit condition for the loop is canonicalized to compare the induction value against the exit value. This turns loops like:

  ```
  for (i = 7; i*i < 1000; ++i)

  into

  for (i = 0; i != 25; ++i)
  ```

- Any use outside of the loop of an expression derived from the indvar is changed to compute the derived value outside of the loop, eliminating the dependence on the exit value of the induction variable. If the only purpose of the loop is to compute the exit value of some derived expression, this transformation will make the loop dead.

This transformation should be followed by strength reduction after all of the desired loop transformations have been performed. Additionally, on targets where it is profitable, the loop could be transformed to count down to zero (the "do loop" optimization).

`-inline`: Function Integration/Inlining

Bottom-up inlining of functions into callees.

`-instcombine`: Combine redundant instructions

Combine instructions to form fewer, simple instructions. This pass does not modify the CFG. This pass is where algebraic simplification happens.

This pass combines things like:

```
%Y = add i32 %X, 1
%Z = add i32 %Y, 1
```

into:

```
%Z = add i32 %X, 2
```

This is a simple worklist driven algorithm.

This pass guarantees that the following canonicalizations are performed on the program:

1. If a binary operator has a constant operand, it is moved to the right-hand side.
2. Bitwise operators with constant operands are always grouped so that shifts are performed first, then `or`s, then `and`s, then `xor`s.
3. Compare instructions are converted from <, >, $\leq$, or $\geq$ to = or $\neq$ if possible.
4. All `cmp` instructions on boolean values are replaced with logical operations.
5. `add X, X` is represented as `mul X, 2` $\Rightarrow$ `shl X, 1`
6. Multiplies with a constant power-of-two argument are transformed into shifts.
7. ... etc.

This pass can also simplify calls to specific well-known function calls (e.g. runtime library functions). For example, a call `exit(3)` that occurs within the `main()` function can be transformed into simply `return 3`. Whether or not library calls are simplified is controlled by the *-functionattrs* pass and LLVM's knowledge of library calls on different targets.

`-internalize`: Internalize Global Symbols

This pass loops over all of the functions in the input module, looking for a main function. If a main function is found, all other functions and all global variables with initializers are marked as internal.

`-ipconstprop`: Interprocedural constant propagation

This pass implements an *extremely* simple interprocedural constant propagation pass. It could certainly be improved in many different ways, like using a worklist. This pass makes arguments dead, but does not remove them. The existing dead argument elimination pass should be run after this to clean up the mess.

`-ipsccp`: Interprocedural Sparse Conditional Constant Propagation

An interprocedural variant of *Sparse Conditional Constant Propagation*.

`-jump-threading`: Jump Threading

Jump threading tries to find distinct threads of control flow running through a basic block. This pass looks at blocks that have multiple predecessors and multiple successors. If one or more of the predecessors of the block can be proven to always cause a jump to one of the successors, we forward the edge from the predecessor to the successor by duplicating the contents of this block.

An example of when this can occur is code like this:

```
if () { ...
  X = 4;
}
if (X < 3) {
```

In this case, the unconditional branch at the end of the first if can be revectored to the false side of the second if.

`-lcssa`: Loop-Closed SSA Form Pass

This pass transforms loops by placing phi nodes at the end of the loops for all values that are live across the loop boundary. For example, it turns the left into the right code:

```
for (...)                 for (...)
    if (c)                    if (c)
        X1 = ...                  X1 = ...
    else                      else
        X2 = ...                  X2 = ...
    X3 = phi(X1, X2)          X3 = phi(X1, X2)
... = X3 + 4                  X4 = phi(X3)
                              ... = X4 + 4
```

This is still valid LLVM; the extra phi nodes are purely redundant, and will be trivially eliminated by `InstCombine`. The major benefit of this transformation is that it makes many other loop optimizations, such as `LoopUnswitch`ing, simpler.

`-licm`: Loop Invariant Code Motion

This pass performs loop invariant code motion, attempting to remove as much code from the body of a loop as possible. It does this by either hoisting code into the preheader block, or by sinking code to the exit blocks if it is safe. This pass also promotes must-aliased memory locations in the loop to live in registers, thus hoisting and sinking "invariant" loads and stores.

This pass uses alias analysis for two purposes:

1. Moving loop invariant loads and calls out of loops. If we can determine that a load or call inside of a loop never aliases anything stored to, we can hoist it or sink it like any other instruction.
2. Scalar Promotion of Memory. If there is a store instruction inside of the loop, we try to move the store to happen AFTER the loop instead of inside of the loop. This can only happen if a few conditions are true:
   (a) The pointer stored through is loop invariant.
   (b) There are no stores or loads in the loop which *may* alias the pointer. There are no calls in the loop which mod/ref the pointer.

   If these conditions are true, we can promote the loads and stores in the loop of the pointer to use a temporary alloca'd variable. We then use the *mem2reg* functionality to construct the appropriate SSA form for the variable.

`-loop-deletion`: Delete dead loops

This file implements the Dead Loop Deletion Pass. This pass is responsible for eliminating loops with non-infinite computable trip counts that have no side effects or volatile instructions, and do not contribute to the computation of the function's return value.

`-loop-extract`: Extract loops into new functions

A pass wrapper around the `ExtractLoop()` scalar transformation to extract each top-level loop into its own new function. If the loop is the *only* loop in a given function, it is not touched. This is a pass most useful for debugging via bugpoint.

`-loop-extract-single`: Extract at most one loop into a new function

Similar to *Extract loops into new functions*, this pass extracts one natural loop from the program into a function if it can. This is used by **bugpoint**.

`-loop-reduce`: Loop Strength Reduction

This pass performs a strength reduction on array references inside loops that have as one or more of their components the loop induction variable. This is accomplished by creating a new value to hold the initial value of the array access for the first iteration, and then creating a new GEP instruction in the loop to increment the value by the appropriate amount.

`-loop-rotate`: Rotate Loops

A simple loop rotation transformation.

`-loop-simplify`: Canonicalize natural loops

This pass performs several transformations to transform natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective.

Loop pre-header insertion guarantees that there is a single, non-critical entry edge from outside of the loop to the loop header. This simplifies a number of analyses and transformations, such as *LICM*.

Loop exit-block insertion guarantees that all exit blocks from the loop (blocks which are outside of the loop that have predecessors inside of the loop) only have predecessors from inside of the loop (and are thus dominated by the loop header). This simplifies transformations such as store-sinking that are built into LICM.

This pass also guarantees that loops will have exactly one backedge.

Note that the *simplifycfg* pass will clean up blocks which are split out but end up being unnecessary, so usage of this pass should not pessimize generated code.

This pass obviously modifies the CFG, but updates loop information and dominator information.

`-loop-unroll`: Unroll loops

This pass implements a simple loop unroller. It works best when loops have been canonicalized by the *indvars* pass, allowing it to determine the trip counts of loops easily.

-loop-unswitch: Unswitch loops

This pass transforms loops that contain branches on loop-invariant conditions to have multiple loops. For example, it turns the left into the right code:

```
for (...)                  if (lic)
    A                          for (...)
    if (lic)                       A; B; C
        B                  else
    C                          for (...)
                                   A; C
```

This can increase the size of the code exponentially (doubling it every time a loop is unswitched) so we only unswitch if the resultant code will be smaller than a threshold.

This pass expects *LICM* to be run before it to hoist invariant conditions out of the loop, to make the unswitching opportunity obvious.

-loweratomic: Lower atomic intrinsics to non-atomic form

This pass lowers atomic intrinsics to non-atomic form for use in a known non-preemptible environment.

The pass does not verify that the environment is non-preemptible (in general this would require knowledge of the entire call graph of the program including any libraries which may not be available in bitcode form); it simply lowers every atomic intrinsic.

-lowerinvoke: Lower invokes to calls, for unwindless code generators

This transformation is designed for use by code generators which do not yet support stack unwinding. This pass converts `invoke` instructions to `call` instructions, so that any exception-handling `landingpad` blocks become dead code (which can be removed by running the `-simplifycfg` pass afterwards).

-lowerswitch: Lower SwitchInsts to branches

Rewrites switch instructions with a sequence of branches, which allows targets to get away with not implementing the switch instruction until it is convenient.

-mem2reg: Promote Memory to Register

This file promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses. An `alloca` is transformed by using dominator frontiers to place phi nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate. This is just the standard SSA construction algorithm to construct "pruned" SSA form.

-memcpyopt: MemCpy Optimization

This pass performs various transformations related to eliminating `memcpy` calls, or transforming sets of stores into `memsets`.

`-mergefunc`: Merge Functions

This pass looks for equivalent functions that are mergable and folds them.

A hash is computed from the function, based on its type and number of basic blocks.

Once all hashes are computed, we perform an expensive equality comparison on each function pair. This takes n^2/2 comparisons per bucket, so it's important that the hash function be high quality. The equality comparison iterates through each instruction in each basic block.

When a match is found the functions are folded. If both functions are overridable, we move the functionality into a new internal function and leave two overridable thunks to it.

`-mergereturn`: Unify function exit nodes

Ensure that functions have at most one `ret` instruction in them. Additionally, it keeps track of which node is the new exit node of the CFG.

`-partial-inliner`: Partial Inliner

This pass performs partial inlining, typically by inlining an `if` statement that surrounds the body of the function.

`-prune-eh`: Remove unused exception handling info

This file implements a simple interprocedural pass which walks the call-graph, turning invoke instructions into call instructions if and only if the callee cannot throw an exception. It implements this as a bottom-up traversal of the call-graph.

`-reassociate`: Reassociate expressions

This pass reassociates commutative expressions in an order that is designed to promote better constant propagation, GCSE, *LICM*, PRE, etc.

For example: 4 + (x + 5) $\Rightarrow$ x + (4 + 5)

In the implementation of this algorithm, constants are assigned rank = 0, function arguments are rank = 1, and other values are assigned ranks corresponding to the reverse post order traversal of current function (starting at 2), which effectively gives values in deep loops higher rank than values not in loops.

`-reg2mem`: Demote all values to stack slots

This file demotes all registers to memory references. It is intended to be the inverse of *mem2reg*. By converting to `load` instructions, the only values live across basic blocks are `alloca` instructions and `load` instructions before `phi` nodes. It is intended that this should make CFG hacking much easier. To make later hacking easier, the entry block is split into two, such that all introduced `alloca` instructions (and nothing else) are in the entry block.

`-scalarrepl`: Scalar Replacement of Aggregates (DT)

The well-known scalar replacement of aggregates transformation. This transform breaks up `alloca` instructions of aggregate type (structure or array) into individual `alloca` instructions for each member if possible. Then, if possible, it transforms the individual `alloca` instructions into nice clean scalar SSA form.

This combines a simple scalar replacement of aggregates algorithm with the *mem2reg* algorithm because they often interact, especially for C++ programs. As such, iterating between `scalarrepl`, then *mem2reg* until we run out of things to promote works well.

`-sccp`: Sparse Conditional Constant Propagation

Sparse conditional constant propagation and merging, which can be summarized as:

- Assumes values are constant unless proven otherwise
- Assumes BasicBlocks are dead unless proven otherwise
- Proves values to be constant, and replaces them with constants
- Proves conditional branches to be unconditional

Note that this pass has a habit of making definitions be dead. It is a good idea to run a *DCE* pass sometime after running this pass.

`-simplifycfg`: Simplify the CFG

Performs dead code elimination and basic block merging. Specifically:

- Removes basic blocks with no predecessors.
- Merges a basic block into its predecessor if there is only one and the predecessor only has one successor.
- Eliminates PHI nodes for basic blocks with a single predecessor.
- Eliminates a basic block that only contains an unconditional branch.

`-sink`: Code sinking

This pass moves instructions into successor blocks, when possible, so that they aren't executed on paths where their results aren't needed.

`-strip`: Strip all symbols from a module

Performs code stripping. This transformation can delete:

- names for virtual registers
- symbols for internal globals and functions
- debug information

Note that this transformation makes code much less readable, so it should only be used in situations where the strip utility would be used, such as reducing code size or making it harder to reverse engineer code.

`-strip-dead-debug-info`: Strip debug info for unused symbols

performs code stripping. this transformation can delete:

- names for virtual registers
- symbols for internal globals and functions
- debug information

note that this transformation makes code much less readable, so it should only be used in situations where the strip utility would be used, such as reducing code size or making it harder to reverse engineer code.

`-strip-dead-prototypes`: Strip Unused Function Prototypes

This pass loops over all of the functions in the input module, looking for dead declarations and removes them. Dead declarations are declarations of functions for which no implementation is available (i.e., declarations for unused library functions).

`-strip-debug-declare`: Strip all `llvm.dbg.declare` intrinsics

This pass implements code stripping. Specifically, it can delete:

1. names for virtual registers
2. symbols for internal globals and functions
3. debug information

Note that this transformation makes code much less readable, so it should only be used in situations where the 'strip' utility would be used, such as reducing code size or making it harder to reverse engineer code.

`-strip-nondebug`: Strip all symbols, except dbg symbols, from a module

This pass implements code stripping. Specifically, it can delete:

1. names for virtual registers
2. symbols for internal globals and functions
3. debug information

Note that this transformation makes code much less readable, so it should only be used in situations where the 'strip' utility would be used, such as reducing code size or making it harder to reverse engineer code.

`-tailcallelim`: Tail Call Elimination

This file transforms calls of the current function (self recursion) followed by a return instruction with a branch to the entry of the function, creating a loop. This pass also implements the following extensions to the basic algorithm:

1. Trivial instructions between the call and return do not prevent the transformation from taking place, though currently the analysis cannot support moving any really useful instructions (only dead ones).
2. This pass transforms functions that are prevented from being tail recursive by an associative expression to use an accumulator variable, thus compiling the typical naive factorial or fib implementation into efficient code.
3. TRE is performed if the function returns void, if the return returns the result returned by the call, or if the function returns a run-time constant on all exits from the function. It is possible, though unlikely, that the return returns something else (like constant 0), and can still be TRE'd. It can be TRE'd if *all other* return instructions in the function return the exact same value.
4. If it can prove that callees do not access theier caller stack frame, they are marked as eligible for tail call elimination (by the code generator).

2.17.4 Utility Passes

This section describes the LLVM Utility Passes.

`-deadarghaX0r`: Dead Argument Hacking (BUGPOINT USE ONLY; DO NOT USE)

Same as dead argument elimination, but deletes arguments to functions which are external. This is only for use by *bugpoint*.

`-extract-blocks`: Extract Basic Blocks From Module (for bugpoint use)

This pass is used by bugpoint to extract all blocks from the module into their own functions.

`-instnamer`: Assign names to anonymous instructions

This is a little utility pass that gives instructions names, this is mostly useful when diffing the effect of an optimization because deleting an unnamed instruction can change all other instruction numbering, making the diff very noisy.

`-preverify`: Preliminary module verification

Ensures that the module is in the form required by the *Module Verifier* pass. Running the verifier runs this pass automatically, so there should be no need to use it directly.

`-verify`: Module Verifier

Verifies an LLVM IR code. This is useful to run after an optimization which is undergoing testing. Note that llvm-as verifies its input before emitting bitcode, and also that malformed bitcode is likely to make LLVM crash. All language front-ends are therefore encouraged to verify their output before performing optimizing transformations.

1. Both of a binary operator's parameters are of the same type.
2. Verify that the indices of mem access instructions match other operands.
3. Verify that arithmetic and other things are only performed on first-class types. Verify that shifts and logicals only happen on integrals f.e.
4. All of the constants in a switch statement are of the correct type.
5. The code is in valid SSA form.
6. It is illegal to put a label into any other type (like a structure) or to return one.
7. Only phi nodes can be self referential: `%x = add i32 %x`, `%x` is invalid.
8. PHI nodes must have an entry for each predecessor, with no extras.
9. PHI nodes must be the first thing in a basic block, all grouped together.
10. PHI nodes must have at least one entry.
11. All basic blocks should only end with terminator insts, not contain them.
12. The entry node to a function must not have predecessors.
13. All Instructions must be embedded into a basic block.
14. Functions cannot take a void-typed parameter.

15. Verify that a function's argument list agrees with its declared type.
16. It is illegal to specify a name for a void value.
17. It is illegal to have an internal global value with no initializer.
18. It is illegal to have a `ret` instruction that returns a value that does not agree with the function return value type.
19. Function call argument types match the function prototype.
20. All other things that are tested by asserts spread about the code.

Note that this does not provide full security verification (like Java), but instead just tries to ensure that code is well-formed.

`-view-cfg`: View CFG of function

Displays the control flow graph using the GraphViz tool.

`-view-cfg-only`: View CFG of function (with no function bodies)

Displays the control flow graph using the GraphViz tool, but omitting function bodies.

`-view-dom`: View dominance tree of function

Displays the dominator tree using the GraphViz tool.

`-view-dom-only`: View dominance tree of function (with no function bodies)

Displays the dominator tree using the GraphViz tool, but omitting function bodies.

`-view-postdom`: View postdominance tree of function

Displays the post dominator tree using the GraphViz tool.

`-view-postdom-only`: View postdominance tree of function (with no function bodies)

Displays the post dominator tree using the GraphViz tool, but omitting function bodies.

2.18 YAML I/O

2.18.1 Introduction to YAML

YAML is a human readable data serialization language. The full YAML language spec can be read at yaml.org. The simplest form of yaml is just "scalars", "mappings", and "sequences". A scalar is any number or string. The pound/hash symbol (#) begins a comment line. A mapping is a set of key-value pairs where the key ends with a colon. For example:

```
# a mapping
name:      Tom
hat-size:  7
```

A sequence is a list of items where each item starts with a leading dash ('-'). For example:

```
# a sequence
- x86
- x86_64
- PowerPC
```

You can combine mappings and sequences by indenting. For example a sequence of mappings in which one of the mapping values is itself a sequence:

```
# a sequence of mappings with one key's value being a sequence
- name:      Tom
  cpus:
   - x86
   - x86_64
- name:      Bob
  cpus:
   - x86
```

```
- name:      Dan
  cpus:
   - PowerPC
   - x86
```

Sometime sequences are known to be short and the one entry per line is too verbose, so YAML offers an alternate syntax for sequences called a "Flow Sequence" in which you put comma separated sequence elements into square brackets. The above example could then be simplified to :

```
# a sequence of mappings with one key's value being a flow sequence
- name:      Tom
  cpus:      [ x86, x86_64 ]
- name:      Bob
  cpus:      [ x86 ]
- name:      Dan
  cpus:      [ PowerPC, x86 ]
```

2.18.2 Introduction to YAML I/O

The use of indenting makes the YAML easy for a human to read and understand, but having a program read and write YAML involves a lot of tedious details. The YAML I/O library structures and simplifies reading and writing YAML documents.

YAML I/O assumes you have some "native" data structures which you want to be able to dump as YAML and recreate from YAML. The first step is to try writing example YAML for your data structures. You may find after looking at possible YAML representations that a direct mapping of your data structures to YAML is not very readable. Often the fields are not in the order that a human would find readable. Or the same information is replicated in multiple locations, making it hard for a human to write such YAML correctly.

In relational database theory there is a design step called normalization in which you reorganize fields and tables. The same considerations need to go into the design of your YAML encoding. But, you may not want to change your existing native data structures. Therefore, when writing out YAML there may be a normalization step, and when reading YAML there would be a corresponding denormalization step.

YAML I/O uses a non-invasive, traits based design. YAML I/O defines some abstract base templates. You specialize those templates on your data types. For instance, if you have an enumerated type FooBar you could specialize ScalarEnumerationTraits on that type and define the enumeration() method:

```
using llvm::yaml::ScalarEnumerationTraits;
using llvm::yaml::IO;

template <>
struct ScalarEnumerationTraits<FooBar> {
  static void enumeration(IO &io, FooBar &value) {
  ...
  }
};
```

As with all YAML I/O template specializations, the ScalarEnumerationTraits is used for both reading and writing YAML. That is, the mapping between in-memory enum values and the YAML string representation is only in one place. This assures that the code for writing and parsing of YAML stays in sync.

To specify a YAML mappings, you define a specialization on llvm::yaml::MappingTraits. If your native data structure happens to be a struct that is already normalized, then the specialization is simple. For example:

```
using llvm::yaml::MappingTraits;
using llvm::yaml::IO;
```

```
template <>
struct MappingTraits<Person> {
  static void mapping(IO &io, Person &info) {
    io.mapRequired("name",         info.name);
    io.mapOptional("hat-size",     info.hatSize);
  }
};
```

A YAML sequence is automatically inferred if you data type has begin()/end() iterators and a push_back() method. Therefore any of the STL containers (such as std::vector<>) will automatically translate to YAML sequences.

Once you have defined specializations for your data types, you can programmatically use YAML I/O to write a YAML document:

```
using llvm::yaml::Output;

Person tom;
tom.name = "Tom";
tom.hatSize = 8;
Person dan;
dan.name = "Dan";
dan.hatSize = 7;
std::vector<Person> persons;
persons.push_back(tom);
persons.push_back(dan);

Output yout(llvm::outs());
yout << persons;
```

This would write the following:

```
- name:      Tom
  hat-size:  8
- name:      Dan
  hat-size:  7
```

And you can also read such YAML documents with the following code:

```
using llvm::yaml::Input;

typedef std::vector<Person> PersonList;
std::vector<PersonList> docs;

Input yin(document.getBuffer());
yin >> docs;

if ( yin.error() )
  return;

// Process read document
for ( PersonList &pl : docs ) {
  for ( Person &person : pl ) {
    cout << "name=" << person.name;
  }
}
```

One other feature of YAML is the ability to define multiple documents in a single file. That is why reading YAML produces a vector of your document type.

2.18.3 Error Handling

When parsing a YAML document, if the input does not match your schema (as expressed in your XxxTraits<> specializations). YAML I/O will print out an error message and your Input object's error() method will return true. For instance the following document:

```
- name:      Tom
  shoe-size: 12
- name:      Dan
  hat-size:  7
```

Has a key (shoe-size) that is not defined in the schema. YAML I/O will automatically generate this error:

```
YAML:2:2: error: unknown key 'shoe-size'
  shoe-size:       12
  ^~~~~~~~~
```

Similar errors are produced for other input not conforming to the schema.

2.18.4 Scalars

YAML scalars are just strings (i.e. not a sequence or mapping). The YAML I/O library provides support for translating between YAML scalars and specific C++ types.

Built-in types

The following types have built-in support in YAML I/O:

- bool
- float
- double
- StringRef
- std::string
- int64_t
- int32_t
- int16_t
- int8_t
- uint64_t
- uint32_t
- uint16_t
- uint8_t

That is, you can use those types in fields of MappingTraits or as element type in sequence. When reading, YAML I/O will validate that the string found is convertible to that type and error out if not.

Unique types

Given that YAML I/O is trait based, the selection of how to convert your data to YAML is based on the type of your data. But in C++ type matching, typedefs do not generate unique type names. That means if you have two typedefs of unsigned int, to YAML I/O both types look exactly like unsigned int. To facilitate make unique type names, YAML I/O provides a macro which is used like a typedef on built-in types, but expands to create a class with conversion operators to and from the base type. For example:

```
LLVM_YAML_STRONG_TYPEDEF(uint32_t, MyFooFlags)
LLVM_YAML_STRONG_TYPEDEF(uint32_t, MyBarFlags)
```

This generates two classes MyFooFlags and MyBarFlags which you can use in your native data structures instead of uint32_t. They are implicitly converted to and from uint32_t. The point of creating these unique types is that you can now specify traits on them to get different YAML conversions.

Hex types

An example use of a unique type is that YAML I/O provides fixed sized unsigned integers that are written with YAML I/O as hexadecimal instead of the decimal format used by the built-in integer types:

- Hex64
- Hex32
- Hex16
- Hex8

You can use llvm::yaml::Hex32 instead of uint32_t and the only different will be that when YAML I/O writes out that type it will be formatted in hexadecimal.

ScalarEnumerationTraits

YAML I/O supports translating between in-memory enumerations and a set of string values in YAML documents. This is done by specializing ScalarEnumerationTraits<> on your enumeration type and define a enumeration() method. For instance, suppose you had an enumeration of CPUs and a struct with it as a field:

```
enum CPUs {
  cpu_x86_64  = 5,
  cpu_x86     = 7,
  cpu_PowerPC = 8
};

struct Info {
  CPUs      cpu;
  uint32_t  flags;
};
```

To support reading and writing of this enumeration, you can define a ScalarEnumerationTraits specialization on CPUs, which can then be used as a field type:

```
using llvm::yaml::ScalarEnumerationTraits;
using llvm::yaml::MappingTraits;
using llvm::yaml::IO;

template <>
struct ScalarEnumerationTraits<CPUs> {
  static void enumeration(IO &io, CPUs &value) {
```

```
    io.enumCase(value, "x86_64",  cpu_x86_64);
    io.enumCase(value, "x86",     cpu_x86);
    io.enumCase(value, "PowerPC", cpu_PowerPC);
  }
};

template <>
struct MappingTraits<Info> {
  static void mapping(IO &io, Info &info) {
    io.mapRequired("cpu",       info.cpu);
    io.mapOptional("flags",     info.flags, 0);
  }
};
```

When reading YAML, if the string found does not match any of the the strings specified by enumCase() methods, an error is automatically generated. When writing YAML, if the value being written does not match any of the values specified by the enumCase() methods, a runtime assertion is triggered.

BitValue

Another common data structure in C++ is a field where each bit has a unique meaning. This is often used in a "flags" field. YAML I/O has support for converting such fields to a flow sequence. For instance suppose you had the following bit flags defined:

```
enum {
  flagsPointy = 1
  flagsHollow = 2
  flagsFlat   = 4
  flagsRound  = 8
};

LLVM_YAML_STRONG_TYPEDEF(uint32_t, MyFlags)
```

To support reading and writing of MyFlags, you specialize ScalarBitSetTraits<> on MyFlags and provide the bit values and their names.

```
using llvm::yaml::ScalarBitSetTraits;
using llvm::yaml::MappingTraits;
using llvm::yaml::IO;

template <>
struct ScalarBitSetTraits<MyFlags> {
  static void bitset(IO &io, MyFlags &value) {
    io.bitSetCase(value, "hollow",  flagHollow);
    io.bitSetCase(value, "flat",    flagFlat);
    io.bitSetCase(value, "round",   flagRound);
    io.bitSetCase(value, "pointy",  flagPointy);
  }
};

struct Info {
  StringRef   name;
  MyFlags     flags;
};

template <>
struct MappingTraits<Info> {
```

```
  static void mapping(IO &io, Info& info) {
    io.mapRequired("name",  info.name);
    io.mapRequired("flags", info.flags);
   }
};
```

With the above, YAML I/O (when writing) will test mask each value in the bitset trait against the flags field, and each that matches will cause the corresponding string to be added to the flow sequence. The opposite is done when reading and any unknown string values will result in a error. With the above schema, a same valid YAML document is:

```
name:    Tom
flags:   [ pointy, flat ]
```

Sometimes a "flags" field might contains an enumeration part defined by a bit-mask.

```
enum {
  flagsFeatureA = 1,
  flagsFeatureB = 2,
  flagsFeatureC = 4,

  flagsCPUMask = 24,

  flagsCPU1 = 8,
  flagsCPU2 = 16
};
```

To support reading and writing such fields, you need to use the maskedBitSet() method and provide the bit values, their names and the enumeration mask.

```
template <>
struct ScalarBitSetTraits<MyFlags> {
  static void bitset(IO &io, MyFlags &value) {
    io.bitSetCase(value, "featureA",  flagsFeatureA);
    io.bitSetCase(value, "featureB",  flagsFeatureB);
    io.bitSetCase(value, "featureC",  flagsFeatureC);
    io.maskedBitSetCase(value, "CPU1",  flagsCPU1, flagsCPUMask);
    io.maskedBitSetCase(value, "CPU2",  flagsCPU2, flagsCPUMask);
  }
};
```

YAML I/O (when writing) will apply the enumeration mask to the flags field, and compare the result and values from the bitset. As in case of a regular bitset, each that matches will cause the corresponding string to be added to the flow sequence.

Custom Scalar

Sometimes for readability a scalar needs to be formatted in a custom way. For instance your internal data structure may use a integer for time (seconds since some epoch), but in YAML it would be much nicer to express that integer in some time format (e.g. 4-May-2012 10:30pm). YAML I/O has a way to support custom formatting and parsing of scalar types by specializing ScalarTraits<> on your data type. When writing, YAML I/O will provide the native type and your specialization must create a temporary llvm::StringRef. When reading, YAML I/O will provide an llvm::StringRef of scalar and your specialization must convert that to your native data type. An outline of a custom scalar type looks like:

```
using llvm::yaml::ScalarTraits;
using llvm::yaml::IO;
```

```
template <>
struct ScalarTraits<MyCustomType> {
  static void output(const T &value, llvm::raw_ostream &out) {
    out << value;  // do custom formatting here
  }
  static StringRef input(StringRef scalar, T &value) {
    // do custom parsing here.  Return the empty string on success,
    // or an error message on failure.
    return StringRef();
  }
  // Determine if this scalar needs quotes.
  static bool mustQuote(StringRef) { return true; }
};
```

2.18.5 Mappings

To be translated to or from a YAML mapping for your type T you must specialize llvm::yaml::MappingTraits on T and implement the "void mapping(IO &io, T&)" method. If your native data structures use pointers to a class everywhere, you can specialize on the class pointer. Examples:

```
using llvm::yaml::MappingTraits;
using llvm::yaml::IO;

// Example of struct Foo which is used by value
template <>
struct MappingTraits<Foo> {
  static void mapping(IO &io, Foo &foo) {
    io.mapOptional("size",      foo.size);
  ...
  }
};

// Example of struct Bar which is natively always a pointer
template <>
struct MappingTraits<Bar*> {
  static void mapping(IO &io, Bar *&bar) {
    io.mapOptional("size",    bar->size);
  ...
  }
};
```

No Normalization

The mapping() method is responsible, if needed, for normalizing and denormalizing. In a simple case where the native data structure requires no normalization, the mapping method just uses mapOptional() or mapRequired() to bind the struct's fields to YAML key names. For example:

```
using llvm::yaml::MappingTraits;
using llvm::yaml::IO;

template <>
struct MappingTraits<Person> {
  static void mapping(IO &io, Person &info) {
    io.mapRequired("name",         info.name);
    io.mapOptional("hat-size",     info.hatSize);
```

```
  }
};
```

Normalization

When [de]normalization is required, the mapping() method needs a way to access normalized values as fields. To help with this, there is a template MappingNormalization<> which you can then use to automatically do the normalization and denormalization. The template is used to create a local variable in your mapping() method which contains the normalized keys.

Suppose you have native data type Polar which specifies a position in polar coordinates (distance, angle):

```
struct Polar {
  float distance;
  float angle;
};
```

but you've decided the normalized YAML for should be in x,y coordinates. That is, you want the yaml to look like:

```
x:   10.3
y:   -4.7
```

You can support this by defining a MappingTraits that normalizes the polar coordinates to x,y coordinates when writing YAML and denormalizes x,y coordinates into polar when reading YAML.

```
using llvm::yaml::MappingTraits;
using llvm::yaml::IO;

template <>
struct MappingTraits<Polar> {

  class NormalizedPolar {
  public:
    NormalizedPolar(IO &io)
      : x(0.0), y(0.0) {
    }
    NormalizedPolar(IO &, Polar &polar)
      : x(polar.distance * cos(polar.angle)),
        y(polar.distance * sin(polar.angle)) {
    }
    Polar denormalize(IO &) {
      return Polar(sqrt(x*x+y*y), arctan(x,y));
    }

    float        x;
    float        y;
  };

  static void mapping(IO &io, Polar &polar) {
    MappingNormalization<NormalizedPolar, Polar> keys(io, polar);

    io.mapRequired("x",    keys->x);
    io.mapRequired("y",    keys->y);
  }
};
```

When writing YAML, the local variable "keys" will be a stack allocated instance of NormalizedPolar, constructed from the supplied polar object which initializes it x and y fields. The mapRequired() methods then write out the x and

y values as key/value pairs.

When reading YAML, the local variable "keys" will be a stack allocated instance of NormalizedPolar, constructed by the empty constructor. The mapRequired methods will find the matching key in the YAML document and fill in the x and y fields of the NormalizedPolar object keys. At the end of the mapping() method when the local keys variable goes out of scope, the denormalize() method will automatically be called to convert the read values back to polar coordinates, and then assigned back to the second parameter to mapping().

In some cases, the normalized class may be a subclass of the native type and could be returned by the denormalize() method, except that the temporary normalized instance is stack allocated. In these cases, the utility template MappingNormalizationHeap<> can be used instead. It just like MappingNormalization<> except that it heap allocates the normalized object when reading YAML. It never destroys the normalized object. The denormalize() method can this return "this".

Default values

Within a mapping() method, calls to io.mapRequired() mean that that key is required to exist when parsing YAML documents, otherwise YAML I/O will issue an error.

On the other hand, keys registered with io.mapOptional() are allowed to not exist in the YAML document being read. So what value is put in the field for those optional keys? There are two steps to how those optional fields are filled in. First, the second parameter to the mapping() method is a reference to a native class. That native class must have a default constructor. Whatever value the default constructor initially sets for an optional field will be that field's value. Second, the mapOptional() method has an optional third parameter. If provided it is the value that mapOptional() should set that field to if the YAML document does not have that key.

There is one important difference between those two ways (default constructor and third parameter to mapOptional). When YAML I/O generates a YAML document, if the mapOptional() third parameter is used, if the actual value being written is the same as (using ==) the default value, then that key/value is not written.

Order of Keys

When writing out a YAML document, the keys are written in the order that the calls to mapRequired()/mapOptional() are made in the mapping() method. This gives you a chance to write the fields in an order that a human reader of the YAML document would find natural. This may be different that the order of the fields in the native class.

When reading in a YAML document, the keys in the document can be in any order, but they are processed in the order that the calls to mapRequired()/mapOptional() are made in the mapping() method. That enables some interesting functionality. For instance, if the first field bound is the cpu and the second field bound is flags, and the flags are cpu specific, you can programmatically switch how the flags are converted to and from YAML based on the cpu. This works for both reading and writing. For example:

```
using llvm::yaml::MappingTraits;
using llvm::yaml::IO;

struct Info {
  CPUs        cpu;
  uint32_t    flags;
};

template <>
struct MappingTraits<Info> {
  static void mapping(IO &io, Info &info) {
    io.mapRequired("cpu",       info.cpu);
    // flags must come after cpu for this to work when reading yaml
    if ( info.cpu == cpu_x86_64 )
```

```
      io.mapRequired("flags",   *(My86_64Flags*)info.flags);
    else
      io.mapRequired("flags",   *(My86Flags*)info.flags);
 }
};
```

Tags

The YAML syntax supports tags as a way to specify the type of a node before it is parsed. This allows dynamic types of nodes. But the YAML I/O model uses static typing, so there are limits to how you can use tags with the YAML I/O model. Recently, we added support to YAML I/O for checking/setting the optional tag on a map. Using this functionality it is even possbile to support different mappings, as long as they are convertable.

To check a tag, inside your mapping() method you can use io.mapTag() to specify what the tag should be. This will also add that tag when writing yaml.

Validation

Sometimes in a yaml map, each key/value pair is valid, but the combination is not. This is similar to something having no syntax errors, but still having semantic errors. To support semantic level checking, YAML I/O allows an optional `validate()` method in a MappingTraits template specialization.

When parsing yaml, the `validate()` method is call *after* all key/values in the map have been processed. Any error message returned by the `validate()` method during input will be printed just a like a syntax error would be printed. When writing yaml, the `validate()` method is called *before* the yaml key/values are written. Any error during output will trigger an `assert()` because it is a programming error to have invalid struct values.

```
using llvm::yaml::MappingTraits;
using llvm::yaml::IO;

struct Stuff {
  ...
};

template <>
struct MappingTraits<Stuff> {
  static void mapping(IO &io, Stuff &stuff) {
  ...
  }
  static StringRef validate(IO &io, Stuff &stuff) {
    // Look at all fields in 'stuff' and if there
    // are any bad values return a string describing
    // the error.  Otherwise return an empty string.
    return StringRef();
  }
};
```

2.18.6 Sequence

To be translated to or from a YAML sequence for your type T you must specialize llvm::yaml::SequenceTraits on T and implement two methods: `size_t size(IO &io, T&)` and `T::value_type& element(IO &io, T&, size_t indx)`. For example:

```
template <>
struct SequenceTraits<MySeq> {
  static size_t size(IO &io, MySeq &list) { ... }
  static MySeqEl &element(IO &io, MySeq &list, size_t index) { ... }
};
```

The size() method returns how many elements are currently in your sequence. The element() method returns a reference to the i'th element in the sequence. When parsing YAML, the element() method may be called with an index one bigger than the current size. Your element() method should allocate space for one more element (using default constructor if element is a C++ object) and returns a reference to that new allocated space.

Flow Sequence

A YAML "flow sequence" is a sequence that when written to YAML it uses the inline notation (e.g [foo, bar]). To specify that a sequence type should be written in YAML as a flow sequence, your SequenceTraits specialization should add "static const bool flow = true;". For instance:

```
template <>
struct SequenceTraits<MyList> {
  static size_t size(IO &io, MyList &list) { ... }
  static MyListEl &element(IO &io, MyList &list, size_t index) { ... }

  // The existence of this member causes YAML I/O to use a flow sequence
  static const bool flow = true;
};
```

With the above, if you used MyList as the data type in your native data structures, then then when converted to YAML, a flow sequence of integers will be used (e.g. [10, -3, 4]).

Utility Macros

Since a common source of sequences is std::vector<>, YAML I/O provides macros: LLVM_YAML_IS_SEQUENCE_VECTOR() and LLVM_YAML_IS_FLOW_SEQUENCE_VECTOR() which can be used to easily specify SequenceTraits<> on a std::vector type. YAML I/O does not partial specialize SequenceTraits on std::vector<> because that would force all vectors to be sequences. An example use of the macros:

```
std::vector<MyType1>;
std::vector<MyType2>;
LLVM_YAML_IS_SEQUENCE_VECTOR(MyType1)
LLVM_YAML_IS_FLOW_SEQUENCE_VECTOR(MyType2)
```

2.18.7 Document List

YAML allows you to define multiple "documents" in a single YAML file. Each new document starts with a left aligned "—" token. The end of all documents is denoted with a left aligned "..." token. Many users of YAML will never have need for multiple documents. The top level node in their YAML schema will be a mapping or sequence. For those cases, the following is not needed. But for cases where you do want multiple documents, you can specify a trait for you document list type. The trait has the same methods as SequenceTraits but is named DocumentListTraits. For example:

```
template <>
struct DocumentListTraits<MyDocList> {
  static size_t size(IO &io, MyDocList &list) { ... }
```

```
  static MyDocType element(IO &io, MyDocList &list, size_t index) { ... }
};
```

2.18.8 User Context Data

When an llvm::yaml::Input or llvm::yaml::Output object is created their constructors take an optional "context" parameter. This is a pointer to whatever state information you might need.

For instance, in a previous example we showed how the conversion type for a flags field could be determined at runtime based on the value of another field in the mapping. But what if an inner mapping needs to know some field value of an outer mapping? That is where the "context" parameter comes in. You can set values in the context in the outer map's mapping() method and retrieve those values in the inner map's mapping() method.

The context value is just a void*. All your traits which use the context and operate on your native data types, need to agree what the context value actually is. It could be a pointer to an object or struct which your various traits use to shared context sensitive information.

2.18.9 Output

The llvm::yaml::Output class is used to generate a YAML document from your in-memory data structures, using traits defined on your data types. To instantiate an Output object you need an llvm::raw_ostream, and optionally a context pointer:

```
class Output : public IO {
public:
  Output(llvm::raw_ostream &, void *context=NULL);
```

Once you have an Output object, you can use the C++ stream operator on it to write your native data as YAML. One thing to recall is that a YAML file can contain multiple "documents". If the top level data structure you are streaming as YAML is a mapping, scalar, or sequence, then Output assumes you are generating one document and wraps the mapping output with "---" and trailing "...".

```
using llvm::yaml::Output;

void dumpMyMapDoc(const MyMapType &info) {
  Output yout(llvm::outs());
  yout << info;
}
```

The above could produce output like:

```
---
name:      Tom
hat-size:  7
...
```

On the other hand, if the top level data structure you are streaming as YAML has a DocumentListTraits specialization, then Output walks through each element of your DocumentList and generates a "—" before the start of each element and ends with a "...".

```
using llvm::yaml::Output;

void dumpMyMapDoc(const MyDocListType &docList) {
  Output yout(llvm::outs());
  yout << docList;
}
```

The above could produce output like:

```
---
name:      Tom
hat-size:  7
---
name:      Tom
shoe-size:  11
...
```

2.18.10 Input

The llvm::yaml::Input class is used to parse YAML document(s) into your native data structures. To instantiate an Input object you need a StringRef to the entire YAML file, and optionally a context pointer:

```
class Input : public IO {
public:
  Input(StringRef inputContent, void *context=NULL);
```

Once you have an Input object, you can use the C++ stream operator to read the document(s). If you expect there might be multiple YAML documents in one file, you'll need to specialize DocumentListTraits on a list of your document type and stream in that document list type. Otherwise you can just stream in the document type. Also, you can check if there was any syntax errors in the YAML be calling the error() method on the Input object. For example:

```
// Reading a single document
using llvm::yaml::Input;

Input yin(mb.getBuffer());

// Parse the YAML file
MyDocType theDoc;
yin >> theDoc;

// Check for error
if ( yin.error() )
  return;

// Reading multiple documents in one file
using llvm::yaml::Input;

LLVM_YAML_IS_DOCUMENT_LIST_VECTOR(std::vector<MyDocType>)

Input yin(mb.getBuffer());

// Parse the YAML file
std::vector<MyDocType> theDocList;
yin >> theDocList;

// Check for error
if ( yin.error() )
  return;
```

2.19 The Often Misunderstood GEP Instruction

2.19.1 Introduction

This document seeks to dispel the mystery and confusion surrounding LLVM's GetElementPtr (GEP) instruction. Questions about the wily GEP instruction are probably the most frequently occurring questions once a developer gets down to coding with LLVM. Here we lay out the sources of confusion and show that the GEP instruction is really quite simple.

2.19.2 Address Computation

When people are first confronted with the GEP instruction, they tend to relate it to known concepts from other programming paradigms, most notably C array indexing and field selection. GEP closely resembles C array indexing and field selection, however it is a little different and this leads to the following questions.

What is the first index of the GEP instruction?

Quick answer: The index stepping through the first operand.

The confusion with the first index usually arises from thinking about the GetElementPtr instruction as if it was a C index operator. They aren't the same. For example, when we write, in "C":

```
AType *Foo;
...
X = &Foo->F;
```

it is natural to think that there is only one index, the selection of the field `F`. However, in this example, `Foo` is a pointer. That pointer must be indexed explicitly in LLVM. C, on the other hand, indices through it transparently. To arrive at the same address location as the C code, you would provide the GEP instruction with two index operands. The first operand indexes through the pointer; the second operand indexes the field `F` of the structure, just as if you wrote:

```
X = &Foo[0].F;
```

Sometimes this question gets rephrased as:

> *Why is it okay to index through the first pointer, but subsequent pointers won't be dereferenced?*

The answer is simply because memory does not have to be accessed to perform the computation. The first operand to the GEP instruction must be a value of a pointer type. The value of the pointer is provided directly to the GEP instruction as an operand without any need for accessing memory. It must, therefore be indexed and requires an index operand. Consider this example:

```
struct munger_struct {
  int f1;
  int f2;
};
void munge(struct munger_struct *P) {
  P[0].f1 = P[1].f1 + P[2].f2;
}
...
munger_struct Array[3];
...
munge(Array);
```

In this "C" example, the front end compiler (Clang) will generate three GEP instructions for the three indices through "P" in the assignment statement. The function argument `P` will be the first operand of each of these GEP instructions. The second operand indexes through that pointer. The third operand will be the field offset into the `struct munger_struct` type, for either the `f1` or `f2` field. So, in LLVM assembly the `munge` function looks like:

```
void %munge(%struct.munger_struct* %P) {
entry:
  %tmp = getelementptr %struct.munger_struct* %P, i32 1, i32 0
  %tmp = load i32* %tmp
  %tmp6 = getelementptr %struct.munger_struct* %P, i32 2, i32 1
  %tmp7 = load i32* %tmp6
  %tmp8 = add i32 %tmp7, %tmp
  %tmp9 = getelementptr %struct.munger_struct* %P, i32 0, i32 0
  store i32 %tmp8, i32* %tmp9
  ret void
}
```

In each case the first operand is the pointer through which the GEP instruction starts. The same is true whether the first operand is an argument, allocated memory, or a global variable.

To make this clear, let's consider a more obtuse example:

```
%MyVar = uninitialized global i32
...
%idx1 = getelementptr i32* %MyVar, i64 0
%idx2 = getelementptr i32* %MyVar, i64 1
%idx3 = getelementptr i32* %MyVar, i64 2
```

These GEP instructions are simply making address computations from the base address of `MyVar`. They compute, as follows (using C syntax):

```
idx1 = (char*) &MyVar + 0
idx2 = (char*) &MyVar + 4
idx3 = (char*) &MyVar + 8
```

Since the type `i32` is known to be four bytes long, the indices 0, 1 and 2 translate into memory offsets of 0, 4, and 8, respectively. No memory is accessed to make these computations because the address of `%MyVar` is passed directly to the GEP instructions.

The obtuse part of this example is in the cases of `%idx2` and `%idx3`. They result in the computation of addresses that point to memory past the end of the `%MyVar` global, which is only one `i32` long, not three `i32`s long. While this is legal in LLVM, it is inadvisable because any load or store with the pointer that results from these GEP instructions would produce undefined results.

Why is the extra 0 index required?

Quick answer: there are no superfluous indices.

This question arises most often when the GEP instruction is applied to a global variable which is always a pointer type. For example, consider this:

```
%MyStruct = uninitialized global { float*, i32 }
...
%idx = getelementptr { float*, i32 }* %MyStruct, i64 0, i32 1
```

The GEP above yields an `i32*` by indexing the `i32` typed field of the structure `%MyStruct`. When people first look at it, they wonder why the `i64 0` index is needed. However, a closer inspection of how globals and GEPs work reveals the need. Becoming aware of the following facts will dispel the confusion:

1. The type of `%MyStruct` is *not* `{ float*, i32 }` but rather `{ float*, i32 }*`. That is, `%MyStruct` is a pointer to a structure containing a pointer to a `float` and an `i32`.
2. Point #1 is evidenced by noticing the type of the first operand of the GEP instruction (`%MyStruct`) which is `{ float*, i32 }*`.
3. The first index, `i64 0` is required to step over the global variable `%MyStruct`. Since the first argument to the GEP instruction must always be a value of pointer type, the first index steps through that pointer. A value of 0 means 0 elements offset from that pointer.
4. The second index, `i32 1` selects the second field of the structure (the `i32`).

What is dereferenced by GEP?

Quick answer: nothing.

The GetElementPtr instruction dereferences nothing. That is, it doesn't access memory in any way. That's what the Load and Store instructions are for. GEP is only involved in the computation of addresses. For example, consider this:

```
%MyVar = uninitialized global { [40 x i32 ]* }
...
%idx = getelementptr { [40 x i32]* }* %MyVar, i64 0, i32 0, i64 0, i64 17
```

In this example, we have a global variable, `%MyVar` that is a pointer to a structure containing a pointer to an array of 40 ints. The GEP instruction seems to be accessing the 18th integer of the structure's array of ints. However, this is actually an illegal GEP instruction. It won't compile. The reason is that the pointer in the structure *must* be dereferenced in order to index into the array of 40 ints. Since the GEP instruction never accesses memory, it is illegal.

In order to access the 18th integer in the array, you would need to do the following:

```
%idx = getelementptr { [40 x i32]* }* %, i64 0, i32 0
%arr = load [40 x i32]** %idx
%idx = getelementptr [40 x i32]* %arr, i64 0, i64 17
```

In this case, we have to load the pointer in the structure with a load instruction before we can index into the array. If the example was changed to:

```
%MyVar = uninitialized global { [40 x i32 ] }
...
%idx = getelementptr { [40 x i32] }*, i64 0, i32 0, i64 17
```

then everything works fine. In this case, the structure does not contain a pointer and the GEP instruction can index through the global variable, into the first field of the structure and access the 18th `i32` in the array there.

Why don't GEP x,0,0,1 and GEP x,1 alias?

Quick Answer: They compute different address locations.

If you look at the first indices in these GEP instructions you find that they are different (0 and 1), therefore the address computation diverges with that index. Consider this example:

```
%MyVar = global { [10 x i32 ] }
%idx1 = getelementptr { [10 x i32 ] }* %MyVar, i64 0, i32 0, i64 1
%idx2 = getelementptr { [10 x i32 ] }* %MyVar, i64 1
```

In this example, `idx1` computes the address of the second integer in the array that is in the structure in `%MyVar`, that is `MyVar+4`. The type of `idx1` is `i32*`. However, `idx2` computes the address of *the next* structure after `%MyVar`. The type of `idx2` is `{ [10 x i32] }*` and its value is equivalent to `MyVar + 40` because it indexes past the ten 4-byte integers in `MyVar`. Obviously, in such a situation, the pointers don't alias.

Why do GEP x,1,0,0 and GEP x,1 alias?

Quick Answer: They compute the same address location.

These two GEP instructions will compute the same address because indexing through the 0th element does not change the address. However, it does change the type. Consider this example:

```
%MyVar = global { [10 x i32 ] }
%idx1 = getelementptr { [10 x i32 ] }* %MyVar, i64 1, i32 0, i64 0
%idx2 = getelementptr { [10 x i32 ] }* %MyVar, i64 1
```

In this example, the value of `%idx1` is `%MyVar+40` and its type is `i32*`. The value of `%idx2` is also `MyVar+40` but its type is `{ [10 x i32] }*`.

Can GEP index into vector elements?

This hasn't always been forcefully disallowed, though it's not recommended. It leads to awkward special cases in the optimizers, and fundamental inconsistency in the IR. In the future, it will probably be outright disallowed.

What effect do address spaces have on GEPs?

None, except that the address space qualifier on the first operand pointer type always matches the address space qualifier on the result type.

How is GEP different from `ptrtoint`, arithmetic, and `inttoptr`?

It's very similar; there are only subtle differences.

With ptrtoint, you have to pick an integer type. One approach is to pick i64; this is safe on everything LLVM supports (LLVM internally assumes pointers are never wider than 64 bits in many places), and the optimizer will actually narrow the i64 arithmetic down to the actual pointer size on targets which don't support 64-bit arithmetic in most cases. However, there are some cases where it doesn't do this. With GEP you can avoid this problem.

Also, GEP carries additional pointer aliasing rules. It's invalid to take a GEP from one object, address into a different separately allocated object, and dereference it. IR producers (front-ends) must follow this rule, and consumers (optimizers, specifically alias analysis) benefit from being able to rely on it. See the Rules section for more information.

And, GEP is more concise in common cases.

However, for the underlying integer computation implied, there is no difference.

I'm writing a backend for a target which needs custom lowering for GEP. How do I do this?

You don't. The integer computation implied by a GEP is target-independent. Typically what you'll need to do is make your backend pattern-match expressions trees involving ADD, MUL, etc., which are what GEP is lowered into. This has the advantage of letting your code work correctly in more cases.

GEP does use target-dependent parameters for the size and layout of data types, which targets can customize.

If you require support for addressing units which are not 8 bits, you'll need to fix a lot of code in the backend, with GEP lowering being only a small piece of the overall picture.

How does VLA addressing work with GEPs?

GEPs don't natively support VLAs. LLVM's type system is entirely static, and GEP address computations are guided by an LLVM type.

VLA indices can be implemented as linearized indices. For example, an expression like `X[a][b][c]`, must be effectively lowered into a form like `X[a*m+b*n+c]`, so that it appears to the GEP as a single-dimensional array reference.

This means if you want to write an analysis which understands array indices and you want to support VLAs, your code will have to be prepared to reverse-engineer the linearization. One way to solve this problem is to use the ScalarEvolution library, which always presents VLA and non-VLA indexing in the same manner.

2.19.3 Rules

What happens if an array index is out of bounds?

There are two senses in which an array index can be out of bounds.

First, there's the array type which comes from the (static) type of the first operand to the GEP. Indices greater than the number of elements in the corresponding static array type are valid. There is no problem with out of bounds indices in this sense. Indexing into an array only depends on the size of the array element, not the number of elements.

A common example of how this is used is arrays where the size is not known. It's common to use array types with zero length to represent these. The fact that the static type says there are zero elements is irrelevant; it's perfectly valid to compute arbitrary element indices, as the computation only depends on the size of the array element, not the number of elements. Note that zero-sized arrays are not a special case here.

This sense is unconnected with `inbounds` keyword. The `inbounds` keyword is designed to describe low-level pointer arithmetic overflow conditions, rather than high-level array indexing rules.

Analysis passes which wish to understand array indexing should not assume that the static array type bounds are respected.

The second sense of being out of bounds is computing an address that's beyond the actual underlying allocated object.

With the `inbounds` keyword, the result value of the GEP is undefined if the address is outside the actual underlying allocated object and not the address one-past-the-end.

Without the `inbounds` keyword, there are no restrictions on computing out-of-bounds addresses. Obviously, performing a load or a store requires an address of allocated and sufficiently aligned memory. But the GEP itself is only concerned with computing addresses.

Can array indices be negative?

Yes. This is basically a special case of array indices being out of bounds.

Can I compare two values computed with GEPs?

Yes. If both addresses are within the same allocated object, or one-past-the-end, you'll get the comparison result you expect. If either is outside of it, integer arithmetic wrapping may occur, so the comparison may not be meaningful.

Can I do GEP with a different pointer type than the type of the underlying object?

Yes. There are no restrictions on bitcasting a pointer value to an arbitrary pointer type. The types in a GEP serve only to define the parameters for the underlying integer computation. They need not correspond with the actual type of the underlying object.

Furthermore, loads and stores don't have to use the same types as the type of the underlying object. Types in this context serve only to specify memory size and alignment. Beyond that there are merely a hint to the optimizer indicating how the value will likely be used.

Can I cast an object's address to integer and add it to null?

You can compute an address that way, but if you use GEP to do the add, you can't use that pointer to actually access the object, unless the object is managed outside of LLVM.

The underlying integer computation is sufficiently defined; null has a defined value — zero — and you can add whatever value you want to it.

However, it's invalid to access (load from or store to) an LLVM-aware object with such a pointer. This includes `GlobalVariables`, `Allocas`, and objects pointed to by noalias pointers.

If you really need this functionality, you can do the arithmetic with explicit integer instructions, and use inttoptr to convert the result to an address. Most of GEP's special aliasing rules do not apply to pointers computed from ptrtoint, arithmetic, and inttoptr sequences.

Can I compute the distance between two objects, and add that value to one address to compute the other address?

As with arithmetic on null, you can use GEP to compute an address that way, but you can't use that pointer to actually access the object if you do, unless the object is managed outside of LLVM.

Also as above, ptrtoint and inttoptr provide an alternative way to do this which do not have this restriction.

Can I do type-based alias analysis on LLVM IR?

You can't do type-based alias analysis using LLVM's built-in type system, because LLVM has no restrictions on mixing types in addressing, loads or stores.

LLVM's type-based alias analysis pass uses metadata to describe a different type system (such as the C type system), and performs type-based aliasing on top of that. Further details are in the language reference.

What happens if a GEP computation overflows?

If the GEP lacks the `inbounds` keyword, the value is the result from evaluating the implied two's complement integer computation. However, since there's no guarantee of where an object will be allocated in the address space, such values have limited meaning.

If the GEP has the `inbounds` keyword, the result value is undefined (a "trap value") if the GEP overflows (i.e. wraps around the end of the address space).

As such, there are some ramifications of this for inbounds GEPs: scales implied by array/vector/pointer indices are always known to be "nsw" since they are signed values that are scaled by the element size. These values are also allowed to be negative (e.g. "`gep i32 *%P, i32 -1`") but the pointer itself is logically treated as an unsigned value. This means that GEPs have an asymmetric relation between the pointer base (which is treated as unsigned) and the offset applied to it (which is treated as signed). The result of the additions within the offset calculation cannot have signed overflow, but when applied to the base pointer, there can be signed overflow.

How can I tell if my front-end is following the rules?

There is currently no checker for the getelementptr rules. Currently, the only way to do this is to manually check each place in your front-end where GetElementPtr operators are created.

It's not possible to write a checker which could find all rule violations statically. It would be possible to write a checker which works by instrumenting the code with dynamic checks though. Alternatively, it would be possible to write a static checker which catches a subset of possible problems. However, no such checker exists today.

2.19.4 Rationale

Why is GEP designed this way?

The design of GEP has the following goals, in rough unofficial order of priority:

- Support C, C-like languages, and languages which can be conceptually lowered into C (this covers a lot).
- Support optimizations such as those that are common in C compilers. In particular, GEP is a cornerstone of LLVM's pointer aliasing model.
- Provide a consistent method for computing addresses so that address computations don't need to be a part of load and store instructions in the IR.
- Support non-C-like languages, to the extent that it doesn't interfere with other goals.
- Minimize target-specific information in the IR.

Why do struct member indices always use `i32`?

The specific type i32 is probably just a historical artifact, however it's wide enough for all practical purposes, so there's been no need to change it. It doesn't necessarily imply i32 address arithmetic; it's just an identifier which identifies a field in a struct. Requiring that all struct indices be the same reduces the range of possibilities for cases where two GEPs are effectively the same but have distinct operand types.

What's an uglygep?

Some LLVM optimizers operate on GEPs by internally lowering them into more primitive integer expressions, which allows them to be combined with other integer expressions and/or split into multiple separate integer expressions. If they've made non-trivial changes, translating back into LLVM IR can involve reverse-engineering the structure of the addressing in order to fit it into the static type of the original first operand. It isn't always possibly to fully reconstruct this structure; sometimes the underlying addressing doesn't correspond with the static type at all. In such cases the optimizer instead will emit a GEP with the base pointer casted to a simple address-unit pointer, using the name "uglygep". This isn't pretty, but it's just as valid, and it's sufficient to preserve the pointer aliasing guarantees that GEP provides.

2.19.5 Summary

In summary, here's some things to always remember about the GetElementPtr instruction:

1. The GEP instruction never accesses memory, it only provides pointer computations.
2. The first operand to the GEP instruction is always a pointer and it must be indexed.
3. There are no superfluous indices for the GEP instruction.
4. Trailing zero indices are superfluous for pointer aliasing, but not for the types of the pointers.
5. Leading zero indices are not superfluous for pointer aliasing nor the types of the pointers.

2.20 MCJIT Design and Implementation

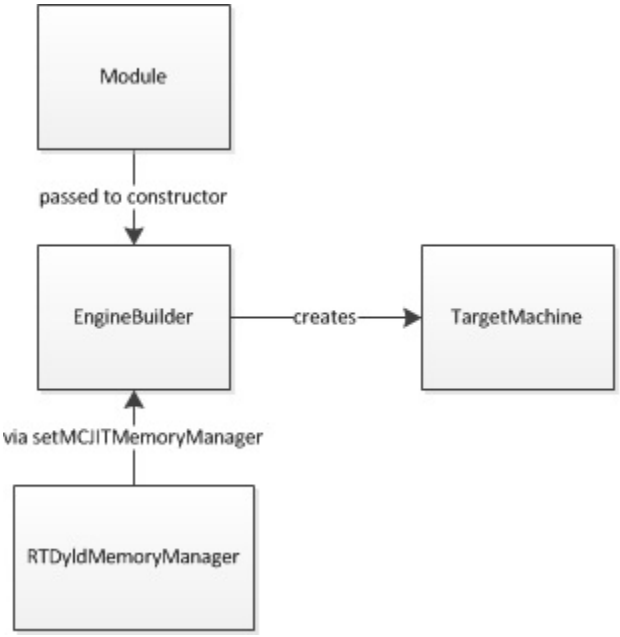
2.20.1 Introduction

This document describes the internal workings of the MCJIT execution engine and the RuntimeDyld component. It is intended as a high level overview of the implementation, showing the flow and interactions of objects throughout the code generation and dynamic loading process.

2.20.2 Engine Creation

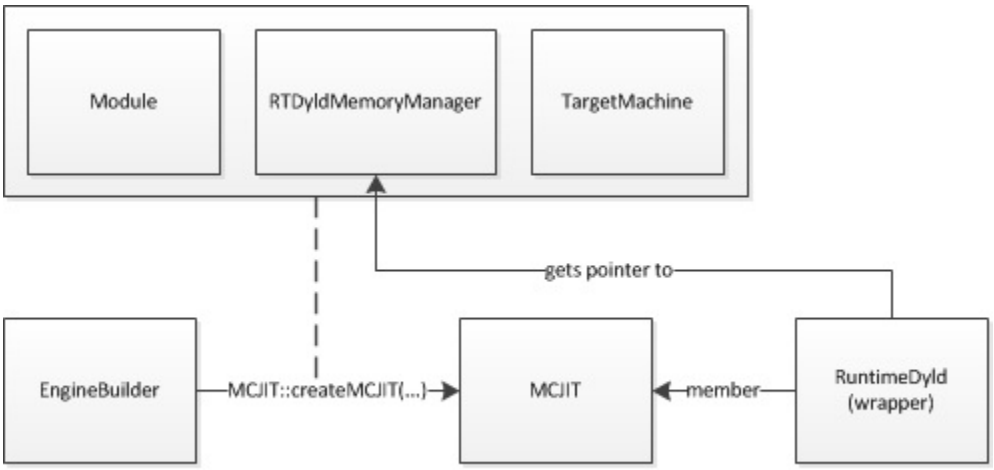
In most cases, an EngineBuilder object is used to create an instance of the MCJIT execution engine. The EngineBuilder takes an llvm::Module object as an argument to its constructor. The client may then set various options that we control the later be passed along to the MCJIT engine, including the selection of MCJIT as the engine type to be created. Of particular interest is the EngineBuilder::setMCJITMemoryManager function. If the client does not explicitly create a memory manager at this time, a default memory manager (specifically SectionMemoryManager) will be created when the MCJIT engine is instantiated.

Once the options have been set, a client calls EngineBuilder::create to create an instance of the MCJIT engine. If the client does not use the form of this function that takes a TargetMachine as a parameter, a new TargetMachine will be created based on the target triple associated with the Module that was used to create the EngineBuilder.



EngineBuilder::create will call the static MCJIT::createJIT function, passing in its pointers to the module, memory manager and target machine objects, all of which will subsequently be owned by the MCJIT object.

The MCJIT class has a member variable, Dyld, which contains an instance of the RuntimeDyld wrapper class. This member will be used for communications between MCJIT and the actual RuntimeDyldImpl object that gets created when an object is loaded.

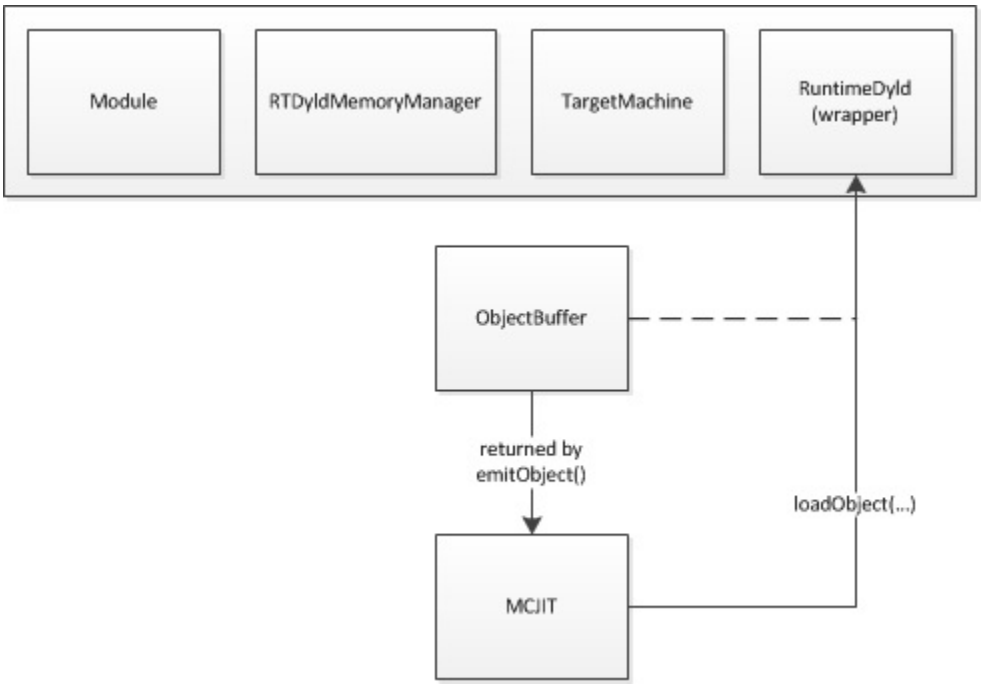


Upon creation, MCJIT holds a pointer to the Module object that it received from EngineBuilder but it does not immediately generate code for this module. Code generation is deferred until either the MCJIT::finalizeObject method is called explicitly or a function such as MCJIT::getPointerToFunction is called which requires the code to have been generated.

2.20.3 Code Generation

When code generation is triggered, as described above, MCJIT will first attempt to retrieve an object image from its ObjectCache member, if one has been set. If a cached object image cannot be retrieved, MCJIT will call its emitObject method. MCJIT::emitObject uses a local PassManager instance and creates a new ObjectBufferStream instance, both

of which it passes to TargetMachine::addPassesToEmitMC before calling PassManager::run on the Module with which it was created.

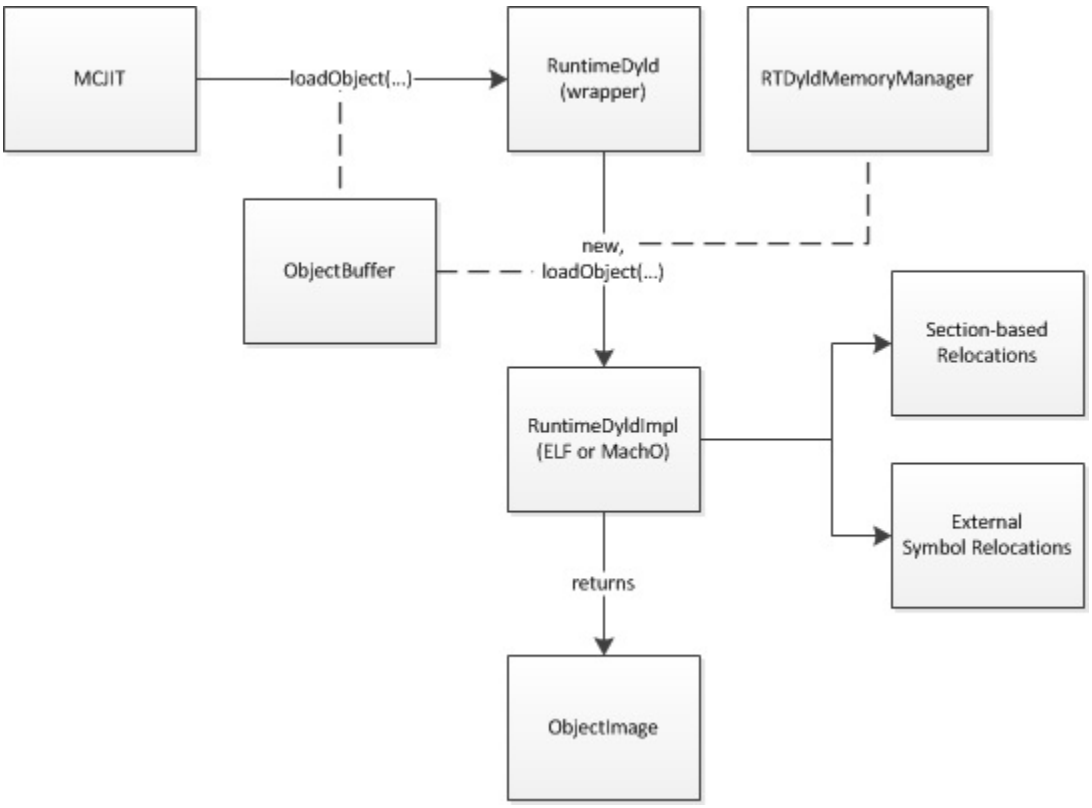


The PassManager::run call causes the MC code generation mechanisms to emit a complete relocatable binary object image (either in either ELF or MachO format, depending on the target) into the ObjectBufferStream object, which is flushed to complete the process. If an ObjectCache is being used, the image will be passed to the ObjectCache here.

At this point, the ObjectBufferStream contains the raw object image. Before the code can be executed, the code and data sections from this image must be loaded into suitable memory, relocations must be applied and memory permission and code cache invalidation (if required) must be completed.

2.20.4 Object Loading

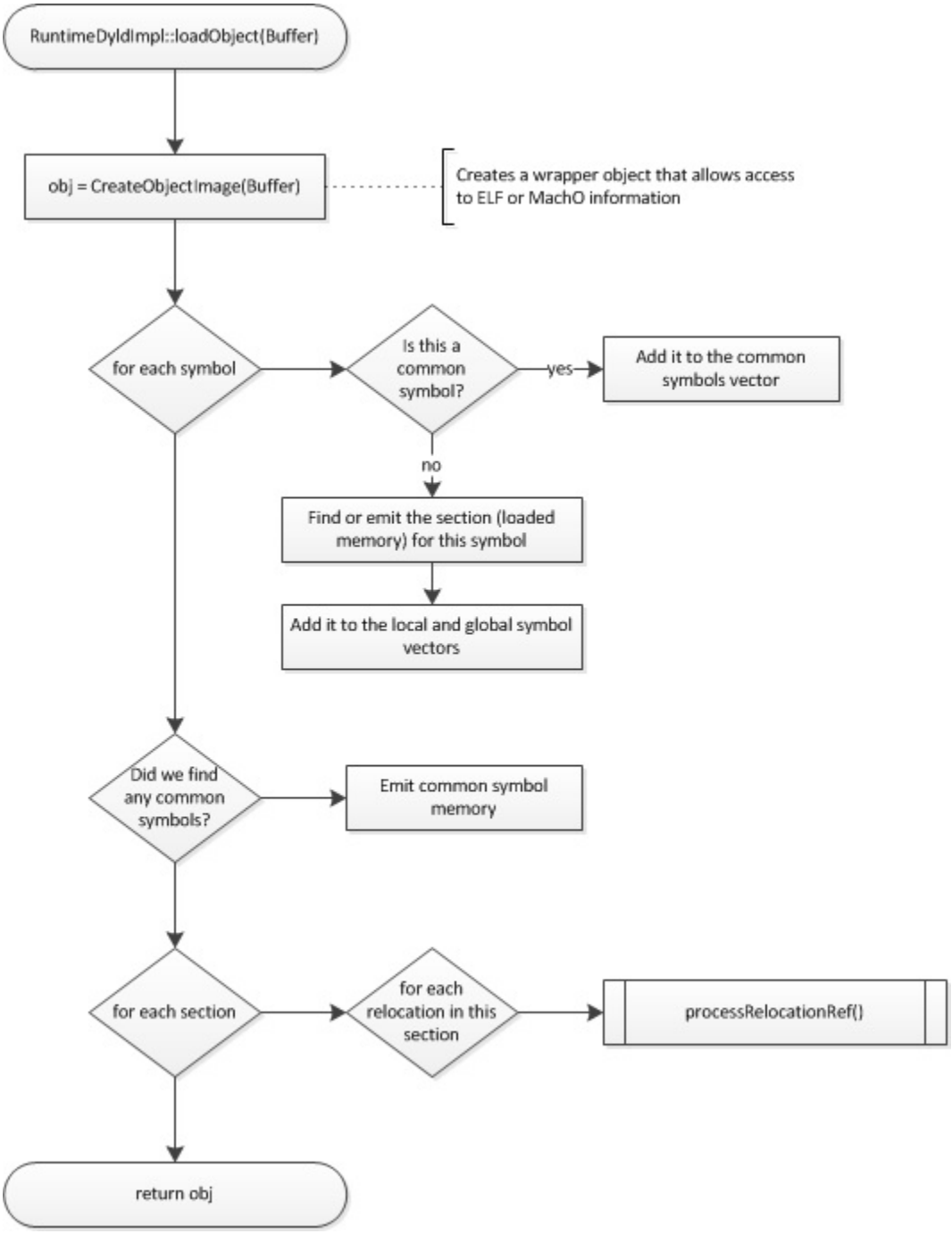
Once an object image has been obtained, either through code generation or having been retrieved from an ObjectCache, it is passed to RuntimeDyld to be loaded. The RuntimeDyld wrapper class examines the object to determine its file format and creates an instance of either RuntimeDyldELF or RuntimeDyldMachO (both of which derive from the RuntimeDyldImpl base class) and calls the RuntimeDyldImpl::loadObject method to perform that actual loading.



RuntimeDyldImpl::loadObject begins by creating an ObjectImage instance from the ObjectBuffer it received. ObjectImage, which wraps the ObjectFile class, is a helper class which parses the binary object image and provides access to the information contained in the format-specific headers, including section, symbol and relocation information.

RuntimeDyldImpl::loadObject then iterates through the symbols in the image. Information about common symbols is collected for later use. For each function or data symbol, the associated section is loaded into memory and the symbol is stored in a symbol table map data structure. When the iteration is complete, a section is emitted for the common symbols.

Next, RuntimeDyldImpl::loadObject iterates through the sections in the object image and for each section iterates through the relocations for that sections. For each relocation, it calls the format-specific processRelocationRef method, which will examine the relocation and store it in one of two data structures, a section-based relocation list map and an external symbol relocation map.



When RuntimeDyldImpl::loadObject returns, all of the code and data sections for the object will have been loaded into memory allocated by the memory manager and relocation information will have been prepared, but the relocations have not yet been applied and the generated code is still not ready to be executed.

[Currently (as of August 2013) the MCJIT engine will immediately apply relocations when loadObject completes. However, this shouldn't be happening. Because the code may have been generated for a remote target, the client should be given a chance to re-map the section addresses before relocations are applied. It is possible to apply relocations multiple times, but in the case where addresses are to be re-mapped, this first application is wasted effort.]

2.20.5 Address Remapping

At any time after initial code has been generated and before finalizeObject is called, the client can remap the address of sections in the object. Typically this is done because the code was generated for an external process and is being mapped into that process' address space. The client remaps the section address by calling MCJIT::mapSectionAddress. This should happen before the section memory is copied to its new location.

When MCJIT::mapSectionAddress is called, MCJIT passes the call on to RuntimeDyldImpl (via its Dyld member). RuntimeDyldImpl stores the new address in an internal data structure but does not update the code at this time, since other sections are likely to change.

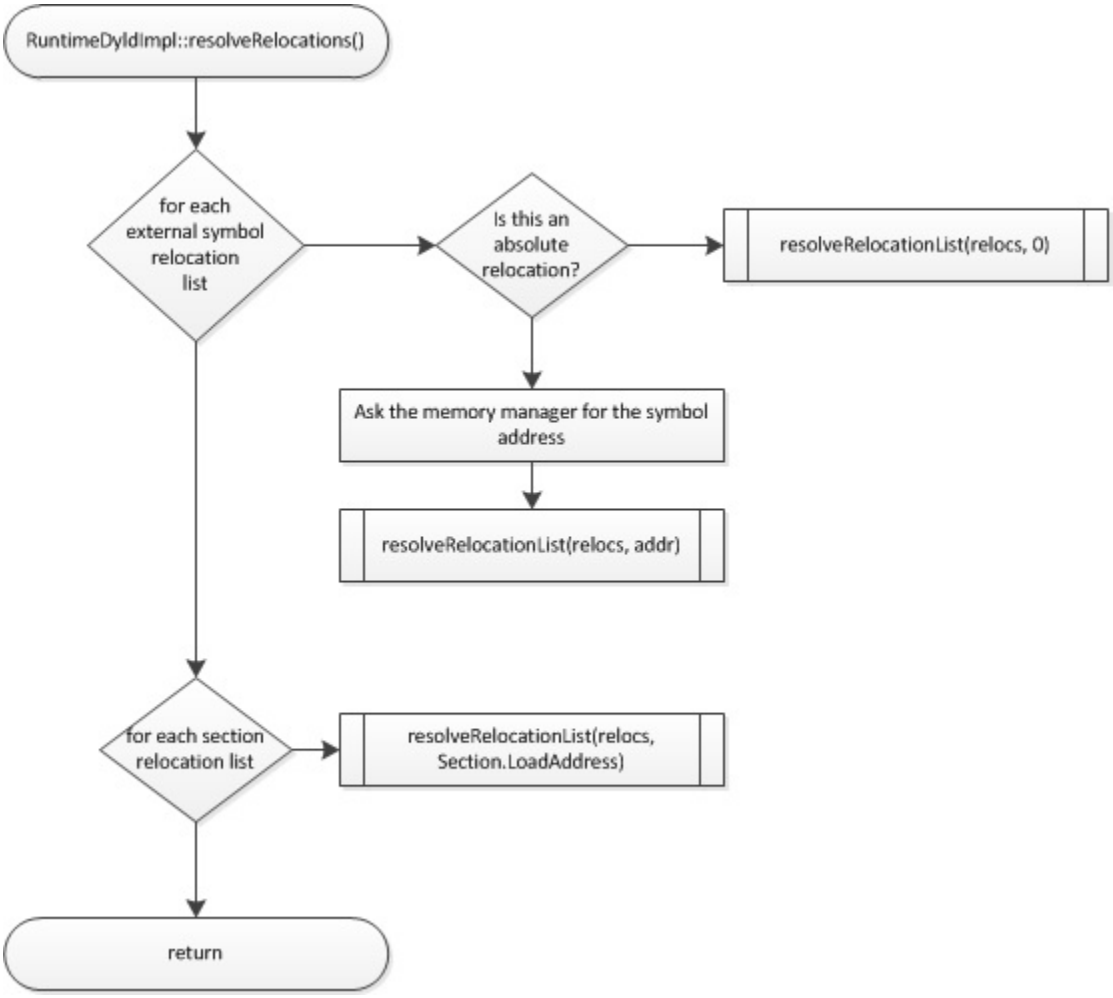
When the client is finished remapping section addresses, it will call MCJIT::finalizeObject to complete the remapping process.

2.20.6 Final Preparations

When MCJIT::finalizeObject is called, MCJIT calls RuntimeDyld::resolveRelocations. This function will attempt to locate any external symbols and then apply all relocations for the object.

External symbols are resolved by calling the memory manager's getPointerToNamedFunction method. The memory manager will return the address of the requested symbol in the target address space. (Note, this may not be a valid pointer in the host process.) RuntimeDyld will then iterate through the list of relocations it has stored which are associated with this symbol and invoke the resolveRelocation method which, through an format-specific implementation, will apply the relocation to the loaded section memory.

Next, RuntimeDyld::resolveRelocations iterates through the list of sections and for each section iterates through a list of relocations that have been saved which reference that symbol and call resolveRelocation for each entry in this list. The relocation list here is a list of relocations for which the symbol associated with the relocation is located in the section associated with the list. Each of these locations will have a target location at which the relocation will be applied that is likely located in a different section.



Once relocations have been applied as described above, MCJIT calls RuntimeDyld::getEHFrameSection, and if a non-zero result is returned passes the section data to the memory manager's registerEHFrames method. This allows the memory manager to call any desired target-specific functions, such as registering the EH frame information with a debugger.

Finally, MCJIT calls the memory manager's finalizeMemory method. In this method, the memory manager will invalidate the target code cache, if necessary, and apply final permissions to the memory pages it has allocated for code and data memory.

Getting Started with the LLVM System Discusses how to get up and running quickly with the LLVM infrastructure. Everything from unpacking and compilation of the distribution to execution of some tools.

Building LLVM with CMake An addendum to the main Getting Started guide for those using the CMake build system.

How To Build On ARM Notes on building and testing LLVM/Clang on ARM.

How To Cross-Compile Clang/LLVM using Clang/LLVM Notes on cross-building and testing LLVM/Clang.

Getting Started with the LLVM System using Microsoft Visual Studio An addendum to the main Getting Started guide for those using Visual Studio on Windows.

LLVM Tutorial: Table of Contents Tutorials about using LLVM. Includes a tutorial about making a custom language with LLVM.

LLVM Command Guide A reference manual for the LLVM command line utilities ("man" pages for LLVM tools).

LLVM's Analysis and Transform Passes A list of optimizations and analyses implemented in LLVM.

Frequently Asked Questions (FAQ) A list of common questions and problems and their solutions.

Release notes for the current release This describes new features, known bugs, and other limitations.

How to submit an LLVM bug report Instructions for properly submitting information about any bugs you run into in the LLVM system.

Sphinx Quickstart Template A template + tutorial for writing new Sphinx documentation. It is meant to be read in source form.

LLVM Testing Infrastructure Guide A reference manual for using the LLVM testing infrastructure.

How to build the C, C++, ObjC, and ObjC++ front end Instructions for building the clang front-end from source.

The LLVM Lexicon Definition of acronyms, terms and concepts used in LLVM.

How To Add Your Build Configuration To LLVM Buildbot Infrastructure Instructions for adding new builder to LLVM buildbot master.

YAML I/O A reference guide for using LLVM's YAML I/O library.

The Often Misunderstood GEP Instruction Answers to some very frequent questions about LLVM's most frequently misunderstood instruction.

CHAPTER

THREE

PROGRAMMING DOCUMENTATION

For developers of applications which use LLVM as a library.

3.1 LLVM Atomic Instructions and Concurrency Guide

3.1.1 Introduction

Historically, LLVM has not had very strong support for concurrency; some minimal intrinsics were provided, and `volatile` was used in some cases to achieve rough semantics in the presence of concurrency. However, this is changing; there are now new instructions which are well-defined in the presence of threads and asynchronous signals, and the model for existing instructions has been clarified in the IR.

The atomic instructions are designed specifically to provide readable IR and optimized code generation for the following:

- The new C++11 `<atomic>` header. (C++11 draft available here.) (C11 draft available here.)
- Proper semantics for Java-style memory, for both `volatile` and regular shared variables. (Java Specification)
- gcc-compatible `__sync_*` builtins. (Description)
- Other scenarios with atomic semantics, including `static` variables with non-trivial constructors in C++.

Atomic and volatile in the IR are orthogonal; "volatile" is the C/C++ volatile, which ensures that every volatile load and store happens and is performed in the stated order. A couple examples: if a SequentiallyConsistent store is

immediately followed by another SequentiallyConsistent store to the same address, the first store can be erased. This transformation is not allowed for a pair of volatile stores. On the other hand, a non-volatile non-atomic load can be moved across a volatile load freely, but not an Acquire load.

This document is intended to provide a guide to anyone either writing a frontend for LLVM or working on optimization passes for LLVM with a guide for how to deal with instructions with special semantics in the presence of concurrency. This is not intended to be a precise guide to the semantics; the details can get extremely complicated and unreadable, and are not usually necessary.

3.1.2 Optimization outside atomic

The basic `'load'` and `'store'` allow a variety of optimizations, but can lead to undefined results in a concurrent environment; see NotAtomic. This section specifically goes into the one optimizer restriction which applies in concurrent environments, which gets a bit more of an extended description because any optimization dealing with stores needs to be aware of it.

From the optimizer's point of view, the rule is that if there are not any instructions with atomic ordering involved, concurrency does not matter, with one exception: if a variable might be visible to another thread or signal handler, a store cannot be inserted along a path where it might not execute otherwise. Take the following example:

```
/* C code, for readability; run through clang -O2 -S -emit-llvm to get
    equivalent IR */
int x;
void f(int* a) {
  for (int i = 0; i < 100; i++) {
    if (a[i])
      x += 1;
  }
}
```

The following is equivalent in non-concurrent situations:

```
int x;
void f(int* a) {
  int xtemp = x;
  for (int i = 0; i < 100; i++) {
    if (a[i])
      xtemp += 1;
  }
  x = xtemp;
}
```

However, LLVM is not allowed to transform the former to the latter: it could indirectly introduce undefined behavior if another thread can access `x` at the same time. (This example is particularly of interest because before the concurrency model was implemented, LLVM would perform this transformation.)

Note that speculative loads are allowed; a load which is part of a race returns `undef`, but does not have undefined behavior.

3.1.3 Atomic instructions

For cases where simple loads and stores are not sufficient, LLVM provides various atomic instructions. The exact guarantees provided depend on the ordering; see Atomic orderings.

`load atomic` and `store atomic` provide the same basic functionality as non-atomic loads and stores, but provide additional guarantees in situations where threads and signals are involved.

`cmpxchg` and `atomicrmw` are essentially like an atomic load followed by an atomic store (where the store is conditional for `cmpxchg`), but no other memory operation can happen on any thread between the load and store.

A `fence` provides Acquire and/or Release ordering which is not part of another operation; it is normally used along with Monotonic memory operations. A Monotonic load followed by an Acquire fence is roughly equivalent to an Acquire load, and a Monotonic store following a Release fence is roughly equivalent to a Release store. Sequentially-Consistent fences behave as both an Acquire and a Release fence, and offer some additional complicated guarantees, see the C++11 standard for details.

Frontends generating atomic instructions generally need to be aware of the target to some degree; atomic instructions are guaranteed to be lock-free, and therefore an instruction which is wider than the target natively supports can be impossible to generate.

3.1.4 Atomic orderings

In order to achieve a balance between performance and necessary guarantees, there are six levels of atomicity. They are listed in order of strength; each level includes all the guarantees of the previous level except for Acquire/Release. (See also LangRef Ordering.)

NotAtomic

NotAtomic is the obvious, a load or store which is not atomic. (This isn't really a level of atomicity, but is listed here for comparison.) This is essentially a regular load or store. If there is a race on a given memory location, loads from that location return undef.

Relevant standard This is intended to match shared variables in C/C++, and to be used in any other context where memory access is necessary, and a race is impossible. (The precise definition is in LangRef Memory Model.)

Notes for frontends The rule is essentially that all memory accessed with basic loads and stores by multiple threads should be protected by a lock or other synchronization; otherwise, you are likely to run into undefined behavior. If your frontend is for a "safe" language like Java, use Unordered to load and store any shared variable. Note that NotAtomic volatile loads and stores are not properly atomic; do not try to use them as a substitute. (Per the C/C++ standards, volatile does provide some limited guarantees around asynchronous signals, but atomics are generally a better solution.)

Notes for optimizers Introducing loads to shared variables along a codepath where they would not otherwise exist is allowed; introducing stores to shared variables is not. See Optimization outside atomic.

Notes for code generation The one interesting restriction here is that it is not allowed to write to bytes outside of the bytes relevant to a store. This is mostly relevant to unaligned stores: it is not allowed in general to convert an unaligned store into two aligned stores of the same width as the unaligned store. Backends are also expected to generate an i8 store as an i8 store, and not an instruction which writes to surrounding bytes. (If you are writing a backend for an architecture which cannot satisfy these restrictions and cares about concurrency, please send an email to llvmdev.)

Unordered

Unordered is the lowest level of atomicity. It essentially guarantees that races produce somewhat sane results instead of having undefined behavior. It also guarantees the operation to be lock-free, so it does not depend on the data being part of a special atomic structure or depend on a separate per-process global lock. Note that code generation will fail for unsupported atomic operations; if you need such an operation, use explicit locking.

Relevant standard This is intended to match the Java memory model for shared variables.

Notes for frontends This cannot be used for synchronization, but is useful for Java and other "safe" languages which need to guarantee that the generated code never exhibits undefined behavior. Note that this guarantee is cheap on common platforms for loads of a native width, but can be expensive or unavailable for wider loads, like a 64-bit store on ARM. (A frontend for Java or other "safe" languages would normally split a 64-bit store on ARM into two 32-bit unordered stores.)

Notes for optimizers In terms of the optimizer, this prohibits any transformation that transforms a single load into multiple loads, transforms a store into multiple stores, narrows a store, or stores a value which would not be stored otherwise. Some examples of unsafe optimizations are narrowing an assignment into a bitfield, rematerializing a load, and turning loads and stores into a memcpy call. Reordering unordered operations is safe, though, and optimizers should take advantage of that because unordered operations are common in languages that need them.

Notes for code generation These operations are required to be atomic in the sense that if you use unordered loads and unordered stores, a load cannot see a value which was never stored. A normal load or store instruction is usually sufficient, but note that an unordered load or store cannot be split into multiple instructions (or an instruction which does multiple memory operations, like `LDRD` on ARM without LPAE, or not naturally-aligned `LDRD` on LPAE ARM).

Monotonic

Monotonic is the weakest level of atomicity that can be used in synchronization primitives, although it does not provide any general synchronization. It essentially guarantees that if you take all the operations affecting a specific address, a consistent ordering exists.

Relevant standard This corresponds to the C++11/C11 `memory_order_relaxed`; see those standards for the exact definition.

Notes for frontends If you are writing a frontend which uses this directly, use with caution. The guarantees in terms of synchronization are very weak, so make sure these are only used in a pattern which you know is correct. Generally, these would either be used for atomic operations which do not protect other memory (like an atomic counter), or along with a `fence`.

Notes for optimizers In terms of the optimizer, this can be treated as a read+write on the relevant memory location (and alias analysis will take advantage of that). In addition, it is legal to reorder non-atomic and Unordered loads around Monotonic loads. CSE/DSE and a few other optimizations are allowed, but Monotonic operations are unlikely to be used in ways which would make those optimizations useful.

Notes for code generation Code generation is essentially the same as that for unordered for loads and stores. No fences are required. `cmpxchg` and `atomicrmw` are required to appear as a single operation.

Acquire

Acquire provides a barrier of the sort necessary to acquire a lock to access other memory with normal loads and stores.

Relevant standard This corresponds to the C++11/C11 `memory_order_acquire`. It should also be used for C++11/C11 `memory_order_consume`.

Notes for frontends If you are writing a frontend which uses this directly, use with caution. Acquire only provides a semantic guarantee when paired with a Release operation.

Notes for optimizers Optimizers not aware of atomics can treat this like a nothrow call. It is also possible to move stores from before an Acquire load or read-modify-write operation to after it, and move non-Acquire loads from before an Acquire operation to after it.

Notes for code generation Architectures with weak memory ordering (essentially everything relevant today except x86 and SPARC) require some sort of fence to maintain the Acquire semantics. The precise fences required varies widely by architecture, but for a simple implementation, most architectures provide a barrier which is

strong enough for everything (`dmb` on ARM, `sync` on PowerPC, etc.). Putting such a fence after the equivalent Monotonic operation is sufficient to maintain Acquire semantics for a memory operation.

Release

Release is similar to Acquire, but with a barrier of the sort necessary to release a lock.

Relevant standard This corresponds to the C++11/C11 `memory_order_release`.

Notes for frontends If you are writing a frontend which uses this directly, use with caution. Release only provides a semantic guarantee when paired with a Acquire operation.

Notes for optimizers Optimizers not aware of atomics can treat this like a nothrow call. It is also possible to move loads from after a Release store or read-modify-write operation to before it, and move non-Release stores from after an Release operation to before it.

Notes for code generation See the section on Acquire; a fence before the relevant operation is usually sufficient for Release. Note that a store-store fence is not sufficient to implement Release semantics; store-store fences are generally not exposed to IR because they are extremely difficult to use correctly.

AcquireRelease

AcquireRelease (`acq_rel` in IR) provides both an Acquire and a Release barrier (for fences and operations which both read and write memory).

Relevant standard This corresponds to the C++11/C11 `memory_order_acq_rel`.

Notes for frontends If you are writing a frontend which uses this directly, use with caution. Acquire only provides a semantic guarantee when paired with a Release operation, and vice versa.

Notes for optimizers In general, optimizers should treat this like a nothrow call; the possible optimizations are usually not interesting.

Notes for code generation This operation has Acquire and Release semantics; see the sections on Acquire and Release.

SequentiallyConsistent

SequentiallyConsistent (`seq_cst` in IR) provides Acquire semantics for loads and Release semantics for stores. Additionally, it guarantees that a total ordering exists between all SequentiallyConsistent operations.

Relevant standard This corresponds to the C++11/C11 `memory_order_seq_cst`, Java volatile, and the gcc-compatible `__sync_*` builtins which do not specify otherwise.

Notes for frontends If a frontend is exposing atomic operations, these are much easier to reason about for the programmer than other kinds of operations, and using them is generally a practical performance tradeoff.

Notes for optimizers Optimizers not aware of atomics can treat this like a nothrow call. For SequentiallyConsistent loads and stores, the same reorderings are allowed as for Acquire loads and Release stores, except that SequentiallyConsistent operations may not be reordered.

Notes for code generation SequentiallyConsistent loads minimally require the same barriers as Acquire operations and SequentiallyConsistent stores require Release barriers. Additionally, the code generator must enforce ordering between SequentiallyConsistent stores followed by SequentiallyConsistent loads. This is usually done by emitting either a full fence before the loads or a full fence after the stores; which is preferred varies by architecture.

3.1.5 Atomics and IR optimization

Predicates for optimizer writers to query:

- `isSimple()`: A load or store which is not volatile or atomic. This is what, for example, memcpyopt would check for operations it might transform.
- `isUnordered()`: A load or store which is not volatile and at most Unordered. This would be checked, for example, by LICM before hoisting an operation.
- `mayReadFromMemory()`/`mayWriteToMemory()`: Existing predicate, but note that they return true for any operation which is volatile or at least Monotonic.
- `isAtLeastAcquire()`/`isAtLeastRelease()`: These are predicates on orderings. They can be useful for passes that are aware of atomics, for example to do DSE across a single atomic access, but not across a release-acquire pair (see MemoryDependencyAnalysis for an example of this)
- Alias analysis: Note that AA will return ModRef for anything Acquire or Release, and for the address accessed by any Monotonic operation.

To support optimizing around atomic operations, make sure you are using the right predicates; everything should work if that is done. If your pass should optimize some atomic operations (Unordered operations in particular), make sure it doesn't replace an atomic load or store with a non-atomic operation.

Some examples of how optimizations interact with various kinds of atomic operations:

- `memcpyopt`: An atomic operation cannot be optimized into part of a memcpy/memset, including unordered loads/stores. It can pull operations across some atomic operations.
- LICM: Unordered loads/stores can be moved out of a loop. It just treats monotonic operations like a read+write to a memory location, and anything stricter than that like a nothrow call.
- DSE: Unordered stores can be DSE'ed like normal stores. Monotonic stores can be DSE'ed in some cases, but it's tricky to reason about, and not especially important. It is possible in some case for DSE to operate across a stronger atomic operation, but it is fairly tricky. DSE delegates this reasoning to MemoryDependencyAnalysis (which is also used by other passes like GVN).
- Folding a load: Any atomic load from a constant global can be constant-folded, because it cannot be observed. Similar reasoning allows scalarrepl with atomic loads and stores.

3.1.6 Atomics and Codegen

Atomic operations are represented in the SelectionDAG with `ATOMIC_*` opcodes. On architectures which use barrier instructions for all atomic ordering (like ARM), appropriate fences can be emitted by the AtomicExpand Codegen pass if `setInsertFencesForAtomic()` was used.

The MachineMemOperand for all atomic operations is currently marked as volatile; this is not correct in the IR sense of volatile, but CodeGen handles anything marked volatile very conservatively. This should get fixed at some point.

Common architectures have some way of representing at least a pointer-sized lock-free `cmpxchg`; such an operation can be used to implement all the other atomic operations which can be represented in IR up to that size. Backends are expected to implement all those operations, but not operations which cannot be implemented in a lock-free manner. It is expected that backends will give an error when given an operation which cannot be implemented. (The LLVM code generator is not very helpful here at the moment, but hopefully that will change.)

On x86, all atomic loads generate a `MOV`. SequentiallyConsistent stores generate an `XCHG`, other stores generate a `MOV`. SequentiallyConsistent fences generate an `MFENCE`, other fences do not cause any code to be generated. cmpxchg uses the `LOCK CMPXCHG` instruction. `atomicrmw xchg` uses `XCHG`, `atomicrmw add` and `atomicrmw sub` use `XADD`, and all other `atomicrmw` operations generate a loop with `LOCK CMPXCHG`. Depending on the users of

the result, some `atomicrmw` operations can be translated into operations like `LOCK AND`, but that does not work in general.

On ARM (before v8), MIPS, and many other RISC architectures, Acquire, Release, and SequentiallyConsistent semantics require barrier instructions for every such operation. Loads and stores generate normal instructions. `cmpxchg` and `atomicrmw` can be represented using a loop with LL/SC-style instructions which take some sort of exclusive lock on a cache line (`LDREX` and `STREX` on ARM, etc.).

It is often easiest for backends to use AtomicExpandPass to lower some of the atomic constructs. Here are some lowerings it can do:

- cmpxchg -> loop with load-linked/store-conditional by overriding `hasLoadLinkedStoreConditional()`, `emitLoadLinked()`, `emitStoreConditional()`
- large loads/stores -> ll-sc/cmpxchg by overriding `shouldExpandAtomicStoreInIR()`/`shouldExpandAtomicLoadIn`
- strong atomic accesses -> monotonic accesses + fences by using `setInsertFencesForAtomic()` and overriding `emitLeadingFence()` and `emitTrailingFence()`
- atomic rmw -> loop with cmpxchg or load-linked/store-conditional by overriding `expandAtomicRMWInIR()`

For an example of all of these, look at the ARM backend.

3.2 LLVM Coding Standards

3.2.1 Introduction

This document attempts to describe a few coding standards that are being used in the LLVM source tree. Although no coding standards should be regarded as absolute requirements to be followed in all instances, coding standards are particularly important for large-scale code bases that follow a library-based design (like LLVM).

While this document may provide guidance for some mechanical formatting issues, whitespace, or other "microscopic details", these are not fixed standards. Always follow the golden rule:

> **If you are extending, enhancing, or bug fixing already implemented code, use the style that is already being used so that the source is uniform and easy to follow.**

Note that some code bases (e.g. `libc++`) have really good reasons to deviate from the coding standards. In the case of `libc++`, this is because the naming and other conventions are dictated by the C++ standard. If you think there is a specific good reason to deviate from the standards here, please bring it up on the LLVMdev mailing list.

There are some conventions that are not uniformly followed in the code base (e.g. the naming convention). This is because they are relatively new, and a lot of code was written before they were put in place. Our long term goal is for the entire codebase to follow the convention, but we explicitly *do not* want patches that do large-scale reformating of existing code. On the other hand, it is reasonable to rename the methods of a class if you're about to change it in some other way. Just do the reformating as a separate commit from the functionality change.

The ultimate goal of these guidelines is the increase readability and maintainability of our common source base. If you have suggestions for topics to be included, please mail them to Chris.

3.2.2 Languages, Libraries, and Standards

Most source code in LLVM and other LLVM projects using these coding standards is C++ code. There are some places where C code is used either due to environment restrictions, historical restrictions, or due to third-party source code imported into the tree. Generally, our preference is for standards conforming, modern, and portable C++ code as the implementation language of choice.

C++ Standard Versions

LLVM, Clang, and LLD are currently written using C++11 conforming code, although we restrict ourselves to features which are available in the major toolchains supported as host compilers. The LLDB project is even more aggressive in the set of host compilers supported and thus uses still more features. Regardless of the supported features, code is expected to (when reasonable) be standard, portable, and modern C++11 code. We avoid unnecessary vendor-specific extensions, etc.

C++ Standard Library

Use the C++ standard library facilities whenever they are available for a particular task. LLVM and related projects emphasize and rely on the standard library facilities for as much as possible. Common support libraries providing functionality missing from the standard library for which there are standard interfaces or active work on adding standard interfaces will often be implemented in the LLVM namespace following the expected standard interface.

There are some exceptions such as the standard I/O streams library which are avoided. Also, there is much more detailed information on these subjects in the *LLVM Programmer's Manual*.

Supported C++11 Language and Library Features

While LLVM, Clang, and LLD use C++11, not all features are available in all of the toolchains which we support. The set of features supported for use in LLVM is the intersection of those supported in MSVC 2012, GCC 4.7, and Clang

3.1. The ultimate definition of this set is what build bots with those respective toolchains accept. Don't argue with the build bots. However, we have some guidance below to help you know what to expect.

Each toolchain provides a good reference for what it accepts:

- Clang: http://clang.llvm.org/cxx_status.html
- GCC: http://gcc.gnu.org/projects/cxx0x.html
- MSVC: http://msdn.microsoft.com/en-us/library/hh567368.aspx

In most cases, the MSVC list will be the dominating factor. Here is a summary of the features that are expected to work. Features not on this list are unlikely to be supported by our host compilers.

- Rvalue references: N2118
  - But *not* Rvalue references for `*this` or member qualifiers (N2439)
- Static assert: N1720
- `auto` type deduction: N1984, N1737
- Trailing return types: N2541
- Lambdas: N2927
  - But *not* lambdas with default arguments.
- `decltype`: N2343
- Nested closing right angle brackets: N1757
- Extern templates: N1987
- `nullptr`: N2431
- Strongly-typed and forward declarable enums: N2347, N2764
- Local and unnamed types as template arguments: N2657
- Range-based for-loop: N2930
  - But `{}` are required around inner `do {} while()` loops. As a result, `{}` are required around function-like macros inside range-based for loops.
- `override` and `final`: N2928, N3206, N3272
- Atomic operations and the C++11 memory model: N2429

The supported features in the C++11 standard libraries are less well tracked, but also much greater. Most of the standard libraries implement most of C++11's library. The most likely lowest common denominator is Linux support. For libc++, the support is just poorly tested and undocumented but expected to be largely complete. YMMV. For libstdc++, the support is documented in detail in the libstdc++ manual. There are some very minor missing facilities that are unlikely to be common problems, and there are a few larger gaps that are worth being aware of:

- Not all of the type traits are implemented
- No regular expression library.
- While most of the atomics library is well implemented, the fences are missing. Fortunately, they are rarely needed.
- The locale support is incomplete.
- `std::initializer_list` (and the constructors and functions that take it as an argument) are not always available, so you cannot (for example) initialize a `std::vector` with a braced initializer list.
- `std::equal()` (and other algorithms) incorrectly assert in MSVC when given `nullptr` as an iterator.

Other than these areas you should assume the standard library is available and working as expected until some build bot tells you otherwise. If you're in an uncertain area of one of the above points, but you cannot test on a Linux system, your best approach is to minimize your use of these features, and watch the Linux build bots to find out if your usage triggered a bug. For example, if you hit a type trait which doesn't work we can then add support to LLVM's traits header to emulate it.

Other Languages

Any code written in the Go programming language is not subject to the formatting rules below. Instead, we adopt the formatting rules enforced by the gofmt tool.

Go code should strive to be idiomatic. Two good sets of guidelines for what this means are Effective Go and Go Code Review Comments.

3.2.3 Mechanical Source Issues

Source Code Formatting

Commenting

Comments are one critical part of readability and maintainability. Everyone knows they should comment their code, and so should you. When writing comments, write them as English prose, which means they should use proper capitalization, punctuation, etc. Aim to describe what the code is trying to do and why, not *how* it does it at a micro level. Here are a few critical things to document:

File Headers Every source file should have a header on it that describes the basic purpose of the file. If a file does not have a header, it should not be checked into the tree. The standard header looks like this:

```
//===-- llvm/Instruction.h - Instruction class definition -------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
///
/// \file
/// \brief This file contains the declaration of the Instruction class, which is
/// the base class for all of the VM instructions.
///
//===----------------------------------------------------------------------===//
```

A few things to note about this particular format: The "`-*- C++ -*-`" string on the first line is there to tell Emacs that the source file is a C++ file, not a C file (Emacs assumes `.h` files are C files by default).

Note: This tag is not necessary in `.cpp` files. The name of the file is also on the first line, along with a very short description of the purpose of the file. This is important when printing out code and flipping though lots of pages.

The next section in the file is a concise note that defines the license that the file is released under. This makes it perfectly clear what terms the source code can be distributed under and should not be modified in any way.

The main body is a `doxygen` comment describing the purpose of the file. It should have a `\brief` command that describes the file in one or two sentences. Any additional information should be separated by a blank line. If an algorithm is being implemented or something tricky is going on, a reference to the paper where it is published should be included, as well as any notes or *gotchas* in the code to watch out for.

Class overviews Classes are one fundamental part of a good object oriented design. As such, a class definition should have a comment block that explains what the class is used for and how it works. Every non-trivial class is expected to have a `doxygen` comment block.

Method information Methods defined in a class (as well as any global functions) should also be documented properly. A quick note about what it does and a description of the borderline behaviour is all that is necessary here (unless something particularly tricky or insidious is going on). The hope is that people can figure out how to use your interfaces without reading the code itself.

Good things to talk about here are what happens when something unexpected happens: does the method return null? Abort? Format your hard disk?

Comment Formatting

In general, prefer C++ style (`//`) comments. They take less space, require less typing, don't have nesting problems, etc. There are a few cases when it is useful to use C style (`/* */`) comments however:

1. When writing C code: Obviously if you are writing C code, use C style comments.
2. When writing a header file that may be `#include`d by a C source file.
3. When writing a source file that is used by a tool that only accepts C style comments.

To comment out a large block of code, use `#if 0` and `#endif`. These nest properly and are better behaved in general than C style comments.

Doxygen Use in Documentation Comments

Use the `\file` command to turn the standard file header into a file-level comment.

Include descriptive `\brief` paragraphs for all public interfaces (public classes, member and non-member functions). Explain API use and purpose in `\brief` paragraphs, don't just restate the information that can be inferred from the API name. Put detailed discussion into separate paragraphs.

To refer to parameter names inside a paragraph, use the `\p name` command. Don't use the `\arg name` command since it starts a new paragraph that contains documentation for the parameter.

Wrap non-inline code examples in `\code ... \endcode`.

To document a function parameter, start a new paragraph with the `\param name` command. If the parameter is used as an out or an in/out parameter, use the `\param [out] name` or `\param [in,out] name` command, respectively.

To describe function return value, start a new paragraph with the `\returns` command.

A minimal documentation comment:

```
/// \brief Does foo and bar.
void fooBar(bool Baz);
```

A documentation comment that uses all Doxygen features in a preferred way:

```
/// \brief Does foo and bar.
///
/// Does not do foo the usual way if \p Baz is true.
///
/// Typical usage:
/// \code
///   fooBar(false, "quux", Res);
/// \endcode
///
/// \param Quux kind of foo to do.
/// \param [out] Result filled with bar sequence on foo success.
///
/// \returns true on success.
bool fooBar(bool Baz, StringRef Quux, std::vector<int> &Result);
```

Don't duplicate the documentation comment in the header file and in the implementation file. Put the documentation comments for public APIs into the header file. Documentation comments for private APIs can go to the implementation file. In any case, implementation files can include additional comments (not necessarily in Doxygen markup) to explain implementation details as needed.

Don't duplicate function or class name at the beginning of the comment. For humans it is obvious which function or class is being documented; automatic documentation processing tools are smart enough to bind the comment to the correct declaration.

Wrong:

```
// In Something.h:

/// Something - An abstraction for some complicated thing.
class Something {
public:
  /// fooBar - Does foo and bar.
  void fooBar();
};

// In Something.cpp:

/// fooBar - Does foo and bar.
void Something::fooBar() { ... }
```

Correct:

```
// In Something.h:

/// \brief An abstraction for some complicated thing.
class Something {
public:
  /// \brief Does foo and bar.
  void fooBar();
};

// In Something.cpp:

// Builds a B-tree in order to do foo.  See paper by...
void Something::fooBar() { ... }
```

It is not required to use additional Doxygen features, but sometimes it might be a good idea to do so.

Consider:

- adding comments to any narrow namespace containing a collection of related functions or types;
- using top-level groups to organize a collection of related functions at namespace scope where the grouping is smaller than the namespace;
- using member groups and additional comments attached to member groups to organize within a class.

For example:

```
class Something {
  /// \name Functions that do Foo.
  /// @{
  void fooBar();
  void fooBaz();
  /// @}
  ...
};
```

`#include` Style

Immediately after the header file comment (and include guards if working on a header file), the minimal list of #includes required by the file should be listed. We prefer these `#includes` to be listed in this order:

1. Main Module Header
2. Local/Private Headers
3. `llvm/...`
4. System `#includes`

and each category should be sorted lexicographically by the full path.

The Main Module Header file applies to `.cpp` files which implement an interface defined by a `.h` file. This `#include` should always be included **first** regardless of where it lives on the file system. By including a header file first in the `.cpp` files that implement the interfaces, we ensure that the header does not have any hidden dependencies which are not explicitly `#include`d in the header, but should be. It is also a form of documentation in the `.cpp` file to indicate where the interfaces it implements are defined.

Source Code Width

Write your code to fit within 80 columns of text. This helps those of us who like to print out code and look at your code in an `xterm` without resizing it.

The longer answer is that there must be some limit to the width of the code in order to reasonably allow developers to have multiple files side-by-side in windows on a modest display. If you are going to pick a width limit, it is somewhat arbitrary but you might as well pick something standard. Going with 90 columns (for example) instead of 80 columns wouldn't add any significant value and would be detrimental to printing out code. Also many other projects have standardized on 80 columns, so some people have already configured their editors for it (vs something else, like 90 columns).

This is one of many contentious issues in coding standards, but it is not up for debate.

Use Spaces Instead of Tabs

In all cases, prefer spaces to tabs in source files. People have different preferred indentation levels, and different styles of indentation that they like; this is fine. What isn't fine is that different editors/viewers expand tabs out to different tab stops. This can cause your code to look completely unreadable, and it is not worth dealing with.

As always, follow the Golden Rule above: follow the style of existing code if you are modifying and extending it. If you like four spaces of indentation, **DO NOT** do that in the middle of a chunk of code with two spaces of indentation. Also, do not reindent a whole source file: it makes for incredible diffs that are absolutely worthless.

Indent Code Consistently

Okay, in your first year of programming you were told that indentation is important. If you didn't believe and internalize this then, now is the time. Just do it. With the introduction of C++11, there are some new formatting challenges that merit some suggestions to help have consistent, maintainable, and tool-friendly formatting and indentation.

Format Lambdas Like Blocks Of Code When formatting a multi-line lambda, format it like a block of code, that's what it is. If there is only one multi-line lambda in a statement, and there are no expressions lexically after it in the statement, drop the indent to the standard two space indent for a block of code, as if it were an if-block opened by the preceding part of the statement:

```
std::sort(foo.begin(), foo.end(), [&](Foo a, Foo b) -> bool {
  if (a.blah < b.blah)
    return true;
  if (a.baz < b.baz)
    return true;
  return a.bam < b.bam;
});
```

To take best advantage of this formatting, if you are designing an API which accepts a continuation or single callable argument (be it a functor, or a `std::function`), it should be the last argument if at all possible.

If there are multiple multi-line lambdas in a statement, or there is anything interesting after the lambda in the statement, indent the block two spaces from the indent of the `[]`:

```
dyn_switch(V->stripPointerCasts(),
           [] (PHINode *PN) {
             // process phis...
           },
           [] (SelectInst *SI) {
             // process selects...
           },
           [] (LoadInst *LI) {
             // process loads...
           },
           [] (AllocaInst *AI) {
             // process allocas...
           });
```

Braced Initializer Lists With C++11, there are significantly more uses of braced lists to perform initialization. These allow you to easily construct aggregate temporaries in expressions among other niceness. They now have a natural way of ending up nested within each other and within function calls in order to build up aggregates (such as option structs) from local variables. To make matters worse, we also have many more uses of braces in an expression context that are *not* performing initialization.

The historically common formatting of braced initialization of aggregate variables does not mix cleanly with deep nesting, general expression contexts, function arguments, and lambdas. We suggest new code use a simple rule for formatting braced initialization lists: act as-if the braces were parentheses in a function call. The formatting rules exactly match those already well understood for formatting nested function calls. Examples:

```
foo({a, b, c}, {1, 2, 3});

llvm::Constant *Mask[] = {
    llvm::ConstantInt::get(llvm::Type::getInt32Ty(getLLVMContext()), 0),
    llvm::ConstantInt::get(llvm::Type::getInt32Ty(getLLVMContext()), 1),
    llvm::ConstantInt::get(llvm::Type::getInt32Ty(getLLVMContext()), 2)};
```

This formatting scheme also makes it particularly easy to get predictable, consistent, and automatic formatting with tools like Clang Format.

Language and Compiler Issues

Treat Compiler Warnings Like Errors

If your code has compiler warnings in it, something is wrong — you aren't casting values correctly, you have "questionable" constructs in your code, or you are doing something legitimately wrong. Compiler warnings can cover up legitimate errors in output and make dealing with a translation unit difficult.

It is not possible to prevent all warnings from all compilers, nor is it desirable. Instead, pick a standard compiler (like `gcc`) that provides a good thorough set of warnings, and stick to it. At least in the case of `gcc`, it is possible to work around any spurious errors by changing the syntax of the code slightly. For example, a warning that annoys me occurs when I write code like this:

```
if (V = getValue()) {
  ...
}
```

`gcc` will warn me that I probably want to use the == operator, and that I probably mistyped it. In most cases, I haven't, and I really don't want the spurious errors. To fix this particular problem, I rewrite the code like this:

```
if ((V = getValue())) {
  ...
}
```

which shuts `gcc` up. Any `gcc` warning that annoys you can be fixed by massaging the code appropriately.

Write Portable Code

In almost all cases, it is possible and within reason to write completely portable code. If there are cases where it isn't possible to write portable code, isolate it behind a well defined (and well documented) interface.

In practice, this means that you shouldn't assume much about the host compiler (and Visual Studio tends to be the lowest common denominator). If advanced features are used, they should only be an implementation detail of a library which has a simple exposed API, and preferably be buried in `libSystem`.

Do not use RTTI or Exceptions

In an effort to reduce code and executable size, LLVM does not use RTTI (e.g. `dynamic_cast<>;`) or exceptions. These two language features violate the general C++ principle of *"you only pay for what you use"*, causing executable bloat even if exceptions are never used in the code base, or if RTTI is never used for a class. Because of this, we turn them off globally in the code.

That said, LLVM does make extensive use of a hand-rolled form of RTTI that use templates like *isa<>, cast<>, and dyn_cast<>*. This form of RTTI is opt-in and can be *added to any class*. It is also substantially more efficient than `dynamic_cast<>`.

Do not use Static Constructors

Static constructors and destructors (e.g. global variables whose types have a constructor or destructor) should not be added to the code base, and should be removed wherever possible. Besides well known problems where the order of initialization is undefined between globals in different source files, the entire concept of static constructors is at odds with the common use case of LLVM as a library linked into a larger application.

Consider the use of LLVM as a JIT linked into another application (perhaps for OpenGL, custom languages, shaders in movies, etc). Due to the design of static constructors, they must be executed at startup time of the entire application, regardless of whether or how LLVM is used in that larger application. There are two problems with this:

- The time to run the static constructors impacts startup time of applications — a critical time for GUI apps, among others.
- The static constructors cause the app to pull many extra pages of memory off the disk: both the code for the constructor in each `.o` file and the small amount of data that gets touched. In addition, touched/dirty pages put more pressure on the VM system on low-memory machines.

We would really like for there to be zero cost for linking in an additional LLVM target or other library into an application, but static constructors violate this goal.

That said, LLVM unfortunately does contain static constructors. It would be a great project for someone to purge all static constructors from LLVM, and then enable the `-Wglobal-constructors` warning flag (when building with Clang) to ensure we do not regress in the future.

Use of `class` and `struct` Keywords

In C++, the `class` and `struct` keywords can be used almost interchangeably. The only difference is when they are used to declare a class: `class` makes all members private by default while `struct` makes all members public by default.

Unfortunately, not all compilers follow the rules and some will generate different symbols based on whether `class` or `struct` was used to declare the symbol (e.g., MSVC). This can lead to problems at link time.

- All declarations and definitions of a given `class` or `struct` must use the same keyword. For example:

```
class Foo;

// Breaks mangling in MSVC.
struct Foo { int Data; };
```

- As a rule of thumb, `struct` should be kept to structures where *all* members are declared public.

```
// Foo feels like a class... this is strange.
struct Foo {
private:
  int Data;
public:
  Foo() : Data(0) { }
  int getData() const { return Data; }
  void setData(int D) { Data = D; }
};

// Bar isn't POD, but it does look like a struct.
struct Bar {
  int Data;
  Foo() : Data(0) { }
};
```

Do not use Braced Initializer Lists to Call a Constructor

In C++11 there is a "generalized initialization syntax" which allows calling constructors using braced initializer lists. Do not use these to call constructors with any interesting logic or if you care that you're calling some *particular* constructor. Those should look like function calls using parentheses rather than like aggregate initialization. Similarly, if you need to explicitly name the type and call its constructor to create a temporary, don't use a braced initializer list. Instead, use a braced initializer list (without any type for temporaries) when doing aggregate initialization or something notionally equivalent. Examples:

```
class Foo {
public:
  // Construct a Foo by reading data from the disk in the whizbang format, ...
  Foo(std::string filename);

  // Construct a Foo by looking up the Nth element of some global data ...
  Foo(int N);

  // ...
};

// The Foo constructor call is very deliberate, no braces.
std::fill(foo.begin(), foo.end(), Foo("name"));

// The pair is just being constructed like an aggregate, use braces.
bar_map.insert({my_key, my_value});
```

If you use a braced initializer list when initializing a variable, use an equals before the open curly brace:

```
int data[] = {0, 1, 2, 3};
```

Use `auto` Type Deduction to Make Code More Readable

Some are advocating a policy of "almost always `auto`" in C++11, however LLVM uses a more moderate stance. Use `auto` if and only if it makes the code more readable or easier to maintain. Don't "almost always" use `auto`, but do use `auto` with initializers like `cast<Foo>(...)` or other places where the type is already obvious from the context. Another time when `auto` works well for these purposes is when the type would have been abstracted away anyways, often behind a container's typedef such as `std::vector<T>::iterator`.

Beware unnecessary copies with `auto`

The convenience of `auto` makes it easy to forget that its default behavior is a copy. Particularly in range-based `for` loops, careless copies are expensive.

As a rule of thumb, use `auto &` unless you need to copy the result, and use `auto *` when copying pointers.

```
// Typically there's no reason to copy.
for (const auto &Val : Container) { observe(Val); }
for (auto &Val : Container) { Val.change(); }

// Remove the reference if you really want a new copy.
for (auto Val : Container) { Val.change(); saveSomewhere(Val); }

// Copy pointers, but make it clear that they're pointers.
for (const auto *Ptr : Container) { observe(*Ptr); }
for (auto *Ptr : Container) { Ptr->change(); }
```

3.2.4 Style Issues

The High-Level Issues

A Public Header File is a Module

C++ doesn't do too well in the modularity department. There is no real encapsulation or data hiding (unless you use expensive protocol classes), but it is what we have to work with. When you write a public header file (in the LLVM source tree, they live in the top level "`include`" directory), you are defining a module of functionality.

Ideally, modules should be completely independent of each other, and their header files should only `#include` the absolute minimum number of headers possible. A module is not just a class, a function, or a namespace: it's a collection of these that defines an interface. This interface may be several functions, classes, or data structures, but the important issue is how they work together.

In general, a module should be implemented by one or more `.cpp` files. Each of these `.cpp` files should include the header that defines their interface first. This ensures that all of the dependences of the module header have been properly added to the module header itself, and are not implicit. System headers should be included after user headers for a translation unit.

`#include` as Little as Possible

`#include` hurts compile time performance. Don't do it unless you have to, especially in header files.

But wait! Sometimes you need to have the definition of a class to use it, or to inherit from it. In these cases go ahead and `#include` that header file. Be aware however that there are many cases where you don't need to have the full definition of a class. If you are using a pointer or reference to a class, you don't need the header file. If you are simply returning a class instance from a prototyped function or method, you don't need it. In fact, for most cases, you simply don't need the definition of a class. And not `#include`ing speeds up compilation.

It is easy to try to go too overboard on this recommendation, however. You **must** include all of the header files that you are using — you can include them either directly or indirectly through another header file. To make sure that you don't accidentally forget to include a header file in your module header, make sure to include your module header **first** in the implementation file (as mentioned above). This way there won't be any hidden dependencies that you'll find out about later.

Keep "Internal" Headers Private

Many modules have a complex implementation that causes them to use more than one implementation (`.cpp`) file. It is often tempting to put the internal communication interface (helper classes, extra functions, etc) in the public module header file. Don't do this!

If you really need to do something like this, put a private header file in the same directory as the source files, and include it locally. This ensures that your private interface remains private and undisturbed by outsiders.

Note: It's okay to put extra implementation methods in a public class itself. Just make them private (or protected) and all is well.

Use Early Exits and `continue` to Simplify Code

When reading code, keep in mind how much state and how many previous decisions have to be remembered by the reader to understand a block of code. Aim to reduce indentation where possible when it doesn't make it more difficult

to understand the code. One great way to do this is by making use of early exits and the `continue` keyword in long loops. As an example of using an early exit from a function, consider this "bad" code:

```
Value *doSomething(Instruction *I) {
  if (!isa<TerminatorInst>(I) &&
      I->hasOneUse() && doOtherThing(I)) {
    ... some long code ....
  }

  return 0;
}
```

This code has several problems if the body of the `'if'` is large. When you're looking at the top of the function, it isn't immediately clear that this *only* does interesting things with non-terminator instructions, and only applies to things with the other predicates. Second, it is relatively difficult to describe (in comments) why these predicates are important because the `if` statement makes it difficult to lay out the comments. Third, when you're deep within the body of the code, it is indented an extra level. Finally, when reading the top of the function, it isn't clear what the result is if the predicate isn't true; you have to read to the end of the function to know that it returns null.

It is much preferred to format the code like this:

```
Value *doSomething(Instruction *I) {
  // Terminators never need 'something' done to them because ...
  if (isa<TerminatorInst>(I))
    return 0;

  // We conservatively avoid transforming instructions with multiple uses
  // because goats like cheese.
  if (!I->hasOneUse())
    return 0;

  // This is really just here for example.
  if (!doOtherThing(I))
    return 0;

  ... some long code ....
}
```

This fixes these problems. A similar problem frequently happens in `for` loops. A silly example is something like this:

```
for (BasicBlock::iterator II = BB->begin(), E = BB->end(); II != E; ++II) {
  if (BinaryOperator *BO = dyn_cast<BinaryOperator>(II)) {
    Value *LHS = BO->getOperand(0);
    Value *RHS = BO->getOperand(1);
    if (LHS != RHS) {
      ...
    }
  }
}
```

When you have very, very small loops, this sort of structure is fine. But if it exceeds more than 10-15 lines, it becomes difficult for people to read and understand at a glance. The problem with this sort of code is that it gets very nested very quickly. Meaning that the reader of the code has to keep a lot of context in their brain to remember what is going immediately on in the loop, because they don't know if/when the `if` conditions will have `else`s etc. It is strongly preferred to structure the loop like this:

```
for (BasicBlock::iterator II = BB->begin(), E = BB->end(); II != E; ++II) {
  BinaryOperator *BO = dyn_cast<BinaryOperator>(II);
  if (!BO) continue;
```

```
  Value *LHS = BO->getOperand(0);
  Value *RHS = BO->getOperand(1);
  if (LHS == RHS) continue;

  ...
}
```

This has all the benefits of using early exits for functions: it reduces nesting of the loop, it makes it easier to describe why the conditions are true, and it makes it obvious to the reader that there is no `else` coming up that they have to push context into their brain for. If a loop is large, this can be a big understandability win.

Don't use `else` after a `return`

For similar reasons above (reduction of indentation and easier reading), please do not use `'else'` or `'else if'` after something that interrupts control flow — like `return`, `break`, `continue`, `goto`, etc. For example, this is *bad*:

```
case 'J': {
  if (Signed) {
    Type = Context.getsigjmp_bufType();
    if (Type.isNull()) {
      Error = ASTContext::GE_Missing_sigjmp_buf;
      return QualType();
    } else {
      break;
    }
  } else {
    Type = Context.getjmp_bufType();
    if (Type.isNull()) {
      Error = ASTContext::GE_Missing_jmp_buf;
      return QualType();
    } else {
      break;
    }
  }
}
```

It is better to write it like this:

```
case 'J':
  if (Signed) {
    Type = Context.getsigjmp_bufType();
    if (Type.isNull()) {
      Error = ASTContext::GE_Missing_sigjmp_buf;
      return QualType();
    }
  } else {
    Type = Context.getjmp_bufType();
    if (Type.isNull()) {
      Error = ASTContext::GE_Missing_jmp_buf;
      return QualType();
    }
  }
  break;
```

Or better yet (in this case) as:

```
case 'J':
  if (Signed)
    Type = Context.getsigjmp_bufType();
  else
    Type = Context.getjmp_bufType();

  if (Type.isNull()) {
    Error = Signed ? ASTContext::GE_Missing_sigjmp_buf :
                     ASTContext::GE_Missing_jmp_buf;
    return QualType();
  }
  break;
```

The idea is to reduce indentation and the amount of code you have to keep track of when reading the code.

Turn Predicate Loops into Predicate Functions

It is very common to write small loops that just compute a boolean value. There are a number of ways that people commonly write these, but an example of this sort of thing is:

```
bool FoundFoo = false;
for (unsigned I = 0, E = BarList.size(); I != E; ++I)
  if (BarList[I]->isFoo()) {
    FoundFoo = true;
    break;
  }

if (FoundFoo) {
  ...
}
```

This sort of code is awkward to write, and is almost always a bad sign. Instead of this sort of loop, we strongly prefer to use a predicate function (which may be static) that uses early exits to compute the predicate. We prefer the code to be structured like this:

```
/// \returns true if the specified list has an element that is a foo.
static bool containsFoo(const std::vector<Bar*> &List) {
  for (unsigned I = 0, E = List.size(); I != E; ++I)
    if (List[I]->isFoo())
      return true;
  return false;
}
...

if (containsFoo(BarList)) {
  ...
}
```

There are many reasons for doing this: it reduces indentation and factors out code which can often be shared by other code that checks for the same predicate. More importantly, it *forces you to pick a name* for the function, and forces you to write a comment for it. In this silly example, this doesn't add much value. However, if the condition is complex, this can make it a lot easier for the reader to understand the code that queries for this predicate. Instead of being faced with the in-line details of how we check to see if the BarList contains a foo, we can trust the function name and continue reading with better locality.

The Low-Level Issues

Name Types, Functions, Variables, and Enumerators Properly

Poorly-chosen names can mislead the reader and cause bugs. We cannot stress enough how important it is to use *descriptive* names. Pick names that match the semantics and role of the underlying entities, within reason. Avoid abbreviations unless they are well known. After picking a good name, make sure to use consistent capitalization for the name, as inconsistency requires clients to either memorize the APIs or to look it up to find the exact spelling.

In general, names should be in camel case (e.g. `TextFileReader` and `isLValue()`). Different kinds of declarations have different rules:

- **Type names** (including classes, structs, enums, typedefs, etc) should be nouns and start with an upper-case letter (e.g. `TextFileReader`).
- **Variable names** should be nouns (as they represent state). The name should be camel case, and start with an upper case letter (e.g. `Leader` or `Boats`).
- **Function names** should be verb phrases (as they represent actions), and command-like function should be imperative. The name should be camel case, and start with a lower case letter (e.g. `openFile()` or `isFoo()`).
- **Enum declarations** (e.g. `enum Foo {...}`) are types, so they should follow the naming conventions for types. A common use for enums is as a discriminator for a union, or an indicator of a subclass. When an enum is used for something like this, it should have a `Kind` suffix (e.g. `ValueKind`).
- **Enumerators** (e.g. `enum { Foo, Bar }`) and **public member variables** should start with an upper-case letter, just like types. Unless the enumerators are defined in their own small namespace or inside a class, enumerators should have a prefix corresponding to the enum declaration name. For example, `enum ValueKind { ... };` may contain enumerators like `VK_Argument`, `VK_BasicBlock`, etc. Enumerators that are just convenience constants are exempt from the requirement for a prefix. For instance:

  ```
  enum {
    MaxSize = 42,
    Density = 12
  };
  ```

As an exception, classes that mimic STL classes can have member names in STL's style of lower-case words separated by underscores (e.g. `begin()`, `push_back()`, and `empty()`). Classes that provide multiple iterators should add a singular prefix to `begin()` and `end()` (e.g. `global_begin()` and `use_begin()`).

Here are some examples of good and bad names:

```
class VehicleMaker {
  ...
  Factory<Tire> F;            // Bad -- abbreviation and non-descriptive.
  Factory<Tire> Factory;      // Better.
  Factory<Tire> TireFactory;  // Even better -- if VehicleMaker has more than one
                              // kind of factories.
};

Vehicle MakeVehicle(VehicleType Type) {
  VehicleMaker M;                         // Might be OK if having a short life-span.
  Tire Tmp1 = M.makeTire();               // Bad -- 'Tmp1' provides no information.
  Light Headlight = M.makeLight("head");  // Good -- descriptive.
  ...
}
```

Assert Liberally

Use the "`assert`" macro to its fullest. Check all of your preconditions and assumptions, you never know when a bug (not necessarily even yours) might be caught early by an assertion, which reduces debugging time dramatically. The "`<cassert>`" header file is probably already included by the header files you are using, so it doesn't cost anything to use it.

To further assist with debugging, make sure to put some kind of error message in the assertion statement, which is printed if the assertion is tripped. This helps the poor debugger make sense of why an assertion is being made and enforced, and hopefully what to do about it. Here is one complete example:

```
inline Value *getOperand(unsigned I) {
  assert(I < Operands.size() && "getOperand() out of range!");
  return Operands[I];
}
```

Here are more examples:

```
assert(Ty->isPointerType() && "Can't allocate a non-pointer type!");

assert((Opcode == Shl || Opcode == Shr) && "ShiftInst Opcode invalid!");

assert(idx < getNumSuccessors() && "Successor # out of range!");

assert(V1.getType() == V2.getType() && "Constant types must be identical!");

assert(isa<PHINode>(Succ->front()) && "Only works on PHId BBs!");
```

You get the idea.

In the past, asserts were used to indicate a piece of code that should not be reached. These were typically of the form:

```
assert(0 && "Invalid radix for integer literal");
```

This has a few issues, the main one being that some compilers might not understand the assertion, or warn about a missing return in builds where assertions are compiled out.

Today, we have something much better: `llvm_unreachable`:

```
llvm_unreachable("Invalid radix for integer literal");
```

When assertions are enabled, this will print the message if it's ever reached and then exit the program. When assertions are disabled (i.e. in release builds), `llvm_unreachable` becomes a hint to compilers to skip generating code for this branch. If the compiler does not support this, it will fall back to the "abort" implementation.

Another issue is that values used only by assertions will produce an "unused value" warning when assertions are disabled. For example, this code will warn:

```
unsigned Size = V.size();
assert(Size > 42 && "Vector smaller than it should be");

bool NewToSet = Myset.insert(Value);
assert(NewToSet && "The value shouldn't be in the set yet");
```

These are two interesting different cases. In the first case, the call to `V.size()` is only useful for the assert, and we don't want it executed when assertions are disabled. Code like this should move the call into the assert itself. In the second case, the side effects of the call must happen whether the assert is enabled or not. In this case, the value should be cast to void to disable the warning. To be specific, it is preferred to write the code like this:

```
assert(V.size() > 42 && "Vector smaller than it should be");

bool NewToSet = Myset.insert(Value); (void)NewToSet;
assert(NewToSet && "The value shouldn't be in the set yet");
```

Do Not Use `using namespace std`

In LLVM, we prefer to explicitly prefix all identifiers from the standard namespace with an "`std::`" prefix, rather than rely on "`using namespace std;`".

In header files, adding a '`using namespace XXX`' directive pollutes the namespace of any source file that `#include`s the header. This is clearly a bad thing.

In implementation files (e.g. `.cpp` files), the rule is more of a stylistic rule, but is still important. Basically, using explicit namespace prefixes makes the code **clearer**, because it is immediately obvious what facilities are being used and where they are coming from. And **more portable**, because namespace clashes cannot occur between LLVM code and other namespaces. The portability rule is important because different standard library implementations expose different symbols (potentially ones they shouldn't), and future revisions to the C++ standard will add more symbols to the `std` namespace. As such, we never use '`using namespace std;`' in LLVM.

The exception to the general rule (i.e. it's not an exception for the `std` namespace) is for implementation files. For example, all of the code in the LLVM project implements code that lives in the 'llvm' namespace. As such, it is ok, and actually clearer, for the `.cpp` files to have a '`using namespace llvm;`' directive at the top, after the `#include`s. This reduces indentation in the body of the file for source editors that indent based on braces, and keeps the conceptual context cleaner. The general form of this rule is that any `.cpp` file that implements code in any namespace may use that namespace (and its parents'), but should not use any others.

Provide a Virtual Method Anchor for Classes in Headers

If a class is defined in a header file and has a vtable (either it has virtual methods or it derives from classes with virtual methods), it must always have at least one out-of-line virtual method in the class. Without this, the compiler will copy the vtable and RTTI into every `.o` file that `#include`s the header, bloating `.o` file sizes and increasing link times.

Don't use default labels in fully covered switches over enumerations

`-Wswitch` warns if a switch, without a default label, over an enumeration does not cover every enumeration value. If you write a default label on a fully covered switch over an enumeration then the `-Wswitch` warning won't fire when new elements are added to that enumeration. To help avoid adding these kinds of defaults, Clang has the warning `-Wcovered-switch-default` which is off by default but turned on when building LLVM with a version of Clang that supports the warning.

A knock-on effect of this stylistic requirement is that when building LLVM with GCC you may get warnings related to "control may reach end of non-void function" if you return from each case of a covered switch-over-enum because GCC assumes that the enum expression may take any representable value, not just those of individual enumerators. To suppress this warning, use `llvm_unreachable` after the switch.

Use `LLVM_DELETED_FUNCTION` to mark uncallable methods

Prior to C++11, a common pattern to make a class uncopyable was to declare an unimplemented copy constructor and copy assignment operator and make them private. This would give a compiler error for accessing a private method or a linker error because it wasn't implemented.

With C++11, we can mark methods that won't be implemented with `= delete`. This will trigger a much better error message and tell the compiler that the method will never be implemented. This enables other checks like `-Wunused-private-field` to run correctly on classes that contain these methods.

For compatibility with MSVC, `LLVM_DELETED_FUNCTION` should be used which will expand to `= delete` on compilers that support it. These methods should still be declared private. Example of the uncopyable pattern:

```
class DontCopy {
private:
  DontCopy(const DontCopy&) LLVM_DELETED_FUNCTION;
  DontCopy &operator =(const DontCopy&) LLVM_DELETED_FUNCTION;
public:
  ...
};
```

Don't evaluate `end()` every time through a loop

Because C++ doesn't have a standard "`foreach`" loop (though it can be emulated with macros and may be coming in C++'0x) we end up writing a lot of loops that manually iterate from begin to end on a variety of containers or through other data structures. One common mistake is to write a loop in this style:

```
BasicBlock *BB = ...
for (BasicBlock::iterator I = BB->begin(); I != BB->end(); ++I)
  ... use I ...
```

The problem with this construct is that it evaluates "`BB->end()`" every time through the loop. Instead of writing the loop like this, we strongly prefer loops to be written so that they evaluate it once before the loop starts. A convenient way to do this is like so:

```
BasicBlock *BB = ...
for (BasicBlock::iterator I = BB->begin(), E = BB->end(); I != E; ++I)
  ... use I ...
```

The observant may quickly point out that these two loops may have different semantics: if the container (a basic block in this case) is being mutated, then "`BB->end()`" may change its value every time through the loop and the second loop may not in fact be correct. If you actually do depend on this behavior, please write the loop in the first form and add a comment indicating that you did it intentionally.

Why do we prefer the second form (when correct)? Writing the loop in the first form has two problems. First it may be less efficient than evaluating it at the start of the loop. In this case, the cost is probably minor — a few extra loads every time through the loop. However, if the base expression is more complex, then the cost can rise quickly. I've seen loops where the end expression was actually something like: "`SomeMap[X]->end()`" and map lookups really aren't cheap. By writing it in the second form consistently, you eliminate the issue entirely and don't even have to think about it.

The second (even bigger) issue is that writing the loop in the first form hints to the reader that the loop is mutating the container (a fact that a comment would handily confirm!). If you write the loop in the second form, it is immediately obvious without even looking at the body of the loop that the container isn't being modified, which makes it easier to read the code and understand what it does.

While the second form of the loop is a few extra keystrokes, we do strongly prefer it.

`#include <iostream>` is Forbidden

The use of `#include <iostream>` in library files is hereby **forbidden**, because many common implementations transparently inject a static constructor into every translation unit that includes it.

Note that using the other stream headers (<sstream> for example) is not problematic in this regard — just <iostream>. However, raw_ostream provides various APIs that are better performing for almost every use than std::ostream style APIs.

Note: New code should always use raw_ostream for writing, or the llvm::MemoryBuffer API for reading files.

Use raw_ostream

LLVM includes a lightweight, simple, and efficient stream implementation in llvm/Support/raw_ostream.h, which provides all of the common features of std::ostream. All new code should use raw_ostream instead of ostream.

Unlike std::ostream, raw_ostream is not a template and can be forward declared as class raw_ostream. Public headers should generally not include the raw_ostream header, but use forward declarations and constant references to raw_ostream instances.

Avoid std::endl

The std::endl modifier, when used with iostreams outputs a newline to the output stream specified. In addition to doing this, however, it also flushes the output stream. In other words, these are equivalent:

```
std::cout << std::endl;
std::cout << '\n' << std::flush;
```

Most of the time, you probably have no reason to flush the output stream, so it's better to use a literal '\n'.

Don't use inline when defining a function in a class definition

A member function defined in a class definition is implicitly inline, so don't put the inline keyword in this case.

Don't:

```
class Foo {
public:
  inline void bar() {
    // ...
  }
};
```

Do:

```
class Foo {
public:
  void bar() {
    // ...
  }
};
```

Microscopic Details

This section describes preferred low-level formatting guidelines along with reasoning on why we prefer them.

Spaces Before Parentheses

We prefer to put a space before an open parenthesis only in control flow statements, but not in normal function call expressions and function-like macros. For example, this is good:

```
if (X) ...
for (I = 0; I != 100; ++I) ...
while (LLVMRocks) ...

somefunc(42);
assert(3 != 4 && "laws of math are failing me");

A = foo(42, 92) + bar(X);
```

and this is bad:

```
if(X) ...
for(I = 0; I != 100; ++I) ...
while(LLVMRocks) ...

somefunc (42);
assert (3 != 4 && "laws of math are failing me");

A = foo (42, 92) + bar (X);
```

The reason for doing this is not completely arbitrary. This style makes control flow operators stand out more, and makes expressions flow better. The function call operator binds very tightly as a postfix operator. Putting a space after a function name (as in the last example) makes it appear that the code might bind the arguments of the left-hand-side of a binary operator with the argument list of a function and the name of the right side. More specifically, it is easy to misread the "A" example as:

```
A = foo ((42, 92) + bar) (X);
```

when skimming through the code. By avoiding a space in a function, we avoid this misinterpretation.

Prefer Preincrement

Hard fast rule: Preincrement (++X) may be no slower than postincrement (X++) and could very well be a lot faster than it. Use preincrementation whenever possible.

The semantics of postincrement include making a copy of the value being incremented, returning it, and then preincrementing the "work value". For primitive types, this isn't a big deal. But for iterators, it can be a huge issue (for example, some iterators contains stack and set objects in them... copying an iterator could invoke the copy ctor's of these as well). In general, get in the habit of always using preincrement, and you won't have a problem.

Namespace Indentation

In general, we strive to reduce indentation wherever possible. This is useful because we want code to fit into 80 columns without wrapping horribly, but also because it makes it easier to understand the code. To facilitate this and avoid some insanely deep nesting on occasion, don't indent namespaces. If it helps readability, feel free to add a comment indicating what namespace is being closed by a }. For example:

```
namespace llvm {
namespace knowledge {

/// This class represents things that Smith can have an intimate
```

```
/// understanding of and contains the data associated with it.
class Grokable {
...
public:
  explicit Grokable() { ... }
  virtual ~Grokable() = 0;

  ...

};

} // end namespace knowledge
} // end namespace llvm
```

Feel free to skip the closing comment when the namespace being closed is obvious for any reason. For example, the outer-most namespace in a header file is rarely a source of confusion. But namespaces both anonymous and named in source files that are being closed half way through the file probably could use clarification.

Anonymous Namespaces

After talking about namespaces in general, you may be wondering about anonymous namespaces in particular. Anonymous namespaces are a great language feature that tells the C++ compiler that the contents of the namespace are only visible within the current translation unit, allowing more aggressive optimization and eliminating the possibility of symbol name collisions. Anonymous namespaces are to C++ as "static" is to C functions and global variables. While "`static`" is available in C++, anonymous namespaces are more general: they can make entire classes private to a file.

The problem with anonymous namespaces is that they naturally want to encourage indentation of their body, and they reduce locality of reference: if you see a random function definition in a C++ file, it is easy to see if it is marked static, but seeing if it is in an anonymous namespace requires scanning a big chunk of the file.

Because of this, we have a simple guideline: make anonymous namespaces as small as possible, and only use them for class declarations. For example, this is good:

```
namespace {
class StringSort {
...
public:
  StringSort(...)
  bool operator<(const char *RHS) const;
};
} // end anonymous namespace

static void runHelper() {
  ...
}

bool StringSort::operator<(const char *RHS) const {
  ...
}
```

This is bad:

```
namespace {

class StringSort {
...
```

```
public:
  StringSort(...)
  bool operator<(const char *RHS) const;
};

void runHelper() {
  ...
}

bool StringSort::operator<(const char *RHS) const {
  ...
}

} // end anonymous namespace
```

This is bad specifically because if you're looking at "`runHelper`" in the middle of a large C++ file, that you have no immediate way to tell if it is local to the file. When it is marked static explicitly, this is immediately obvious. Also, there is no reason to enclose the definition of "`operator<`" in the namespace just because it was declared there.

3.2.5 See Also

A lot of these comments and recommendations have been culled from other sources. Two particularly important books for our work are:

1. Effective C++ by Scott Meyers. Also interesting and useful are "More Effective C++" and "Effective STL" by the same author.
2. Large-Scale C++ Software Design by John Lakos

If you get some free time, and you haven't read them: do so, you might learn something.

3.3 CommandLine 2.0 Library Manual

3.3.1 Introduction

This document describes the CommandLine argument processing library. It will show you how to use it, and what it can do. The CommandLine library uses a declarative approach to specifying the command line options that your program takes. By default, these options declarations implicitly hold the value parsed for the option declared (of course this can be changed).

Although there are a **lot** of command line argument parsing libraries out there in many different languages, none of them fit well with what I needed. By looking at the features and problems of other libraries, I designed the CommandLine library to have the following features:

1. Speed: The CommandLine library is very quick and uses little resources. The parsing time of the library is directly proportional to the number of arguments parsed, not the number of options recognized. Additionally,

command line argument values are captured transparently into user defined global variables, which can be accessed like any other variable (and with the same performance).

2. Type Safe: As a user of CommandLine, you don't have to worry about remembering the type of arguments that you want (is it an int? a string? a bool? an enum?) and keep casting it around. Not only does this help prevent error prone constructs, it also leads to dramatically cleaner source code.
3. No subclasses required: To use CommandLine, you instantiate variables that correspond to the arguments that you would like to capture, you don't subclass a parser. This means that you don't have to write **any** boilerplate code.
4. Globally accessible: Libraries can specify command line arguments that are automatically enabled in any tool that links to the library. This is possible because the application doesn't have to keep a list of arguments to pass to the parser. This also makes supporting dynamically loaded options trivial.
5. Cleaner: CommandLine supports enum and other types directly, meaning that there is less error and more security built into the library. You don't have to worry about whether your integral command line argument accidentally got assigned a value that is not valid for your enum type.
6. Powerful: The CommandLine library supports many different types of arguments, from simple boolean flags to scalars arguments (strings, integers, enums, doubles), to lists of arguments. This is possible because CommandLine is...
7. Extensible: It is very simple to add a new argument type to CommandLine. Simply specify the parser that you want to use with the command line option when you declare it. Custom parsers are no problem.
8. Labor Saving: The CommandLine library cuts down on the amount of grunt work that you, the user, have to do. For example, it automatically provides a `-help` option that shows the available command line options for your tool. Additionally, it does most of the basic correctness checking for you.
9. Capable: The CommandLine library can handle lots of different forms of options often found in real programs. For example, positional arguments, `ls` style grouping options (to allow processing '`ls -lad`' naturally), `ld` style prefix options (to parse '`-lmalloc -L/usr/lib`'), and interpreter style options.

This document will hopefully let you jump in and start using CommandLine in your utility quickly and painlessly. Additionally it should be a simple reference manual to figure out how stuff works.

3.3.2 Quick Start Guide

This section of the manual runs through a simple CommandLine'ification of a basic compiler tool. This is intended to show you how to jump into using the CommandLine library in your own program, and show you some of the cool things it can do.

To start out, you need to include the CommandLine header file into your program:

```
#include "llvm/Support/CommandLine.h"
```

Additionally, you need to add this as the first line of your main program:

```
int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv);
  ...
}
```

... which actually parses the arguments and fills in the variable declarations.

Now that you are ready to support command line arguments, we need to tell the system which ones we want, and what type of arguments they are. The CommandLine library uses a declarative syntax to model command line arguments with the global variable declarations that capture the parsed values. This means that for every command line option that you would like to support, there should be a global variable declaration to capture the result. For example, in a

compiler, we would like to support the Unix-standard '`-o <filename>`' option to specify where to put the output. With the CommandLine library, this is represented like this:

```
cl::opt<string> OutputFilename("o", cl::desc("Specify output filename"), cl::value_desc("filename"));
```

This declares a global variable "`OutputFilename`" that is used to capture the result of the "`o`" argument (first parameter). We specify that this is a simple scalar option by using the "`cl::opt`" template (as opposed to the "`cl::list`" template), and tell the CommandLine library that the data type that we are parsing is a string.

The second and third parameters (which are optional) are used to specify what to output for the "`-help`" option. In this case, we get a line that looks like this:

```
USAGE: compiler [options]

OPTIONS:
  -help              - display available options (-help-hidden for more)
  -o <filename>      - Specify output filename
```

Because we specified that the command line option should parse using the `string` data type, the variable declared is automatically usable as a real string in all contexts that a normal C++ string object may be used. For example:

```
...
std::ofstream Output(OutputFilename.c_str());
if (Output.good()) ...
...
```

There are many different options that you can use to customize the command line option handling library, but the above example shows the general interface to these options. The options can be specified in any order, and are specified with helper functions like cl::desc(...), so there are no positional dependencies to remember. The available options are discussed in detail in the Reference Guide.

Continuing the example, we would like to have our compiler take an input filename as well as an output filename, but we do not want the input filename to be specified with a hyphen (ie, not `-filename.c`). To support this style of argument, the CommandLine library allows for positional arguments to be specified for the program. These positional arguments are filled with command line parameters that are not in option form. We use this feature like this:

```
cl::opt<string> InputFilename(cl::Positional, cl::desc("<input file>"), cl::init("-"));
```

This declaration indicates that the first positional argument should be treated as the input filename. Here we use the cl::init option to specify an initial value for the command line option, which is used if the option is not specified (if you do not specify a cl::init modifier for an option, then the default constructor for the data type is used to initialize the value). Command line options default to being optional, so if we would like to require that the user always specify an input filename, we would add the cl::Required flag, and we could eliminate the cl::init modifier, like this:

```
cl::opt<string> InputFilename(cl::Positional, cl::desc("<input file>"), cl::Required);
```

Again, the CommandLine library does not require the options to be specified in any particular order, so the above declaration is equivalent to:

```
cl::opt<string> InputFilename(cl::Positional, cl::Required, cl::desc("<input file>"));
```

By simply adding the cl::Required flag, the CommandLine library will automatically issue an error if the argument is not specified, which shifts all of the command line option verification code out of your application into the library. This is just one example of how using flags can alter the default behaviour of the library, on a per-option basis. By adding one of the declarations above, the `-help` option synopsis is now extended to:

```
USAGE: compiler [options] <input file>

OPTIONS:
```

```
  -help              - display available options (-help-hidden for more)
  -o <filename>      - Specify output filename
```

... indicating that an input filename is expected.

Boolean Arguments

In addition to input and output filenames, we would like the compiler example to support three boolean flags: "`-f`" to force writing binary output to a terminal, "`--quiet`" to enable quiet mode, and "`-q`" for backwards compatibility with some of our users. We can support these by declaring options of boolean type like this:

```
cl::opt<bool> Force ("f", cl::desc("Enable binary output on terminals"));
cl::opt<bool> Quiet ("quiet", cl::desc("Don't print informational messages"));
cl::opt<bool> Quiet2("q", cl::desc("Don't print informational messages"), cl::Hidden);
```

This does what you would expect: it declares three boolean variables ("`Force`", "`Quiet`", and "`Quiet2`") to recognize these options. Note that the "`-q`" option is specified with the "cl::Hidden" flag. This modifier prevents it from being shown by the standard "`-help`" output (note that it is still shown in the "`-help-hidden`" output).

The CommandLine library uses a different parser for different data types. For example, in the string case, the argument passed to the option is copied literally into the content of the string variable... we obviously cannot do that in the boolean case, however, so we must use a smarter parser. In the case of the boolean parser, it allows no options (in which case it assigns the value of true to the variable), or it allows the values "`true`" or "`false`" to be specified, allowing any of the following inputs:

```
compiler -f          # No value, 'Force' == true
compiler -f=true     # Value specified, 'Force' == true
compiler -f=TRUE     # Value specified, 'Force' == true
compiler -f=FALSE    # Value specified, 'Force' == false
```

... you get the idea. The bool parser just turns the string values into boolean values, and rejects things like '`compiler -f=foo`'. Similarly, the float, double, and int parsers work like you would expect, using the '`strtol`' and '`strtod`' C library calls to parse the string value into the specified data type.

With the declarations above, "`compiler -help`" emits this:

```
USAGE: compiler [options] <input file>

OPTIONS:
  -f     - Enable binary output on terminals
  -o     - Override output filename
  -quiet - Don't print informational messages
  -help  - display available options (-help-hidden for more)
```

and "`compiler -help-hidden`" prints this:

```
USAGE: compiler [options] <input file>

OPTIONS:
  -f     - Enable binary output on terminals
  -o     - Override output filename
  -q     - Don't print informational messages
  -quiet - Don't print informational messages
  -help  - display available options (-help-hidden for more)
```

This brief example has shown you how to use the 'cl::opt' class to parse simple scalar command line arguments. In addition to simple scalar arguments, the CommandLine library also provides primitives to support CommandLine option aliases, and lists of options.

Argument Aliases

So far, the example works well, except for the fact that we need to check the quiet condition like this now:

```
...
  if (!Quiet && !Quiet2) printInformationalMessage(...);
...
```

... which is a real pain! Instead of defining two values for the same condition, we can use the "cl::alias" class to make the "-q" option an **alias** for the "-quiet" option, instead of providing a value itself:

```
cl::opt<bool> Force ("f", cl::desc("Overwrite output files"));
cl::opt<bool> Quiet ("quiet", cl::desc("Don't print informational messages"));
cl::alias     QuietA("q", cl::desc("Alias for -quiet"), cl::aliasopt(Quiet));
```

The third line (which is the only one we modified from above) defines a "-q" alias that updates the "Quiet" variable (as specified by the cl::aliasopt modifier) whenever it is specified. Because aliases do not hold state, the only thing the program has to query is the Quiet variable now. Another nice feature of aliases is that they automatically hide themselves from the -help output (although, again, they are still visible in the -help-hidden output).

Now the application code can simply use:

```
...
  if (!Quiet) printInformationalMessage(...);
...
```

... which is much nicer! The "cl::alias" can be used to specify an alternative name for any variable type, and has many uses.

Selecting an alternative from a set of possibilities

So far we have seen how the CommandLine library handles builtin types like std::string, bool and int, but how does it handle things it doesn't know about, like enums or 'int*'s?

The answer is that it uses a table-driven generic parser (unless you specify your own parser, as described in the Extension Guide). This parser maps literal strings to whatever type is required, and requires you to tell it what this mapping should be.

Let's say that we would like to add four optimization levels to our optimizer, using the standard flags "-g", "-O0", "-O1", and "-O2". We could easily implement this with boolean options like above, but there are several problems with this strategy:

1. A user could specify more than one of the options at a time, for example, "compiler -O3 -O2". The CommandLine library would not be able to catch this erroneous input for us.
2. We would have to test 4 different variables to see which ones are set.
3. This doesn't map to the numeric levels that we want... so we cannot easily see if some level >= "-O1" is enabled.

To cope with these problems, we can use an enum value, and have the CommandLine library fill it in with the appropriate level directly, which is used like this:

```
enum OptLevel {
  g, O1, O2, O3
};

cl::opt<OptLevel> OptimizationLevel(cl::desc("Choose optimization level:"),
  cl::values(
    clEnumVal(g , "No optimizations, enable debugging"),
    clEnumVal(O1, "Enable trivial optimizations"),
```

```
    clEnumVal(O2, "Enable default optimizations"),
    clEnumVal(O3, "Enable expensive optimizations"),
   clEnumValEnd));

...
  if (OptimizationLevel >= O2) doPartialRedundancyElimination(...);
...
```

This declaration defines a variable "`OptimizationLevel`" of the "`OptLevel`" enum type. This variable can be assigned any of the values that are listed in the declaration (Note that the declaration list must be terminated with the "`clEnumValEnd`" argument!). The CommandLine library enforces that the user can only specify one of the options, and it ensure that only valid enum values can be specified. The "`clEnumVal`" macros ensure that the command line arguments matched the enum values. With this option added, our help output now is:

```
USAGE: compiler [options] <input file>

OPTIONS:
  Choose optimization level:
    -g          - No optimizations, enable debugging
    -O1         - Enable trivial optimizations
    -O2         - Enable default optimizations
    -O3         - Enable expensive optimizations
  -f            - Enable binary output on terminals
  -help         - display available options (-help-hidden for more)
  -o <filename> - Specify output filename
  -quiet        - Don't print informational messages
```

In this case, it is sort of awkward that flag names correspond directly to enum names, because we probably don't want a enum definition named "`g`" in our program. Because of this, we can alternatively write this example like this:

```
enum OptLevel {
  Debug, O1, O2, O3
};

cl::opt<OptLevel> OptimizationLevel(cl::desc("Choose optimization level:"),
  cl::values(
   clEnumValN(Debug, "g", "No optimizations, enable debugging"),
    clEnumVal(O1        , "Enable trivial optimizations"),
    clEnumVal(O2        , "Enable default optimizations"),
    clEnumVal(O3        , "Enable expensive optimizations"),
   clEnumValEnd));

...
  if (OptimizationLevel == Debug) outputDebugInfo(...);
...
```

By using the "`clEnumValN`" macro instead of "`clEnumVal`", we can directly specify the name that the flag should get. In general a direct mapping is nice, but sometimes you can't or don't want to preserve the mapping, which is when you would use it.

Named Alternatives

Another useful argument form is a named alternative style. We shall use this style in our compiler to specify different debug levels that can be used. Instead of each debug level being its own switch, we want to support the following options, of which only one can be specified at a time: "`--debug-level=none`", "`--debug-level=quick`", "`--debug-level=detailed`". To do this, we use the exact same format as our optimization level flags, but we also specify an option name. For this case, the code looks like this:

```
enum DebugLev {
  nodebuginfo, quick, detailed
};

// Enable Debug Options to be specified on the command line
cl::opt<DebugLev> DebugLevel("debug_level", cl::desc("Set the debugging level:"),
  cl::values(
    clEnumValN(nodebuginfo, "none", "disable debug information"),
     clEnumVal(quick,               "enable quick debug information"),
     clEnumVal(detailed,            "enable detailed debug information"),
    clEnumValEnd));
```

This definition defines an enumerated command line variable of type "`enum DebugLev`", which works exactly the same way as before. The difference here is just the interface exposed to the user of your program and the help output by the "`-help`" option:

```
USAGE: compiler [options] <input file>

OPTIONS:
  Choose optimization level:
    -g          - No optimizations, enable debugging
    -O1         - Enable trivial optimizations
    -O2         - Enable default optimizations
    -O3         - Enable expensive optimizations
  -debug_level  - Set the debugging level:
    =none       - disable debug information
    =quick      - enable quick debug information
    =detailed   - enable detailed debug information
  -f            - Enable binary output on terminals
  -help         - display available options (-help-hidden for more)
  -o <filename> - Specify output filename
  -quiet        - Don't print informational messages
```

Again, the only structural difference between the debug level declaration and the optimization level declaration is that the debug level declaration includes an option name (`"debug_level"`), which automatically changes how the library processes the argument. The CommandLine library supports both forms so that you can choose the form most appropriate for your application.

Parsing a list of options

Now that we have the standard run-of-the-mill argument types out of the way, lets get a little wild and crazy. Lets say that we want our optimizer to accept a **list** of optimizations to perform, allowing duplicates. For example, we might want to run: "`compiler -dce -constprop -inline -dce -strip`". In this case, the order of the arguments and the number of appearances is very important. This is what the "`cl::list`" template is for. First, start by defining an enum of the optimizations that you would like to perform:

```
enum Opts {
  // 'inline' is a C++ keyword, so name it 'inlining'
  dce, constprop, inlining, strip
};
```

Then define your "`cl::list`" variable:

```
cl::list<Opts> OptimizationList(cl::desc("Available Optimizations:"),
  cl::values(
    clEnumVal(dce               , "Dead Code Elimination"),
    clEnumVal(constprop         , "Constant Propagation"),
```

```
   clEnumValN(inlining, "inline", "Procedure Integration"),
    clEnumVal(strip             , "Strip Symbols"),
  clEnumValEnd));
```

This defines a variable that is conceptually of the type "`std::vector<enum Opts>`". Thus, you can access it with standard vector methods:

```
for (unsigned i = 0; i != OptimizationList.size(); ++i)
  switch (OptimizationList[i])
     ...
```

... to iterate through the list of options specified.

Note that the "`cl::list`" template is completely general and may be used with any data types or other arguments that you can use with the "`cl::opt`" template. One especially useful way to use a list is to capture all of the positional arguments together if there may be more than one specified. In the case of a linker, for example, the linker takes several '`.o`' files, and needs to capture them into a list. This is naturally specified as:

```
...
cl::list<std::string> InputFilenames(cl::Positional, cl::desc("<Input files>"), cl::OneOrMore);
...
```

This variable works just like a "`vector<string>`" object. As such, accessing the list is simple, just like above. In this example, we used the cl::OneOrMore modifier to inform the CommandLine library that it is an error if the user does not specify any `.o` files on our command line. Again, this just reduces the amount of checking we have to do.

Collecting options as a set of flags

Instead of collecting sets of options in a list, it is also possible to gather information for enum values in a **bit vector**. The representation used by the cl::bits class is an `unsigned` integer. An enum value is represented by a 0/1 in the enum's ordinal value bit position. 1 indicating that the enum was specified, 0 otherwise. As each specified value is parsed, the resulting enum's bit is set in the option's bit vector:

```
bits |= 1 << (unsigned)enum;
```

Options that are specified multiple times are redundant. Any instances after the first are discarded.

Reworking the above list example, we could replace cl::list with cl::bits:

```
cl::bits<Opts> OptimizationBits(cl::desc("Available Optimizations:"),
  cl::values(
    clEnumVal(dce               , "Dead Code Elimination"),
    clEnumVal(constprop         , "Constant Propagation"),
   clEnumValN(inlining, "inline", "Procedure Integration"),
    clEnumVal(strip             , "Strip Symbols"),
  clEnumValEnd));
```

To test to see if `constprop` was specified, we can use the `cl:bits::isSet` function:

```
if (OptimizationBits.isSet(constprop)) {
  ...
}
```

It's also possible to get the raw bit vector using the `cl::bits::getBits` function:

```
unsigned bits = OptimizationBits.getBits();
```

Finally, if external storage is used, then the location specified must be of **type** `unsigned`. In all other ways a cl::bits option is equivalent to a cl::list option.

Adding freeform text to help output

As our program grows and becomes more mature, we may decide to put summary information about what it does into the help output. The help output is styled to look similar to a Unix `man` page, providing concise information about a program. Unix `man` pages, however often have a description about what the program does. To add this to your CommandLine program, simply pass a third argument to the cl::ParseCommandLineOptions call in main. This additional argument is then printed as the overview information for your program, allowing you to include any additional information that you want. For example:

```
int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv, " CommandLine compiler example\n\n"
                              "  This program blah blah blah...\n");
  ...
}
```

would yield the help output:

```
**OVERVIEW: CommandLine compiler example

  This program blah blah blah...**

USAGE: compiler [options] <input file>

OPTIONS:
  ...
  -help              - display available options (-help-hidden for more)
  -o <filename>      - Specify output filename
```

Grouping options into categories

If our program has a large number of options it may become difficult for users of our tool to navigate the output of `-help`. To alleviate this problem we can put our options into categories. This can be done by declaring option categories (cl::OptionCategory objects) and then placing our options into these categories using the cl::cat option attribute. For example:

```
cl::OptionCategory StageSelectionCat("Stage Selection Options",
                                     "These control which stages are run.");

cl::opt<bool> Preprocessor("E",cl::desc("Run preprocessor stage."),
                           cl::cat(StageSelectionCat));

cl::opt<bool> NoLink("c",cl::desc("Run all stages except linking."),
                     cl::cat(StageSelectionCat));
```

The output of `-help` will become categorized if an option category is declared. The output looks something like

```
OVERVIEW: This is a small program to demo the LLVM CommandLine API
USAGE: Sample [options]

OPTIONS:

  General options:

    -help              - Display available options (-help-hidden for more)
    -help-list         - Display list of available options (-help-list-hidden for more)
```

```
  Stage Selection Options:
  These control which stages are run.

    -E                - Run preprocessor stage.
    -c                - Run all stages except linking.
```

In addition to the behaviour of `-help` changing when an option category is declared, the command line option `-help-list` becomes visible which will print the command line options as uncategorized list.

Note that Options that are not explicitly categorized will be placed in the `cl::GeneralCategory` category.

3.3.3 Reference Guide

Now that you know the basics of how to use the CommandLine library, this section will give you the detailed information you need to tune how command line options work, as well as information on more "advanced" command line option processing capabilities.

Positional Arguments

Positional arguments are those arguments that are not named, and are not specified with a hyphen. Positional arguments should be used when an option is specified by its position alone. For example, the standard Unix `grep` tool takes a regular expression argument, and an optional filename to search through (which defaults to standard input if a filename is not specified). Using the CommandLine library, this would be specified as:

```
cl::opt<string> Regex   (cl::Positional, cl::desc("<regular expression>"), cl::Required);
cl::opt<string> Filename(cl::Positional, cl::desc("<input file>"), cl::init("-"));
```

Given these two option declarations, the `-help` output for our grep replacement would look like this:

```
USAGE: spiffygrep [options] <regular expression> <input file>

OPTIONS:
  -help - display available options (-help-hidden for more)
```

... and the resultant program could be used just like the standard `grep` tool.

Positional arguments are sorted by their order of construction. This means that command line options will be ordered according to how they are listed in a .cpp file, but will not have an ordering defined if the positional arguments are defined in multiple .cpp files. The fix for this problem is simply to define all of your positional arguments in one .cpp file.

Specifying positional options with hyphens

Sometimes you may want to specify a value to your positional argument that starts with a hyphen (for example, searching for '`-foo`' in a file). At first, you will have trouble doing this, because it will try to find an argument named '`-foo`', and will fail (and single quotes will not save you). Note that the system `grep` has the same problem:

```
$ spiffygrep '-foo' test.txt
Unknown command line argument '-foo'.  Try: spiffygrep -help'

$ grep '-foo' test.txt
grep: illegal option -- f
grep: illegal option -- o
grep: illegal option -- o
Usage: grep -hblcnsviw pattern file . . .
```

The solution for this problem is the same for both your tool and the system version: use the '--' marker. When the user specifies '--' on the command line, it is telling the program that all options after the '--' should be treated as positional arguments, not options. Thus, we can use it like this:

```
$ spiffygrep -- -foo test.txt
  ...output...
```

Determining absolute position with getPosition()

Sometimes an option can affect or modify the meaning of another option. For example, consider `gcc`'s `-x LANG` option. This tells `gcc` to ignore the suffix of subsequent positional arguments and force the file to be interpreted as if it contained source code in language `LANG`. In order to handle this properly, you need to know the absolute position of each argument, especially those in lists, so their interaction(s) can be applied correctly. This is also useful for options like `-llibname` which is actually a positional argument that starts with a dash.

So, generally, the problem is that you have two `cl::list` variables that interact in some way. To ensure the correct interaction, you can use the `cl::list::getPosition(optnum)` method. This method returns the absolute position (as found on the command line) of the `optnum` item in the `cl::list`.

The idiom for usage is like this:

```
static cl::list<std::string> Files(cl::Positional, cl::OneOrMore);
static cl::list<std::string> Libraries("l", cl::ZeroOrMore);

int main(int argc, char**argv) {
  // ...
  std::vector<std::string>::iterator fileIt = Files.begin();
  std::vector<std::string>::iterator libIt  = Libraries.begin();
  unsigned libPos = 0, filePos = 0;
  while ( 1 ) {
    if ( libIt != Libraries.end() )
      libPos = Libraries.getPosition( libIt - Libraries.begin() );
    else
      libPos = 0;
    if ( fileIt != Files.end() )
      filePos = Files.getPosition( fileIt - Files.begin() );
    else
      filePos = 0;

    if ( filePos != 0 && (libPos == 0 || filePos < libPos) ) {
      // Source File Is next
      ++fileIt;
    }
    else if ( libPos != 0 && (filePos == 0 || libPos < filePos) ) {
      // Library is next
      ++libIt;
    }
    else
      break; // we're done with the list
  }
}
```

Note that, for compatibility reasons, the `cl::opt` also supports an `unsigned getPosition()` option that will provide the absolute position of that option. You can apply the same approach as above with a `cl::opt` and a `cl::list` option as you can with two lists.

The `cl::ConsumeAfter` modifier

The `cl::ConsumeAfter` formatting option is used to construct programs that use "interpreter style" option processing. With this style of option processing, all arguments specified after the last positional argument are treated as special interpreter arguments that are not interpreted by the command line argument.

As a concrete example, lets say we are developing a replacement for the standard Unix Bourne shell (`/bin/sh`). To run `/bin/sh`, first you specify options to the shell itself (like `-x` which turns on trace output), then you specify the name of the script to run, then you specify arguments to the script. These arguments to the script are parsed by the Bourne shell command line option processor, but are not interpreted as options to the shell itself. Using the CommandLine library, we would specify this as:

```
cl::opt<string> Script(cl::Positional, cl::desc("<input script>"), cl::init("-"));
cl::list<string>  Argv(cl::ConsumeAfter, cl::desc("<program arguments>..."));
cl::opt<bool>    Trace("x", cl::desc("Enable trace output"));
```

which automatically provides the help output:

```
USAGE: spiffysh [options] <input script> <program arguments>...

OPTIONS:
  -help - display available options (-help-hidden for more)
  -x    - Enable trace output
```

At runtime, if we run our new shell replacement as '`spiffysh -x test.sh -a -x -y bar`', the `Trace` variable will be set to true, the `Script` variable will be set to "`test.sh`", and the `Argv` list will contain `["-a", "-x", "-y", "bar"]`, because they were specified after the last positional argument (which is the script name).

There are several limitations to when `cl::ConsumeAfter` options can be specified. For example, only one `cl::ConsumeAfter` can be specified per program, there must be at least one positional argument specified, there must not be any cl::list positional arguments, and the `cl::ConsumeAfter` option should be a cl::list option.

Internal vs External Storage

By default, all command line options automatically hold the value that they parse from the command line. This is very convenient in the common case, especially when combined with the ability to define command line options in the files that use them. This is called the internal storage model.

Sometimes, however, it is nice to separate the command line option processing code from the storage of the value parsed. For example, lets say that we have a '`-debug`' option that we would like to use to enable debug information across the entire body of our program. In this case, the boolean value controlling the debug code should be globally accessible (in a header file, for example) yet the command line option processing code should not be exposed to all of these clients (requiring lots of .cpp files to `#include CommandLine.h`).

To do this, set up your .h file with your option, like this for example:

```
// DebugFlag.h - Get access to the '-debug' command line option
//

// DebugFlag - This boolean is set to true if the '-debug' command line option
// is specified.  This should probably not be referenced directly, instead, use
// the DEBUG macro below.
//
extern bool DebugFlag;

// DEBUG macro - This macro should be used by code to emit debug information.
// In the '-debug' option is specified on the command line, and if this is a
// debug build, then the code specified as the option to the macro will be
```

```
// executed.  Otherwise it will not be.
#ifdef NDEBUG
#define DEBUG(X)
#else
#define DEBUG(X) do { if (DebugFlag) { X; } } while (0)
#endif
```

This allows clients to blissfully use the `DEBUG()` macro, or the `DebugFlag` explicitly if they want to. Now we just need to be able to set the `DebugFlag` boolean when the option is set. To do this, we pass an additional argument to our command line argument processor, and we specify where to fill in with the cl::location attribute:

```
bool DebugFlag;                   // the actual value
static cl::opt<bool, true>        // The parser
Debug("debug", cl::desc("Enable debug output"), cl::Hidden, cl::location(DebugFlag));
```

In the above example, we specify "`true`" as the second argument to the cl::opt template, indicating that the template should not maintain a copy of the value itself. In addition to this, we specify the cl::location attribute, so that `DebugFlag` is automatically set.

Option Attributes

This section describes the basic attributes that you can specify on options.

- The option name attribute (which is required for all options, except positional options) specifies what the option name is. This option is specified in simple double quotes:

  ```
  cl::opt<bool> Quiet("quiet");
  ```

- The **cl::desc** attribute specifies a description for the option to be shown in the `-help` output for the program. This attribute supports multi-line descriptions with lines separated by 'n'.

- The **cl::value_desc** attribute specifies a string that can be used to fine tune the `-help` output for a command line option. Look here for an example.

- The **cl::init** attribute specifies an initial value for a scalar option. If this attribute is not specified then the command line option value defaults to the value created by the default constructor for the type.

  > **Warning:** If you specify both **cl::init** and **cl::location** for an option, you must specify **cl::location** first, so that when the command-line parser sees **cl::init**, it knows where to put the initial value. (You will get an error at runtime if you don't put them in the right order.)

- The **cl::location** attribute where to store the value for a parsed command line option if using external storage. See the section on Internal vs External Storage for more information.

- The **cl::aliasopt** attribute specifies which option a cl::alias option is an alias for.

- The **cl::values** attribute specifies the string-to-value mapping to be used by the generic parser. It takes a **clEnumValEnd terminated** list of (option, value, description) triplets that specify the option name, the value mapped to, and the description shown in the `-help` for the tool. Because the generic parser is used most frequently with enum values, two macros are often useful:

  1. The **clEnumVal** macro is used as a nice simple way to specify a triplet for an enum. This macro automatically makes the option name be the same as the enum name. The first option to the macro is the enum, the second is the description for the command line option.

  2. The **clEnumValN** macro is used to specify macro options where the option name doesn't equal the enum name. For this macro, the first argument is the enum value, the second is the flag name, and the second is the description.

You will get a compile time error if you try to use cl::values with a parser that does not support it.

- The **cl::multi_val** attribute specifies that this option takes has multiple values (example: `-sectalign segname sectname sectvalue`). This attribute takes one unsigned argument - the number of values for the option. This attribute is valid only on `cl::list` options (and will fail with compile error if you try to use it with other option types). It is allowed to use all of the usual modifiers on multi-valued options (besides `cl::ValueDisallowed`, obviously).
- The **cl::cat** attribute specifies the option category that the option belongs to. The category should be a cl::OptionCategory object.

Option Modifiers

Option modifiers are the flags and expressions that you pass into the constructors for cl::opt and cl::list. These modifiers give you the ability to tweak how options are parsed and how `-help` output is generated to fit your application well.

These options fall into five main categories:

1. Hiding an option from `-help` output
2. Controlling the number of occurrences required and allowed
3. Controlling whether or not a value must be specified
4. Controlling other formatting options
5. Miscellaneous option modifiers

It is not possible to specify two options from the same category (you'll get a runtime error) to a single option, except for options in the miscellaneous category. The CommandLine library specifies defaults for all of these settings that are the most useful in practice and the most common, which mean that you usually shouldn't have to worry about these.

Hiding an option from `-help` output

The `cl::NotHidden`, `cl::Hidden`, and `cl::ReallyHidden` modifiers are used to control whether or not an option appears in the `-help` and `-help-hidden` output for the compiled program:

- The **cl::NotHidden** modifier (which is the default for cl::opt and cl::list options) indicates the option is to appear in both help listings.
- The **cl::Hidden** modifier (which is the default for cl::alias options) indicates that the option should not appear in the `-help` output, but should appear in the `-help-hidden` output.
- The **cl::ReallyHidden** modifier indicates that the option should not appear in any help output.

Controlling the number of occurrences required and allowed

This group of options is used to control how many time an option is allowed (or required) to be specified on the command line of your program. Specifying a value for this setting allows the CommandLine library to do error checking for you.

The allowed values for this option group are:

- The **cl::Optional** modifier (which is the default for the cl::opt and cl::alias classes) indicates that your program will allow either zero or one occurrence of the option to be specified.
- The **cl::ZeroOrMore** modifier (which is the default for the cl::list class) indicates that your program will allow the option to be specified zero or more times.

- The **cl::Required** modifier indicates that the specified option must be specified exactly one time.
- The **cl::OneOrMore** modifier indicates that the option must be specified at least one time.
- The **cl::ConsumeAfter** modifier is described in the Positional arguments section.

If an option is not specified, then the value of the option is equal to the value specified by the cl::init attribute. If the `cl::init` attribute is not specified, the option value is initialized with the default constructor for the data type.

If an option is specified multiple times for an option of the cl::opt class, only the last value will be retained.

Controlling whether or not a value must be specified

This group of options is used to control whether or not the option allows a value to be present. In the case of the CommandLine library, a value is either specified with an equal sign (e.g. '`-index-depth=17`') or as a trailing string (e.g. '`-o a.out`').

The allowed values for this option group are:

- The **cl::ValueOptional** modifier (which is the default for `bool` typed options) specifies that it is acceptable to have a value, or not. A boolean argument can be enabled just by appearing on the command line, or it can have an explicit '`-foo=true`'. If an option is specified with this mode, it is illegal for the value to be provided without the equal sign. Therefore '`-foo true`' is illegal. To get this behavior, you must use the cl::ValueRequired modifier.
- The **cl::ValueRequired** modifier (which is the default for all other types except for unnamed alternatives using the generic parser) specifies that a value must be provided. This mode informs the command line library that if an option is not provides with an equal sign, that the next argument provided must be the value. This allows things like '`-o a.out`' to work.
- The **cl::ValueDisallowed** modifier (which is the default for unnamed alternatives using the generic parser) indicates that it is a runtime error for the user to specify a value. This can be provided to disallow users from providing options to boolean options (like '`-foo=true`').

In general, the default values for this option group work just like you would want them to. As mentioned above, you can specify the cl::ValueDisallowed modifier to a boolean argument to restrict your command line parser. These options are mostly useful when extending the library.

Controlling other formatting options

The formatting option group is used to specify that the command line option has special abilities and is otherwise different from other command line arguments. As usual, you can only specify one of these arguments at most.

- The **cl::NormalFormatting** modifier (which is the default all options) specifies that this option is "normal".
- The **cl::Positional** modifier specifies that this is a positional argument that does not have a command line option associated with it. See the Positional Arguments section for more information.
- The **cl::ConsumeAfter** modifier specifies that this option is used to capture "interpreter style" arguments. See this section for more information.
- The **cl::Prefix** modifier specifies that this option prefixes its value. With 'Prefix' options, the equal sign does not separate the value from the option name specified. Instead, the value is everything after the prefix, including any equal sign if present. This is useful for processing odd arguments like `-lmalloc` and `-L/usr/lib` in a linker tool or `-DNAME=value` in a compiler tool. Here, the '`l`', '`D`' and '`L`' options are normal string (or list) options, that have the **cl::Prefix** modifier added to allow the CommandLine library to recognize them. Note that **cl::Prefix** options must not have the **cl::ValueDisallowed** modifier specified.

- The **cl::Grouping** modifier is used to implement Unix-style tools (like `ls`) that have lots of single letter arguments, but only require a single dash. For example, the '`ls -labF`' command actually enables four different options, all of which are single letters. Note that **cl::Grouping** options cannot have values.

The CommandLine library does not restrict how you use the **cl::Prefix** or **cl::Grouping** modifiers, but it is possible to specify ambiguous argument settings. Thus, it is possible to have multiple letter options that are prefix or grouping options, and they will still work as designed.

To do this, the CommandLine library uses a greedy algorithm to parse the input option into (potentially multiple) prefix and grouping options. The strategy basically looks like this:

```
parse(string OrigInput) {

1. string input = OrigInput;
2. if (isOption(input)) return getOption(input).parse();  // Normal option
3. while (!isOption(input) && !input.empty()) input.pop_back();  // Remove the last letter
4. if (input.empty()) return error();  // No matching option
5. if (getOption(input).isPrefix())
     return getOption(input).parse(input);
6. while (!input.empty()) {  // Must be grouping options
     getOption(input).parse();
     OrigInput.erase(OrigInput.begin(), OrigInput.begin()+input.length());
     input = OrigInput;
     while (!isOption(input) && !input.empty()) input.pop_back();
   }
7. if (!OrigInput.empty()) error();

}
```

Miscellaneous option modifiers

The miscellaneous option modifiers are the only flags where you can specify more than one flag from the set: they are not mutually exclusive. These flags specify boolean properties that modify the option.

- The **cl::CommaSeparated** modifier indicates that any commas specified for an option's value should be used to split the value up into multiple values for the option. For example, these two options are equivalent when `cl::CommaSeparated` is specified: "`-foo=a -foo=b -foo=c`" and "`-foo=a,b,c`". This option only makes sense to be used in a case where the option is allowed to accept one or more values (i.e. it is a cl::list option).
- The **cl::PositionalEatsArgs** modifier (which only applies to positional arguments, and only makes sense for lists) indicates that positional argument should consume any strings after it (including strings that start with a "-") up until another recognized positional argument. For example, if you have two "eating" positional arguments, "`pos1`" and "`pos2`", the string "`-pos1 -foo -bar baz -pos2 -bork`" would cause the "`-foo -bar -baz`" strings to be applied to the "`-pos1`" option and the "`-bork`" string to be applied to the "`-pos2`" option.
- The **cl::Sink** modifier is used to handle unknown options. If there is at least one option with `cl::Sink` modifier specified, the parser passes unrecognized option strings to it as values instead of signaling an error. As with `cl::CommaSeparated`, this modifier only makes sense with a cl::list option.

So far, these are the only three miscellaneous option modifiers.

Response files

Some systems, such as certain variants of Microsoft Windows and some older Unices have a relatively low limit on command-line length. It is therefore customary to use the so-called 'response files' to circumvent this restriction.

These files are mentioned on the command-line (using the "@file") syntax. The program reads these files and inserts the contents into argv, thereby working around the command-line length limits. Response files are enabled by an optional fourth argument to cl::ParseEnvironmentOptions and cl::ParseCommandLineOptions.

Top-Level Classes and Functions

Despite all of the built-in flexibility, the CommandLine option library really only consists of one function cl::ParseCommandLineOptions) and three main classes: cl::opt, cl::list, and cl::alias. This section describes these three classes in detail.

The `cl::getRegisteredOptions` function

The `cl::getRegisteredOptions` function is designed to give a programmer access to declared non-positional command line options so that how they appear in `-help` can be modified prior to calling cl::ParseCommandLineOptions. Note this method should not be called during any static initialisation because it cannot be guaranteed that all options will have been initialised. Hence it should be called from `main`.

This function can be used to gain access to options declared in libraries that the tool writter may not have direct access to.

The function retrieves a *StringMap* that maps the option string (e.g. `-help`) to an `Option*`.

Here is an example of how the function could be used:

```
using namespace llvm;
int main(int argc, char **argv) {
  cl::OptionCategory AnotherCategory("Some options");

  StringMap<cl::Option*> Map;
  cl::getRegisteredOptions(Map);

  //Unhide useful option and put it in a different category
  assert(Map.count("print-all-options") > 0);
  Map["print-all-options"]->setHiddenFlag(cl::NotHidden);
  Map["print-all-options"]->setCategory(AnotherCategory);

  //Hide an option we don't want to see
  assert(Map.count("enable-no-infs-fp-math") > 0);
  Map["enable-no-infs-fp-math"]->setHiddenFlag(cl::Hidden);

  //Change --version to --show-version
  assert(Map.count("version") > 0);
  Map["version"]->setArgStr("show-version");

  //Change --help description
  assert(Map.count("help") > 0);
  Map["help"]->setDescription("Shows help");

  cl::ParseCommandLineOptions(argc, argv, "This is a small program to demo the LLVM CommandLine API")
  ...
}
```

The `cl::ParseCommandLineOptions` function

The `cl::ParseCommandLineOptions` function is designed to be called directly from `main`, and is used to fill in the values of all of the command line option variables once `argc` and `argv` are available.

The `cl::ParseCommandLineOptions` function requires two parameters (`argc` and `argv`), but may also take an optional third parameter which holds additional extra text to emit when the `-help` option is invoked, and a fourth boolean parameter that enables response files.

The `cl::ParseEnvironmentOptions` function

The `cl::ParseEnvironmentOptions` function has mostly the same effects as cl::ParseCommandLineOptions, except that it is designed to take values for options from an environment variable, for those cases in which reading the command line is not convenient or desired. It fills in the values of all the command line option variables just like cl::ParseCommandLineOptions does.

It takes four parameters: the name of the program (since `argv` may not be available, it can't just look in `argv[0]`), the name of the environment variable to examine, the optional additional extra text to emit when the `-help` option is invoked, and the boolean switch that controls whether response files should be read.

`cl::ParseEnvironmentOptions` will break the environment variable's value up into words and then process them using cl::ParseCommandLineOptions. **Note:** Currently `cl::ParseEnvironmentOptions` does not support quoting, so an environment variable containing `-option "foo bar"` will be parsed as three words, `-option`, `"foo`, and `bar"`, which is different from what you would get from the shell with the same input.

The `cl::SetVersionPrinter` function

The `cl::SetVersionPrinter` function is designed to be called directly from `main` and *before* `cl::ParseCommandLineOptions`. Its use is optional. It simply arranges for a function to be called in response to the `--version` option instead of having the `CommandLine` library print out the usual version string for LLVM. This is useful for programs that are not part of LLVM but wish to use the `CommandLine` facilities. Such programs should just define a small function that takes no arguments and returns `void` and that prints out whatever version information is appropriate for the program. Pass the address of that function to `cl::SetVersionPrinter` to arrange for it to be called when the `--version` option is given by the user.

The `cl::opt` class

The `cl::opt` class is the class used to represent scalar command line options, and is the one used most of the time. It is a templated class which can take up to three arguments (all except for the first have default values though):

```
namespace cl {
  template <class DataType, bool ExternalStorage = false,
            class ParserClass = parser<DataType> >
  class opt;
}
```

The first template argument specifies what underlying data type the command line argument is, and is used to select a default parser implementation. The second template argument is used to specify whether the option should contain the storage for the option (the default) or whether external storage should be used to contain the value parsed for the option (see Internal vs External Storage for more information).

The third template argument specifies which parser to use. The default value selects an instantiation of the `parser` class based on the underlying data type of the option. In general, this default works well for most applications, so this option is only used when using a custom parser.

The `cl::list` class

The `cl::list` class is the class used to represent a list of command line options. It too is a templated class which can take up to three arguments:

```
namespace cl {
  template <class DataType, class Storage = bool,
            class ParserClass = parser<DataType> >
  class list;
}
```

This class works the exact same as the cl::opt class, except that the second argument is the **type** of the external storage, not a boolean value. For this class, the marker type '`bool`' is used to indicate that internal storage should be used.

The `cl::bits` class

The `cl::bits` class is the class used to represent a list of command line options in the form of a bit vector. It is also a templated class which can take up to three arguments:

```
namespace cl {
  template <class DataType, class Storage = bool,
            class ParserClass = parser<DataType> >
  class bits;
}
```

This class works the exact same as the cl::list class, except that the second argument must be of **type** `unsigned` if external storage is used.

The `cl::alias` class

The `cl::alias` class is a nontemplated class that is used to form aliases for other arguments.

```
namespace cl {
  class alias;
}
```

The cl::aliasopt attribute should be used to specify which option this is an alias for. Alias arguments default to being cl::Hidden, and use the aliased options parser to do the conversion from string to data.

The `cl::extrahelp` class

The `cl::extrahelp` class is a nontemplated class that allows extra help text to be printed out for the `-help` option.

```
namespace cl {
  struct extrahelp;
}
```

To use the extrahelp, simply construct one with a `const char*` parameter to the constructor. The text passed to the constructor will be printed at the bottom of the help message, verbatim. Note that multiple `cl::extrahelp` **can** be used, but this practice is discouraged. If your tool needs to print additional help information, put all that help into a single `cl::extrahelp` instance.

For example:

```
cl::extrahelp("\nADDITIONAL HELP:\n\n  This is the extra help\n");
```

The `cl::OptionCategory` class

The `cl::OptionCategory` class is a simple class for declaring option categories.

```
namespace cl {
  class OptionCategory;
}
```

An option category must have a name and optionally a description which are passed to the constructor as `const char*`.

Note that declaring an option category and associating it with an option before parsing options (e.g. statically) will change the output of `-help` from uncategorized to categorized. If an option category is declared but not associated with an option then it will be hidden from the output of `-help` but will be shown in the output of `-help-hidden`.

Builtin parsers

Parsers control how the string value taken from the command line is translated into a typed value, suitable for use in a C++ program. By default, the CommandLine library uses an instance of `parser<type>` if the command line option specifies that it uses values of type '`type`'. Because of this, custom option processing is specified with specializations of the '`parser`' class.

The CommandLine library provides the following builtin parser specializations, which are sufficient for most applications. It can, however, also be extended to work with new data types and new ways of interpreting the same data. See the Writing a Custom Parser for more details on this type of library extension.

- The generic `parser<t>` parser can be used to map strings values to any data type, through the use of the cl::values property, which specifies the mapping information. The most common use of this parser is for parsing enum values, which allows you to use the CommandLine library for all of the error checking to make sure that only valid enum values are specified (as opposed to accepting arbitrary strings). Despite this, however, the generic parser class can be used for any data type.
- The **parser<bool> specialization** is used to convert boolean strings to a boolean value. Currently accepted strings are "`true`", "`TRUE`", "`True`", "`1`", "`false`", "`FALSE`", "`False`", and "`0`".
- The **parser<boolOrDefault> specialization** is used for cases where the value is boolean, but we also need to know whether the option was specified at all. boolOrDefault is an enum with 3 values, BOU_UNSET, BOU_TRUE and BOU_FALSE. This parser accepts the same strings as **"parser<bool>"**.
- The **parser<string> specialization** simply stores the parsed string into the string value specified. No conversion or modification of the data is performed.
- The **parser<int> specialization** uses the C `strtol` function to parse the string input. As such, it will accept a decimal number (with an optional '+' or '-' prefix) which must start with a non-zero digit. It accepts octal numbers, which are identified with a '`0`' prefix digit, and hexadecimal numbers with a prefix of '`0x`' or '`0X`'.
- The **parser<double>** and **parser<float> specializations** use the standard C `strtod` function to convert floating point strings into floating point values. As such, a broad range of string formats is supported, including exponential notation (ex: `1.7e15`) and properly supports locales.

3.3.4 Extension Guide

Although the CommandLine library has a lot of functionality built into it already (as discussed previously), one of its true strengths lie in its extensibility. This section discusses how the CommandLine library works under the covers and

illustrates how to do some simple, common, extensions.

Writing a custom parser

One of the simplest and most common extensions is the use of a custom parser. As discussed previously, parsers are the portion of the CommandLine library that turns string input from the user into a particular parsed data type, validating the input in the process.

There are two ways to use a new parser:

1. Specialize the cl::parser template for your custom data type.

   This approach has the advantage that users of your custom data type will automatically use your custom parser whenever they define an option with a value type of your data type. The disadvantage of this approach is that it doesn't work if your fundamental data type is something that is already supported.

2. Write an independent class, using it explicitly from options that need it.

   This approach works well in situations where you would line to parse an option using special syntax for a not-very-special data-type. The drawback of this approach is that users of your parser have to be aware that they are using your parser instead of the builtin ones.

To guide the discussion, we will discuss a custom parser that accepts file sizes, specified with an optional unit after the numeric size. For example, we would like to parse "102kb", "41M", "1G" into the appropriate integer value. In this case, the underlying data type we want to parse into is 'unsigned'. We choose approach #2 above because we don't want to make this the default for all unsigned options.

To start out, we declare our new FileSizeParser class:

```
struct FileSizeParser : public cl::parser<unsigned> {
  // parse - Return true on error.
  bool parse(cl::Option &O, StringRef ArgName, const std::string &ArgValue,
             unsigned &Val);
};
```

Our new class inherits from the cl::parser template class to fill in the default, boiler plate code for us. We give it the data type that we parse into, the last argument to the parse method, so that clients of our custom parser know what object type to pass in to the parse method. (Here we declare that we parse into 'unsigned' variables.)

For most purposes, the only method that must be implemented in a custom parser is the parse method. The parse method is called whenever the option is invoked, passing in the option itself, the option name, the string to parse, and a reference to a return value. If the string to parse is not well-formed, the parser should output an error message and return true. Otherwise it should return false and set 'Val' to the parsed value. In our example, we implement parse as:

```
bool FileSizeParser::parse(cl::Option &O, StringRef ArgName,
                           const std::string &Arg, unsigned &Val) {
  const char *ArgStart = Arg.c_str();
  char *End;

  // Parse integer part, leaving 'End' pointing to the first non-integer char
  Val = (unsigned)strtol(ArgStart, &End, 0);

  while (1) {
    switch (*End++) {
    case 0: return false;   // No error
    case 'i':               // Ignore the 'i' in KiB if people use that
    case 'b': case 'B':     // Ignore B suffix
      break;
```

```
    case 'g': case 'G': Val *= 1024*1024*1024; break;
    case 'm': case 'M': Val *= 1024*1024;      break;
    case 'k': case 'K': Val *= 1024;           break;

    default:
      // Print an error message if unrecognized character!
      return O.error("'" + Arg + "' value invalid for file size argument!");
    }
  }
}
```

This function implements a very simple parser for the kinds of strings we are interested in. Although it has some holes (it allows "123KKK" for example), it is good enough for this example. Note that we use the option itself to print out the error message (the `error` method always returns true) in order to get a nice error message (shown below). Now that we have our parser class, we can use it like this:

```
static cl::opt<unsigned, false, FileSizeParser>
MFS("max-file-size", cl::desc("Maximum file size to accept"),
    cl::value_desc("size"));
```

Which adds this to the output of our program:

```
OPTIONS:
  -help                - display available options (-help-hidden for more)
  ...
  -max-file-size=<size> - Maximum file size to accept
```

And we can test that our parse works correctly now (the test program just prints out the max-file-size argument value):

```
$ ./test
MFS: 0
$ ./test -max-file-size=123MB
MFS: 128974848
$ ./test -max-file-size=3G
MFS: 3221225472
$ ./test -max-file-size=dog
-max-file-size option: 'dog' value invalid for file size argument!
```

It looks like it works. The error message that we get is nice and helpful, and we seem to accept reasonable file sizes. This wraps up the "custom parser" tutorial.

Exploiting external storage

Several of the LLVM libraries define static `cl::opt` instances that will automatically be included in any program that links with that library. This is a feature. However, sometimes it is necessary to know the value of the command line option outside of the library. In these cases the library does or should provide an external storage location that is accessible to users of the library. Examples of this include the `llvm::DebugFlag` exported by the `lib/Support/Debug.cpp` file and the `llvm::TimePassesIsEnabled` flag exported by the `lib/VMCore/PassManager.cpp` file. Dynamically adding command line options

3.4 Architecture & Platform Information for Compiler Writers

Note: This document is a work-in-progress. Additions and clarifications are welcome.

3.4.1 Hardware

ARM

- ARM documentation (Processor Cores Cores)
- ABI
- ABI Addenda and Errata
- ARM C Language Extensions

AArch64

- ARMv8 Instruction Set Overview
- ARM C Language Extensions

Itanium (ia64)

- Itanium documentation

MIPS

- MIPS Processor Architecture

PowerPC

IBM - Official manuals and docs

- Power Instruction Set Architecture, Versions 2.03 through 2.06 (authentication required, free sign-up)
- PowerPC Compiler Writer's Guide
- Intro to PowerPC Architecture
- PowerPC Processor Manuals (embedded)
- Various IBM specifications and white papers
- IBM AIX/5L for POWER Assembly Reference

Other documents, collections, notes

- PowerPC ABI documents
- PowerPC64 alignment of long doubles (from GCC)
- Long branch stubs for powerpc64-linux (from binutils)

R600

- AMD R6xx shader ISA
- AMD R7xx shader ISA
- AMD Evergreen shader ISA
- AMD Cayman/Trinity shader ISA
- AMD Southern Islands Series ISA
- AMD Sea Islands Series ISA
- AMD GPU Programming Guide
- AMD Compute Resources

SPARC

- SPARC standards
- SPARC V9 ABI
- SPARC V8 ABI

SystemZ

- z/Architecture Principles of Operation (registration required, free sign-up)

X86

AMD - Official manuals and docs

- AMD processor manuals
- X86-64 ABI

Intel - Official manuals and docs

- Intel 64 and IA-32 manuals
- Intel Itanium documentation

Other x86-specific information

- Calling conventions for different C++ compilers and operating systems

XCore

- The XMOS XS1 Architecture (ISA)
- Tools Development Guide (includes ABI)

Other relevant lists

- GCC reading list

3.4.2 ABI

- System V Application Binary Interface
- Itanium C++ ABI

Linux

- PowerPC 64-bit ELF ABI Supplement
- Procedure Call Standard for the AArch64 Architecture
- ELF for the ARM Architecture
- ELF for the ARM 64-bit Architecture (AArch64)
- System z ELF ABI Supplement

OS X

- Mach-O Runtime Architecture
- Notes on Mach-O ABI

Windows

- Microsoft PE/COFF Specification

3.4.3 NVPTX

- CUDA Documentation includes the PTX ISA and Driver API documentation

3.4.4 Miscellaneous Resources

- Executable File Format library
- GCC prefetch project page has a good survey of the prefetching capabilities of a variety of modern processors.

3.5 Extending LLVM: Adding instructions, intrinsics, types, etc.

3.5.1 Introduction and Warning

During the course of using LLVM, you may wish to customize it for your research project or for experimentation. At this point, you may realize that you need to add something to LLVM, whether it be a new fundamental type, a new intrinsic function, or a whole new instruction.

When you come to this realization, stop and think. Do you really need to extend LLVM? Is it a new fundamental capability that LLVM does not support at its current incarnation or can it be synthesized from already pre-existing LLVM elements? If you are not sure, ask on the LLVM-dev list. The reason is that extending LLVM will get involved as you need to update all the different passes that you intend to use with your extension, and there are `many` LLVM analyses and transformations, so it may be quite a bit of work.

Adding an intrinsic function is far easier than adding an instruction, and is transparent to optimization passes. If your added functionality can be expressed as a function call, an intrinsic function is the method of choice for LLVM extension.

Before you invest a significant amount of effort into a non-trivial extension, **ask on the list** if what you are looking to do can be done with already-existing infrastructure, or if maybe someone else is already working on it. You will save yourself a lot of time and effort by doing so.

3.5.2 Adding a new intrinsic function

Adding a new intrinsic function to LLVM is much easier than adding a new instruction. Almost all extensions to LLVM should start as an intrinsic function and then be turned into an instruction if warranted.

1. `llvm/docs/LangRef.html`:

   Document the intrinsic. Decide whether it is code generator specific and what the restrictions are. Talk to other people about it so that you are sure it's a good idea.

2. `llvm/include/llvm/IR/Intrinsics*.td`:

   Add an entry for your intrinsic. Describe its memory access characteristics for optimization (this controls whether it will be DCE'd, CSE'd, etc). Note that any intrinsic using the `llvm_int_ty` type for an argument will be deemed by `tblgen` as overloaded and the corresponding suffix will be required on the intrinsic's name.

3. `llvm/lib/Analysis/ConstantFolding.cpp`:

   If it is possible to constant fold your intrinsic, add support to it in the `canConstantFoldCallTo` and `ConstantFoldCall` functions.

4. `llvm/test/Regression/*`:

   Add test cases for your test cases to the test suite

Once the intrinsic has been added to the system, you must add code generator support for it. Generally you must do the following steps:

Add support to the .td file for the target(s) of your choice in `lib/Target/*/*.td`.

> This is usually a matter of adding a pattern to the .td file that matches the intrinsic, though it may obviously require adding the instructions you want to generate as well. There are lots of examples in the PowerPC and X86 backend to follow.

3.5.3 Adding a new SelectionDAG node

As with intrinsics, adding a new SelectionDAG node to LLVM is much easier than adding a new instruction. New nodes are often added to help represent instructions common to many targets. These nodes often map to an LLVM instruction (add, sub) or intrinsic (byteswap, population count). In other cases, new nodes have been added to allow many targets to perform a common task (converting between floating point and integer representation) or capture more complicated behavior in a single node (rotate).

1. `include/llvm/CodeGen/ISDOpcodes.h`:

   Add an enum value for the new SelectionDAG node.

2. `lib/CodeGen/SelectionDAG/SelectionDAG.cpp`:

   Add code to print the node to `getOperationName`. If your new node can be evaluated at compile time when given constant arguments (such as an add of a constant with another constant), find the `getNode` method that takes the appropriate number of arguments, and add a case for your node to the switch statement that performs constant folding for nodes that take the same number of arguments as your new node.

3. `lib/CodeGen/SelectionDAG/LegalizeDAG.cpp`:

   Add code to legalize, promote, and expand the node as necessary. At a minimum, you will need to add a case statement for your node in `LegalizeOp` which calls LegalizeOp on the node's operands, and returns a new node if any of the operands changed as a result of being legalized. It is likely that not all targets supported by the SelectionDAG framework will natively support the new node. In this case, you must also add code in your node's case statement in `LegalizeOp` to Expand your node into simpler, legal operations. The case for `ISD::UREM` for expanding a remainder into a divide, multiply, and a subtract is a good example.

4. `lib/CodeGen/SelectionDAG/LegalizeDAG.cpp`:

   If targets may support the new node being added only at certain sizes, you will also need to add code to your node's case statement in `LegalizeOp` to Promote your node's operands to a larger size, and perform the correct operation. You will also need to add code to `PromoteOp` to do this as well. For a good example, see `ISD::BSWAP`, which promotes its operand to a wider size, performs the byteswap, and then shifts the correct bytes right to emulate the narrower byteswap in the wider type.

5. `lib/CodeGen/SelectionDAG/LegalizeDAG.cpp`:

   Add a case for your node in `ExpandOp` to teach the legalizer how to perform the action represented by the new node on a value that has been split into high and low halves. This case will be used to support your node with a 64 bit operand on a 32 bit target.

6. `lib/CodeGen/SelectionDAG/DAGCombiner.cpp`:

   If your node can be combined with itself, or other existing nodes in a peephole-like fashion, add a visit function for it, and call that function from. There are several good examples for simple combines you can do; `visitFABS` and `visitSRL` are good starting places.

7. `lib/Target/PowerPC/PPCISelLowering.cpp`:

   Each target has an implementation of the `TargetLowering` class, usually in its own file (although some targets include it in the same file as the DAGToDAGISel). The default behavior for a target is to assume that your new node is legal for all types that are legal for that target. If this target does not natively support your node, then tell the target to either Promote it (if it is supported at a larger type) or Expand it. This will cause the code you wrote in `LegalizeOp` above to decompose your new node into other legal nodes for this target.

8. `lib/Target/TargetSelectionDAG.td`:

   Most current targets supported by LLVM generate code using the DAGToDAG method, where SelectionDAG nodes are pattern matched to target-specific nodes, which represent individual instructions. In order for the targets to match an instruction to your new node, you must add a def for that node to the list in this file, with the appropriate type constraints. Look at `add`, `bswap`, and `fadd` for examples.

9. `lib/Target/PowerPC/PPCInstrInfo.td`:

   Each target has a tablegen file that describes the target's instruction set. For targets that use the DAGToDAG instruction selection framework, add a pattern for your new node that uses one or more target nodes. Documentation for this is a bit sparse right now, but there are several decent examples. See the patterns for `rotl` in `PPCInstrInfo.td`.

10. TODO: document complex patterns.

11. `llvm/test/Regression/CodeGen/*`:

    Add test cases for your new node to the test suite. `llvm/test/Regression/CodeGen/X86/bswap.ll` is a good example.

3.5.4 Adding a new instruction

Warning: Adding instructions changes the bitcode format, and it will take some effort to maintain compatibility with the previous version. Only add an instruction if it is absolutely necessary.

1. `llvm/include/llvm/Instruction.def`:

   add a number for your instruction and an enum name

2. `llvm/include/llvm/Instructions.h`:

   add a definition for the class that will represent your instruction

3. `llvm/include/llvm/Support/InstVisitor.h`:

   add a prototype for a visitor to your new instruction type

4. `llvm/lib/AsmParser/Lexer.l`:

   add a new token to parse your instruction from assembly text file

5. `llvm/lib/AsmParser/llvmAsmParser.y`:

   add the grammar on how your instruction can be read and what it will construct as a result

6. `llvm/lib/Bitcode/Reader/Reader.cpp`:

   add a case for your instruction and how it will be parsed from bitcode

7. `llvm/lib/VMCore/Instruction.cpp`:

   add a case for how your instruction will be printed out to assembly

8. `llvm/lib/VMCore/Instructions.cpp`:

   implement the class you defined in `llvm/include/llvm/Instructions.h`

9. Test your instruction

10. `llvm/lib/Target/*`:

    add support for your instruction to code generators, or add a lowering pass.

11. `llvm/test/Regression/*`:

    add your test cases to the test suite.

Also, you need to implement (or modify) any analyses or passes that you want to understand this new instruction.

3.5.5 Adding a new type

Warning: Adding new types changes the bitcode format, and will break compatibility with currently-existing LLVM installations. Only add new types if it is absolutely necessary.

Adding a fundamental type

1. `llvm/include/llvm/Type.h`:

   add enum for the new type; add static `Type*` for this type

2. `llvm/lib/VMCore/Type.cpp`:

   add mapping from `TypeID` => `Type*`; initialize the static `Type*`

3. `llvm/lib/AsmReader/Lexer.l`:

   add ability to parse in the type from text assembly

4. `llvm/lib/AsmReader/llvmAsmParser.y`:

   add a token for that type

Adding a derived type

1. `llvm/include/llvm/Type.h`:

   add enum for the new type; add a forward declaration of the type also

2. `llvm/include/llvm/DerivedTypes.h`:

   add new class to represent new class in the hierarchy; add forward declaration to the TypeMap value type

3. `llvm/lib/VMCore/Type.cpp`:

   add support for derived type to:

```
std::string getTypeDescription(const Type &Ty,
                               std::vector<const Type*> &TypeStack)
bool TypesEqual(const Type *Ty, const Type *Ty2,
                std::map<const Type*, const Type*> &EqTypes)
```

add necessary member functions for type, and factory methods

4. `llvm/lib/AsmReader/Lexer.l`:

   add ability to parse in the type from text assembly

5. `llvm/lib/Bitcode/Writer/Writer.cpp`:

   modify `void BitcodeWriter::outputType(const Type *T)` to serialize your type

6. `llvm/lib/Bitcode/Reader/Reader.cpp`:

   modify `const Type *BitcodeReader::ParseType()` to read your data type

7. `llvm/lib/VMCore/AsmWriter.cpp`:

   modify

```
void calcTypeName(const Type *Ty,
                  std::vector<const Type*> &TypeStack,
                  std::map<const Type*,std::string> &TypeNames,
                  std::string &Result)
```

   to output the new derived type

3.6 How to set up LLVM-style RTTI for your class hierarchy

Contents

3.6.1 Background

LLVM avoids using C++'s built in RTTI. Instead, it pervasively uses its own hand-rolled form of RTTI which is much more efficient and flexible, although it requires a bit more work from you as a class author.

A description of how to use LLVM-style RTTI from a client's perspective is given in the Programmer's Manual. This document, in contrast, discusses the steps you need to take as a class hierarchy author to make LLVM-style RTTI available to your clients.

Before diving in, make sure that you are familiar with the Object Oriented Programming concept of "is-a".

3.6.2 Basic Setup

This section describes how to set up the most basic form of LLVM-style RTTI (which is sufficient for 99.9% of the cases). We will set up LLVM-style RTTI for this class hierarchy:

```
class Shape {
public:
  Shape() {}
  virtual double computeArea() = 0;
};

class Square : public Shape {
  double SideLength;
public:
  Square(double S) : SideLength(S) {}
  double computeArea() /* override */;
};

class Circle : public Shape {
  double Radius;
public:
  Circle(double R) : Radius(R) {}
  double computeArea() /* override */;
};
```

The most basic working setup for LLVM-style RTTI requires the following steps:

1. In the header where you declare `Shape`, you will want to `#include "llvm/Support/Casting.h"`, which declares LLVM's RTTI templates. That way your clients don't even have to think about it.

   ```
   #include "llvm/Support/Casting.h"
   ```

2. In the base class, introduce an enum which discriminates all of the different concrete classes in the hierarchy, and stash the enum value somewhere in the base class.

   Here is the code after introducing this change:

   ```
    class Shape {
    public:
   +  /// Discriminator for LLVM-style RTTI (dyn_cast<> et al.)
   +  enum ShapeKind {
   +    SK_Square,
   +    SK_Circle
   +  };
   +private:
   +  const ShapeKind Kind;
   +public:
   +  ShapeKind getKind() const { return Kind; }
   +
      Shape() {}
      virtual double computeArea() = 0;
    };
   ```

   You will usually want to keep the `Kind` member encapsulated and private, but let the enum `ShapeKind` be public along with providing a `getKind()` method. This is convenient for clients so that they can do a `switch` over the enum.

   A common naming convention is that these enums are "kind"s, to avoid ambiguity with the words "type" or "class" which have overloaded meanings in many contexts within LLVM. Sometimes there will be a natural name for it, like "opcode". Don't bikeshed over this; when in doubt use `Kind`.

   You might wonder why the `Kind` enum doesn't have an entry for `Shape`. The reason for this is that since `Shape` is abstract (`computeArea() = 0;`), you will never actually have non-derived instances of exactly that class (only subclasses). See Concrete Bases and Deeper Hierarchies for information on how to deal with

non-abstract bases. It's worth mentioning here that unlike `dynamic_cast<>`, LLVM-style RTTI can be used (and is often used) for classes that don't have v-tables.

3. Next, you need to make sure that the `Kind` gets initialized to the value corresponding to the dynamic type of the class. Typically, you will want to have it be an argument to the constructor of the base class, and then pass in the respective `XXXKind` from subclass constructors.

   Here is the code after that change:

```
class Shape {
public:
  /// Discriminator for LLVM-style RTTI (dyn_cast<> et al.)
  enum ShapeKind {
    SK_Square,
    SK_Circle
  };
private:
  const ShapeKind Kind;
public:
  ShapeKind getKind() const { return Kind; }

-  Shape() {}
+  Shape(ShapeKind K) : Kind(K) {}
  virtual double computeArea() = 0;
};

class Square : public Shape {
  double SideLength;
public:
-  Square(double S) : SideLength(S) {}
+  Square(double S) : Shape(SK_Square), SideLength(S) {}
  double computeArea() /* override */;
};

class Circle : public Shape {
  double Radius;
public:
-  Circle(double R) : Radius(R) {}
+  Circle(double R) : Shape(SK_Circle), Radius(R) {}
  double computeArea() /* override */;
};
```

4. Finally, you need to inform LLVM's RTTI templates how to dynamically determine the type of a class (i.e. whether the `isa<>`/`dyn_cast<>` should succeed). The default "99.9% of use cases" way to accomplish this is through a small static member function `classof`. In order to have proper context for an explanation, we will display this code first, and then below describe each part:

```
class Shape {
public:
  /// Discriminator for LLVM-style RTTI (dyn_cast<> et al.)
  enum ShapeKind {
    SK_Square,
    SK_Circle
  };
private:
  const ShapeKind Kind;
public:
  ShapeKind getKind() const { return Kind; }
```

```
   Shape(ShapeKind K) : Kind(K) {}
   virtual double computeArea() = 0;
 };

 class Square : public Shape {
   double SideLength;
 public:
   Square(double S) : Shape(SK_Square), SideLength(S) {}
   double computeArea() /* override */;
+
+  static bool classof(const Shape *S) {
+    return S->getKind() == SK_Square;
+  }
 };

 class Circle : public Shape {
   double Radius;
 public:
   Circle(double R) : Shape(SK_Circle), Radius(R) {}
   double computeArea() /* override */;
+
+  static bool classof(const Shape *S) {
+    return S->getKind() == SK_Circle;
+  }
 };
```

The job of `classof` is to dynamically determine whether an object of a base class is in fact of a particular derived class. In order to downcast a type `Base` to a type `Derived`, there needs to be a `classof` in `Derived` which will accept an object of type `Base`.

To be concrete, consider the following code:

```
Shape *S = ...;
if (isa<Circle>(S)) {
  /* do something ... */
}
```

The code of the `isa<>` test in this code will eventually boil down—after template instantiation and some other machinery—to a check roughly like `Circle::classof(S)`. For more information, see *The Contract of classof*.

The argument to `classof` should always be an *ancestor* class because the implementation has logic to allow and optimize away upcasts/up-`isa<>`'s automatically. It is as though every class `Foo` automatically has a `classof` like:

```
class Foo {
  [...]
  template <class T>
  static bool classof(const T *,
                      ::std::enable_if<
                        ::std::is_base_of<Foo, T>::value
                      >::type* = 0) { return true; }
  [...]
};
```

Note that this is the reason that we did not need to introduce a `classof` into `Shape`: all relevant classes derive from `Shape`, and `Shape` itself is abstract (has no entry in the `Kind` enum), so this notional inferred `classof` is all we need. See Concrete Bases and Deeper Hierarchies for more information about how to extend this example to more general hierarchies.

Although for this small example setting up LLVM-style RTTI seems like a lot of "boilerplate", if your classes are doing anything interesting then this will end up being a tiny fraction of the code.

3.6.3 Concrete Bases and Deeper Hierarchies

For concrete bases (i.e. non-abstract interior nodes of the inheritance tree), the `Kind` check inside `classof` needs to be a bit more complicated. The situation differs from the example above in that

- Since the class is concrete, it must itself have an entry in the `Kind` enum because it is possible to have objects with this class as a dynamic type.
- Since the class has children, the check inside `classof` must take them into account.

Say that `SpecialSquare` and `OtherSpecialSquare` derive from `Square`, and so `ShapeKind` becomes:

```
 enum ShapeKind {
   SK_Square,
+  SK_SpecialSquare,
+  SK_OtherSpecialSquare,
   SK_Circle
 }
```

Then in `Square`, we would need to modify the `classof` like so:

```
-  static bool classof(const Shape *S) {
-    return S->getKind() == SK_Square;
-  }
+  static bool classof(const Shape *S) {
+    return S->getKind() >= SK_Square &&
+           S->getKind() <= SK_OtherSpecialSquare;
+  }
```

The reason that we need to test a range like this instead of just equality is that both `SpecialSquare` and `OtherSpecialSquare` "is-a" `Square`, and so `classof` needs to return `true` for them.

This approach can be made to scale to arbitrarily deep hierarchies. The trick is that you arrange the enum values so that they correspond to a preorder traversal of the class hierarchy tree. With that arrangement, all subclass tests can be done with two comparisons as shown above. If you just list the class hierarchy like a list of bullet points, you'll get the ordering right:

```
| Shape
  | Square
    | SpecialSquare
    | OtherSpecialSquare
  | Circle
```

A Bug to be Aware Of

The example just given opens the door to bugs where the `classof`s are not updated to match the `Kind` enum when adding (or removing) classes to (from) the hierarchy.

Continuing the example above, suppose we add a `SomewhatSpecialSquare` as a subclass of `Square`, and update the `ShapeKind` enum like so:

```
 enum ShapeKind {
   SK_Square,
   SK_SpecialSquare,
   SK_OtherSpecialSquare,
```

```
+  SK_SomewhatSpecialSquare,
   SK_Circle
 }
```

Now, suppose that we forget to update `Square::classof()`, so it still looks like:

```
static bool classof(const Shape *S) {
  // BUG: Returns false when S->getKind() == SK_SomewhatSpecialSquare,
  // even though SomewhatSpecialSquare "is a" Square.
  return S->getKind() >= SK_Square &&
         S->getKind() <= SK_OtherSpecialSquare;
}
```

As the comment indicates, this code contains a bug. A straightforward and non-clever way to avoid this is to introduce an explicit `SK_LastSquare` entry in the enum when adding the first subclass(es). For example, we could rewrite the example at the beginning of Concrete Bases and Deeper Hierarchies as:

```
 enum ShapeKind {
   SK_Square,
+  SK_SpecialSquare,
+  SK_OtherSpecialSquare,
+  SK_LastSquare,
   SK_Circle
 }
...
// Square::classof()
-  static bool classof(const Shape *S) {
-    return S->getKind() == SK_Square;
-  }
+  static bool classof(const Shape *S) {
+    return S->getKind() >= SK_Square &&
+           S->getKind() <= SK_LastSquare;
+  }
```

Then, adding new subclasses is easy:

```
 enum ShapeKind {
   SK_Square,
   SK_SpecialSquare,
   SK_OtherSpecialSquare,
+  SK_SomewhatSpecialSquare,
   SK_LastSquare,
   SK_Circle
 }
```

Notice that `Square::classof` does not need to be changed.

The Contract of `classof`

To be more precise, let `classof` be inside a class `C`. Then the contract for `classof` is "return `true` if the dynamic type of the argument is-a `C`". As long as your implementation fulfills this contract, you can tweak and optimize it as much as you want.

3.6.4 Rules of Thumb

1. The `Kind` enum should have one entry per concrete class, ordered according to a preorder traversal of the inheritance tree.

2. The argument to `classof` should be a `const Base *`, where `Base` is some ancestor in the inheritance hierarchy. The argument should *never* be a derived class or the class itself: the template machinery for `isa<>` already handles this case and optimizes it.
3. For each class in the hierarchy that has no children, implement a `classof` that checks only against its `Kind`.
4. For each class in the hierarchy that has children, implement a `classof` that checks a range of the first child's `Kind` and the last child's `Kind`.

3.7 LLVM Programmer's Manual

Warning: This is always a work in progress.

3.7.1 Introduction

This document is meant to highlight some of the important classes and interfaces available in the LLVM source-base. This manual is not intended to explain what LLVM is, how it works, and what LLVM code looks like. It assumes that you know the basics of LLVM and are interested in writing transformations or otherwise analyzing or manipulating the code.

This document should get you oriented so that you can find your way in the continuously growing source code that makes up the LLVM infrastructure. Note that this manual is not intended to serve as a replacement for reading the source code, so if you think there should be a method in one of these classes to do something, but it's not listed, check the source. Links to the doxygen sources are provided to make this as easy as possible.

The first section of this document describes general information that is useful to know when working in the LLVM infrastructure, and the second describes the Core LLVM classes. In the future this manual will be extended with information describing how to use extension libraries, such as dominator information, CFG traversal routines, and useful utilities like the `InstVisitor` (doxygen) template.

3.7.2 General Information

This section contains general information that is useful if you are working in the LLVM source-base, but that isn't specific to any particular API.

The C++ Standard Template Library

LLVM makes heavy use of the C++ Standard Template Library (STL), perhaps much more than you are used to, or have seen before. Because of this, you might want to do a little background reading in the techniques used and capabilities of the library. There are many good pages that discuss the STL, and several books on the subject that you can get, so it will not be discussed in this document.

Here are some useful links:

1. cppreference.com - an excellent reference for the STL and other parts of the standard C++ library.
2. C++ In a Nutshell - This is an O'Reilly book in the making. It has a decent Standard Library Reference that rivals Dinkumware's, and is unfortunately no longer free since the book has been published.
3. C++ Frequently Asked Questions.
4. SGI's STL Programmer's Guide - Contains a useful Introduction to the STL.
5. Bjarne Stroustrup's C++ Page.
6. Bruce Eckel's Thinking in C++, 2nd ed. Volume 2 Revision 4.0 (even better, get the book).

You are also encouraged to take a look at the *LLVM Coding Standards* guide which focuses on how to write maintainable code more than where to put your curly braces.

Other useful references

1. Using static and shared libraries across platforms

3.7.3 Important and useful LLVM APIs

Here we highlight some LLVM APIs that are generally useful and good to know about when writing transformations.

The `isa<>`, `cast<>` and `dyn_cast<>` templates

The LLVM source-base makes extensive use of a custom form of RTTI. These templates have many similarities to the C++ `dynamic_cast<>` operator, but they don't have some drawbacks (primarily stemming from the fact that `dynamic_cast<>` only works on classes that have a v-table). Because they are used so often, you must know what they do and how they work. All of these templates are defined in the `llvm/Support/Casting.h` (doxygen) file (note that you very rarely have to include this file directly).

`isa<>`: The `isa<>` operator works exactly like the Java "`instanceof`" operator. It returns true or false depending on whether a reference or pointer points to an instance of the specified class. This can be very useful for constraint checking of various sorts (example below).

`cast<>`: The `cast<>` operator is a "checked cast" operation. It converts a pointer or reference from a base class to a derived class, causing an assertion failure if it is not really an instance of the right type. This should be used in cases where you have some information that makes you believe that something is of the right type. An example of the `isa<>` and `cast<>` template is:

```
static bool isLoopInvariant(const Value *V, const Loop *L) {
  if (isa<Constant>(V) || isa<Argument>(V) || isa<GlobalValue>(V))
    return true;

  // Otherwise, it must be an instruction...
  return !L->contains(cast<Instruction>(V)->getParent());
}
```

Note that you should **not** use an `isa<>` test followed by a `cast<>`, for that use the `dyn_cast<>` operator.

`dyn_cast<>`: The `dyn_cast<>` operator is a "checking cast" operation. It checks to see if the operand is of the specified type, and if so, returns a pointer to it (this operator does not work with references). If the operand is not of the correct type, a null pointer is returned. Thus, this works very much like the `dynamic_cast<>` operator in C++, and should be used in the same circumstances. Typically, the `dyn_cast<>` operator is used in an `if` statement or some other flow control statement like this:

```
if (AllocationInst *AI = dyn_cast<AllocationInst>(Val)) {
  // ...
}
```

This form of the `if` statement effectively combines together a call to `isa<>` and a call to `cast<>` into one statement, which is very convenient.

Note that the `dyn_cast<>` operator, like C++'s `dynamic_cast<>` or Java's `instanceof` operator, can be abused. In particular, you should not use big chained `if/then/else` blocks to check for lots of different variants of classes. If you find yourself wanting to do this, it is much cleaner and more efficient to use the `InstVisitor` class to dispatch over the instruction type directly.

`cast_or_null<>`: The `cast_or_null<>` operator works just like the `cast<>` operator, except that it allows for a null pointer as an argument (which it then propagates). This can sometimes be useful, allowing you to combine several null checks into one.

`dyn_cast_or_null<>`: The `dyn_cast_or_null<>` operator works just like the `dyn_cast<>` operator, except that it allows for a null pointer as an argument (which it then propagates). This can sometimes be useful, allowing you to combine several null checks into one.

These five templates can be used with any classes, whether they have a v-table or not. If you want to add support for these templates, see the document *How to set up LLVM-style RTTI for your class hierarchy*

Passing strings (the `StringRef` and `Twine` classes)

Although LLVM generally does not do much string manipulation, we do have several important APIs which take strings. Two important examples are the Value class – which has names for instructions, functions, etc. – and the `StringMap` class which is used extensively in LLVM and Clang.

These are generic classes, and they need to be able to accept strings which may have embedded null characters. Therefore, they cannot simply take a `const char *`, and taking a `const std::string&` requires clients to perform a heap allocation which is usually unnecessary. Instead, many LLVM APIs use a `StringRef` or a `const Twine&` for passing strings efficiently.

The `StringRef` class

The `StringRef` data type represents a reference to a constant string (a character array and a length) and supports the common operations available on `std::string`, but does not require heap allocation.

It can be implicitly constructed using a C style null-terminated string, an `std::string`, or explicitly with a character pointer and length. For example, the `StringRef` find function is declared as:

```
iterator find(StringRef Key);
```

and clients can call it using any one of:

```
Map.find("foo");                 // Lookup "foo"
Map.find(std::string("bar"));    // Lookup "bar"
Map.find(StringRef("\0baz", 4)); // Lookup "\0baz"
```

Similarly, APIs which need to return a string may return a `StringRef` instance, which can be used directly or converted to an `std::string` using the `str` member function. See `llvm/ADT/StringRef.h` (doxygen) for more information.

You should rarely use the `StringRef` class directly, because it contains pointers to external memory it is not generally safe to store an instance of the class (unless you know that the external storage will not be freed). `StringRef` is small and pervasive enough in LLVM that it should always be passed by value.

The `Twine` class

The `Twine` (doxygen) class is an efficient way for APIs to accept concatenated strings. For example, a common LLVM paradigm is to name one instruction based on the name of another instruction with a suffix, for example:

```
New = CmpInst::Create(..., SO->getName() + ".cmp");
```

The `Twine` class is effectively a lightweight rope which points to temporary (stack allocated) objects. Twines can be implicitly constructed as the result of the plus operator applied to strings (i.e., a C strings, an `std::string`, or a `StringRef`). The twine delays the actual concatenation of strings until it is actually required, at which point it can be efficiently rendered directly into a character array. This avoids unnecessary heap allocation involved in constructing the temporary results of string concatenation. See `llvm/ADT/Twine.h` (doxygen) and *here* for more information.

As with a `StringRef`, `Twine` objects point to external memory and should almost never be stored or mentioned directly. They are intended solely for use when defining a function which should be able to efficiently accept concatenated strings.

Passing functions and other callable objects

Sometimes you may want a function to be passed a callback object. In order to support lambda expressions and other function objects, you should not use the traditional C approach of taking a function pointer and an opaque cookie:

```
void takeCallback(bool (*Callback)(Function *, void *), void *Cookie);
```

Instead, use one of the following approaches:

Function template

If you don't mind putting the definition of your function into a header file, make it a function template that is templated on the callable type.

```
template<typename Callable>
void takeCallback(Callable Callback) {
  Callback(1, 2, 3);
}
```

The `function_ref` class template

The `function_ref` (doxygen) class template represents a reference to a callable object, templated over the type of the callable. This is a good choice for passing a callback to a function, if you don't need to hold onto the callback after the function returns. In this way, `function_ref` is to `std::function` as `StringRef` is to `std::string`.

`function_ref<Ret(Param1, Param2, ...)>` can be implicitly constructed from any callable object that can be called with arguments of type `Param1`, `Param2`, ..., and returns a value that can be converted to type `Ret`. For example:

```
void visitBasicBlocks(Function *F, function_ref<bool (BasicBlock*)> Callback) {
  for (BasicBlock &BB : *F)
    if (Callback(&BB))
      return;
}
```

can be called using:

```
visitBasicBlocks(F, [&](BasicBlock *BB) {
  if (process(BB))
    return isEmpty(BB);
  return false;
});
```

Note that a `function_ref` object contains pointers to external memory, so it is not generally safe to store an instance of the class (unless you know that the external storage will not be freed). If you need this ability, consider using `std::function`. `function_ref` is small enough that it should always be passed by value.

The `DEBUG()` macro and `-debug` option

Often when working on your pass you will put a bunch of debugging printouts and other code into your pass. After you get it working, you want to remove it, but you may need it again in the future (to work out new bugs that you run across).

Naturally, because of this, you don't want to delete the debug printouts, but you don't want them to always be noisy. A standard compromise is to comment them out, allowing you to enable them if you need them in the future.

The `llvm/Support/Debug.h` (doxygen) file provides a macro named `DEBUG()` that is a much nicer solution to this problem. Basically, you can put arbitrary code into the argument of the `DEBUG` macro, and it is only executed if '`opt`' (or any other tool) is run with the '`-debug`' command line argument:

```
DEBUG(errs() << "I am here!\n");
```

Then you can run your pass like this:

```
$ opt < a.bc > /dev/null -mypass
<no output>
$ opt < a.bc > /dev/null -mypass -debug
I am here!
```

Using the `DEBUG()` macro instead of a home-brewed solution allows you to not have to create "yet another" command line option for the debug output for your pass. Note that `DEBUG()` macros are disabled for optimized builds, so they do not cause a performance impact at all (for the same reason, they should also not contain side-effects!).

One additional nice thing about the `DEBUG()` macro is that you can enable or disable it directly in gdb. Just use "`set DebugFlag=0`" or "`set DebugFlag=1`" from the gdb if the program is running. If the program hasn't been started yet, you can always just run it with `-debug`.

Fine grained debug info with `DEBUG_TYPE` and the `-debug-only` option

Sometimes you may find yourself in a situation where enabling `-debug` just turns on **too much** information (such as when working on the code generator). If you want to enable debug information with more fine-grained control, you can define the `DEBUG_TYPE` macro and use the `-debug-only` option as follows:

```
#undef  DEBUG_TYPE
DEBUG(errs() << "No debug type\n");
#define DEBUG_TYPE "foo"
DEBUG(errs() << "'foo' debug type\n");
#undef  DEBUG_TYPE
#define DEBUG_TYPE "bar"
DEBUG(errs() << "'bar' debug type\n"));
#undef  DEBUG_TYPE
#define DEBUG_TYPE ""
DEBUG(errs() << "No debug type (2)\n");
```

Then you can run your pass like this:

```
$ opt < a.bc > /dev/null -mypass
<no output>
$ opt < a.bc > /dev/null -mypass -debug
No debug type
'foo' debug type
'bar' debug type
No debug type (2)
$ opt < a.bc > /dev/null -mypass -debug-only=foo
'foo' debug type
$ opt < a.bc > /dev/null -mypass -debug-only=bar
'bar' debug type
```

Of course, in practice, you should only set `DEBUG_TYPE` at the top of a file, to specify the debug type for the entire module (if you do this before you `#include "llvm/Support/Debug.h"`, you don't have to insert the ugly `#undef`'s). Also, you should use names more meaningful than "foo" and "bar", because there is no system in place to ensure that names do not conflict. If two different modules use the same string, they will all be turned on when the name is specified. This allows, for example, all debug information for instruction scheduling to be enabled with `-debug-only=InstrSched`, even if the source lives in multiple files.

For performance reasons, -debug-only is not available in optimized build (`--enable-optimized`) of LLVM.

The DEBUG_WITH_TYPE macro is also available for situations where you would like to set DEBUG_TYPE, but only for one specific DEBUG statement. It takes an additional first parameter, which is the type to use. For example, the preceding example could be written as:

```
DEBUG_WITH_TYPE("", errs() << "No debug type\n");
DEBUG_WITH_TYPE("foo", errs() << "'foo' debug type\n");
DEBUG_WITH_TYPE("bar", errs() << "'bar' debug type\n"));
DEBUG_WITH_TYPE("", errs() << "No debug type (2)\n");
```

The Statistic class & -stats option

The llvm/ADT/Statistic.h (doxygen) file provides a class named Statistic that is used as a unified way to keep track of what the LLVM compiler is doing and how effective various optimizations are. It is useful to see what optimizations are contributing to making a particular program run faster.

Often you may run your pass on some big program, and you're interested to see how many times it makes a certain transformation. Although you can do this with hand inspection, or some ad-hoc method, this is a real pain and not very useful for big programs. Using the Statistic class makes it very easy to keep track of this information, and the calculated information is presented in a uniform manner with the rest of the passes being executed.

There are many examples of Statistic uses, but the basics of using it are as follows:

1. Define your statistic like this:

   ```
   #define DEBUG_TYPE "mypassname"   // This goes before any #includes.
   STATISTIC(NumXForms, "The # of times I did stuff");
   ```

   The STATISTIC macro defines a static variable, whose name is specified by the first argument. The pass name is taken from the DEBUG_TYPE macro, and the description is taken from the second argument. The variable defined ("NumXForms" in this case) acts like an unsigned integer.

1. Whenever you make a transformation, bump the counter:

   ```
   ++NumXForms;   // I did stuff!
   ```

That's all you have to do. To get 'opt' to print out the statistics gathered, use the '-stats' option:

```
$ opt -stats -mypassname < program.bc > /dev/null
... statistics output ...
```

When running opt on a C file from the SPEC benchmark suite, it gives a report that looks like this:

```
  7646 bitcodewriter   - Number of normal instructions
   725 bitcodewriter   - Number of oversized instructions
129996 bitcodewriter   - Number of bitcode bytes written
  2817 raise           - Number of insts DCEd or constprop'd
  3213 raise           - Number of cast-of-self removed
  5046 raise           - Number of expression trees converted
    75 raise           - Number of other getelementptr's formed
   138 raise           - Number of load/store peepholes
    42 deadtypeelim    - Number of unused typenames removed from symtab
   392 funcresolve     - Number of varargs functions resolved
    27 globaldce       - Number of global variables removed
     2 adce            - Number of basic blocks removed
   134 cee             - Number of branches revectored
    49 cee             - Number of setcc instruction eliminated
   532 gcse            - Number of loads removed
  2919 gcse            - Number of instructions removed
    86 indvars         - Number of canonical indvars added
```

```
  87 indvars         - Number of aux indvars removed
  25 instcombine     - Number of dead inst eliminate
 434 instcombine     - Number of insts combined
 248 licm            - Number of load insts hoisted
1298 licm            - Number of insts hoisted to a loop pre-header
   3 licm            - Number of insts hoisted to multiple loop preds (bad, no loop pre-header)
  75 mem2reg         - Number of alloca's promoted
1444 cfgsimplify     - Number of blocks simplified
```

Obviously, with so many optimizations, having a unified framework for this stuff is very nice. Making your pass fit well into the framework makes it more maintainable and useful.

Viewing graphs while debugging code

Several of the important data structures in LLVM are graphs: for example CFGs made out of LLVM *BasicBlocks*, CFGs made out of LLVM *MachineBasicBlocks*, and *Instruction Selection DAGs*. In many cases, while debugging various parts of the compiler, it is nice to instantly visualize these graphs.

LLVM provides several callbacks that are available in a debug build to do exactly that. If you call the `Function::viewCFG()` method, for example, the current LLVM tool will pop up a window containing the CFG for the function where each basic block is a node in the graph, and each node contains the instructions in the block. Similarly, there also exists `Function::viewCFGOnly()` (does not include the instructions), the `MachineFunction::viewCFG()` and `MachineFunction::viewCFGOnly()`, and the `SelectionDAG::viewGraph()` methods. Within GDB, for example, you can usually use something like `call DAG.viewGraph()` to pop up a window. Alternatively, you can sprinkle calls to these functions in your code in places you want to debug.

Getting this to work requires a small amount of setup. On Unix systems with X11, install the graphviz toolkit, and make sure 'dot' and 'gv' are in your path. If you are running on Mac OS X, download and install the Mac OS X Graphviz program and add `/Applications/Graphviz.app/Contents/MacOS/` (or wherever you install it) to your path. The programs need not be present when configuring, building or running LLVM and can simply be installed when needed during an active debug session.

`SelectionDAG` has been extended to make it easier to locate *interesting* nodes in large complex graphs. From gdb, if you `call DAG.setGraphColor(node, "color")`, then the next `call DAG.viewGraph()` would highlight the node in the specified color (choices of colors can be found at colors.) More complex node attributes can be provided with `call DAG.setGraphAttrs(node, "attributes")` (choices can be found at Graph attributes.) If you want to restart and clear all the current graph attributes, then you can `call DAG.clearGraphAttrs()`.

Note that graph visualization features are compiled out of Release builds to reduce file size. This means that you need a Debug+Asserts or Release+Asserts build to use these features.

3.7.4 Picking the Right Data Structure for a Task

LLVM has a plethora of data structures in the `llvm/ADT/` directory, and we commonly use STL data structures. This section describes the trade-offs you should consider when you pick one.

The first step is a choose your own adventure: do you want a sequential container, a set-like container, or a map-like container? The most important thing when choosing a container is the algorithmic properties of how you plan to access the container. Based on that, you should use:

- a *map-like* container if you need efficient look-up of a value based on another value. Map-like containers also support efficient queries for containment (whether a key is in the map). Map-like containers generally do not support efficient reverse mapping (values to keys). If you need that, use two maps. Some map-like containers also support efficient iteration through the keys in sorted order. Map-like containers are the most expensive sort, only use them if you need one of these capabilities.

- a *set-like* container if you need to put a bunch of stuff into a container that automatically eliminates duplicates. Some set-like containers support efficient iteration through the elements in sorted order. Set-like containers are more expensive than sequential containers.
- a *sequential* container provides the most efficient way to add elements and keeps track of the order they are added to the collection. They permit duplicates and support efficient iteration, but do not support efficient look-up based on a key.
- a *string* container is a specialized sequential container or reference structure that is used for character or byte arrays.
- a *bit* container provides an efficient way to store and perform set operations on sets of numeric id's, while automatically eliminating duplicates. Bit containers require a maximum of 1 bit for each identifier you want to store.

Once the proper category of container is determined, you can fine tune the memory use, constant factors, and cache behaviors of access by intelligently picking a member of the category. Note that constant factors and cache behavior can be a big deal. If you have a vector that usually only contains a few elements (but could contain many), for example, it's much better to use *SmallVector* than *vector*. Doing so avoids (relatively) expensive malloc/free calls, which dwarf the cost of adding the elements to the container.

Sequential Containers (std::vector, std::list, etc)

There are a variety of sequential containers available for you, based on your needs. Pick the first in this section that will do what you want.

llvm/ADT/ArrayRef.h

The `llvm::ArrayRef` class is the preferred class to use in an interface that accepts a sequential list of elements in memory and just reads from them. By taking an `ArrayRef`, the API can be passed a fixed size array, an `std::vector`, an `llvm::SmallVector` and anything else that is contiguous in memory.

Fixed Size Arrays

Fixed size arrays are very simple and very fast. They are good if you know exactly how many elements you have, or you have a (low) upper bound on how many you have.

Heap Allocated Arrays

Heap allocated arrays (`new[]` + `delete[]`) are also simple. They are good if the number of elements is variable, if you know how many elements you will need before the array is allocated, and if the array is usually large (if not, consider a *SmallVector*). The cost of a heap allocated array is the cost of the new/delete (aka malloc/free). Also note that if you are allocating an array of a type with a constructor, the constructor and destructors will be run for every element in the array (re-sizable vectors only construct those elements actually used).

llvm/ADT/TinyPtrVector.h

`TinyPtrVector<Type>` is a highly specialized collection class that is optimized to avoid allocation in the case when a vector has zero or one elements. It has two major restrictions: 1) it can only hold values of pointer type, and 2) it cannot hold a null pointer.

Since this container is highly specialized, it is rarely used.

llvm/ADT/SmallVector.h

`SmallVector<Type, N>` is a simple class that looks and smells just like `vector<Type>`: it supports efficient iteration, lays out elements in memory order (so you can do pointer arithmetic between elements), supports efficient push_back/pop_back operations, supports efficient random access to its elements, etc.

The advantage of SmallVector is that it allocates space for some number of elements (N) **in the object itself**. Because of this, if the SmallVector is dynamically smaller than N, no malloc is performed. This can be a big win in cases where the malloc/free call is far more expensive than the code that fiddles around with the elements.

This is good for vectors that are "usually small" (e.g. the number of predecessors/successors of a block is usually less than 8). On the other hand, this makes the size of the SmallVector itself large, so you don't want to allocate lots of them (doing so will waste a lot of space). As such, SmallVectors are most useful when on the stack.

SmallVector also provides a nice portable and efficient replacement for `alloca`.

Note: Prefer to use `SmallVectorImpl<T>` as a parameter type.

In APIs that don't care about the "small size" (most?), prefer to use the `SmallVectorImpl<T>` class, which is basically just the "vector header" (and methods) without the elements allocated after it. Note that `SmallVector<T, N>` inherits from `SmallVectorImpl<T>` so the conversion is implicit and costs nothing. E.g.

```
// BAD: Clients cannot pass e.g. SmallVector<Foo, 4>.
hardcodedSmallSize(SmallVector<Foo, 2> &Out);
// GOOD: Clients can pass any SmallVector<Foo, N>.
allowsAnySmallSize(SmallVectorImpl<Foo> &Out);

void someFunc() {
  SmallVector<Foo, 8> Vec;
  hardcodedSmallSize(Vec); // Error.
  allowsAnySmallSize(Vec); // Works.
}
```

Even though it has "`Impl`" in the name, this is so widely used that it really isn't "private to the implementation" anymore. A name like `SmallVectorHeader` would be more appropriate.

<vector>

`std::vector` is well loved and respected. It is useful when SmallVector isn't: when the size of the vector is often large (thus the small optimization will rarely be a benefit) or if you will be allocating many instances of the vector itself (which would waste space for elements that aren't in the container). vector is also useful when interfacing with code that expects vectors :).

One worthwhile note about std::vector: avoid code like this:

```
for ( ... ) {
   std::vector<foo> V;
   // make use of V.
}
```

Instead, write this as:

```
std::vector<foo> V;
for ( ... ) {
   // make use of V.
   V.clear();
}
```

Doing so will save (at least) one heap allocation and free per iteration of the loop.

<deque>

`std::deque` is, in some senses, a generalized version of `std::vector`. Like `std::vector`, it provides constant time random access and other similar properties, but it also provides efficient access to the front of the list. It does not guarantee continuity of elements within memory.

In exchange for this extra flexibility, `std::deque` has significantly higher constant factor costs than `std::vector`. If possible, use `std::vector` or something cheaper.

<list>

`std::list` is an extremely inefficient class that is rarely useful. It performs a heap allocation for every element inserted into it, thus having an extremely high constant factor, particularly for small data types. `std::list` also only supports bidirectional iteration, not random access iteration.

In exchange for this high cost, std::list supports efficient access to both ends of the list (like `std::deque`, but unlike `std::vector` or `SmallVector`). In addition, the iterator invalidation characteristics of std::list are stronger than that of a vector class: inserting or removing an element into the list does not invalidate iterator or pointers to other elements in the list.

llvm/ADT/ilist.h

`ilist<T>` implements an 'intrusive' doubly-linked list. It is intrusive, because it requires the element to store and provide access to the prev/next pointers for the list.

`ilist` has the same drawbacks as `std::list`, and additionally requires an `ilist_traits` implementation for the element type, but it provides some novel characteristics. In particular, it can efficiently store polymorphic objects, the traits class is informed when an element is inserted or removed from the list, and `ilist`s are guaranteed to support a constant-time splice operation.

These properties are exactly what we want for things like `Instruction`s and basic blocks, which is why these are implemented with `ilist`s.

Related classes of interest are explained in the following subsections:

- *ilist_traits*
- *iplist*
- *llvm/ADT/ilist_node.h*
- *Sentinels*

llvm/ADT/PackedVector.h

Useful for storing a vector of values using only a few number of bits for each value. Apart from the standard operations of a vector-like container, it can also perform an 'or' set operation.

For example:

```
enum State {
    None = 0x0,
    FirstCondition = 0x1,
    SecondCondition = 0x2,
```

```
    Both = 0x3
};

State get() {
    PackedVector<State, 2> Vec1;
    Vec1.push_back(FirstCondition);

    PackedVector<State, 2> Vec2;
    Vec2.push_back(SecondCondition);

    Vec1 |= Vec2;
    return Vec1[0]; // returns 'Both'.
}
```

ilist_traits

`ilist_traits<T>` is `ilist<T>`'s customization mechanism. `iplist<T>` (and consequently `ilist<T>`) publicly derive from this traits class.

iplist

`iplist<T>` is `ilist<T>`'s base and as such supports a slightly narrower interface. Notably, inserters from `T&` are absent.

`ilist_traits<T>` is a public base of this class and can be used for a wide variety of customizations.

llvm/ADT/ilist_node.h

`ilist_node<T>` implements the forward and backward links that are expected by the `ilist<T>` (and analogous containers) in the default manner.

`ilist_node<T>`s are meant to be embedded in the node type `T`, usually `T` publicly derives from `ilist_node<T>`.

Sentinels

`ilists` have another specialty that must be considered. To be a good citizen in the C++ ecosystem, it needs to support the standard container operations, such as `begin` and `end` iterators, etc. Also, the `operator--` must work correctly on the `end` iterator in the case of non-empty `ilists`.

The only sensible solution to this problem is to allocate a so-called *sentinel* along with the intrusive list, which serves as the `end` iterator, providing the back-link to the last element. However conforming to the C++ convention it is illegal to `operator++` beyond the sentinel and it also must not be dereferenced.

These constraints allow for some implementation freedom to the `ilist` how to allocate and store the sentinel. The corresponding policy is dictated by `ilist_traits<T>`. By default a `T` gets heap-allocated whenever the need for a sentinel arises.

While the default policy is sufficient in most cases, it may break down when `T` does not provide a default constructor. Also, in the case of many instances of `ilists`, the memory overhead of the associated sentinels is wasted. To alleviate the situation with numerous and voluminous `T`-sentinels, sometimes a trick is employed, leading to *ghostly sentinels*.

Ghostly sentinels are obtained by specially-crafted `ilist_traits<T>` which superpose the sentinel with the `ilist` instance in memory. Pointer arithmetic is used to obtain the sentinel, which is relative to the `ilist`'s `this`

pointer. The `ilist` is augmented by an extra pointer, which serves as the back-link of the sentinel. This is the only field in the ghostly sentinel which can be legally accessed.

Other Sequential Container options

Other STL containers are available, such as `std::string`.

There are also various STL adapter classes such as `std::queue`, `std::priority_queue`, `std::stack`, etc. These provide simplified access to an underlying container but don't affect the cost of the container itself.

String-like containers

There are a variety of ways to pass around and use strings in C and C++, and LLVM adds a few new options to choose from. Pick the first option on this list that will do what you need, they are ordered according to their relative cost.

Note that is is generally preferred to *not* pass strings around as `const char*`'s. These have a number of problems, including the fact that they cannot represent embedded nul ("0") characters, and do not have a length available efficiently. The general replacement for '`const char*`' is StringRef.

For more information on choosing string containers for APIs, please see *Passing Strings*.

llvm/ADT/StringRef.h

The StringRef class is a simple value class that contains a pointer to a character and a length, and is quite related to the *ArrayRef* class (but specialized for arrays of characters). Because StringRef carries a length with it, it safely handles strings with embedded nul characters in it, getting the length does not require a strlen call, and it even has very convenient APIs for slicing and dicing the character range that it represents.

StringRef is ideal for passing simple strings around that are known to be live, either because they are C string literals, std::string, a C array, or a SmallVector. Each of these cases has an efficient implicit conversion to StringRef, which doesn't result in a dynamic strlen being executed.

StringRef has a few major limitations which make more powerful string containers useful:

1. You cannot directly convert a StringRef to a 'const char*' because there is no way to add a trailing nul (unlike the .c_str() method on various stronger classes).
2. StringRef doesn't own or keep alive the underlying string bytes. As such it can easily lead to dangling pointers, and is not suitable for embedding in datastructures in most cases (instead, use an std::string or something like that).
3. For the same reason, StringRef cannot be used as the return value of a method if the method "computes" the result string. Instead, use std::string.
4. StringRef's do not allow you to mutate the pointed-to string bytes and it doesn't allow you to insert or remove bytes from the range. For editing operations like this, it interoperates with the *Twine* class.

Because of its strengths and limitations, it is very common for a function to take a StringRef and for a method on an object to return a StringRef that points into some string that it owns.

llvm/ADT/Twine.h

The Twine class is used as an intermediary datatype for APIs that want to take a string that can be constructed inline with a series of concatenations. Twine works by forming recursive instances of the Twine datatype (a simple value object) on the stack as temporary objects, linking them together into a tree which is then linearized when the Twine is consumed. Twine is only safe to use as the argument to a function, and should always be a const reference, e.g.:

```
void foo(const Twine &T);
...
StringRef X = ...
unsigned i = ...
foo(X + "." + Twine(i));
```

This example forms a string like "blarg.42" by concatenating the values together, and does not form intermediate strings containing "blarg" or "blarg.".

Because Twine is constructed with temporary objects on the stack, and because these instances are destroyed at the end of the current statement, it is an inherently dangerous API. For example, this simple variant contains undefined behavior and will probably crash:

```
void foo(const Twine &T);
...
StringRef X = ...
unsigned i = ...
const Twine &Tmp = X + "." + Twine(i);
foo(Tmp);
```

... because the temporaries are destroyed before the call. That said, Twine's are much more efficient than intermediate std::string temporaries, and they work really well with StringRef. Just be aware of their limitations.

llvm/ADT/SmallString.h

SmallString is a subclass of *SmallVector* that adds some convenience APIs like += that takes StringRef's. SmallString avoids allocating memory in the case when the preallocated space is enough to hold its data, and it calls back to general heap allocation when required. Since it owns its data, it is very safe to use and supports full mutation of the string.

Like SmallVector's, the big downside to SmallString is their sizeof. While they are optimized for small strings, they themselves are not particularly small. This means that they work great for temporary scratch buffers on the stack, but should not generally be put into the heap: it is very rare to see a SmallString as the member of a frequently-allocated heap data structure or returned by-value.

std::string

The standard C++ std::string class is a very general class that (like SmallString) owns its underlying data. sizeof(std::string) is very reasonable so it can be embedded into heap data structures and returned by-value. On the other hand, std::string is highly inefficient for inline editing (e.g. concatenating a bunch of stuff together) and because it is provided by the standard library, its performance characteristics depend a lot of the host standard library (e.g. libc++ and MSVC provide a highly optimized string class, GCC contains a really slow implementation).

The major disadvantage of std::string is that almost every operation that makes them larger can allocate memory, which is slow. As such, it is better to use SmallVector or Twine as a scratch buffer, but then use std::string to persist the result.

Set-Like Containers (std::set, SmallSet, SetVector, etc)

Set-like containers are useful when you need to canonicalize multiple values into a single representation. There are several different choices for how to do this, providing various trade-offs.

A sorted 'vector'

If you intend to insert a lot of elements, then do a lot of queries, a great approach is to use a vector (or other sequential container) with std::sort+std::unique to remove duplicates. This approach works really well if your usage pattern has these two distinct phases (insert then query), and can be coupled with a good choice of *sequential container*.

This combination provides the several nice properties: the result data is contiguous in memory (good for cache locality), has few allocations, is easy to address (iterators in the final vector are just indices or pointers), and can be efficiently queried with a standard binary search (e.g. `std::lower_bound`; if you want the whole range of elements comparing equal, use `std::equal_range`).

llvm/ADT/SmallSet.h

If you have a set-like data structure that is usually small and whose elements are reasonably small, a `SmallSet<Type, N>` is a good choice. This set has space for N elements in place (thus, if the set is dynamically smaller than N, no malloc traffic is required) and accesses them with a simple linear search. When the set grows beyond 'N' elements, it allocates a more expensive representation that guarantees efficient access (for most types, it falls back to std::set, but for pointers it uses something far better, *SmallPtrSet*.

The magic of this class is that it handles small sets extremely efficiently, but gracefully handles extremely large sets without loss of efficiency. The drawback is that the interface is quite small: it supports insertion, queries and erasing, but does not support iteration.

llvm/ADT/SmallPtrSet.h

SmallPtrSet has all the advantages of `SmallSet` (and a `SmallSet` of pointers is transparently implemented with a `SmallPtrSet`), but also supports iterators. If more than 'N' insertions are performed, a single quadratically probed hash table is allocated and grows as needed, providing extremely efficient access (constant time insertion/deleting/queries with low constant factors) and is very stingy with malloc traffic.

Note that, unlike `std::set`, the iterators of `SmallPtrSet` are invalidated whenever an insertion occurs. Also, the values visited by the iterators are not visited in sorted order.

llvm/ADT/DenseSet.h

DenseSet is a simple quadratically probed hash table. It excels at supporting small values: it uses a single allocation to hold all of the pairs that are currently inserted in the set. DenseSet is a great way to unique small values that are not simple pointers (use *SmallPtrSet* for pointers). Note that DenseSet has the same requirements for the value type that *DenseMap* has.

llvm/ADT/SparseSet.h

SparseSet holds a small number of objects identified by unsigned keys of moderate size. It uses a lot of memory, but provides operations that are almost as fast as a vector. Typical keys are physical registers, virtual registers, or numbered basic blocks.

SparseSet is useful for algorithms that need very fast clear/find/insert/erase and fast iteration over small sets. It is not intended for building composite data structures.

llvm/ADT/SparseMultiSet.h

SparseMultiSet adds multiset behavior to SparseSet, while retaining SparseSet's desirable attributes. Like SparseSet, it typically uses a lot of memory, but provides operations that are almost as fast as a vector. Typical keys are physical registers, virtual registers, or numbered basic blocks.

SparseMultiSet is useful for algorithms that need very fast clear/find/insert/erase of the entire collection, and iteration over sets of elements sharing a key. It is often a more efficient choice than using composite data structures (e.g. vector-of-vectors, map-of-vectors). It is not intended for building composite data structures.

llvm/ADT/FoldingSet.h

FoldingSet is an aggregate class that is really good at uniquing expensive-to-create or polymorphic objects. It is a combination of a chained hash table with intrusive links (uniqued objects are required to inherit from FoldingSetNode) that uses *SmallVector* as part of its ID process.

Consider a case where you want to implement a "getOrCreateFoo" method for a complex object (for example, a node in the code generator). The client has a description of **what** it wants to generate (it knows the opcode and all the operands), but we don't want to 'new' a node, then try inserting it into a set only to find out it already exists, at which point we would have to delete it and return the node that already exists.

To support this style of client, FoldingSet perform a query with a FoldingSetNodeID (which wraps SmallVector) that can be used to describe the element that we want to query for. The query either returns the element matching the ID or it returns an opaque ID that indicates where insertion should take place. Construction of the ID usually does not require heap traffic.

Because FoldingSet uses intrusive links, it can support polymorphic objects in the set (for example, you can have SDNode instances mixed with LoadSDNodes). Because the elements are individually allocated, pointers to the elements are stable: inserting or removing elements does not invalidate any pointers to other elements.

<set>

`std::set` is a reasonable all-around set class, which is decent at many things but great at nothing. std::set allocates memory for each element inserted (thus it is very malloc intensive) and typically stores three pointers per element in the set (thus adding a large amount of per-element space overhead). It offers guaranteed log(n) performance, which is not particularly fast from a complexity standpoint (particularly if the elements of the set are expensive to compare, like strings), and has extremely high constant factors for lookup, insertion and removal.

The advantages of std::set are that its iterators are stable (deleting or inserting an element from the set does not affect iterators or pointers to other elements) and that iteration over the set is guaranteed to be in sorted order. If the elements in the set are large, then the relative overhead of the pointers and malloc traffic is not a big deal, but if the elements of the set are small, std::set is almost never a good choice.

llvm/ADT/SetVector.h

LLVM's `SetVector<Type>` is an adapter class that combines your choice of a set-like container along with a *Sequential Container* The important property that this provides is efficient insertion with uniquing (duplicate elements are ignored) with iteration support. It implements this by inserting elements into both a set-like container and the sequential container, using the set-like container for uniquing and the sequential container for iteration.

The difference between SetVector and other sets is that the order of iteration is guaranteed to match the order of insertion into the SetVector. This property is really important for things like sets of pointers. Because pointer values are non-deterministic (e.g. vary across runs of the program on different machines), iterating over the pointers in the set will not be in a well-defined order.

The drawback of SetVector is that it requires twice as much space as a normal set and has the sum of constant factors from the set-like container and the sequential container that it uses. Use it **only** if you need to iterate over the elements in a deterministic order. SetVector is also expensive to delete elements out of (linear time), unless you use its "pop_back" method, which is faster.

`SetVector` is an adapter class that defaults to using `std::vector` and a size 16 `SmallSet` for the underlying containers, so it is quite expensive. However, `"llvm/ADT/SetVector.h"` also provides a `SmallSetVector` class, which defaults to using a `SmallVector` and `SmallSet` of a specified size. If you use this, and if your sets are dynamically smaller than `N`, you will save a lot of heap traffic.

llvm/ADT/UniqueVector.h

UniqueVector is similar to *SetVector* but it retains a unique ID for each element inserted into the set. It internally contains a map and a vector, and it assigns a unique ID for each value inserted into the set.

UniqueVector is very expensive: its cost is the sum of the cost of maintaining both the map and vector, it has high complexity, high constant factors, and produces a lot of malloc traffic. It should be avoided.

llvm/ADT/ImmutableSet.h

ImmutableSet is an immutable (functional) set implementation based on an AVL tree. Adding or removing elements is done through a Factory object and results in the creation of a new ImmutableSet object. If an ImmutableSet already exists with the given contents, then the existing one is returned; equality is compared with a FoldingSetNodeID. The time and space complexity of add or remove operations is logarithmic in the size of the original set.

There is no method for returning an element of the set, you can only check for membership.

Other Set-Like Container Options

The STL provides several other options, such as std::multiset and the various "hash_set" like containers (whether from C++ TR1 or from the SGI library). We never use hash_set and unordered_set because they are generally very expensive (each insertion requires a malloc) and very non-portable.

std::multiset is useful if you're not interested in elimination of duplicates, but has all the drawbacks of std::set. A sorted vector (where you don't delete duplicate entries) or some other approach is almost always better.

Map-Like Containers (std::map, DenseMap, etc)

Map-like containers are useful when you want to associate data to a key. As usual, there are a lot of different ways to do this. :)

A sorted 'vector'

If your usage pattern follows a strict insert-then-query approach, you can trivially use the same approach as *sorted vectors for set-like containers*. The only difference is that your query function (which uses std::lower_bound to get efficient log(n) lookup) should only compare the key, not both the key and value. This yields the same advantages as sorted vectors for sets.

llvm/ADT/StringMap.h

Strings are commonly used as keys in maps, and they are difficult to support efficiently: they are variable length, inefficient to hash and compare when long, expensive to copy, etc. StringMap is a specialized container designed to cope with these issues. It supports mapping an arbitrary range of bytes to an arbitrary other object.

The StringMap implementation uses a quadratically-probed hash table, where the buckets store a pointer to the heap allocated entries (and some other stuff). The entries in the map must be heap allocated because the strings are variable length. The string data (key) and the element object (value) are stored in the same allocation with the string data immediately after the element object. This container guarantees the "`(char*)(&Value+1)`" points to the key string for a value.

The StringMap is very fast for several reasons: quadratic probing is very cache efficient for lookups, the hash value of strings in buckets is not recomputed when looking up an element, StringMap rarely has to touch the memory for unrelated objects when looking up a value (even when hash collisions happen), hash table growth does not recompute the hash values for strings already in the table, and each pair in the map is store in a single allocation (the string data is stored in the same allocation as the Value of a pair).

StringMap also provides query methods that take byte ranges, so it only ever copies a string if a value is inserted into the table.

StringMap iteratation order, however, is not guaranteed to be deterministic, so any uses which require that should instead use a std::map.

llvm/ADT/IndexedMap.h

IndexedMap is a specialized container for mapping small dense integers (or values that can be mapped to small dense integers) to some other type. It is internally implemented as a vector with a mapping function that maps the keys to the dense integer range.

This is useful for cases like virtual registers in the LLVM code generator: they have a dense mapping that is offset by a compile-time constant (the first virtual register ID).

llvm/ADT/DenseMap.h

DenseMap is a simple quadratically probed hash table. It excels at supporting small keys and values: it uses a single allocation to hold all of the pairs that are currently inserted in the map. DenseMap is a great way to map pointers to pointers, or map other small types to each other.

There are several aspects of DenseMap that you should be aware of, however. The iterators in a DenseMap are invalidated whenever an insertion occurs, unlike map. Also, because DenseMap allocates space for a large number of key/value pairs (it starts with 64 by default), it will waste a lot of space if your keys or values are large. Finally, you must implement a partial specialization of DenseMapInfo for the key that you want, if it isn't already supported. This is required to tell DenseMap about two special marker values (which can never be inserted into the map) that it needs internally.

DenseMap's find_as() method supports lookup operations using an alternate key type. This is useful in cases where the normal key type is expensive to construct, but cheap to compare against. The DenseMapInfo is responsible for defining the appropriate comparison and hashing methods for each alternate key type used.

llvm/IR/ValueMap.h

ValueMap is a wrapper around a *DenseMap* mapping `Value*`s (or subclasses) to another type. When a Value is deleted or RAUW'ed, ValueMap will update itself so the new version of the key is mapped to the same value, just as

if the key were a WeakVH. You can configure exactly how this happens, and what else happens on these two events, by passing a `Config` parameter to the ValueMap template.

llvm/ADT/IntervalMap.h

IntervalMap is a compact map for small keys and values. It maps key intervals instead of single keys, and it will automatically coalesce adjacent intervals. When then map only contains a few intervals, they are stored in the map object itself to avoid allocations.

The IntervalMap iterators are quite big, so they should not be passed around as STL iterators. The heavyweight iterators allow a smaller data structure.

<map>

std::map has similar characteristics to *std::set*: it uses a single allocation per pair inserted into the map, it offers log(n) lookup with an extremely large constant factor, imposes a space penalty of 3 pointers per pair in the map, etc.

std::map is most useful when your keys or values are very large, if you need to iterate over the collection in sorted order, or if you need stable iterators into the map (i.e. they don't get invalidated if an insertion or deletion of another element takes place).

llvm/ADT/MapVector.h

`MapVector<KeyT,ValueT>` provides a subset of the DenseMap interface. The main difference is that the iteration order is guaranteed to be the insertion order, making it an easy (but somewhat expensive) solution for non-deterministic iteration over maps of pointers.

It is implemented by mapping from key to an index in a vector of key,value pairs. This provides fast lookup and iteration, but has two main drawbacks: the key is stored twice and removing elements takes linear time. If it is necessary to remove elements, it's best to remove them in bulk using `remove_if()`.

llvm/ADT/IntEqClasses.h

IntEqClasses provides a compact representation of equivalence classes of small integers. Initially, each integer in the range 0..n-1 has its own equivalence class. Classes can be joined by passing two class representatives to the join(a, b) method. Two integers are in the same class when findLeader() returns the same representative.

Once all equivalence classes are formed, the map can be compressed so each integer 0..n-1 maps to an equivalence class number in the range 0..m-1, where m is the total number of equivalence classes. The map must be uncompressed before it can be edited again.

llvm/ADT/ImmutableMap.h

ImmutableMap is an immutable (functional) map implementation based on an AVL tree. Adding or removing elements is done through a Factory object and results in the creation of a new ImmutableMap object. If an ImmutableMap already exists with the given key set, then the existing one is returned; equality is compared with a FoldingSetNodeID. The time and space complexity of add or remove operations is logarithmic in the size of the original map.

Other Map-Like Container Options

The STL provides several other options, such as std::multimap and the various "hash_map" like containers (whether from C++ TR1 or from the SGI library). We never use hash_set and unordered_set because they are generally very expensive (each insertion requires a malloc) and very non-portable.

std::multimap is useful if you want to map a key to multiple values, but has all the drawbacks of std::map. A sorted vector or some other approach is almost always better.

Bit storage containers (BitVector, SparseBitVector)

Unlike the other containers, there are only two bit storage containers, and choosing when to use each is relatively straightforward.

One additional option is `std::vector<bool>`: we discourage its use for two reasons 1) the implementation in many common compilers (e.g. commonly available versions of GCC) is extremely inefficient and 2) the C++ standards committee is likely to deprecate this container and/or change it significantly somehow. In any case, please don't use it.

BitVector

The BitVector container provides a dynamic size set of bits for manipulation. It supports individual bit setting/testing, as well as set operations. The set operations take time O(size of bitvector), but operations are performed one word at a time, instead of one bit at a time. This makes the BitVector very fast for set operations compared to other containers. Use the BitVector when you expect the number of set bits to be high (i.e. a dense set).

SmallBitVector

The SmallBitVector container provides the same interface as BitVector, but it is optimized for the case where only a small number of bits, less than 25 or so, are needed. It also transparently supports larger bit counts, but slightly less efficiently than a plain BitVector, so SmallBitVector should only be used when larger counts are rare.

At this time, SmallBitVector does not support set operations (and, or, xor), and its operator[] does not provide an assignable lvalue.

SparseBitVector

The SparseBitVector container is much like BitVector, with one major difference: Only the bits that are set, are stored. This makes the SparseBitVector much more space efficient than BitVector when the set is sparse, as well as making set operations O(number of set bits) instead of O(size of universe). The downside to the SparseBitVector is that setting and testing of random bits is O(N), and on large SparseBitVectors, this can be slower than BitVector. In our implementation, setting or testing bits in sorted order (either forwards or reverse) is O(1) worst case. Testing and setting bits within 128 bits (depends on size) of the current bit is also O(1). As a general statement, testing/setting bits in a SparseBitVector is O(distance away from last set bit).

3.7.5 Helpful Hints for Common Operations

This section describes how to perform some very simple transformations of LLVM code. This is meant to give examples of common idioms used, showing the practical side of LLVM transformations.

Because this is a "how-to" section, you should also read about the main classes that you will be working with. The *Core LLVM Class Hierarchy Reference* contains details and descriptions of the main classes that you should know about.

Basic Inspection and Traversal Routines

The LLVM compiler infrastructure have many different data structures that may be traversed. Following the example of the C++ standard template library, the techniques used to traverse these various data structures are all basically the same. For a enumerable sequence of values, the `XXXbegin()` function (or method) returns an iterator to the start of the sequence, the `XXXend()` function returns an iterator pointing to one past the last valid element of the sequence, and there is some `XXXiterator` data type that is common between the two operations.

Because the pattern for iteration is common across many different aspects of the program representation, the standard template library algorithms may be used on them, and it is easier to remember how to iterate. First we show a few common examples of the data structures that need to be traversed. Other data structures are traversed in very similar ways.

Iterating over the `BasicBlock` in a `Function`

It's quite common to have a `Function` instance that you'd like to transform in some way; in particular, you'd like to manipulate its `BasicBlock`s. To facilitate this, you'll need to iterate over all of the `BasicBlock`s that constitute the `Function`. The following is an example that prints the name of a `BasicBlock` and the number of `Instruction`s it contains:

```
// func is a pointer to a Function instance
for (Function::iterator i = func->begin(), e = func->end(); i != e; ++i)
  // Print out the name of the basic block if it has one, and then the
  // number of instructions that it contains
  errs() << "Basic block (name=" << i->getName() << ") has "
             << i->size() << " instructions.\n";
```

Note that i can be used as if it were a pointer for the purposes of invoking member functions of the `Instruction` class. This is because the indirection operator is overloaded for the iterator classes. In the above code, the expression `i->size()` is exactly equivalent to `(*i).size()` just like you'd expect.

Iterating over the `Instruction` in a `BasicBlock`

Just like when dealing with `BasicBlock`s in `Function`s, it's easy to iterate over the individual instructions that make up `BasicBlock`s. Here's a code snippet that prints out each instruction in a `BasicBlock`:

```
// blk is a pointer to a BasicBlock instance
for (BasicBlock::iterator i = blk->begin(), e = blk->end(); i != e; ++i)
   // The next statement works since operator<<(ostream&,...)
   // is overloaded for Instruction&
   errs() << *i << "\n";
```

However, this isn't really the best way to print out the contents of a `BasicBlock`! Since the ostream operators are overloaded for virtually anything you'll care about, you could have just invoked the print routine on the basic block itself: `errs() << *blk << "\n";`.

Iterating over the `Instruction` in a `Function`

If you're finding that you commonly iterate over a `Function`'s `BasicBlock`s and then that `BasicBlock`'s `Instruction`s, `InstIterator` should be used instead. You'll need to include `llvm/IR/InstIterator.h` (doxygen) and then instantiate `InstIterator`s explicitly in your code. Here's a small example that shows how to dump all instructions in a function to the standard error stream:

```
#include "llvm/IR/InstIterator.h"

// F is a pointer to a Function instance
for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I)
  errs() << *I << "\n";
```

Easy, isn't it? You can also use `InstIterators` to fill a work list with its initial contents. For example, if you wanted to initialize a work list to contain all instructions in a `Function` F, all you would need to do is something like:

```
std::set<Instruction*> worklist;
// or better yet, SmallPtrSet<Instruction*, 64> worklist;

for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I)
  worklist.insert(&*I);
```

The STL set `worklist` would now contain all instructions in the `Function` pointed to by F.

Turning an iterator into a class pointer (and vice-versa)

Sometimes, it'll be useful to grab a reference (or pointer) to a class instance when all you've got at hand is an iterator. Well, extracting a reference or a pointer from an iterator is very straight-forward. Assuming that `i` is a `BasicBlock::iterator` and `j` is a `BasicBlock::const_iterator`:

```
Instruction& inst = *i;   // Grab reference to instruction reference
Instruction* pinst = &*i; // Grab pointer to instruction reference
const Instruction& inst = *j;
```

However, the iterators you'll be working with in the LLVM framework are special: they will automatically convert to a ptr-to-instance type whenever they need to. Instead of derferencing the iterator and then taking the address of the result, you can simply assign the iterator to the proper pointer type and you get the dereference and address-of operation as a result of the assignment (behind the scenes, this is a result of overloading casting mechanisms). Thus the last line of the last example,

```
Instruction *pinst = &*i;
```

is semantically equivalent to

```
Instruction *pinst = i;
```

It's also possible to turn a class pointer into the corresponding iterator, and this is a constant time operation (very efficient). The following code snippet illustrates use of the conversion constructors provided by LLVM iterators. By using these, you can explicitly grab the iterator of something without actually obtaining it via iteration over some structure:

```
void printNextInstruction(Instruction* inst) {
  BasicBlock::iterator it(inst);
  ++it; // After this line, it refers to the instruction after *inst
  if (it != inst->getParent()->end()) errs() << *it << "\n";
}
```

Unfortunately, these implicit conversions come at a cost; they prevent these iterators from conforming to standard iterator conventions, and thus from being usable with standard algorithms and containers. For example, they prevent the following code, where `B` is a `BasicBlock`, from compiling:

```
llvm::SmallVector<llvm::Instruction *, 16>(B->begin(), B->end());
```

Because of this, these implicit conversions may be removed some day, and `operator*` changed to return a pointer instead of a reference.

Finding call sites: a slightly more complex example

Say that you're writing a FunctionPass and would like to count all the locations in the entire module (that is, across every `Function`) where a certain function (i.e., some `Function *`) is already in scope. As you'll learn later, you may want to use an `InstVisitor` to accomplish this in a much more straight-forward manner, but this example will allow us to explore how you'd do it if you didn't have `InstVisitor` around. In pseudo-code, this is what we want to do:

```
initialize callCounter to zero
for each Function f in the Module
  for each BasicBlock b in f
    for each Instruction i in b
      if (i is a CallInst and calls the given function)
        increment callCounter
```

And the actual code is (remember, because we're writing a `FunctionPass`, our `FunctionPass`-derived class simply has to override the `runOnFunction` method):

```
Function* targetFunc = ...;

class OurFunctionPass : public FunctionPass {
  public:
    OurFunctionPass(): callCounter(0) { }

    virtual runOnFunction(Function& F) {
      for (Function::iterator b = F.begin(), be = F.end(); b != be; ++b) {
        for (BasicBlock::iterator i = b->begin(), ie = b->end(); i != ie; ++i) {
          if (CallInst* callInst = dyn_cast<CallInst>(&*i)) {
            // We know we've encountered a call instruction, so we
            // need to determine if it's a call to the
            // function pointed to by m_func or not.
            if (callInst->getCalledFunction() == targetFunc)
              ++callCounter;
          }
        }
      }
    }

  private:
    unsigned callCounter;
};
```

Treating calls and invokes the same way

You may have noticed that the previous example was a bit oversimplified in that it did not deal with call sites generated by 'invoke' instructions. In this, and in other situations, you may find that you want to treat `CallInsts` and `InvokeInsts` the same way, even though their most-specific common base class is `Instruction`, which includes lots of less closely-related things. For these cases, LLVM provides a handy wrapper class called `CallSite` (doxygen) It is essentially a wrapper around an `Instruction` pointer, with some methods that provide functionality common to `CallInsts` and `InvokeInsts`.

This class has "value semantics": it should be passed by value, not by reference and it should not be dynamically allocated or deallocated using `operator new` or `operator delete`. It is efficiently copyable, assignable and

constructable, with costs equivalents to that of a bare pointer. If you look at its definition, it has only a single pointer member.

Iterating over def-use & use-def chains

Frequently, we might have an instance of the `Value` class (doxygen) and we want to determine which `User` s use the `Value`. The list of all `Users` of a particular `Value` is called a *def-use* chain. For example, let's say we have a `Function*` named `F` to a particular function `foo`. Finding all of the instructions that *use* `foo` is as simple as iterating over the *def-use* chain of `F`:

```
Function *F = ...;

for (User *U : GV->users()) {
  if (Instruction *Inst = dyn_cast<Instruction>(U)) {
    errs() << "F is used in instruction:\n";
    errs() << *Inst << "\n";
  }
```

Alternatively, it's common to have an instance of the `User` Class (doxygen) and need to know what `Values` are used by it. The list of all `Values` used by a `User` is known as a *use-def* chain. Instances of class `Instruction` are common `User` s, so we might want to iterate over all of the values that a particular instruction uses (that is, the operands of the particular `Instruction`):

```
Instruction *pi = ...;

for (Use &U : pi->operands()) {
  Value *v = U.get();
  // ...
}
```

Declaring objects as `const` is an important tool of enforcing mutation free algorithms (such as analyses, etc.). For this purpose above iterators come in constant flavors as `Value::const_use_iterator` and `Value::const_op_iterator`. They automatically arise when calling `use/op_begin()` on `const Value*`s or `const User*`s respectively. Upon dereferencing, they return `const Use*`s. Otherwise the above patterns remain unchanged.

Iterating over predecessors & successors of blocks

Iterating over the predecessors and successors of a block is quite easy with the routines defined in `"llvm/Support/CFG.h"`. Just use code like this to iterate over all predecessors of BB:

```
#include "llvm/Support/CFG.h"
BasicBlock *BB = ...;

for (pred_iterator PI = pred_begin(BB), E = pred_end(BB); PI != E; ++PI) {
  BasicBlock *Pred = *PI;
  // ...
}
```

Similarly, to iterate over successors use `succ_iterator/succ_begin/succ_end`.

Making simple changes

There are some primitive transformation operations present in the LLVM infrastructure that are worth knowing about. When performing transformations, it's fairly common to manipulate the contents of basic blocks. This section de-

scribes some of the common methods for doing so and gives example code.

Creating and inserting new `Instruction`s

Instantiating Instructions

Creation of `Instruction`s is straight-forward: simply call the constructor for the kind of instruction to instantiate and provide the necessary parameters. For example, an `AllocaInst` only *requires* a (const-ptr-to) `Type`. Thus:

```
AllocaInst* ai = new AllocaInst(Type::Int32Ty);
```

will create an `AllocaInst` instance that represents the allocation of one integer in the current stack frame, at run time. Each `Instruction` subclass is likely to have varying default parameters which change the semantics of the instruction, so refer to the doxygen documentation for the subclass of Instruction that you're interested in instantiating.

Naming values

It is very useful to name the values of instructions when you're able to, as this facilitates the debugging of your transformations. If you end up looking at generated LLVM machine code, you definitely want to have logical names associated with the results of instructions! By supplying a value for the `Name` (default) parameter of the `Instruction` constructor, you associate a logical name with the result of the instruction's execution at run time. For example, say that I'm writing a transformation that dynamically allocates space for an integer on the stack, and that integer is going to be used as some kind of index by some other code. To accomplish this, I place an `AllocaInst` at the first point in the first `BasicBlock` of some `Function`, and I'm intending to use it within the same `Function`. I might do:

```
AllocaInst* pa = new AllocaInst(Type::Int32Ty, 0, "indexLoc");
```

where `indexLoc` is now the logical name of the instruction's execution value, which is a pointer to an integer on the run time stack.

Inserting instructions

There are essentially three ways to insert an `Instruction` into an existing sequence of instructions that form a `BasicBlock`:

- Insertion into an explicit instruction list

  Given a `BasicBlock* pb`, an `Instruction* pi` within that `BasicBlock`, and a newly-created instruction we wish to insert before `*pi`, we do the following:

  ```
  BasicBlock *pb = ...;
  Instruction *pi = ...;
  Instruction *newInst = new Instruction(...);

  pb->getInstList().insert(pi, newInst); // Inserts newInst before pi in pb
  ```

  Appending to the end of a `BasicBlock` is so common that the `Instruction` class and `Instruction`-derived classes provide constructors which take a pointer to a `BasicBlock` to be appended to. For example code that looked like:

  ```
  BasicBlock *pb = ...;
  Instruction *newInst = new Instruction(...);

  pb->getInstList().push_back(newInst); // Appends newInst to pb
  ```

  becomes:

  ```
  BasicBlock *pb = ...;
  Instruction *newInst = new Instruction(..., pb);
  ```

which is much cleaner, especially if you are creating long instruction streams.

- Insertion into an implicit instruction list

  `Instruction` instances that are already in `BasicBlock`s are implicitly associated with an existing instruction list: the instruction list of the enclosing basic block. Thus, we could have accomplished the same thing as the above code without being given a `BasicBlock` by doing:

  ```
  Instruction *pi = ...;
  Instruction *newInst = new Instruction(...);

  pi->getParent()->getInstList().insert(pi, newInst);
  ```

  In fact, this sequence of steps occurs so frequently that the `Instruction` class and `Instruction`-derived classes provide constructors which take (as a default parameter) a pointer to an `Instruction` which the newly-created `Instruction` should precede. That is, `Instruction` constructors are capable of inserting the newly-created instance into the `BasicBlock` of a provided instruction, immediately before that instruction. Using an `Instruction` constructor with a `insertBefore` (default) parameter, the above code becomes:

  ```
  Instruction* pi = ...;
  Instruction* newInst = new Instruction(..., pi);
  ```

  which is much cleaner, especially if you're creating a lot of instructions and adding them to `BasicBlock`s.

- Insertion using an instance of `IRBuilder`

  Inserting several `Instruction`s can be quite laborious using the previous methods. The `IRBuilder` is a convenience class that can be used to add several instructions to the end of a `BasicBlock` or before a particular `Instruction`. It also supports constant folding and renaming named registers (see `IRBuilder`'s template arguments).

  The example below demonstrates a very simple use of the `IRBuilder` where three instructions are inserted before the instruction `pi`. The first two instructions are Call instructions and third instruction multiplies the return value of the two calls.

  ```
  Instruction *pi = ...;
  IRBuilder<> Builder(pi);
  CallInst* callOne = Builder.CreateCall(...);
  CallInst* callTwo = Builder.CreateCall(...);
  Value* result = Builder.CreateMul(callOne, callTwo);
  ```

  The example below is similar to the above example except that the created `IRBuilder` inserts instructions at the end of the `BasicBlock` `pb`.

  ```
  BasicBlock *pb = ...;
  IRBuilder<> Builder(pb);
  CallInst* callOne = Builder.CreateCall(...);
  CallInst* callTwo = Builder.CreateCall(...);
  Value* result = Builder.CreateMul(callOne, callTwo);
  ```

  See *Kaleidoscope: Code generation to LLVM IR* for a practical use of the `IRBuilder`.

Deleting Instructions

Deleting an instruction from an existing sequence of instructions that form a BasicBlock is very straight-forward: just call the instruction's `eraseFromParent()` method. For example:

```
Instruction *I = .. ;
I->eraseFromParent();
```

This unlinks the instruction from its containing basic block and deletes it. If you'd just like to unlink the instruction from its containing basic block but not delete it, you can use the `removeFromParent()` method.

Replacing an Instruction with another Value

Replacing individual instructions Including "llvm/Transforms/Utils/BasicBlockUtils.h" permits use of two very useful replace functions: `ReplaceInstWithValue` and `ReplaceInstWithInst`.

Deleting Instructions

- `ReplaceInstWithValue`

  This function replaces all uses of a given instruction with a value, and then removes the original instruction. The following example illustrates the replacement of the result of a particular `AllocaInst` that allocates memory for a single integer with a null pointer to an integer.

```
AllocaInst* instToReplace = ...;
BasicBlock::iterator ii(instToReplace);

ReplaceInstWithValue(instToReplace->getParent()->getInstList(), ii,
                     Constant::getNullValue(PointerType::getUnqual(Type::Int32Ty)));
```

- `ReplaceInstWithInst`

  This function replaces a particular instruction with another instruction, inserting the new instruction into the basic block at the location where the old instruction was, and replacing any uses of the old instruction with the new instruction. The following example illustrates the replacement of one `AllocaInst` with another.

```
AllocaInst* instToReplace = ...;
BasicBlock::iterator ii(instToReplace);

ReplaceInstWithInst(instToReplace->getParent()->getInstList(), ii,
                    new AllocaInst(Type::Int32Ty, 0, "ptrToReplacedInt"));
```

Replacing multiple uses of Users and Values You can use `Value::replaceAllUsesWith` and `User::replaceUsesOfWith` to change more than one use at a time. See the doxygen documentation for the Value Class and User Class, respectively, for more information.

Deleting GlobalVariables

Deleting a global variable from a module is just as easy as deleting an Instruction. First, you must have a pointer to the global variable that you wish to delete. You use this pointer to erase it from its parent, the module. For example:

```
GlobalVariable *GV = .. ;

GV->eraseFromParent();
```

How to Create Types

In generating IR, you may need some complex types. If you know these types statically, you can use `TypeBuilder<...>::get()`, defined in `llvm/Support/TypeBuilder.h`, to retrieve them. `TypeBuilder` has two forms depending on whether you're building types for cross-compilation or native library use. `TypeBuilder<T, true>` requires that `T` be independent of the host environment, meaning that it's built

out of types from the `llvm::types` (doxygen) namespace and pointers, functions, arrays, etc. built of those. `TypeBuilder<T, false>` additionally allows native C types whose size may depend on the host compiler. For example,

```
FunctionType *ft = TypeBuilder<types::i<8>(types::i<32>*), true>::get();
```

is easier to read and write than the equivalent

```
std::vector<const Type*> params;
params.push_back(PointerType::getUnqual(Type::Int32Ty));
FunctionType *ft = FunctionType::get(Type::Int8Ty, params, false);
```

See the class comment for more details.

3.7.6 Threads and LLVM

This section describes the interaction of the LLVM APIs with multithreading, both on the part of client applications, and in the JIT, in the hosted application.

Note that LLVM's support for multithreading is still relatively young. Up through version 2.5, the execution of threaded hosted applications was supported, but not threaded client access to the APIs. While this use case is now supported, clients *must* adhere to the guidelines specified below to ensure proper operation in multithreaded mode.

Note that, on Unix-like platforms, LLVM requires the presence of GCC's atomic intrinsics in order to support threaded operation. If you need a multhreading-capable LLVM on a platform without a suitably modern system compiler, consider compiling LLVM and LLVM-GCC in single-threaded mode, and using the resultant compiler to build a copy of LLVM with multithreading support.

Ending Execution with `llvm_shutdown()`

When you are done using the LLVM APIs, you should call `llvm_shutdown()` to deallocate memory used for internal structures.

Lazy Initialization with `ManagedStatic`

`ManagedStatic` is a utility class in LLVM used to implement static initialization of static resources, such as the global type tables. In a single-threaded environment, it implements a simple lazy initialization scheme. When LLVM is compiled with support for multi-threading, however, it uses double-checked locking to implement thread-safe lazy initialization.

Achieving Isolation with `LLVMContext`

`LLVMContext` is an opaque class in the LLVM API which clients can use to operate multiple, isolated instances of LLVM concurrently within the same address space. For instance, in a hypothetical compile-server, the compilation of an individual translation unit is conceptually independent from all the others, and it would be desirable to be able to compile incoming translation units concurrently on independent server threads. Fortunately, `LLVMContext` exists to enable just this kind of scenario!

Conceptually, `LLVMContext` provides isolation. Every LLVM entity (`Modules`, `Values`, `Types`, `Constants`, etc.) in LLVM's in-memory IR belongs to an `LLVMContext`. Entities in different contexts *cannot* interact with each other: `Modules` in different contexts cannot be linked together, `Functions` cannot be added to `Modules` in different contexts, etc. What this means is that is is safe to compile on multiple threads simultaneously, as long as no two threads operate on entities within the same context.

In practice, very few places in the API require the explicit specification of a `LLVMContext`, other than the `Type` creation/lookup APIs. Because every `Type` carries a reference to its owning context, most other entities can determine what context they belong to by looking at their own `Type`. If you are adding new entities to LLVM IR, please try to maintain this interface design.

For clients that do *not* require the benefits of isolation, LLVM provides a convenience API `getGlobalContext()`. This returns a global, lazily initialized `LLVMContext` that may be used in situations where isolation is not a concern.

Threads and the JIT

LLVM's "eager" JIT compiler is safe to use in threaded programs. Multiple threads can call `ExecutionEngine::getPointerToFunction()` or `ExecutionEngine::runFunction()` concurrently, and multiple threads can run code output by the JIT concurrently. The user must still ensure that only one thread accesses IR in a given `LLVMContext` while another thread might be modifying it. One way to do that is to always hold the JIT lock while accessing IR outside the JIT (the JIT *modifies* the IR by adding `CallbackVH`s). Another way is to only call `getPointerToFunction()` from the `LLVMContext`'s thread.

When the JIT is configured to compile lazily (using `ExecutionEngine::DisableLazyCompilation(false)`), there is currently a race condition in updating call sites after a function is lazily-jitted. It's still possible to use the lazy JIT in a threaded program if you ensure that only one thread at a time can call any particular lazy stub and that the JIT lock guards any IR access, but we suggest using only the eager JIT in threaded programs.

3.7.7 Advanced Topics

This section describes some of the advanced or obscure API's that most clients do not need to be aware of. These API's tend manage the inner workings of the LLVM system, and only need to be accessed in unusual circumstances.

The `ValueSymbolTable` class

The `ValueSymbolTable` (doxygen) class provides a symbol table that the *Function* and Module classes use for naming value definitions. The symbol table can provide a name for any Value.

Note that the `SymbolTable` class should not be directly accessed by most clients. It should only be used when iteration over the symbol table names themselves are required, which is very special purpose. Note that not all LLVM Values have names, and those without names (i.e. they have an empty name) do not exist in the symbol table.

Symbol tables support iteration over the values in the symbol table with `begin/end/iterator` and supports querying to see if a specific name is in the symbol table (with `lookup`). The `ValueSymbolTable` class exposes no public mutator methods, instead, simply call `setName` on a value, which will autoinsert it into the appropriate symbol table.

The `User` and owned `Use` classes' memory layout

The `User` (doxygen) class provides a basis for expressing the ownership of `User` towards other Value instances. The `Use` (doxygen) helper class is employed to do the bookkeeping and to facilitate *O(1)* addition and removal.

Interaction and relationship between `User` and `Use` objects

A subclass of `User` can choose between incorporating its `Use` objects or refer to them out-of-line by means of a pointer. A mixed variant (some `Use` s inline others hung off) is impractical and breaks the invariant that the `Use` objects belonging to the same `User` form a contiguous array.

We have 2 different layouts in the `User` (sub)classes:

- Layout a)

  The `Use` object(s) are inside (resp. at fixed offset) of the `User` object and there are a fixed number of them.

- Layout b)

  The `Use` object(s) are referenced by a pointer to an array from the `User` object and there may be a variable number of them.

As of v2.4 each layout still possesses a direct pointer to the start of the array of `Use`s. Though not mandatory for layout a), we stick to this redundancy for the sake of simplicity. The `User` object also stores the number of `Use` objects it has. (Theoretically this information can also be calculated given the scheme presented below.)

Special forms of allocation operators (`operator new`) enforce the following memory layouts:

- Layout a) is modelled by prepending the `User` object by the `Use[]` array.

```
...---.---.---.---.-------...
  | P | P | P | P | User
'''---'---'---'---'-------'''
```

- Layout b) is modelled by pointing at the `Use[]` array.

```
.-------...
| User
'-------'''
    |
    v
    .---.---.---.---...
    | P | P | P | P |
    '---'---'---'---'''
```

(In the above figures '`P`' stands for the `Use**` that is stored in each `Use` object in the member `Use::Prev`)

The waymarking algorithm

Since the `Use` objects are deprived of the direct (back)pointer to their `User` objects, there must be a fast and exact method to recover it. This is accomplished by the following scheme:

A bit-encoding in the 2 LSBits (least significant bits) of the `Use::Prev` allows to find the start of the `User` object:

- `00` — binary digit 0
- `01` — binary digit 1
- `10` — stop and calculate (`s`)
- `11` — full stop (`S`)

Given a `Use*`, all we have to do is to walk till we get a stop and we either have a `User` immediately behind or we have to walk to the next stop picking up digits and calculating the offset:

```
.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.----------------
| 1 | s | 1 | 0 | 1 | 0 | s | 1 | 1 | 0 | s | 1 | 1 | s | 1 | S | User (or User*)
'---'---'---'---'---'---'---'---'---'---'---'---'---'---'---'---'----------------
    |+15               |+10            |+6         |+3     |+1
    |                  |               |           |       | __>
    |                  |               |           | ___________>
    |                  |               | _________________________>
    |                  | _____________________________________________>
    | ____________________________________________________________________>
```

Only the significant number of bits need to be stored between the stops, so that the *worst case is 20 memory accesses* when there are 1000 `Use` objects associated with a `User`.

Reference implementation

The following literate Haskell fragment demonstrates the concept:

```
> import Test.QuickCheck
>
> digits :: Int -> [Char] -> [Char]
> digits 0 acc = '0' : acc
> digits 1 acc = '1' : acc
> digits n acc = digits (n `div` 2) $ digits (n `mod` 2) acc
>
> dist :: Int -> [Char] -> [Char]
> dist 0 [] = ['S']
> dist 0 acc = acc
> dist 1 acc = let r = dist 0 acc in 's' : digits (length r) r
> dist n acc = dist (n - 1) $ dist 1 acc
>
> takeLast n ss = reverse $ take n $ reverse ss
>
> test = takeLast 40 $ dist 20 []
>
```

Printing <test> gives: `"1s100000s11010s10100s1111s1010s110s11s1S"`

The reverse algorithm computes the length of the string just by examining a certain prefix:

```
> pref :: [Char] -> Int
> pref "S" = 1
> pref ('s':'1':rest) = decode 2 1 rest
> pref (_:rest) = 1 + pref rest
>
> decode walk acc ('0':rest) = decode (walk + 1) (acc * 2) rest
> decode walk acc ('1':rest) = decode (walk + 1) (acc * 2 + 1) rest
> decode walk acc _ = walk + acc
>
```

Now, as expected, printing <pref test> gives `40`.

We can *quickCheck* this with following property:

```
> testcase = dist 2000 []
> testcaseLength = length testcase
>
> identityProp n = n > 0 && n <= testcaseLength ==> length arr == pref arr
>     where arr = takeLast n testcase
>
```

As expected <quickCheck identityProp> gives:

```
*Main> quickCheck identityProp
OK, passed 100 tests.
```

Let's be a bit more exhaustive:

```
>
> deepCheck p = check (defaultConfig { configMaxTest = 500 }) p
>
```

And here is the result of <deepCheck identityProp>:

```
*Main> deepCheck identityProp
OK, passed 500 tests.
```

Tagging considerations

To maintain the invariant that the 2 LSBits of each `Use**` in `Use` never change after being set up, setters of `Use::Prev` must re-tag the new `Use**` on every modification. Accordingly getters must strip the tag bits.

For layout b) instead of the `User` we find a pointer (`User*` with LSBit set). Following this pointer brings us to the `User`. A portable trick ensures that the first bytes of `User` (if interpreted as a pointer) never has the LSBit set. (Portability is relying on the fact that all known compilers place the `vptr` in the first word of the instances.)

3.7.8 The Core LLVM Class Hierarchy Reference

`#include "llvm/IR/Type.h"`

header source: Type.h

doxygen info: Type Clases

The Core LLVM classes are the primary means of representing the program being inspected or transformed. The core LLVM classes are defined in header files in the `include/llvm/` directory, and implemented in the `lib/VMCore` directory.

The Type class and Derived Types

`Type` is a superclass of all type classes. Every `Value` has a `Type`. `Type` cannot be instantiated directly but only through its subclasses. Certain primitive types (`VoidType`, `LabelType`, `FloatType` and `DoubleType`) have hidden subclasses. They are hidden because they offer no useful functionality beyond what the `Type` class offers except to distinguish themselves from other subclasses of `Type`.

All other types are subclasses of `DerivedType`. Types can be named, but this is not a requirement. There exists exactly one instance of a given shape at any one time. This allows type equality to be performed with address equality of the Type Instance. That is, given two `Type*` values, the types are identical if the pointers are identical.

Important Public Methods

- `bool isIntegerTy() const`: Returns true for any integer type.
- `bool isFloatingPointTy()`: Return true if this is one of the five floating point types.
- `bool isSized()`: Return true if the type has known size. Things that don't have a size are abstract types, labels and void.

Important Derived Types

`IntegerType` Subclass of DerivedType that represents integer types of any bit width. Any bit width between `IntegerType::MIN_INT_BITS` (1) and `IntegerType::MAX_INT_BITS` (~8 million) can be represented.

- `static const IntegerType* get(unsigned NumBits)`: get an integer type of a specific bit width.

- `unsigned getBitWidth() const`: Get the bit width of an integer type.

`SequentialType` This is subclassed by ArrayType, PointerType and VectorType.

- `const Type * getElementType() const`: Returns the type of each of the elements in the sequential type.

`ArrayType` This is a subclass of SequentialType and defines the interface for array types.

- `unsigned getNumElements() const`: Returns the number of elements in the array.

`PointerType` Subclass of SequentialType for pointer types.

`VectorType` Subclass of SequentialType for vector types. A vector type is similar to an ArrayType but is distinguished because it is a first class type whereas ArrayType is not. Vector types are used for vector operations and are usually small vectors of of an integer or floating point type.

`StructType` Subclass of DerivedTypes for struct types.

`FunctionType` Subclass of DerivedTypes for function types.

- `bool isVarArg() const`: Returns true if it's a vararg function.
- `const Type * getReturnType() const`: Returns the return type of the function.
- `const Type * getParamType (unsigned i)`: Returns the type of the ith parameter.
- `const unsigned getNumParams() const`: Returns the number of formal parameters.

The `Module` class

`#include "llvm/IR/Module.h"`

header source: Module.h

doxygen info: Module Class

The `Module` class represents the top level structure present in LLVM programs. An LLVM module is effectively either a translation unit of the original program or a combination of several translation units merged by the linker. The `Module` class keeps track of a list of *Function*s, a list of GlobalVariables, and a SymbolTable. Additionally, it contains a few helpful member functions that try to make common operations easy.

Important Public Members of the `Module` class

- `Module::Module(std::string name = "")`

  Constructing a Module is easy. You can optionally provide a name for it (probably based on the name of the translation unit).

- `Module::iterator` - Typedef for function list iterator
  `Module::const_iterator` - Typedef for const_iterator.
  `begin(),end(),size(),empty()`

  These are forwarding methods that make it easy to access the contents of a `Module` object's *Function* list.

- `Module::FunctionListType &getFunctionList()`

  Returns the list of *Function*s. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `Module::global_iterator` - Typedef for global variable list iterator

`Module::const_global_iterator` - Typedef for const_iterator.
`global_begin(), global_end(), global_size(), global_empty()`

These are forwarding methods that make it easy to access the contents of a `Module` object's GlobalVariable list.

- `Module::GlobalListType &getGlobalList()`

  Returns the list of GlobalVariables. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `SymbolTable *getSymbolTable()`

  Return a reference to the SymbolTable for this `Module`.

- `Function *getFunction(StringRef Name) const`

  Look up the specified function in the `Module` SymbolTable. If it does not exist, return `null`.

- `Function *getOrInsertFunction(const std::string &Name, const FunctionType *T)`

  Look up the specified function in the `Module` SymbolTable. If it does not exist, add an external declaration for the function and return it.

- `std::string getTypeName(const Type *Ty)`

  If there is at least one entry in the SymbolTable for the specified Type, return it. Otherwise return the empty string.

- `bool addTypeName(const std::string &Name, const Type *Ty)`

  Insert an entry in the SymbolTable mapping `Name` to `Ty`. If there is already an entry for this name, true is returned and the SymbolTable is not modified.

The `Value` class

```
#include "llvm/IR/Value.h"
```

header source: Value.h

doxygen info: Value Class

The `Value` class is the most important class in the LLVM Source base. It represents a typed value that may be used (among other things) as an operand to an instruction. There are many different types of `Values`, such as Constants, Arguments. Even Instructions and *Function*s are `Values`.

A particular `Value` may be used many times in the LLVM representation for a program. For example, an incoming argument to a function (represented with an instance of the Argument class) is "used" by every instruction in the function that references the argument. To keep track of this relationship, the `Value` class keeps a list of all of the `Users` that is using it (the User class is a base class for all nodes in the LLVM graph that can refer to `Values`). This use list is how LLVM represents def-use information in the program, and is accessible through the `use_*` methods, shown below.

Because LLVM is a typed representation, every LLVM `Value` is typed, and this Type is available through the `getType()` method. In addition, all LLVM values can be named. The "name" of the `Value` is a symbolic string printed in the LLVM code:

```
%foo = add i32 1, 2
```

The name of this instruction is "foo". **NOTE** that the name of any value may be missing (an empty string), so names should **ONLY** be used for debugging (making the source code easier to read, debugging printouts), they should not be used to keep track of values or map between them. For this purpose, use a `std::map` of pointers to the `Value` itself instead.

One important aspect of LLVM is that there is no distinction between an SSA variable and the operation that produces it. Because of this, any reference to the value produced by an instruction (or the value available as an incoming argument, for example) is represented as a direct pointer to the instance of the class that represents this value. Although this may take some getting used to, it simplifies the representation and makes it easier to manipulate.

Important Public Members of the `Value` class

- `Value::use_iterator` - Typedef for iterator over the use-list
  `Value::const_use_iterator` - Typedef for const_iterator over the use-list
  `unsigned use_size()` - Returns the number of users of the value.
  `bool use_empty()` - Returns true if there are no users.
  `use_iterator use_begin()` - Get an iterator to the start of the use-list.
  `use_iterator use_end()` - Get an iterator to the end of the use-list.
  `User *use_back()` - Returns the last element in the list.

  These methods are the interface to access the def-use information in LLVM. As with all other iterators in LLVM, the naming conventions follow the conventions defined by the STL.
- `Type *getType() const` This method returns the Type of the Value.
- `bool hasName() const`
  `std::string getName() const`
  `void setName(const std::string &Name)`

  This family of methods is used to access and assign a name to a `Value`, be aware of the *precaution above*.
- `void replaceAllUsesWith(Value *V)`

  This method traverses the use list of a `Value` changing all Users of the current value to refer to "`V`" instead. For example, if you detect that an instruction always produces a constant value (for example through constant folding), you can replace all uses of the instruction with the constant like this:

  ```
  Inst->replaceAllUsesWith(ConstVal);
  ```

The `User` class

`#include "llvm/IR/User.h"`

header source: User.h

doxygen info: User Class

Superclass: Value

The `User` class is the common base class of all LLVM nodes that may refer to `Values`. It exposes a list of "Operands" that are all of the `Values` that the User is referring to. The `User` class itself is a subclass of `Value`.

The operands of a `User` point directly to the LLVM `Value` that it refers to. Because LLVM uses Static Single Assignment (SSA) form, there can only be one definition referred to, allowing this direct connection. This connection provides the use-def information in LLVM.

Important Public Members of the `User` class

The `User` class exposes the operand list in two ways: through an index access interface and through an iterator based interface.

- `Value *getOperand(unsigned i)`
  `unsigned getNumOperands()`

  These two methods expose the operands of the `User` in a convenient form for direct access.

- `User::op_iterator` - Typedef for iterator over the operand list
  `op_iterator op_begin()` - Get an iterator to the start of the operand list.
  `op_iterator op_end()` - Get an iterator to the end of the operand list.

  Together, these methods make up the iterator based interface to the operands of a `User`.

The `Instruction` class

```
#include "llvm/IR/Instruction.h"
```

header source: Instruction.h

doxygen info: Instruction Class

Superclasses: User, Value

The `Instruction` class is the common base class for all LLVM instructions. It provides only a few methods, but is a very commonly used class. The primary data tracked by the `Instruction` class itself is the opcode (instruction type) and the parent BasicBlock the `Instruction` is embedded into. To represent a specific type of instruction, one of many subclasses of `Instruction` are used.

Because the `Instruction` class subclasses the User class, its operands can be accessed in the same way as for other `Users` (with the `getOperand()`/`getNumOperands()` and `op_begin()`/`op_end()` methods). An important file for the `Instruction` class is the `llvm/Instruction.def` file. This file contains some meta-data about the various different types of instructions in LLVM. It describes the enum values that are used as opcodes (for example `Instruction::Add` and `Instruction::ICmp`), as well as the concrete sub-classes of `Instruction` that implement the instruction (for example BinaryOperator and CmpInst). Unfortunately, the use of macros in this file confuses doxygen, so these enum values don't show up correctly in the doxygen output.

Important Subclasses of the `Instruction` class

- `BinaryOperator`

  This subclasses represents all two operand instructions whose operands must be the same type, except for the comparison instructions.

- `CastInst` This subclass is the parent of the 12 casting instructions. It provides common operations on cast instructions.

- `CmpInst`

  This subclass respresents the two comparison instructions, ICmpInst (integer opreands), and FCmpInst (floating point operands).

- `TerminatorInst`

  This subclass is the parent of all terminator instructions (those which can terminate a block).

Important Public Members of the `Instruction` class

- `BasicBlock *getParent()`

  Returns the BasicBlock that this `Instruction` is embedded into.

- `bool mayWriteToMemory()`

  Returns true if the instruction writes to memory, i.e. it is a `call`, `free`, `invoke`, or `store`.

- `unsigned getOpcode()`

  Returns the opcode for the `Instruction`.

- `Instruction *clone() const`

  Returns another instance of the specified instruction, identical in all ways to the original except that the instruction has no parent (i.e. it's not embedded into a BasicBlock), and it has no name.

The `Constant` class and subclasses

Constant represents a base class for different types of constants. It is subclassed by ConstantInt, ConstantArray, etc. for representing the various types of Constants. GlobalValue is also a subclass, which represents the address of a global variable or function.

Important Subclasses of Constant

- ConstantInt : This subclass of Constant represents an integer constant of any width.
  - `const APInt& getValue() const`: Returns the underlying value of this constant, an APInt value.
  - `int64_t getSExtValue() const`: Converts the underlying APInt value to an int64_t via sign extension. If the value (not the bit width) of the APInt is too large to fit in an int64_t, an assertion will result. For this reason, use of this method is discouraged.
  - `uint64_t getZExtValue() const`: Converts the underlying APInt value to a uint64_t via zero extension. IF the value (not the bit width) of the APInt is too large to fit in a uint64_t, an assertion will result. For this reason, use of this method is discouraged.
  - `static ConstantInt* get(const APInt& Val)`: Returns the ConstantInt object that represents the value provided by `Val`. The type is implied as the IntegerType that corresponds to the bit width of `Val`.
  - `static ConstantInt* get(const Type *Ty, uint64_t Val)`: Returns the ConstantInt object that represents the value provided by `Val` for integer type `Ty`.
- ConstantFP : This class represents a floating point constant.
  - `double getValue() const`: Returns the underlying value of this constant.
- ConstantArray : This represents a constant array.
  - `const std::vector<Use> &getValues() const`: Returns a vector of component constants that makeup this array.
- ConstantStruct : This represents a constant struct.
  - `const std::vector<Use> &getValues() const`: Returns a vector of component constants that makeup this array.

- GlobalValue : This represents either a global variable or a function. In either case, the value is a constant fixed address (after linking).

The `GlobalValue` class

`#include "llvm/IR/GlobalValue.h"`

header source: GlobalValue.h

doxygen info: GlobalValue Class

Superclasses: Constant, User, Value

Global values (GlobalVariables or *Functions*) are the only LLVM values that are visible in the bodies of all *Functions*. Because they are visible at global scope, they are also subject to linking with other globals defined in different translation units. To control the linking process, `GlobalValues` know their linkage rules. Specifically, `GlobalValues` know whether they have internal or external linkage, as defined by the `LinkageTypes` enumeration.

If a `GlobalValue` has internal linkage (equivalent to being `static` in C), it is not visible to code outside the current translation unit, and does not participate in linking. If it has external linkage, it is visible to external code, and does participate in linking. In addition to linkage information, `GlobalValues` keep track of which Module they are currently part of.

Because `GlobalValues` are memory objects, they are always referred to by their **address**. As such, the Type of a global is always a pointer to its contents. It is important to remember this when using the `GetElementPtrInst` instruction because this pointer must be dereferenced first. For example, if you have a `GlobalVariable` (a subclass of `GlobalValue`) that is an array of 24 ints, type `[24 x i32]`, then the `GlobalVariable` is a pointer to that array. Although the address of the first element of this array and the value of the `GlobalVariable` are the same, they have different types. The `GlobalVariable`'s type is `[24 x i32]`. The first element's type is `i32`. Because of this, accessing a global value requires you to dereference the pointer with `GetElementPtrInst` first, then its elements can be accessed. This is explained in the LLVM Language Reference Manual.

Important Public Members of the `GlobalValue` class

- ```
  bool hasInternalLinkage() const
  bool hasExternalLinkage() const
  void setInternalLinkage(bool HasInternalLinkage)
  ```

  These methods manipulate the linkage characteristics of the `GlobalValue`.

- `Module *getParent()`

  This returns the Module that the GlobalValue is currently embedded into.

The `Function` class

`#include "llvm/IR/Function.h"`

header source: Function.h

doxygen info: Function Class

Superclasses: GlobalValue, Constant, User, Value

The `Function` class represents a single procedure in LLVM. It is actually one of the more complex classes in the LLVM hierarchy because it must keep track of a large amount of data. The `Function` class keeps track of a list of BasicBlocks, a list of formal Arguments, and a SymbolTable.
  ```

The list of BasicBlocks is the most commonly used part of `Function` objects. The list imposes an implicit ordering of the blocks in the function, which indicate how the code will be laid out by the backend. Additionally, the first BasicBlock is the implicit entry node for the `Function`. It is not legal in LLVM to explicitly branch to this initial block. There are no implicit exit nodes, and in fact there may be multiple exit nodes from a single `Function`. If the BasicBlock list is empty, this indicates that the `Function` is actually a function declaration: the actual body of the function hasn't been linked in yet.

In addition to a list of BasicBlocks, the `Function` class also keeps track of the list of formal Arguments that the function receives. This container manages the lifetime of the Argument nodes, just like the BasicBlock list does for the BasicBlocks.

The SymbolTable is a very rarely used LLVM feature that is only used when you have to look up a value by name. Aside from that, the SymbolTable is used internally to make sure that there are not conflicts between the names of Instructions, BasicBlocks, or Arguments in the function body.

Note that `Function` is a GlobalValue and therefore also a Constant. The value of the function is its address (after linking) which is guaranteed to be constant.

Important Public Members of the `Function`

- `Function(const FunctionType *Ty, LinkageTypes Linkage, const std::string &N = "", Module* Parent = 0)`

  Constructor used when you need to create new `Function`s to add the program. The constructor must specify the type of the function to create and what type of linkage the function should have. The FunctionType argument specifies the formal arguments and return value for the function. The same FunctionType value can be used to create multiple functions. The `Parent` argument specifies the Module in which the function is defined. If this argument is provided, the function will automatically be inserted into that module's list of functions.

- `bool isDeclaration()`

  Return whether or not the `Function` has a body defined. If the function is "external", it does not have a body, and thus must be resolved by linking with a function defined in a different translation unit.

- `Function::iterator` - Typedef for basic block list iterator
  `Function::const_iterator` - Typedef for const_iterator.
  `begin(), end(), size(), empty()`

  These are forwarding methods that make it easy to access the contents of a `Function` object's BasicBlock list.

- `Function::BasicBlockListType &getBasicBlockList()`

  Returns the list of BasicBlocks. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `Function::arg_iterator` - Typedef for the argument list iterator
  `Function::const_arg_iterator` - Typedef for const_iterator.
  `arg_begin(), arg_end(), arg_size(), arg_empty()`

  These are forwarding methods that make it easy to access the contents of a `Function` object's Argument list.

- `Function::ArgumentListType &getArgumentList()`

  Returns the list of Argument. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `BasicBlock &getEntryBlock()`

  Returns the entry `BasicBlock` for the function. Because the entry block for the function is always the first block, this returns the first block of the `Function`.

- `Type *getReturnType()`
  `FunctionType *getFunctionType()`

  This traverses the Type of the `Function` and returns the return type of the function, or the FunctionType of the actual function.

- `SymbolTable *getSymbolTable()`

  Return a pointer to the SymbolTable for this `Function`.

The `GlobalVariable` class

`#include "llvm/IR/GlobalVariable.h"`

header source: GlobalVariable.h

doxygen info: GlobalVariable Class

Superclasses: GlobalValue, Constant, User, Value

Global variables are represented with the (surprise surprise) `GlobalVariable` class. Like functions, `GlobalVariables` are also subclasses of GlobalValue, and as such are always referenced by their address (global values must live in memory, so their "name" refers to their constant address). See GlobalValue for more on this. Global variables may have an initial value (which must be a Constant), and if they have an initializer, they may be marked as "constant" themselves (indicating that their contents never change at runtime).

Important Public Members of the `GlobalVariable` class

- `GlobalVariable(const Type *Ty, bool isConstant, LinkageTypes &Linkage, Constant *Initializer = 0, const std::string &Name = "", Module* Parent = 0)`

  Create a new global variable of the specified type. If `isConstant` is true then the global variable will be marked as unchanging for the program. The Linkage parameter specifies the type of linkage (internal, external, weak, linkonce, appending) for the variable. If the linkage is InternalLinkage, WeakAnyLinkage, WeakODR-Linkage, LinkOnceAnyLinkage or LinkOnceODRLinkage, then the resultant global variable will have internal linkage. AppendingLinkage concatenates together all instances (in different translation units) of the variable into a single variable but is only applicable to arrays. See the LLVM Language Reference for further details on linkage types. Optionally an initializer, a name, and the module to put the variable into may be specified for the global variable as well.

- `bool isConstant() const`

  Returns true if this is a global variable that is known not to be modified at runtime.

- `bool hasInitializer()`

  Returns true if this `GlobalVariable` has an intializer.

- `Constant *getInitializer()`

  Returns the initial value for a `GlobalVariable`. It is not legal to call this method if there is no initializer.

The `BasicBlock` class

`#include "llvm/IR/BasicBlock.h"`

header source: BasicBlock.h

doxygen info: BasicBlock Class

Superclass: Value

This class represents a single entry single exit section of the code, commonly known as a basic block by the compiler community. The `BasicBlock` class maintains a list of Instructions, which form the body of the block. Matching the language definition, the last element of this list of instructions is always a terminator instruction (a subclass of the TerminatorInst class).

In addition to tracking the list of instructions that make up the block, the `BasicBlock` class also keeps track of the *Function* that it is embedded into.

Note that `BasicBlock`s themselves are Values, because they are referenced by instructions like branches and can go in the switch tables. `BasicBlock`s have type `label`.

Important Public Members of the `BasicBlock` class

- `BasicBlock(const std::string &Name = "", Function *Parent = 0)`

  The `BasicBlock` constructor is used to create new basic blocks for insertion into a function. The constructor optionally takes a name for the new block, and a *Function* to insert it into. If the `Parent` parameter is specified, the new `BasicBlock` is automatically inserted at the end of the specified *Function*, if not specified, the BasicBlock must be manually inserted into the *Function*.

- `BasicBlock::iterator` - Typedef for instruction list iterator
  `BasicBlock::const_iterator` - Typedef for const_iterator.
  `begin()`, `end()`, `front()`, `back()`, `size()`, `empty()` STL-style functions for accessing the instruction list.

  These methods and typedefs are forwarding functions that have the same semantics as the standard library methods of the same names. These methods expose the underlying instruction list of a basic block in a way that is easy to manipulate. To get the full complement of container operations (including operations to update the list), you must use the `getInstList()` method.

- `BasicBlock::InstListType &getInstList()`

  This method is used to get access to the underlying container that actually holds the Instructions. This method must be used when there isn't a forwarding function in the `BasicBlock` class for the operation that you would like to perform. Because there are no forwarding functions for "updating" operations, you need to use this if you want to update the contents of a `BasicBlock`.

- `Function *getParent()`

  Returns a pointer to *Function* the block is embedded into, or a null pointer if it is homeless.

- `TerminatorInst *getTerminator()`

  Returns a pointer to the terminator instruction that appears at the end of the `BasicBlock`. If there is no terminator instruction, or if the last instruction in the block is not a terminator, then a null pointer is returned.

The `Argument` class

This subclass of Value defines the interface for incoming formal arguments to a function. A Function maintains a list of its formal arguments. An argument has a pointer to the parent Function.

3.8 LLVM Extensions

3.8.1 Introduction

This document describes extensions to tools and formats LLVM seeks compatibility with.

3.8.2 General Assembly Syntax

C99-style Hexadecimal Floating-point Constants

LLVM's assemblers allow floating-point constants to be written in C99's hexadecimal format instead of decimal if desired.

```
.section .data
.float 0x1c2.2ap3
```

3.8.3 Machine-specific Assembly Syntax

X86/COFF-Dependent

Relocations

The following additional relocation types are supported:

@IMGREL (AT&T syntax only) generates an image-relative relocation that corresponds to the COFF relocation types `IMAGE_REL_I386_DIR32NB` (32-bit) or `IMAGE_REL_AMD64_ADDR32NB` (64-bit).

```
.text
fun:
  mov foo@IMGREL(%ebx, %ecx, 4), %eax

.section .pdata
  .long fun@IMGREL
  .long (fun@imgrel + 0x3F)
  .long $unwind$fun@imgrel
```

.secrel32 generates a relocation that corresponds to the COFF relocation types `IMAGE_REL_I386_SECREL` (32-bit) or `IMAGE_REL_AMD64_SECREL` (64-bit).

.secidx relocation generates an index of the section that contains the target. It corresponds to the COFF relocation types `IMAGE_REL_I386_SECTION` (32-bit) or `IMAGE_REL_AMD64_SECTION` (64-bit).

```
.section .debug$S,"rn"
  .long 4
  .long 242
  .long 40
  .secrel32 _function_name
  .secidx   _function_name
  ...
```

`.linkonce` Directive

Syntax:

```
.linkonce [ comdat type ]
```

Supported COMDAT types:

`discard` Discards duplicate sections with the same COMDAT symbol. This is the default if no type is specified.

`one_only` If the symbol is defined multiple times, the linker issues an error.

`same_size` Duplicates are discarded, but the linker issues an error if any have different sizes.

`same_contents` Duplicates are discarded, but the linker issues an error if any duplicates do not have exactly the same content.

`largest` Links the largest section from among the duplicates.

`newest` Links the newest section from among the duplicates.

```
.section .text$foo
.linkonce
  ...
```

`.section` Directive

MC supports passing the information in `.linkonce` at the end of `.section`. For example, these two codes are equivalent

```
.section secName, "dr", discard, "Symbol1"
.globl Symbol1
Symbol1:
.long 1

.section secName, "dr"
.linkonce discard
.globl Symbol1
Symbol1:
.long 1
```

Note that in the combined form the COMDAT symbol is explicit. This extension exists to support multiple sections with the same name in different COMDATs:

```
.section secName, "dr", discard, "Symbol1"
.globl Symbol1
Symbol1:
.long 1
```

```
.section secName, "dr", discard, "Symbol2"
.globl Symbol2
Symbol2:
.long 1
```

In addition to the types allowed with `.linkonce`, `.section` also accepts `associative`. The meaning is that the section is linked if a certain other COMDAT section is linked. This other section is indicated by the comdat symbol in this directive. It can be any symbol defined in the associated section, but is usually the associated section's comdat.

The following restrictions apply to the associated section:

1. It must be a COMDAT section.
2. It cannot be another associative COMDAT section.

In the following example the symobl `sym` is the comdat symbol of `.foo` and `.bar` is associated to `.foo`.

```
.section        .foo,"bw",discard, "sym"
.section        .bar,"rd",associative, "sym"
```

3.8.4 Target Specific Behaviour

Windows on ARM

Stack Probe Emission

The reference implementation (Microsoft Visual Studio 2012) emits stack probes in the following fashion:

```
movw r4, #constant
bl __chkstk
sub.w sp, sp, r4
```

However, this has the limitation of 32 MiB (±16MiB). In order to accommodate larger binaries, LLVM supports the use of `-mcode-model=large` to allow a 4GiB range via a slight deviation. It will generate an indirect jump as follows:

```
movw r4, #constant
movw r12, :lower16:__chkstk
movt r12, :upper16:__chkstk
blx r12
sub.w sp, sp, r4
```

Variable Length Arrays

The reference implementation (Microsoft Visual Studio 2012) does not permit the emission of Variable Length Arrays (VLAs).

The Windows ARM Itanium ABI extends the base ABI by adding support for emitting a dynamic stack allocation. When emitting a variable stack allocation, a call to `__chkstk` is emitted unconditionally to ensure that guard pages are setup properly. The emission of this stack probe emission is handled similar to the standard stack probe emission.

The MSVC environment does not emit code for VLAs currently.

LLVM Language Reference Manual Defines the LLVM intermediate representation and the assembly form of the different nodes.

LLVM Atomic Instructions and Concurrency Guide Information about LLVM's concurrency model.

LLVM Programmer's Manual Introduction to the general layout of the LLVM sourcebase, important classes and APIs, and some tips & tricks.

LLVM Extensions LLVM-specific extensions to tools and formats LLVM seeks compatibility with.

CommandLine 2.0 Library Manual Provides information on using the command line parsing library.

LLVM Coding Standards Details the LLVM coding standards and provides useful information on writing efficient C++ code.

How to set up LLVM-style RTTI for your class hierarchy How to make `isa<>`, `dyn_cast<>`, etc. available for clients of your class hierarchy.

Extending LLVM: Adding instructions, intrinsics, types, etc. Look here to see how to add instructions and intrinsics to LLVM.

Doxygen generated documentation (classes) (tarball)

ViewVC Repository Browser

Architecture & Platform Information for Compiler Writers A list of helpful links for compiler writers.

CHAPTER

FOUR

SUBSYSTEM DOCUMENTATION

For API clients and LLVM developers.

4.1 LLVM Alias Analysis Infrastructure

4.1.1 Introduction

Alias Analysis (aka Pointer Analysis) is a class of techniques which attempt to determine whether or not two pointers ever can point to the same object in memory. There are many different algorithms for alias analysis and many different ways of classifying them: flow-sensitive vs. flow-insensitive, context-sensitive vs. context-insensitive, field-sensitive vs. field-insensitive, unification-based vs. subset-based, etc. Traditionally, alias analyses respond to a query with a

Must, May, or No alias response, indicating that two pointers always point to the same object, might point to the same object, or are known to never point to the same object.

The LLVM AliasAnalysis class is the primary interface used by clients and implementations of alias analyses in the LLVM system. This class is the common interface between clients of alias analysis information and the implementations providing it, and is designed to support a wide range of implementations and clients (but currently all clients are assumed to be flow-insensitive). In addition to simple alias analysis information, this class exposes Mod/Ref information from those implementations which can provide it, allowing for powerful analyses and transformations to work well together.

This document contains information necessary to successfully implement this interface, use it, and to test both sides. It also explains some of the finer points about what exactly results mean. If you feel that something is unclear or should be added, please let me know.

4.1.2 `AliasAnalysis` Class Overview

The AliasAnalysis class defines the interface that the various alias analysis implementations should support. This class exports two important enums: `AliasResult` and `ModRefResult` which represent the result of an alias query or a mod/ref query, respectively.

The `AliasAnalysis` interface exposes information about memory, represented in several different ways. In particular, memory objects are represented as a starting address and size, and function calls are represented as the actual `call` or `invoke` instructions that performs the call. The `AliasAnalysis` interface also exposes some helper methods which allow you to get mod/ref information for arbitrary instructions.

All `AliasAnalysis` interfaces require that in queries involving multiple values, values which are not *constants* are all defined within the same function.

Representation of Pointers

Most importantly, the `AliasAnalysis` class provides several methods which are used to query whether or not two memory objects alias, whether function calls can modify or read a memory object, etc. For all of these queries, memory objects are represented as a pair of their starting address (a symbolic LLVM `Value*`) and a static size.

Representing memory objects as a starting address and a size is critically important for correct Alias Analyses. For example, consider this (silly, but possible) C code:

```
int i;
char C[2];
char A[10];
/* ... */
for (i = 0; i != 10; ++i) {
  C[0] = A[i];          /* One byte store */
  C[1] = A[9-i];        /* One byte store */
}
```

In this case, the `basicaa` pass will disambiguate the stores to `C[0]` and `C[1]` because they are accesses to two distinct locations one byte apart, and the accesses are each one byte. In this case, the Loop Invariant Code Motion (LICM) pass can use store motion to remove the stores from the loop. In constrast, the following code:

```
int i;
char C[2];
char A[10];
/* ... */
for (i = 0; i != 10; ++i) {
  ((short*)C)[0] = A[i];  /* Two byte store! */
```

```
  C[1] = A[9-i];           /* One byte store */
}
```

In this case, the two stores to C do alias each other, because the access to the `&C[0]` element is a two byte access. If size information wasn't available in the query, even the first case would have to conservatively assume that the accesses alias.

The `alias` method

The `alias` method is the primary interface used to determine whether or not two memory objects alias each other. It takes two memory objects as input and returns MustAlias, PartialAlias, MayAlias, or NoAlias as appropriate.

Like all `AliasAnalysis` interfaces, the `alias` method requires that either the two pointer values be defined within the same function, or at least one of the values is a *constant*.

Must, May, and No Alias Responses

The `NoAlias` response may be used when there is never an immediate dependence between any memory reference *based* on one pointer and any memory reference *based* the other. The most obvious example is when the two pointers point to non-overlapping memory ranges. Another is when the two pointers are only ever used for reading memory. Another is when the memory is freed and reallocated between accesses through one pointer and accesses through the other — in this case, there is a dependence, but it's mediated by the free and reallocation.

As an exception to this is with the *noalias* keyword; the "irrelevant" dependencies are ignored.

The `MayAlias` response is used whenever the two pointers might refer to the same object.

The `PartialAlias` response is used when the two memory objects are known to be overlapping in some way, but do not start at the same address.

The `MustAlias` response may only be returned if the two memory objects are guaranteed to always start at exactly the same location. A `MustAlias` response implies that the pointers compare equal.

The `getModRefInfo` methods

The `getModRefInfo` methods return information about whether the execution of an instruction can read or modify a memory location. Mod/Ref information is always conservative: if an instruction **might** read or write a location, `ModRef` is returned.

The `AliasAnalysis` class also provides a `getModRefInfo` method for testing dependencies between function calls. This method takes two call sites (`CS1` & `CS2`), returns `NoModRef` if neither call writes to memory read or written by the other, `Ref` if `CS1` reads memory written by `CS2`, `Mod` if `CS1` writes to memory read or written by `CS2`, or `ModRef` if `CS1` might read or write memory written to by `CS2`. Note that this relation is not commutative.

Other useful `AliasAnalysis` methods

Several other tidbits of information are often collected by various alias analysis implementations and can be put to good use by various clients.

The `pointsToConstantMemory` method

The `pointsToConstantMemory` method returns true if and only if the analysis can prove that the pointer only points to unchanging memory locations (functions, constant global variables, and the null pointer). This information can be used to refine mod/ref information: it is impossible for an unchanging memory location to be modified.

The `doesNotAccessMemory` and `onlyReadsMemory` methods

These methods are used to provide very simple mod/ref information for function calls. The `doesNotAccessMemory` method returns true for a function if the analysis can prove that the function never reads or writes to memory, or if the function only reads from constant memory. Functions with this property are side-effect free and only depend on their input arguments, allowing them to be eliminated if they form common subexpressions or be hoisted out of loops. Many common functions behave this way (e.g., `sin` and `cos`) but many others do not (e.g., `acos`, which modifies the `errno` variable).

The `onlyReadsMemory` method returns true for a function if analysis can prove that (at most) the function only reads from non-volatile memory. Functions with this property are side-effect free, only depending on their input arguments and the state of memory when they are called. This property allows calls to these functions to be eliminated and moved around, as long as there is no store instruction that changes the contents of memory. Note that all functions that satisfy the `doesNotAccessMemory` method also satisfies `onlyReadsMemory`.

4.1.3 Writing a new `AliasAnalysis` Implementation

Writing a new alias analysis implementation for LLVM is quite straight-forward. There are already several implementations that you can use for examples, and the following information should help fill in any details. For a examples, take a look at the various alias analysis implementations included with LLVM.

Different Pass styles

The first step to determining what type of *LLVM pass* you need to use for your Alias Analysis. As is the case with most other analyses and transformations, the answer should be fairly obvious from what type of problem you are trying to solve:

1. If you require interprocedural analysis, it should be a `Pass`.
2. If you are a function-local analysis, subclass `FunctionPass`.
3. If you don't need to look at the program at all, subclass `ImmutablePass`.

In addition to the pass that you subclass, you should also inherit from the `AliasAnalysis` interface, of course, and use the `RegisterAnalysisGroup` template to register as an implementation of `AliasAnalysis`.

Required initialization calls

Your subclass of `AliasAnalysis` is required to invoke two methods on the `AliasAnalysis` base class: `getAnalysisUsage` and `InitializeAliasAnalysis`. In particular, your implementation of `getAnalysisUsage` should explicitly call into the `AliasAnalysis::getAnalysisUsage` method in addition to doing any declaring any pass dependencies your pass has. Thus you should have something like this:

```
void getAnalysisUsage(AnalysisUsage &AU) const {
  AliasAnalysis::getAnalysisUsage(AU);
  // declare your dependencies here.
}
```

Additionally, your must invoke the `InitializeAliasAnalysis` method from your analysis run method (`run` for a `Pass`, `runOnFunction` for a `FunctionPass`, or `InitializePass` for an `ImmutablePass`). For example (as part of a `Pass`):

```
bool run(Module &M) {
  InitializeAliasAnalysis(this);
  // Perform analysis here...
  return false;
}
```

Required methods to override

You must override the `getAdjustedAnalysisPointer` method on all subclasses of `AliasAnalysis`. An example implementation of this method would look like:

```
void *getAdjustedAnalysisPointer(const void* ID) override {
  if (ID == &AliasAnalysis::ID)
    return (AliasAnalysis*)this;
  return this;
}
```

Interfaces which may be specified

All of the AliasAnalysis virtual methods default to providing *chaining* to another alias analysis implementation, which ends up returning conservatively correct information (returning "May" Alias and "Mod/Ref" for alias and mod/ref queries respectively). Depending on the capabilities of the analysis you are implementing, you just override the interfaces you can improve.

`AliasAnalysis` chaining behavior

With only one special exception (the *-no-aa* pass) every alias analysis pass chains to another alias analysis implementation (for example, the user can specify "`-basicaa -ds-aa -licm`" to get the maximum benefit from both alias analyses). The alias analysis class automatically takes care of most of this for methods that you don't override. For methods that you do override, in code paths that return a conservative MayAlias or Mod/Ref result, simply return whatever the superclass computes. For example:

```
AliasAnalysis::AliasResult alias(const Value *V1, unsigned V1Size,
                                 const Value *V2, unsigned V2Size) {
  if (...)
    return NoAlias;
  ...

  // Couldn't determine a must or no-alias result.
  return AliasAnalysis::alias(V1, V1Size, V2, V2Size);
}
```

In addition to analysis queries, you must make sure to unconditionally pass LLVM update notification methods to the superclass as well if you override them, which allows all alias analyses in a change to be updated.

Updating analysis results for transformations

Alias analysis information is initially computed for a static snapshot of the program, but clients will use this information to make transformations to the code. All but the most trivial forms of alias analysis will need to have their analysis results updated to reflect the changes made by these transformations.

The `AliasAnalysis` interface exposes four methods which are used to communicate program changes from the clients to the analysis implementations. Various alias analysis implementations should use these methods to ensure that their internal data structures are kept up-to-date as the program changes (for example, when an instruction is deleted), and clients of alias analysis must be sure to call these interfaces appropriately.

The `deleteValue` method

The `deleteValue` method is called by transformations when they remove an instruction or any other value from the program (including values that do not use pointers). Typically alias analyses keep data structures that have entries for each value in the program. When this method is called, they should remove any entries for the specified value, if they exist.

The `copyValue` method

The `copyValue` method is used when a new value is introduced into the program. There is no way to introduce a value into the program that did not exist before (this doesn't make sense for a safe compiler transformation), so this is the only way to introduce a new value. This method indicates that the new value has exactly the same properties as the value being copied.

The `replaceWithNewValue` method

This method is a simple helper method that is provided to make clients easier to use. It is implemented by copying the old analysis information to the new value, then deleting the old value. This method cannot be overridden by alias analysis implementations.

The `addEscapingUse` method

The `addEscapingUse` method is used when the uses of a pointer value have changed in ways that may invalidate precomputed analysis information. Implementations may either use this callback to provide conservative responses for points whose uses have change since analysis time, or may recompute some or all of their internal state to continue providing accurate responses.

In general, any new use of a pointer value is considered an escaping use, and must be reported through this callback, *except* for the uses below:

- A `bitcast` or `getelementptr` of the pointer
- A `store` through the pointer (but not a `store` *of* the pointer)
- A `load` through the pointer

Efficiency Issues

From the LLVM perspective, the only thing you need to do to provide an efficient alias analysis is to make sure that alias analysis **queries** are serviced quickly. The actual calculation of the alias analysis results (the "run" method) is only performed once, but many (perhaps duplicate) queries may be performed. Because of this, try to move as much computation to the run method as possible (within reason).

Limitations

The AliasAnalysis infrastructure has several limitations which make writing a new `AliasAnalysis` implementation difficult.

There is no way to override the default alias analysis. It would be very useful to be able to do something like "`opt -my-aa -O2`" and have it use `-my-aa` for all passes which need AliasAnalysis, but there is currently no support for that, short of changing the source code and recompiling. Similarly, there is also no way of setting a chain of analyses as the default.

There is no way for transform passes to declare that they preserve `AliasAnalysis` implementations. The `AliasAnalysis` interface includes `deleteValue` and `copyValue` methods which are intended to allow a pass to keep an AliasAnalysis consistent, however there's no way for a pass to declare in its `getAnalysisUsage` that it does so. Some passes attempt to use `AU.addPreserved<AliasAnalysis>`, however this doesn't actually have any effect.

`AliasAnalysisCounter` (`-count-aa`) and `AliasDebugger` (`-debug-aa`) are implemented as `ModulePass` classes, so if your alias analysis uses `FunctionPass`, it won't be able to use these utilities. If you try to use them, the pass manager will silently route alias analysis queries directly to `BasicAliasAnalysis` instead.

Similarly, the `opt -p` option introduces `ModulePass` passes between each pass, which prevents the use of `FunctionPass` alias analysis passes.

The `AliasAnalysis` API does have functions for notifying implementations when values are deleted or copied, however these aren't sufficient. There are many other ways that LLVM IR can be modified which could be relevant to `AliasAnalysis` implementations which can not be expressed.

The `AliasAnalysisDebugger` utility seems to suggest that `AliasAnalysis` implementations can expect that they will be informed of any relevant `Value` before it appears in an alias query. However, popular clients such as `GVN` don't support this, and are known to trigger errors when run with the `AliasAnalysisDebugger`.

Due to several of the above limitations, the most obvious use for the `AliasAnalysisCounter` utility, collecting stats on all alias queries in a compilation, doesn't work, even if the `AliasAnalysis` implementations don't use `FunctionPass`. There's no way to set a default, much less a default sequence, and there's no way to preserve it.

The `AliasSetTracker` class (which is used by `LICM`) makes a non-deterministic number of alias queries. This can cause stats collected by `AliasAnalysisCounter` to have fluctuations among identical runs, for example. Another consequence is that debugging techniques involving pausing execution after a predetermined number of queries can be unreliable.

Many alias queries can be reformulated in terms of other alias queries. When multiple `AliasAnalysis` queries are chained together, it would make sense to start those queries from the beginning of the chain, with care taken to avoid infinite looping, however currently an implementation which wants to do this can only start such queries from itself.

4.1.4 Using alias analysis results

There are several different ways to use alias analysis results. In order of preference, these are:

Using the `MemoryDependenceAnalysis` Pass

The `memdep` pass uses alias analysis to provide high-level dependence information about memory-using instructions. This will tell you which store feeds into a load, for example. It uses caching and other techniques to be efficient, and is used by Dead Store Elimination, GVN, and memcpy optimizations.

Using the `AliasSetTracker` class

Many transformations need information about alias **sets** that are active in some scope, rather than information about pairwise aliasing. The AliasSetTracker class is used to efficiently build these Alias Sets from the pairwise alias analysis information provided by the `AliasAnalysis` interface.

First you initialize the AliasSetTracker by using the "`add`" methods to add information about various potentially aliasing instructions in the scope you are interested in. Once all of the alias sets are completed, your pass should simply iterate through the constructed alias sets, using the `AliasSetTracker` `begin()`/`end()` methods.

The `AliasSets` formed by the `AliasSetTracker` are guaranteed to be disjoint, calculate mod/ref information and volatility for the set, and keep track of whether or not all of the pointers in the set are Must aliases. The AliasSetTracker also makes sure that sets are properly folded due to call instructions, and can provide a list of pointers in each set.

As an example user of this, the Loop Invariant Code Motion pass uses `AliasSetTrackers` to calculate alias sets for each loop nest. If an `AliasSet` in a loop is not modified, then all load instructions from that set may be hoisted out of the loop. If any alias sets are stored to **and** are must alias sets, then the stores may be sunk to outside of the loop, promoting the memory location to a register for the duration of the loop nest. Both of these transformations only apply if the pointer argument is loop-invariant.

The AliasSetTracker implementation

The AliasSetTracker class is implemented to be as efficient as possible. It uses the union-find algorithm to efficiently merge AliasSets when a pointer is inserted into the AliasSetTracker that aliases multiple sets. The primary data structure is a hash table mapping pointers to the AliasSet they are in.

The AliasSetTracker class must maintain a list of all of the LLVM `Value*s` that are in each AliasSet. Since the hash table already has entries for each LLVM `Value*` of interest, the AliasesSets thread the linked list through these hash-table nodes to avoid having to allocate memory unnecessarily, and to make merging alias sets extremely efficient (the linked list merge is constant time).

You shouldn't need to understand these details if you are just a client of the AliasSetTracker, but if you look at the code, hopefully this brief description will help make sense of why things are designed the way they are.

Using the `AliasAnalysis` interface directly

If neither of these utility class are what your pass needs, you should use the interfaces exposed by the `AliasAnalysis` class directly. Try to use the higher-level methods when possible (e.g., use mod/ref information instead of the alias method directly if possible) to get the best precision and efficiency.

4.1.5 Existing alias analysis implementations and clients

If you're going to be working with the LLVM alias analysis infrastructure, you should know what clients and implementations of alias analysis are available. In particular, if you are implementing an alias analysis, you should be aware of the the clients that are useful for monitoring and evaluating different implementations.

Available `AliasAnalysis` implementations

This section lists the various implementations of the `AliasAnalysis` interface. With the exception of the *-no-aa* implementation, all of these *chain* to other alias analysis implementations.

The `-no-aa` pass

The `-no-aa` pass is just like what it sounds: an alias analysis that never returns any useful information. This pass can be useful if you think that alias analysis is doing something wrong and are trying to narrow down a problem.

The `-basicaa` pass

The `-basicaa` pass is an aggressive local analysis that *knows* many important facts:

- Distinct globals, stack allocations, and heap allocations can never alias.
- Globals, stack allocations, and heap allocations never alias the null pointer.
- Different fields of a structure do not alias.
- Indexes into arrays with statically differing subscripts cannot alias.
- Many common standard C library functions never access memory or only read memory.
- Pointers that obviously point to constant globals "`pointToConstantMemory`".
- Function calls can not modify or references stack allocations if they never escape from the function that allocates them (a common case for automatic arrays).

The `-globalsmodref-aa` pass

This pass implements a simple context-sensitive mod/ref and alias analysis for internal global variables that don't "have their address taken". If a global does not have its address taken, the pass knows that no pointers alias the global. This pass also keeps track of functions that it knows never access memory or never read memory. This allows certain optimizations (e.g. GVN) to eliminate call instructions entirely.

The real power of this pass is that it provides context-sensitive mod/ref information for call instructions. This allows the optimizer to know that calls to a function do not clobber or read the value of the global, allowing loads and stores to be eliminated.

Note: This pass is somewhat limited in its scope (only support non-address taken globals), but is very quick analysis.

The `-steens-aa` pass

The `-steens-aa` pass implements a variation on the well-known "Steensgaard's algorithm" for interprocedural alias analysis. Steensgaard's algorithm is a unification-based, flow-insensitive, context-insensitive, and field-insensitive alias analysis that is also very scalable (effectively linear time).

The LLVM `-steens-aa` pass implements a "speculatively field-**sensitive**" version of Steensgaard's algorithm using the Data Structure Analysis framework. This gives it substantially more precision than the standard algorithm while maintaining excellent analysis scalability.

Note: `-steens-aa` is available in the optional "poolalloc" module. It is not part of the LLVM core.

The `-ds-aa` pass

The `-ds-aa` pass implements the full Data Structure Analysis algorithm. Data Structure Analysis is a modular unification-based, flow-insensitive, context-**sensitive**, and speculatively field-**sensitive** alias analysis that is also quite scalable, usually at `O(n * log(n))`.

This algorithm is capable of responding to a full variety of alias analysis queries, and can provide context-sensitive mod/ref information as well. The only major facility not implemented so far is support for must-alias information.

Note: `-ds-aa` is available in the optional "poolalloc" module. It is not part of the LLVM core.

The `-scev-aa` pass

The `-scev-aa` pass implements AliasAnalysis queries by translating them into ScalarEvolution queries. This gives it a more complete understanding of `getelementptr` instructions and loop induction variables than other alias analyses have.

Alias analysis driven transformations

LLVM includes several alias-analysis driven transformations which can be used with any of the implementations above.

The `-adce` pass

The `-adce` pass, which implements Aggressive Dead Code Elimination uses the `AliasAnalysis` interface to delete calls to functions that do not have side-effects and are not used.

The `-licm` pass

The `-licm` pass implements various Loop Invariant Code Motion related transformations. It uses the `AliasAnalysis` interface for several different transformations:

- It uses mod/ref information to hoist or sink load instructions out of loops if there are no instructions in the loop that modifies the memory loaded.
- It uses mod/ref information to hoist function calls out of loops that do not write to memory and are loop-invariant.
- If uses alias information to promote memory objects that are loaded and stored to in loops to live in a register instead. It can do this if there are no may aliases to the loaded/stored memory location.

The `-argpromotion` pass

The `-argpromotion` pass promotes by-reference arguments to be passed in by-value instead. In particular, if pointer arguments are only loaded from it passes in the value loaded instead of the address to the function. This pass uses alias information to make sure that the value loaded from the argument pointer is not modified between the entry of the function and any load of the pointer.

The `-gvn`, `-memcpyopt`, and `-dse` passes

These passes use AliasAnalysis information to reason about loads and stores.

Clients for debugging and evaluation of implementations

These passes are useful for evaluating the various alias analysis implementations. You can use them with commands like:

```
% opt -ds-aa -aa-eval foo.bc -disable-output -stats
```

The `-print-alias-sets` pass

The `-print-alias-sets` pass is exposed as part of the `opt` tool to print out the Alias Sets formed by the AliasSetTracker class. This is useful if you're using the `AliasSetTracker` class. To use it, use something like:

```
% opt -ds-aa -print-alias-sets -disable-output
```

The `-count-aa` pass

The `-count-aa` pass is useful to see how many queries a particular pass is making and what responses are returned by the alias analysis. As an example:

```
% opt -basicaa -count-aa -ds-aa -count-aa -licm
```

will print out how many queries (and what responses are returned) by the `-licm` pass (of the `-ds-aa` pass) and how many queries are made of the `-basicaa` pass by the `-ds-aa` pass. This can be useful when debugging a transformation or an alias analysis implementation.

The `-aa-eval` pass

The `-aa-eval` pass simply iterates through all pairs of pointers in a function and asks an alias analysis whether or not the pointers alias. This gives an indication of the precision of the alias analysis. Statistics are printed indicating the percent of no/may/must aliases found (a more precise algorithm will have a lower number of may aliases).

4.1.6 Memory Dependence Analysis

If you're just looking to be a client of alias analysis information, consider using the Memory Dependence Analysis interface instead. MemDep is a lazy, caching layer on top of alias analysis that is able to answer the question of what preceding memory operations a given instruction depends on, either at an intra- or inter-block level. Because of its laziness and caching policy, using MemDep can be a significant performance win over accessing alias analysis directly.

4.2 LLVM Bitcode File Format

4.2.1 Abstract

This document describes the LLVM bitstream file format and the encoding of the LLVM IR into it.

4.2.2 Overview

What is commonly known as the LLVM bitcode file format (also, sometimes anachronistically known as bytecode) is actually two things: a bitstream container format and an encoding of LLVM IR into the container format.

The bitstream format is an abstract encoding of structured data, very similar to XML in some ways. Like XML, bitstream files contain tags, and nested structures, and you can parse the file without having to understand the tags. Unlike XML, the bitstream format is a binary encoding, and unlike XML it provides a mechanism for the file to self-describe "abbreviations", which are effectively size optimizations for the content.

LLVM IR files may be optionally embedded into a wrapper structure, or in a native object file. Both of these mechanisms make it easy to embed extra data along with LLVM IR files.

This document first describes the LLVM bitstream format, describes the wrapper format, then describes the record structure used by LLVM IR files.

4.2.3 Bitstream Format

The bitstream format is literally a stream of bits, with a very simple structure. This structure consists of the following concepts:

- A "magic number" that identifies the contents of the stream.
- Encoding primitives like variable bit-rate integers.
- Blocks, which define nested content.
- Data Records, which describe entities within the file.
- Abbreviations, which specify compression optimizations for the file.

Note that the *llvm-bcanalyzer* tool can be used to dump and inspect arbitrary bitstreams, which is very useful for understanding the encoding.

Magic Numbers

The first two bytes of a bitcode file are 'BC' (`0x42`, `0x43`). The second two bytes are an application-specific magic number. Generic bitcode tools can look at only the first two bytes to verify the file is bitcode, while application-specific programs will want to look at all four.

Primitives

A bitstream literally consists of a stream of bits, which are read in order starting with the least significant bit of each byte. The stream is made up of a number of primitive values that encode a stream of unsigned integer values. These integers are encoded in two ways: either as Fixed Width Integers or as Variable Width Integers.

Fixed Width Integers

Fixed-width integer values have their low bits emitted directly to the file. For example, a 3-bit integer value encodes 1 as 001. Fixed width integers are used when there are a well-known number of options for a field. For example, boolean values are usually encoded with a 1-bit wide integer.

Variable Width Integers

Variable-width integer (VBR) values encode values of arbitrary size, optimizing for the case where the values are small. Given a 4-bit VBR field, any 3-bit value (0 through 7) is encoded directly, with the high bit set to zero. Values larger than N-1 bits emit their bits in a series of N-1 bit chunks, where all but the last set the high bit.

For example, the value 27 (0x1B) is encoded as 1011 0011 when emitted as a vbr4 value. The first set of four bits indicates the value 3 (011) with a continuation piece (indicated by a high bit of 1). The next word indicates a value of 24 (011 << 3) with no continuation. The sum (3+24) yields the value 27.

6-bit characters

6-bit characters encode common characters into a fixed 6-bit field. They represent the following characters with the following 6-bit values:

```
'a' .. 'z' ---  0 .. 25
'A' .. 'Z' --- 26 .. 51
'0' .. '9' --- 52 .. 61
       '.' --- 62
       '_' --- 63
```

This encoding is only suitable for encoding characters and strings that consist only of the above characters. It is completely incapable of encoding characters not in the set.

Word Alignment

Occasionally, it is useful to emit zero bits until the bitstream is a multiple of 32 bits. This ensures that the bit position in the stream can be represented as a multiple of 32-bit words.

Abbreviation IDs

A bitstream is a sequential series of Blocks and Data Records. Both of these start with an abbreviation ID encoded as a fixed-bitwidth field. The width is specified by the current block, as described below. The value of the abbreviation ID specifies either a builtin ID (which have special meanings, defined below) or one of the abbreviation IDs defined for the current block by the stream itself.

The set of builtin abbrev IDs is:

- 0 - END_BLOCK — This abbrev ID marks the end of the current block.
- 1 - ENTER_SUBBLOCK — This abbrev ID marks the beginning of a new block.
- 2 - DEFINE_ABBREV — This defines a new abbreviation.
- 3 - UNABBREV_RECORD — This ID specifies the definition of an unabbreviated record.

Abbreviation IDs 4 and above are defined by the stream itself, and specify an abbreviated record encoding.

Blocks

Blocks in a bitstream denote nested regions of the stream, and are identified by a content-specific id number (for example, LLVM IR uses an ID of 12 to represent function bodies). Block IDs 0-7 are reserved for standard blocks whose meaning is defined by Bitcode; block IDs 8 and greater are application specific. Nested blocks capture the hierarchical structure of the data encoded in it, and various properties are associated with blocks as the file is parsed. Block definitions allow the reader to efficiently skip blocks in constant time if the reader wants a summary of blocks,

or if it wants to efficiently skip data it does not understand. The LLVM IR reader uses this mechanism to skip function bodies, lazily reading them on demand.

When reading and encoding the stream, several properties are maintained for the block. In particular, each block maintains:

1. A current abbrev id width. This value starts at 2 at the beginning of the stream, and is set every time a block record is entered. The block entry specifies the abbrev id width for the body of the block.
2. A set of abbreviations. Abbreviations may be defined within a block, in which case they are only defined in that block (neither subblocks nor enclosing blocks see the abbreviation). Abbreviations can also be defined inside a BLOCKINFO block, in which case they are defined in all blocks that match the ID that the `BLOCKINFO` block is describing.

As sub blocks are entered, these properties are saved and the new sub-block has its own set of abbreviations, and its own abbrev id width. When a sub-block is popped, the saved values are restored.

ENTER_SUBBLOCK Encoding

[ENTER_SUBBLOCK, $\text{blockid}_{\text{vbr8}}$, $\text{newabbrevlen}_{\text{vbr4}}$, <align32bits>, blocklen_32]

The `ENTER_SUBBLOCK` abbreviation ID specifies the start of a new block record. The `blockid` value is encoded as an 8-bit VBR identifier, and indicates the type of block being entered, which can be a standard block or an application-specific block. The `newabbrevlen` value is a 4-bit VBR, which specifies the abbrev id width for the sub-block. The `blocklen` value is a 32-bit aligned value that specifies the size of the subblock in 32-bit words. This value allows the reader to skip over the entire block in one jump.

END_BLOCK Encoding

```
[END_BLOCK, <align32bits>]
```

The `END_BLOCK` abbreviation ID specifies the end of the current block record. Its end is aligned to 32-bits to ensure that the size of the block is an even multiple of 32-bits.

Data Records

Data records consist of a record code and a number of (up to) 64-bit integer values. The interpretation of the code and values is application specific and may vary between different block types. Records can be encoded either using an unabbrev record, or with an abbreviation. In the LLVM IR format, for example, there is a record which encodes the target triple of a module. The code is `MODULE_CODE_TRIPLE`, and the values of the record are the ASCII codes for the characters in the string.

UNABBREV_RECORD Encoding

[UNABBREV_RECORD, $\text{code}_{\text{vbr6}}$, $\text{numops}_{\text{vbr6}}$, op0_{vbr6}, op1_{vbr6}, ...]

An `UNABBREV_RECORD` provides a default fallback encoding, which is both completely general and extremely inefficient. It can describe an arbitrary record by emitting the code and operands as VBRs.

For example, emitting an LLVM IR target triple as an unabbreviated record requires emitting the `UNABBREV_RECORD` abbrevid, a vbr6 for the `MODULE_CODE_TRIPLE` code, a vbr6 for the length of the string, which is equal to the number of operands, and a vbr6 for each character. Because there are no letters with values less than 32, each letter would need to be emitted as at least a two-part VBR, which means that each letter would require at least 12 bits. This is not an efficient encoding, but it is fully general.

Abbreviated Record Encoding

`[<abbrevid>, fields...]`

An abbreviated record is a abbreviation id followed by a set of fields that are encoded according to the abbreviation definition. This allows records to be encoded significantly more densely than records encoded with the UNABBREV_RECORD type, and allows the abbreviation types to be specified in the stream itself, which allows the files to be completely self describing. The actual encoding of abbreviations is defined below.

The record code, which is the first field of an abbreviated record, may be encoded in the abbreviation definition (as a literal operand) or supplied in the abbreviated record (as a Fixed or VBR operand value).

Abbreviations

Abbreviations are an important form of compression for bitstreams. The idea is to specify a dense encoding for a class of records once, then use that encoding to emit many records. It takes space to emit the encoding into the file, but the space is recouped (hopefully plus some) when the records that use it are emitted.

Abbreviations can be determined dynamically per client, per file. Because the abbreviations are stored in the bitstream itself, different streams of the same format can contain different sets of abbreviations according to the needs of the specific stream. As a concrete example, LLVM IR files usually emit an abbreviation for binary operators. If a specific LLVM module contained no or few binary operators, the abbreviation does not need to be emitted.

DEFINE_ABBREV Encoding

[DEFINE_ABBREV, numabbrevops$_{vbr5}$, abbrevop0, abbrevop1, ...]

A `DEFINE_ABBREV` record adds an abbreviation to the list of currently defined abbreviations in the scope of this block. This definition only exists inside this immediate block — it is not visible in subblocks or enclosing blocks. Abbreviations are implicitly assigned IDs sequentially starting from 4 (the first application-defined abbreviation ID). Any abbreviations defined in a `BLOCKINFO` record for the particular block type receive IDs first, in order, followed by any abbreviations defined within the block itself. Abbreviated data records reference this ID to indicate what abbreviation they are invoking.

An abbreviation definition consists of the `DEFINE_ABBREV` abbrevid followed by a VBR that specifies the number of abbrev operands, then the abbrev operands themselves. Abbreviation operands come in three forms. They all start with a single bit that indicates whether the abbrev operand is a literal operand (when the bit is 1) or an encoding operand (when the bit is 0).

1. Literal operands — [1_1, litvalue$_{vbr8}$] — Literal operands specify that the value in the result is always a single specific value. This specific value is emitted as a vbr8 after the bit indicating that it is a literal operand.
2. Encoding info without data — [0_1, encoding$_3$] — Operand encodings that do not have extra data are just emitted as their code.
3. Encoding info with data — [0_1, encoding$_3$, value$_{vbr5}$] — Operand encodings that do have extra data are emitted as their code, followed by the extra data.

The possible operand encodings are:

- Fixed (code 1): The field should be emitted as a fixed-width value, whose width is specified by the operand's extra data.
- VBR (code 2): The field should be emitted as a variable-width value, whose width is specified by the operand's extra data.
- Array (code 3): This field is an array of values. The array operand has no extra data, but expects another operand to follow it, indicating the element type of the array. When reading an array in an abbreviated record, the first

integer is a vbr6 that indicates the array length, followed by the encoded elements of the array. An array may only occur as the last operand of an abbreviation (except for the one final operand that gives the array's type).

- Char6 (code 4): This field should be emitted as a char6-encoded value. This operand type takes no extra data. Char6 encoding is normally used as an array element type.
- Blob (code 5): This field is emitted as a vbr6, followed by padding to a 32-bit boundary (for alignment) and an array of 8-bit objects. The array of bytes is further followed by tail padding to ensure that its total length is a multiple of 4 bytes. This makes it very efficient for the reader to decode the data without having to make a copy of it: it can use a pointer to the data in the mapped in file and poke directly at it. A blob may only occur as the last operand of an abbreviation.

For example, target triples in LLVM modules are encoded as a record of the form `[TRIPLE, 'a', 'b', 'c', 'd']`. Consider if the bitstream emitted the following abbrev entry:

```
[0, Fixed, 4]
[0, Array]
[0, Char6]
```

When emitting a record with this abbreviation, the above entry would be emitted as:

$[4_{\text{abbrevwidth}}, 2_4, 4_{\text{vbr6}}, 0_6, 1_6, 2_6, 3_6]$

These values are:

1. The first value, 4, is the abbreviation ID for this abbreviation.
2. The second value, 2, is the record code for `TRIPLE` records within LLVM IR file `MODULE_BLOCK` blocks.
3. The third value, 4, is the length of the array.
4. The rest of the values are the char6 encoded values for `"abcd"`.

With this abbreviation, the triple is emitted with only 37 bits (assuming a abbrev id width of 3). Without the abbreviation, significantly more space would be required to emit the target triple. Also, because the `TRIPLE` value is not emitted as a literal in the abbreviation, the abbreviation can also be used for any other string value.

Standard Blocks

In addition to the basic block structure and record encodings, the bitstream also defines specific built-in block types. These block types specify how the stream is to be decoded or other metadata. In the future, new standard blocks may be added. Block IDs 0-7 are reserved for standard blocks.

#0 - BLOCKINFO Block

The `BLOCKINFO` block allows the description of metadata for other blocks. The currently specified records are:

```
[SETBID (#1), blockid]
[DEFINE_ABBREV, ...]
[BLOCKNAME, ...name...]
[SETRECORDNAME, RecordID, ...name...]
```

The `SETBID` record (code 1) indicates which block ID is being described. `SETBID` records can occur multiple times throughout the block to change which block ID is being described. There must be a `SETBID` record prior to any other records.

Standard `DEFINE_ABBREV` records can occur inside `BLOCKINFO` blocks, but unlike their occurrence in normal blocks, the abbreviation is defined for blocks matching the block ID we are describing, *not* the `BLOCKINFO` block itself. The abbreviations defined in `BLOCKINFO` blocks receive abbreviation IDs as described in DEFINE_ABBREV.

The BLOCKNAME record (code 2) can optionally occur in this block. The elements of the record are the bytes of the string name of the block. llvm-bcanalyzer can use this to dump out bitcode files symbolically.

The SETRECORDNAME record (code 3) can also optionally occur in this block. The first operand value is a record ID number, and the rest of the elements of the record are the bytes for the string name of the record. llvm-bcanalyzer can use this to dump out bitcode files symbolically.

Note that although the data in BLOCKINFO blocks is described as "metadata," the abbreviations they contain are essential for parsing records from the corresponding blocks. It is not safe to skip them.

4.2.4 Bitcode Wrapper Format

Bitcode files for LLVM IR may optionally be wrapped in a simple wrapper structure. This structure contains a simple header that indicates the offset and size of the embedded BC file. This allows additional information to be stored alongside the BC file. The structure of this file header is:

[$Magic_{32}$, $Version_{32}$, $Offset_{32}$, $Size_{32}$, $CPUType_{32}$]

Each of the fields are 32-bit fields stored in little endian form (as with the rest of the bitcode file fields). The Magic number is always 0x0B17C0DE and the version is currently always 0. The Offset field is the offset in bytes to the start of the bitcode stream in the file, and the Size field is the size in bytes of the stream. CPUType is a target-specific value that can be used to encode the CPU of the target.

4.2.5 Native Object File Wrapper Format

Bitcode files for LLVM IR may also be wrapped in a native object file (i.e. ELF, COFF, Mach-O). The bitcode must be stored in a section of the object file named .llvmbc. This wrapper format is useful for accommodating LTO in compilation pipelines where intermediate objects must be native object files which contain metadata in other sections.

Not all tools support this format.

4.2.6 LLVM IR Encoding

LLVM IR is encoded into a bitstream by defining blocks and records. It uses blocks for things like constant pools, functions, symbol tables, etc. It uses records for things like instructions, global variable descriptors, type descriptions, etc. This document does not describe the set of abbreviations that the writer uses, as these are fully self-described in the file, and the reader is not allowed to build in any knowledge of this.

Basics

LLVM IR Magic Number

The magic number for LLVM IR files is:

[$0x0_4$, $0xC_4$, $0xE_4$, $0xD_4$]

When combined with the bitcode magic number and viewed as bytes, this is "BC 0xC0DE".

Signed VBRs

Variable Width Integer encoding is an efficient way to encode arbitrary sized unsigned values, but is an extremely inefficient for encoding signed values, as signed values are otherwise treated as maximally large unsigned values.

As such, signed VBR values of a specific width are emitted as follows:

- Positive values are emitted as VBRs of the specified width, but with their value shifted left by one.
- Negative values are emitted as VBRs of the specified width, but the negated value is shifted left by one, and the low bit is set.

With this encoding, small positive and small negative values can both be emitted efficiently. Signed VBR encoding is used in `CST_CODE_INTEGER` and `CST_CODE_WIDE_INTEGER` records within `CONSTANTS_BLOCK` blocks. It is also used for phi instruction operands in MODULE_CODE_VERSION 1.

LLVM IR Blocks

LLVM IR is defined with the following blocks:

- 8 — MODULE_BLOCK — This is the top-level block that contains the entire module, and describes a variety of per-module information.
- 9 — PARAMATTR_BLOCK — This enumerates the parameter attributes.
- 10 — TYPE_BLOCK — This describes all of the types in the module.
- 11 — CONSTANTS_BLOCK — This describes constants for a module or function.
- 12 — FUNCTION_BLOCK — This describes a function body.
- 13 — TYPE_SYMTAB_BLOCK — This describes the type symbol table.
- 14 — VALUE_SYMTAB_BLOCK — This describes a value symbol table.
- 15 — METADATA_BLOCK — This describes metadata items.
- 16 — METADATA_ATTACHMENT — This contains records associating metadata with function instruction values.

MODULE_BLOCK Contents

The `MODULE_BLOCK` block (id 8) is the top-level block for LLVM bitcode files, and each bitcode file must contain exactly one. In addition to records (described below) containing information about the module, a `MODULE_BLOCK` block may contain the following sub-blocks:

- BLOCKINFO
- PARAMATTR_BLOCK
- TYPE_BLOCK
- TYPE_SYMTAB_BLOCK
- VALUE_SYMTAB_BLOCK
- CONSTANTS_BLOCK
- FUNCTION_BLOCK
- METADATA_BLOCK

MODULE_CODE_VERSION Record

```
[VERSION, version#]
```

The VERSION record (code 1) contains a single value indicating the format version. Versions 0 and 1 are supported at this time. The difference between version 0 and 1 is in the encoding of instruction operands in each FUNCTION_BLOCK.

In version 0, each value defined by an instruction is assigned an ID unique to the function. Function-level value IDs are assigned starting from NumModuleValues since they share the same namespace as module-level values. The value enumerator resets after each function. When a value is an operand of an instruction, the value ID is used to represent the operand. For large functions or large modules, these operand values can be large.

The encoding in version 1 attempts to avoid large operand values in common cases. Instead of using the value ID directly, operands are encoded as relative to the current instruction. Thus, if an operand is the value defined by the previous instruction, the operand will be encoded as 1.

For example, instead of

```
#n = load #n-1
#n+1 = icmp eq #n, #const0
br #n+1, label #(bb1), label #(bb2)
```

version 1 will encode the instructions as

```
#n = load #1
#n+1 = icmp eq #1, (#n+1)-#const0
br #1, label #(bb1), label #(bb2)
```

Note in the example that operands which are constants also use the relative encoding, while operands like basic block labels do not use the relative encoding.

Forward references will result in a negative value. This can be inefficient, as operands are normally encoded as unsigned VBRs. However, forward references are rare, except in the case of phi instructions. For phi instructions, operands are encoded as Signed VBRs to deal with forward references.

MODULE_CODE_TRIPLE Record

```
[TRIPLE, ...string...]
```

The TRIPLE record (code 2) contains a variable number of values representing the bytes of the `target triple` specification string.

MODULE_CODE_DATALAYOUT Record

```
[DATALAYOUT, ...string...]
```

The DATALAYOUT record (code 3) contains a variable number of values representing the bytes of the `target datalayout` specification string.

MODULE_CODE_ASM Record

```
[ASM, ...string...]
```

The ASM record (code 4) contains a variable number of values representing the bytes of `module asm` strings, with individual assembly blocks separated by newline (ASCII 10) characters.

MODULE_CODE_SECTIONNAME Record

```
[SECTIONNAME, ...string...]
```

The SECTIONNAME record (code 5) contains a variable number of values representing the bytes of a single section name string. There should be one SECTIONNAME record for each section name referenced (e.g., in global variable or function section attributes) within the module. These records can be referenced by the 1-based index in the *section* fields of GLOBALVAR or FUNCTION records.

MODULE_CODE_DEPLIB Record

```
[DEPLIB, ...string...]
```

The DEPLIB record (code 6) contains a variable number of values representing the bytes of a single dependent library name string, one of the libraries mentioned in a deplibs declaration. There should be one DEPLIB record for each library name referenced.

MODULE_CODE_GLOBALVAR Record

```
[GLOBALVAR, pointer type, isconst, initid, linkage, alignment, section,
visibility, threadlocal, unnamed_addr, dllstorageclass]
```

The GLOBALVAR record (code 7) marks the declaration or definition of a global variable. The operand fields are:

- *pointer type*: The type index of the pointer type used to point to this global variable
- *isconst*: Non-zero if the variable is treated as constant within the module, or zero if it is not
- *initid*: If non-zero, the value index of the initializer for this variable, plus 1.
- *linkage*: An encoding of the linkage type for this variable: * external: code 0 * weak: code 1 * appending: code 2 * internal: code 3 * linkonce: code 4 * dllimport: code 5 * dllexport: code 6 * extern_weak: code 7 * common: code 8 * private: code 9 * weak_odr: code 10 * linkonce_odr: code 11 * available_externally: code 12 * deprecated : code 13 * deprecated : code 14
- alignment*: The logarithm base 2 of the variable's requested alignment, plus 1
- *section*: If non-zero, the 1-based section index in the table of MODULE_CODE_SECTIONNAME entries.
- *visibility*: If present, an encoding of the visibility of this variable: * default: code 0 * hidden: code 1 * protected: code 2
- *threadlocal*: If present, an encoding of the thread local storage mode of the variable: * not thread local: code 0 * thread local; default TLS model: code 1 * localdynamic: code 2 * initialexec: code 3 * localexec: code 4
- *unnamed_addr*: If present and non-zero, indicates that the variable has unnamed_addr
- *dllstorageclass*: If present, an encoding of the DLL storage class of this variable:
  - default: code 0
  - dllimport: code 1
  - dllexport: code 2

MODULE_CODE_FUNCTION Record

`[FUNCTION, type, callingconv, isproto, linkage, paramattr, alignment, section, visibility, gc, prefix, dllstorageclass]`

The `FUNCTION` record (code 8) marks the declaration or definition of a function. The operand fields are:

- *type*: The type index of the function type describing this function
- *callingconv*: The calling convention number: * `ccc`: code 0 * `fastcc`: code 8 * `coldcc`: code 9 * `webkit_jscc`: code 12 * `anyregcc`: code 13 * `preserve_mostcc`: code 14 * `preserve_allcc`: code 15 * `x86_stdcallcc`: code 64 * `x86_fastcallcc`: code 65 * `arm_apcscc`: code 66 * `arm_aapcscc`: code 67 * `arm_aapcs_vfpcc`: code 68
- isproto*: Non-zero if this entry represents a declaration rather than a definition
- *linkage*: An encoding of the linkage type for this function
- *paramattr*: If nonzero, the 1-based parameter attribute index into the table of PARAMATTR_CODE_ENTRY entries.
- *alignment*: The logarithm base 2 of the function's requested alignment, plus 1
- *section*: If non-zero, the 1-based section index in the table of MODULE_CODE_SECTIONNAME entries.
- *visibility*: An encoding of the visibility of this function
- *gc*: If present and nonzero, the 1-based garbage collector index in the table of MODULE_CODE_GCNAME entries.
- *unnamed_addr*: If present and non-zero, indicates that the function has `unnamed_addr`
- *prefix*: If non-zero, the value index of the prefix data for this function, plus 1.
- *dllstorageclass*: An encoding of the *dllstorageclass* of this function

MODULE_CODE_ALIAS Record

`[ALIAS, alias type, aliasee val#, linkage, visibility, dllstorageclass]`

The `ALIAS` record (code 9) marks the definition of an alias. The operand fields are

- *alias type*: The type index of the alias
- *aliasee val#*: The value index of the aliased value
- *linkage*: An encoding of the linkage type for this alias
- *visibility*: If present, an encoding of the visibility of the alias
- *dllstorageclass*: If present, an encoding of the *dllstorageclass* of the alias

MODULE_CODE_PURGEVALS Record

`[PURGEVALS, numvals]`

The `PURGEVALS` record (code 10) resets the module-level value list to the size given by the single operand value. Module-level value list items are added by `GLOBALVAR`, `FUNCTION`, and `ALIAS` records. After a `PURGEVALS` record is seen, new value indices will start from the given *numvals* value.

MODULE_CODE_GCNAME Record

```
[GCNAME, ...string...]
```

The GCNAME record (code 11) contains a variable number of values representing the bytes of a single garbage collector name string. There should be one GCNAME record for each garbage collector name referenced in function gc attributes within the module. These records can be referenced by 1-based index in the *gc* fields of FUNCTION records.

PARAMATTR_BLOCK Contents

The PARAMATTR_BLOCK block (id 9) contains a table of entries describing the attributes of function parameters. These entries are referenced by 1-based index in the *paramattr* field of module block FUNCTION records, or within the *attr* field of function block INST_INVOKE and INST_CALL records.

Entries within PARAMATTR_BLOCK are constructed to ensure that each is unique (i.e., no two indicies represent equivalent attribute lists).

PARAMATTR_CODE_ENTRY Record

```
[ENTRY, paramidx0, attr0, paramidx1, attr1...]
```

The ENTRY record (code 1) contains an even number of values describing a unique set of function parameter attributes. Each *paramidx* value indicates which set of attributes is represented, with 0 representing the return value attributes, 0xFFFFFFFF representing function attributes, and other values representing 1-based function parameters. Each *attr* value is a bitmap with the following interpretation:

- bit 0: zeroext
- bit 1: signext
- bit 2: noreturn
- bit 3: inreg
- bit 4: sret
- bit 5: nounwind
- bit 6: noalias
- bit 7: byval
- bit 8: nest
- bit 9: readnone
- bit 10: readonly
- bit 11: noinline
- bit 12: alwaysinline
- bit 13: optsize
- bit 14: ssp
- bit 15: sspreq
- bits 16-31: align n
- bit 32: nocapture
- bit 33: noredzone

- bit 34: `noimplicitfloat`
- bit 35: `naked`
- bit 36: `inlinehint`
- bits 37-39: `alignstack n`, represented as the logarithm base 2 of the requested alignment, plus 1

TYPE_BLOCK Contents

The `TYPE_BLOCK` block (id 10) contains records which constitute a table of type operator entries used to represent types referenced within an LLVM module. Each record (with the exception of NUMENTRY) generates a single type table entry, which may be referenced by 0-based index from instructions, constants, metadata, type symbol table entries, or other type operator records.

Entries within `TYPE_BLOCK` are constructed to ensure that each entry is unique (i.e., no two indicies represent structurally equivalent types).

TYPE_CODE_NUMENTRY Record

```
[NUMENTRY, numentries]
```

The `NUMENTRY` record (code 1) contains a single value which indicates the total number of type code entries in the type table of the module. If present, `NUMENTRY` should be the first record in the block.

TYPE_CODE_VOID Record

```
[VOID]
```

The `VOID` record (code 2) adds a `void` type to the type table.

TYPE_CODE_HALF Record

```
[HALF]
```

The `HALF` record (code 10) adds a `half` (16-bit floating point) type to the type table.

TYPE_CODE_FLOAT Record

```
[FLOAT]
```

The `FLOAT` record (code 3) adds a `float` (32-bit floating point) type to the type table.

TYPE_CODE_DOUBLE Record

```
[DOUBLE]
```

The `DOUBLE` record (code 4) adds a `double` (64-bit floating point) type to the type table.

TYPE_CODE_LABEL Record

```
[LABEL]
```

The `LABEL` record (code 5) adds a `label` type to the type table.

TYPE_CODE_OPAQUE Record

```
[OPAQUE]
```

The `OPAQUE` record (code 6) adds an `opaque` type to the type table. Note that distinct `opaque` types are not unified.

TYPE_CODE_INTEGER Record

```
[INTEGER, width]
```

The `INTEGER` record (code 7) adds an integer type to the type table. The single *width* field indicates the width of the integer type.

TYPE_CODE_POINTER Record

```
[POINTER, pointee type, address space]
```

The `POINTER` record (code 8) adds a pointer type to the type table. The operand fields are

- *pointee type*: The type index of the pointed-to type
- *address space*: If supplied, the target-specific numbered address space where the pointed-to object resides. Otherwise, the default address space is zero.

TYPE_CODE_FUNCTION Record

```
[FUNCTION, vararg, ignored, retty, ...paramty...  ]
```

The `FUNCTION` record (code 9) adds a function type to the type table. The operand fields are

- *vararg*: Non-zero if the type represents a varargs function
- *ignored*: This value field is present for backward compatibility only, and is ignored
- *retty*: The type index of the function's return type
- *paramty*: Zero or more type indices representing the parameter types of the function

TYPE_CODE_STRUCT Record

```
[STRUCT, ispacked, ...eltty...]
```

The `STRUCT` record (code 10) adds a struct type to the type table. The operand fields are

- *ispacked*: Non-zero if the type represents a packed structure
- *eltty*: Zero or more type indices representing the element types of the structure

TYPE_CODE_ARRAY Record

`[ARRAY, numelts, eltty]`

The `ARRAY` record (code 11) adds an array type to the type table. The operand fields are

- *numelts*: The number of elements in arrays of this type
- *eltty*: The type index of the array element type

TYPE_CODE_VECTOR Record

`[VECTOR, numelts, eltty]`

The `VECTOR` record (code 12) adds a vector type to the type table. The operand fields are

- *numelts*: The number of elements in vectors of this type
- *eltty*: The type index of the vector element type

TYPE_CODE_X86_FP80 Record

`[X86_FP80]`

The `X86_FP80` record (code 13) adds an `x86_fp80` (80-bit floating point) type to the type table.

TYPE_CODE_FP128 Record

`[FP128]`

The `FP128` record (code 14) adds an `fp128` (128-bit floating point) type to the type table.

TYPE_CODE_PPC_FP128 Record

`[PPC_FP128]`

The `PPC_FP128` record (code 15) adds a `ppc_fp128` (128-bit floating point) type to the type table.

TYPE_CODE_METADATA Record

`[METADATA]`

The `METADATA` record (code 16) adds a `metadata` type to the type table.

CONSTANTS_BLOCK Contents

The `CONSTANTS_BLOCK` block (id 11) ...

FUNCTION_BLOCK Contents

The `FUNCTION_BLOCK` block (id 12) ...

In addition to the record types described below, a `FUNCTION_BLOCK` block may contain the following sub-blocks:

- CONSTANTS_BLOCK
- VALUE_SYMTAB_BLOCK
- METADATA_ATTACHMENT

TYPE_SYMTAB_BLOCK Contents

The `TYPE_SYMTAB_BLOCK` block (id 13) contains entries which map between module-level named types and their corresponding type indices.

TST_CODE_ENTRY Record

```
[ENTRY, typeid, ...string...]
```

The `ENTRY` record (code 1) contains a variable number of values, with the first giving the type index of the designated type, and the remaining values giving the character codes of the type name. Each entry corresponds to a single named type.

VALUE_SYMTAB_BLOCK Contents

The `VALUE_SYMTAB_BLOCK` block (id 14) ...

METADATA_BLOCK Contents

The `METADATA_BLOCK` block (id 15) ...

METADATA_ATTACHMENT Contents

The `METADATA_ATTACHMENT` block (id 16) ...

4.3 LLVM Block Frequency Terminology

4.3.1 Introduction

Block Frequency is a metric for estimating the relative frequency of different basic blocks. This document describes the terminology that the `BlockFrequencyInfo` and `MachineBlockFrequencyInfo` analysis passes use.

4.3.2 Branch Probability

Blocks with multiple successors have probabilities associated with each outgoing edge. These are called branch probabilities. For a given block, the sum of its outgoing branch probabilities should be 1.0.

4.3.3 Branch Weight

Rather than storing fractions on each edge, we store an integer weight. Weights are relative to the other edges of a given predecessor block. The branch probability associated with a given edge is its own weight divided by the sum of the weights on the predecessor's outgoing edges.

For example, consider this IR:

```
define void @foo() {
    ; ...
    A:
        br i1 %cond, label %B, label %C, !prof !0
    ; ...
}
!0 = metadata !{metadata !"branch_weights", i32 7, i32 8}
```

and this simple graph representation:

```
A -> B  (edge-weight: 7)
A -> C  (edge-weight: 8)
```

The probability of branching from block A to block B is 7/15, and the probability of branching from block A to block C is 8/15.

See *LLVM Branch Weight Metadata* for details about the branch weight IR representation.

4.3.4 Block Frequency

Block frequency is a relative metric that represents the number of times a block executes. The ratio of a block frequency to the entry block frequency is the expected number of times the block will execute per entry to the function.

Block frequency is the main output of the `BlockFrequencyInfo` and `MachineBlockFrequencyInfo` analysis passes.

4.3.5 Implementation: a series of DAGs

The implementation of the block frequency calculation analyses each loop, bottom-up, ignoring backedges; i.e., as a DAG. After each loop is processed, it's packaged up to act as a pseudo-node in its parent loop's (or the function's) DAG analysis.

4.3.6 Block Mass

For each DAG, the entry node is assigned a mass of `UINT64_MAX` and mass is distributed to successors according to branch weights. Block Mass uses a fixed-point representation where `UINT64_MAX` represents `1.0` and `0` represents a number just above `0.0`.

After mass is fully distributed, in any cut of the DAG that separates the exit nodes from the entry node, the sum of the block masses of the nodes succeeded by a cut edge should equal `UINT64_MAX`. In other words, mass is conserved as it "falls" through the DAG.

If a function's basic block graph is a DAG, then block masses are valid block frequencies. This works poorly in practise though, since downstream users rely on adding block frequencies together without hitting the maximum.

4.3.7 Loop Scale

Loop scale is a metric that indicates how many times a loop iterates per entry. As mass is distributed through the loop's DAG, the (otherwise ignored) backedge mass is collected. This backedge mass is used to compute the exit frequency, and thus the loop scale.

4.3.8 Implementation: Getting from mass and scale to frequency

After analysing the complete series of DAGs, each block has a mass (local to its containing loop, if any), and each loop pseudo-node has a loop scale and its own mass (from its parent's DAG).

We can get an initial frequency assignment (with entry frequency of 1.0) by multiplying these masses and loop scales together. A given block's frequency is the product of its mass, the mass of containing loops' pseudo nodes, and the containing loops' loop scales.

Since downstream users need integers (not floating point), this initial frequency assignment is shifted as necessary into the range of `uint64_t`.

4.3.9 Block Bias

Block bias is a proposed *absolute* metric to indicate a bias toward or away from a given block during a function's execution. The idea is that bias can be used in isolation to indicate whether a block is relatively hot or cold, or to compare two blocks to indicate whether one is hotter or colder than the other.

The proposed calculation involves calculating a *reference* block frequency, where:

- every branch weight is assumed to be 1 (i.e., every branch probability distribution is even) and
- loop scales are ignored.

This reference frequency represents what the block frequency would be in an unbiased graph.

The bias is the ratio of the block frequency to this reference block frequency.

4.4 LLVM Branch Weight Metadata

4.4.1 Introduction

Branch Weight Metadata represents branch weights as its likeliness to be taken (see *LLVM Block Frequency Terminology*). Metadata is assigned to the `TerminatorInst` as a `MDNode` of the `MD_prof` kind. The first operator is always a `MDString` node with the string "branch_weights". Number of operators depends on the terminator type.

Branch weights might be fetch from the profiling file, or generated based on __builtin_expect instruction.

All weights are represented as an unsigned 32-bit values, where higher value indicates greater chance to be taken.

4.4.2 Supported Instructions

`BranchInst`

Metadata is only assigned to the conditional branches. There are two extra operarands for the true and the false branch.

```
!0 = metadata !{
  metadata !"branch_weights",
  i32 <TRUE_BRANCH_WEIGHT>,
  i32 <FALSE_BRANCH_WEIGHT>
}
```

`SwitchInst`

Branch weights are assigned to every case (including the `default` case which is always case #0).

```
!0 = metadata !{
  metadata !"branch_weights",
  i32 <DEFAULT_BRANCH_WEIGHT>
  [ , i32 <CASE_BRANCH_WEIGHT> ... ]
}
```

`IndirectBrInst`

Branch weights are assigned to every destination.

```
!0 = metadata !{
  metadata !"branch_weights",
  i32 <LABEL_BRANCH_WEIGHT>
```

```
  [ , i32 <LABEL_BRANCH_WEIGHT> ... ]
}
```

Other

Other terminator instructions are not allowed to contain Branch Weight Metadata.

4.4.3 Built-in `expect` Instructions

`__builtin_expect(long exp, long c)` instruction provides branch prediction information. The return value is the value of `exp`.

It is especially useful in conditional statements. Currently Clang supports two conditional statements:

`if` statement

The `exp` parameter is the condition. The `c` parameter is the expected comparison value. If it is equal to 1 (true), the condition is likely to be true, in other case condition is likely to be false. For example:

```
if (__builtin_expect(x > 0, 1)) {
  // This block is likely to be taken.
}
```

`switch` statement

The `exp` parameter is the value. The `c` parameter is the expected value. If the expected value doesn't show on the cases list, the `default` case is assumed to be likely taken.

```
switch (__builtin_expect(x, 5)) {
default: break;
case 0:  // ...
case 3:  // ...
case 5:  // This case is likely to be taken.
}
```

4.4.4 CFG Modifications

Branch Weight Metatada is not proof against CFG changes. If terminator operands' are changed some action should be taken. In other case some misoptimizations may occur due to incorrent branch prediction information.

4.5 LLVM bugpoint tool: design and usage

4.5.1 Description

`bugpoint` narrows down the source of problems in LLVM tools and passes. It can be used to debug three types of failures: optimizer crashes, miscompilations by optimizers, or bad native code generation (including problems in the static and JIT compilers). It aims to reduce large test cases to small, useful ones. For example, if `opt` crashes while optimizing a file, it will identify the optimization (or combination of optimizations) that causes the crash, and reduce the file down to a small example which triggers the crash.

For detailed case scenarios, such as debugging `opt`, or one of the LLVM code generators, see *How to submit an LLVM bug report*.

4.5.2 Design Philosophy

`bugpoint` is designed to be a useful tool without requiring any hooks into the LLVM infrastructure at all. It works with any and all LLVM passes and code generators, and does not need to "know" how they work. Because of this, it may appear to do stupid things or miss obvious simplifications. `bugpoint` is also designed to trade off programmer time for computer time in the compiler-debugging process; consequently, it may take a long period of (unattended) time to reduce a test case, but we feel it is still worth it. Note that `bugpoint` is generally very quick unless debugging a miscompilation where each test of the program (which requires executing it) takes a long time.

Automatic Debugger Selection

`bugpoint` reads each `.bc` or `.ll` file specified on the command line and links them together into a single module, called the test program. If any LLVM passes are specified on the command line, it runs these passes on the test program. If any of the passes crash, or if they produce malformed output (which causes the verifier to abort), `bugpoint` starts the crash debugger.

Otherwise, if the `-output` option was not specified, `bugpoint` runs the test program with the "safe" backend (which is assumed to generate good code) to generate a reference output. Once `bugpoint` has a reference output for the test program, it tries executing it with the selected code generator. If the selected code generator crashes, `bugpoint` starts the crash debugger on the code generator. Otherwise, if the resulting output differs from the reference output, it assumes the difference resulted from a code generator failure, and starts the code generator debugger.

Finally, if the output of the selected code generator matches the reference output, `bugpoint` runs the test program after all of the LLVM passes have been applied to it. If its output differs from the reference output, it assumes the difference resulted from a failure in one of the LLVM passes, and enters the miscompilation debugger. Otherwise, there is no problem `bugpoint` can debug.

Crash debugger

If an optimizer or code generator crashes, `bugpoint` will try as hard as it can to reduce the list of passes (for optimizer crashes) and the size of the test program. First, `bugpoint` figures out which combination of optimizer passes triggers the bug. This is useful when debugging a problem exposed by `opt`, for example, because it runs over 38 passes.

Next, `bugpoint` tries removing functions from the test program, to reduce its size. Usually it is able to reduce a test program to a single function, when debugging intraprocedural optimizations. Once the number of functions has been reduced, it attempts to delete various edges in the control flow graph, to reduce the size of the function as much as possible. Finally, `bugpoint` deletes any individual LLVM instructions whose absence does not eliminate the failure. At the end, `bugpoint` should tell you what passes crash, give you a bitcode file, and give you instructions on how to reproduce the failure with `opt` or `llc`.

Code generator debugger

The code generator debugger attempts to narrow down the amount of code that is being miscompiled by the selected code generator. To do this, it takes the test program and partitions it into two pieces: one piece which it compiles with the "safe" backend (into a shared object), and one piece which it runs with either the JIT or the static LLC compiler. It uses several techniques to reduce the amount of code pushed through the LLVM code generator, to reduce the potential scope of the problem. After it is finished, it emits two bitcode files (called "test" [to be compiled with the code generator] and "safe" [to be compiled with the "safe" backend], respectively), and instructions for reproducing the problem. The code generator debugger assumes that the "safe" backend produces good code.

Miscompilation debugger

The miscompilation debugger works similarly to the code generator debugger. It works by splitting the test program into two pieces, running the optimizations specified on one piece, linking the two pieces back together, and then executing the result. It attempts to narrow down the list of passes to the one (or few) which are causing the miscompilation, then reduce the portion of the test program which is being miscompiled. The miscompilation debugger assumes that the selected code generator is working properly.

4.5.3 Advice for using bugpoint

`bugpoint` can be a remarkably useful tool, but it sometimes works in non-obvious ways. Here are some hints and tips:

- In the code generator and miscompilation debuggers, `bugpoint` only works with programs that have deterministic output. Thus, if the program outputs `argv[0]`, the date, time, or any other "random" data, `bugpoint` may misinterpret differences in these data, when output, as the result of a miscompilation. Programs should be temporarily modified to disable outputs that are likely to vary from run to run.
- In the code generator and miscompilation debuggers, debugging will go faster if you manually modify the program or its inputs to reduce the runtime, but still exhibit the problem.
- `bugpoint` is extremely useful when working on a new optimization: it helps track down regressions quickly. To avoid having to relink `bugpoint` every time you change your optimization however, have `bugpoint` dynamically load your optimization with the `-load` option.
- `bugpoint` can generate a lot of output and run for a long period of time. It is often useful to capture the output of the program to file. For example, in the C shell, you can run:

  ```
  $ bugpoint  ... |& tee bugpoint.log
  ```

  to get a copy of `bugpoint`'s output in the file `bugpoint.log`, as well as on your terminal.

- `bugpoint` cannot debug problems with the LLVM linker. If `bugpoint` crashes before you see its "All input ok" message, you might try `llvm-link -v` on the same set of input files. If that also crashes, you may be experiencing a linker bug.
- `bugpoint` is useful for proactively finding bugs in LLVM. Invoking `bugpoint` with the `-find-bugs` option will cause the list of specified optimizations to be randomized and applied to the program. This process will repeat until a bug is found or the user kills `bugpoint`.

4.5.4 What to do when bugpoint isn't enough

Sometimes, `bugpoint` is not enough. In particular, InstCombine and TargetLowering both have visitor structured code with lots of potential transformations. If the process of using bugpoint has left you with still too much code to figure out and the problem seems to be in instcombine, the following steps may help. These same techniques are useful with TargetLowering as well.

Turn on `-debug-only=instcombine` and see which transformations within instcombine are firing by selecting out lines with "`IC`" in them.

At this point, you have a decision to make. Is the number of transformations small enough to step through them using a debugger? If so, then try that.

If there are too many transformations, then a source modification approach may be helpful. In this approach, you can modify the source code of instcombine to disable just those transformations that are being performed on your test input and perform a binary search over the set of transformations. One set of places to modify are the "`visit*`" methods of `InstCombiner` (*e.g.* `visitICmpInst`) by adding a "`return false`" as the first line of the method.

If that still doesn't remove enough, then change the caller of `InstCombiner::DoOneIteration`, `InstCombiner::runOnFunction` to limit the number of iterations.

You may also find it useful to use "`-stats`" now to see what parts of instcombine are firing. This can guide where to put additional reporting code.

At this point, if the amount of transformations is still too large, then inserting code to limit whether or not to execute the body of the code in the visit function can be helpful. Add a static counter which is incremented on every invocation of the function. Then add code which simply returns false on desired ranges. For example:

```
static int calledCount = 0;
calledCount++;
DEBUG(if (calledCount < 212) return false);
DEBUG(if (calledCount > 217) return false);
DEBUG(if (calledCount == 213) return false);
DEBUG(if (calledCount == 214) return false);
DEBUG(if (calledCount == 215) return false);
DEBUG(if (calledCount == 216) return false);
DEBUG(dbgs() << "visitXOR calledCount: " << calledCount << "\n");
DEBUG(dbgs() << "I: "; I->dump());
```

could be added to `visitXOR` to limit `visitXor` to being applied only to calls 212 and 217. This is from an actual test case and raises an important point—a simple binary search may not be sufficient, as transformations that interact may require isolating more than one call. In TargetLowering, use `return SDNode();` instead of `return false;`.

Now that that the number of transformations is down to a manageable number, try examining the output to see if you can figure out which transformations are being done. If that can be figured out, then do the usual debugging. If which code corresponds to the transformation being performed isn't obvious, set a breakpoint after the call count based disabling and step through the code. Alternatively, you can use "`printf`" style debugging to report waypoints.

4.6 The LLVM Target-Independent Code Generator

Warning: This is a work in progress.

4.6.1 Introduction

The LLVM target-independent code generator is a framework that provides a suite of reusable components for translating the LLVM internal representation to the machine code for a specified target—either in assembly form (suitable for a static compiler) or in binary machine code format (usable for a JIT compiler). The LLVM target-independent code generator consists of six main components:

1. Abstract target description interfaces which capture important properties about various aspects of the machine, independently of how they will be used. These interfaces are defined in `include/llvm/Target/`.
2. Classes used to represent the code being generated for a target. These classes are intended to be abstract enough to represent the machine code for *any* target machine. These classes are defined in `include/llvm/CodeGen/`. At this level, concepts like "constant pool entries" and "jump tables" are explicitly exposed.
3. Classes and algorithms used to represent code as the object file level, the MC Layer. These classes represent assembly level constructs like labels, sections, and instructions. At this level, concepts like "constant pool entries" and "jump tables" don't exist.
4. Target-independent algorithms used to implement various phases of native code generation (register allocation, scheduling, stack frame representation, etc). This code lives in `lib/CodeGen/`.
5. Implementations of the abstract target description interfaces for particular targets. These machine descriptions make use of the components provided by LLVM, and can optionally provide custom target-specific passes, to build complete code generators for a specific target. Target descriptions live in `lib/Target/`.
6. The target-independent JIT components. The LLVM JIT is completely target independent (it uses the `TargetJITInfo` structure to interface for target-specific issues. The code for the target-independent JIT lives in `lib/ExecutionEngine/JIT`.

Depending on which part of the code generator you are interested in working on, different pieces of this will be useful to you. In any case, you should be familiar with the target description and machine code representation classes. If you want to add a backend for a new target, you will need to implement the target description classes for your new target and understand the *LLVM code representation*. If you are interested in implementing a new code generation algorithm, it should only depend on the target-description and machine code representation classes, ensuring that it is portable.

Required components in the code generator

The two pieces of the LLVM code generator are the high-level interface to the code generator and the set of reusable components that can be used to build target-specific backends. The two most important interfaces (TargetMachine and DataLayout) are the only ones that are required to be defined for a backend to fit into the LLVM system, but the others must be defined if the reusable code generator components are going to be used.

This design has two important implications. The first is that LLVM can support completely non-traditional code generation targets. For example, the C backend does not require register allocation, instruction selection, or any of the other standard components provided by the system. As such, it only implements these two interfaces, and does its own thing. Note that C backend was removed from the trunk since LLVM 3.1 release. Another example of a code generator like this is a (purely hypothetical) backend that converts LLVM to the GCC RTL form and uses GCC to emit machine code for a target.

This design also implies that it is possible to design and implement radically different code generators in the LLVM system that do not make use of any of the built-in components. Doing so is not recommended at all, but could be required for radically different targets that do not fit into the LLVM machine description model: FPGAs for example.

The high-level design of the code generator

The LLVM target-independent code generator is designed to support efficient and quality code generation for standard register-based microprocessors. Code generation in this model is divided into the following stages:

1. Instruction Selection — This phase determines an efficient way to express the input LLVM code in the target instruction set. This stage produces the initial code for the program in the target instruction set, then makes use of virtual registers in SSA form and physical registers that represent any required register assignments due to target constraints or calling conventions. This step turns the LLVM code into a DAG of target instructions.
2. Scheduling and Formation — This phase takes the DAG of target instructions produced by the instruction selection phase, determines an ordering of the instructions, then emits the instructions as MachineInstrs with that ordering. Note that we describe this in the instruction selection section because it operates on a SelectionDAG.
3. SSA-based Machine Code Optimizations — This optional stage consists of a series of machine-code optimizations that operate on the SSA-form produced by the instruction selector. Optimizations like modulo-scheduling or peephole optimization work here.
4. Register Allocation — The target code is transformed from an infinite virtual register file in SSA form to the concrete register file used by the target. This phase introduces spill code and eliminates all virtual register references from the program.
5. Prolog/Epilog Code Insertion — Once the machine code has been generated for the function and the amount of stack space required is known (used for LLVM alloca's and spill slots), the prolog and epilog code for the function can be inserted and "abstract stack location references" can be eliminated. This stage is responsible for implementing optimizations like frame-pointer elimination and stack packing.
6. Late Machine Code Optimizations — Optimizations that operate on "final" machine code can go here, such as spill code scheduling and peephole optimizations.
7. Code Emission — The final stage actually puts out the code for the current function, either in the target assembler format or in machine code.

The code generator is based on the assumption that the instruction selector will use an optimal pattern matching selector to create high-quality sequences of native instructions. Alternative code generator designs based on pattern expansion and aggressive iterative peephole optimization are much slower. This design permits efficient compilation (important for JIT environments) and aggressive optimization (used when generating code offline) by allowing components of varying levels of sophistication to be used for any step of compilation.

In addition to these stages, target implementations can insert arbitrary target-specific passes into the flow. For example, the X86 target uses a special pass to handle the 80x87 floating point stack architecture. Other targets with unusual requirements can be supported with custom passes as needed.

Using TableGen for target description

The target description classes require a detailed description of the target architecture. These target descriptions often have a large amount of common information (e.g., an `add` instruction is almost identical to a `sub` instruction). In order to allow the maximum amount of commonality to be factored out, the LLVM code generator uses the *TableGen* tool to describe big chunks of the target machine, which allows the use of domain-specific and target-specific abstractions to reduce the amount of repetition.

As LLVM continues to be developed and refined, we plan to move more and more of the target description to the `.td` form. Doing so gives us a number of advantages. The most important is that it makes it easier to port LLVM because it reduces the amount of C++ code that has to be written, and the surface area of the code generator that needs to be understood before someone can get something working. Second, it makes it easier to change things. In particular, if tables and other things are all emitted by `tblgen`, we only need a change in one place (`tblgen`) to update all of the targets to a new interface.

4.6.2 Target description classes

The LLVM target description classes (located in the `include/llvm/Target` directory) provide an abstract description of the target machine independent of any particular client. These classes are designed to capture the *abstract* properties of the target (such as the instructions and registers it has), and do not incorporate any particular pieces of code generation algorithms.

All of the target description classes (except the DataLayout class) are designed to be subclassed by the concrete target implementation, and have virtual methods implemented. To get to these implementations, the TargetMachine class provides accessors that should be implemented by the target.

The `TargetMachine` class

The `TargetMachine` class provides virtual methods that are used to access the target-specific implementations of the various target description classes via the `get*Info` methods (`getInstrInfo`, `getRegisterInfo`, `getFrameInfo`, etc.). This class is designed to be specialized by a concrete target implementation (e.g., `X86TargetMachine`) which implements the various virtual methods. The only required target description class is the DataLayout class, but if the code generator components are to be used, the other interfaces should be implemented as well.

The `DataLayout` class

The `DataLayout` class is the only required target description class, and it is the only class that is not extensible (you cannot derive a new class from it). `DataLayout` specifies information about how the target lays out memory for structures, the alignment requirements for various data types, the size of pointers in the target, and whether the target is little-endian or big-endian.

The `TargetLowering` class

The `TargetLowering` class is used by SelectionDAG based instruction selectors primarily to describe how LLVM code should be lowered to SelectionDAG operations. Among other things, this class indicates:

- an initial register class to use for various `ValueTypes`,
- which operations are natively supported by the target machine,
- the return type of `setcc` operations,
- the type to use for shift amounts, and
- various high-level characteristics, like whether it is profitable to turn division by a constant into a multiplication sequence.

The `TargetRegisterInfo` class

The `TargetRegisterInfo` class is used to describe the register file of the target and any interactions between the registers.

Registers are represented in the code generator by unsigned integers. Physical registers (those that actually exist in the target description) are unique small numbers, and virtual registers are generally large. Note that register `#0` is reserved as a flag value.

Each register in the processor description has an associated `TargetRegisterDesc` entry, which provides a textual name for the register (used for assembly output and debugging dumps) and a set of aliases (used to indicate whether one register overlaps with another).

In addition to the per-register description, the `TargetRegisterInfo` class exposes a set of processor specific register classes (instances of the `TargetRegisterClass` class). Each register class contains sets of registers that have the same properties (for example, they are all 32-bit integer registers). Each SSA virtual register created by the instruction selector has an associated register class. When the register allocator runs, it replaces virtual registers with a physical register in the set.

The target-specific implementations of these classes is auto-generated from a *TableGen* description of the register file.

The `TargetInstrInfo` class

The `TargetInstrInfo` class is used to describe the machine instructions supported by the target. Descriptions define things like the mnemonic for the opcode, the number of operands, the list of implicit register uses and defs, whether the instruction has certain target-independent properties (accesses memory, is commutable, etc), and holds any target-specific flags.

The `TargetFrameLowering` class

The `TargetFrameLowering` class is used to provide information about the stack frame layout of the target. It holds the direction of stack growth, the known stack alignment on entry to each function, and the offset to the local area. The offset to the local area is the offset from the stack pointer on function entry to the first location where function data (local variables, spill locations) can be stored.

The `TargetSubtarget` class

The `TargetSubtarget` class is used to provide information about the specific chip set being targeted. A sub-target informs code generation of which instructions are supported, instruction latencies and instruction execution itinerary; i.e., which processing units are used, in what order, and for how long.

The `TargetJITInfo` class

The `TargetJITInfo` class exposes an abstract interface used by the Just-In-Time code generator to perform target-specific activities, such as emitting stubs. If a `TargetMachine` supports JIT code generation, it should provide one of these objects through the `getJITInfo` method.

4.6.3 Machine code description classes

At the high-level, LLVM code is translated to a machine specific representation formed out of MachineFunction , MachineBasicBlock , and MachineInstr instances (defined in `include/llvm/CodeGen`). This representation is completely target agnostic, representing instructions in their most abstract form: an opcode and a series of operands. This representation is designed to support both an SSA representation for machine code, as well as a register allocated, non-SSA form.

The `MachineInstr` class

Target machine instructions are represented as instances of the `MachineInstr` class. This class is an extremely abstract way of representing machine instructions. In particular, it only keeps track of an opcode number and a set of operands.

The opcode number is a simple unsigned integer that only has meaning to a specific backend. All of the instructions for a target should be defined in the `*InstrInfo.td` file for the target. The opcode enum values are auto-generated

from this description. The `MachineInstr` class does not have any information about how to interpret the instruction (i.e., what the semantics of the instruction are); for that you must refer to the TargetInstrInfo class.

The operands of a machine instruction can be of several different types: a register reference, a constant integer, a basic block reference, etc. In addition, a machine operand should be marked as a def or a use of the value (though only registers are allowed to be defs).

By convention, the LLVM code generator orders instruction operands so that all register definitions come before the register uses, even on architectures that are normally printed in other orders. For example, the SPARC add instruction: "`add %i1, %i2, %i3`" adds the "%i1", and "%i2" registers and stores the result into the "%i3" register. In the LLVM code generator, the operands should be stored as "`%i3, %i1, %i2`": with the destination first.

Keeping destination (definition) operands at the beginning of the operand list has several advantages. In particular, the debugging printer will print the instruction like this:

```
%r3 = add %i1, %i2
```

Also if the first operand is a def, it is easier to create instructions whose only def is the first operand.

Using the `MachineInstrBuilder.h` functions

Machine instructions are created by using the `BuildMI` functions, located in the `include/llvm/CodeGen/MachineInstrBuilder.h` file. The `BuildMI` functions make it easy to build arbitrary machine instructions. Usage of the `BuildMI` functions look like this:

```
// Create a 'DestReg = mov 42' (rendered in X86 assembly as 'mov DestReg, 42')
// instruction.  The '1' specifies how many operands will be added.
MachineInstr *MI = BuildMI(X86::MOV32ri, 1, DestReg).addImm(42);

// Create the same instr, but insert it at the end of a basic block.
MachineBasicBlock &MBB = ...
BuildMI(MBB, X86::MOV32ri, 1, DestReg).addImm(42);

// Create the same instr, but insert it before a specified iterator point.
MachineBasicBlock::iterator MBBI = ...
BuildMI(MBB, MBBI, X86::MOV32ri, 1, DestReg).addImm(42);

// Create a 'cmp Reg, 0' instruction, no destination reg.
MI = BuildMI(X86::CMP32ri, 2).addReg(Reg).addImm(0);

// Create an 'sahf' instruction which takes no operands and stores nothing.
MI = BuildMI(X86::SAHF, 0);

// Create a self looping branch instruction.
BuildMI(MBB, X86::JNE, 1).addMBB(&MBB);
```

The key thing to remember with the `BuildMI` functions is that you have to specify the number of operands that the machine instruction will take. This allows for efficient memory allocation. You also need to specify if operands default to be uses of values, not definitions. If you need to add a definition operand (other than the optional destination register), you must explicitly mark it as such:

```
MI.addReg(Reg, RegState::Define);
```

Fixed (preassigned) registers

One important issue that the code generator needs to be aware of is the presence of fixed registers. In particular, there are often places in the instruction stream where the register allocator *must* arrange for a particular value to be in a

particular register. This can occur due to limitations of the instruction set (e.g., the X86 can only do a 32-bit divide with the EAX/EDX registers), or external factors like calling conventions. In any case, the instruction selector should emit code that copies a virtual register into or out of a physical register when needed.

For example, consider this simple LLVM example:

```
define i32 @test(i32 %X, i32 %Y) {
  %Z = sdiv i32 %X, %Y
  ret i32 %Z
}
```

The X86 instruction selector might produce this machine code for the div and ret:

```
;; Start of div
%EAX = mov %reg1024           ;; Copy X (in reg1024) into EAX
%reg1027 = sar %reg1024, 31
%EDX = mov %reg1027           ;; Sign extend X into EDX
idiv %reg1025                 ;; Divide by Y (in reg1025)
%reg1026 = mov %EAX           ;; Read the result (Z) out of EAX

;; Start of ret
%EAX = mov %reg1026           ;; 32-bit return value goes in EAX
ret
```

By the end of code generation, the register allocator would coalesce the registers and delete the resultant identity moves producing the following code:

```
;; X is in EAX, Y is in ECX
mov %EAX, %EDX
sar %EDX, 31
idiv %ECX
ret
```

This approach is extremely general (if it can handle the X86 architecture, it can handle anything!) and allows all of the target specific knowledge about the instruction stream to be isolated in the instruction selector. Note that physical registers should have a short lifetime for good code generation, and all physical registers are assumed dead on entry to and exit from basic blocks (before register allocation). Thus, if you need a value to be live across basic block boundaries, it *must* live in a virtual register.

Call-clobbered registers

Some machine instructions, like calls, clobber a large number of physical registers. Rather than adding <def,dead> operands for all of them, it is possible to use an MO_RegisterMask operand instead. The register mask operand holds a bit mask of preserved registers, and everything else is considered to be clobbered by the instruction.

Machine code in SSA form

MachineInstr's are initially selected in SSA-form, and are maintained in SSA-form until register allocation happens. For the most part, this is trivially simple since LLVM is already in SSA form; LLVM PHI nodes become machine code PHI nodes, and virtual registers are only allowed to have a single definition.

After register allocation, machine code is no longer in SSA-form because there are no virtual registers left in the code.

The `MachineBasicBlock` class

The `MachineBasicBlock` class contains a list of machine instructions (MachineInstr instances). It roughly corresponds to the LLVM code input to the instruction selector, but there can be a one-to-many mapping (i.e. one LLVM basic block can map to multiple machine basic blocks). The `MachineBasicBlock` class has a "`getBasicBlock`" method, which returns the LLVM basic block that it comes from.

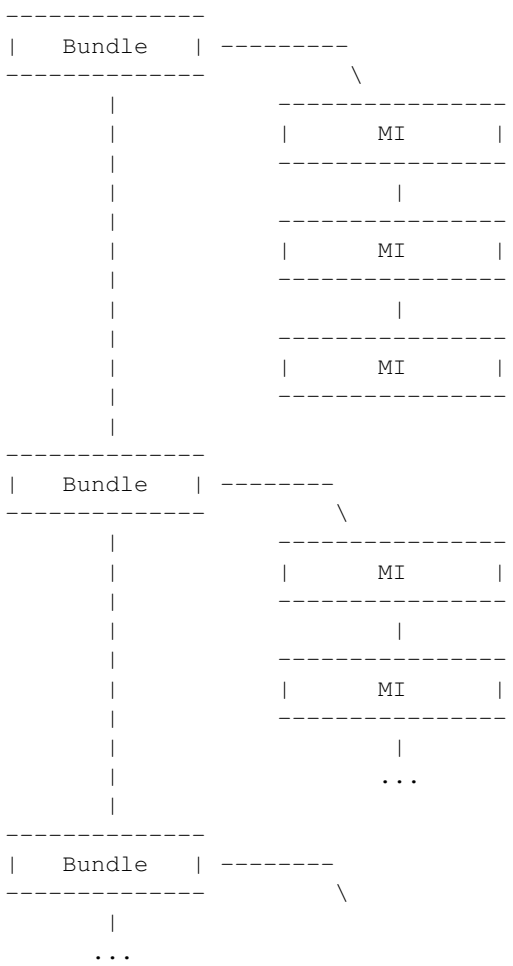
The `MachineFunction` class

The `MachineFunction` class contains a list of machine basic blocks (MachineBasicBlock instances). It corresponds one-to-one with the LLVM function input to the instruction selector. In addition to a list of basic blocks, the `MachineFunction` contains a a `MachineConstantPool`, a `MachineFrameInfo`, a `MachineFunctionInfo`, and a `MachineRegisterInfo`. See `include/llvm/CodeGen/MachineFunction.h` for more information.

`MachineInstr Bundles`

LLVM code generator can model sequences of instructions as MachineInstr bundles. A MI bundle can model a VLIW group / pack which contains an arbitrary number of parallel instructions. It can also be used to model a sequential list of instructions (potentially with data dependencies) that cannot be legally separated (e.g. ARM Thumb2 IT blocks).

Conceptually a MI bundle is a MI with a number of other MIs nested within:



MI bundle support does not change the physical representations of MachineBasicBlock and MachineInstr. All the MIs (including top level and nested ones) are stored as sequential list of MIs. The "bundled" MIs are marked with the 'InsideBundle' flag. A top level MI with the special BUNDLE opcode is used to represent the start of a bundle. It's legal to mix BUNDLE MIs with indiviual MIs that are not inside bundles nor represent bundles.

MachineInstr passes should operate on a MI bundle as a single unit. Member methods have been taught to correctly handle bundles and MIs inside bundles. The MachineBasicBlock iterator has been modified to skip over bundled MIs to enforce the bundle-as-a-single-unit concept. An alternative iterator instr_iterator has been added to MachineBasicBlock to allow passes to iterate over all of the MIs in a MachineBasicBlock, including those which are nested inside bundles. The top level BUNDLE instruction must have the correct set of register MachineOperand's that represent the cumulative inputs and outputs of the bundled MIs.

Packing / bundling of MachineInstr's should be done as part of the register allocation super-pass. More specifically, the pass which determines what MIs should be bundled together must be done after code generator exits SSA form (i.e. after two-address pass, PHI elimination, and copy coalescing). Bundles should only be finalized (i.e. adding BUNDLE MIs and input and output register MachineOperands) after virtual registers have been rewritten into physical registers. This requirement eliminates the need to add virtual register operands to BUNDLE instructions which would effectively double the virtual register def and use lists.

4.6.4 The "MC" Layer

The MC Layer is used to represent and process code at the raw machine code level, devoid of "high level" information like "constant pools", "jump tables", "global variables" or anything like that. At this level, LLVM handles things like label names, machine instructions, and sections in the object file. The code in this layer is used for a number of important purposes: the tail end of the code generator uses it to write a .s or .o file, and it is also used by the llvm-mc tool to implement standalone machine code assemblers and disassemblers.

This section describes some of the important classes. There are also a number of important subsystems that interact at this layer, they are described later in this manual.

The `MCStreamer` API

MCStreamer is best thought of as an assembler API. It is an abstract API which is *implemented* in different ways (e.g. to output a .s file, output an ELF .o file, etc) but whose API correspond directly to what you see in a .s file. MCStreamer has one method per directive, such as EmitLabel, EmitSymbolAttribute, SwitchSection, EmitValue (for .byte, .word), etc, which directly correspond to assembly level directives. It also has an EmitInstruction method, which is used to output an MCInst to the streamer.

This API is most important for two clients: the llvm-mc stand-alone assembler is effectively a parser that parses a line, then invokes a method on MCStreamer. In the code generator, the Code Emission phase of the code generator lowers higher level LLVM IR and Machine* constructs down to the MC layer, emitting directives through MCStreamer.

On the implementation side of MCStreamer, there are two major implementations: one for writing out a .s file (MCAsmStreamer), and one for writing out a .o file (MCObjectStreamer). MCAsmStreamer is a straight-forward implementation that prints out a directive for each method (e.g. `EmitValue -> .byte`), but MCObjectStreamer implements a full assembler.

For target specific directives, the MCStreamer has a MCTargetStreamer instance. Each target that needs it defines a class that inherits from it and is a lot like MCStreamer itself: It has one method per directive and two classes that inherit from it, a target object streamer and a target asm streamer. The target asm streamer just prints it (`emitFnStart -> .fnstrart`), and the object streamer implement the assembler logic for it.

To make llvm use these classes, the target initialization must call TargetRegistry::RegisterAsmStreamer and TargetRegistry::RegisterMCObjectStreamer passing callbacks that allocate the corresponding target streamer and pass it to createAsmStreamer or to the appropriate object streamer constructor.

The `MCContext` class

The MCContext class is the owner of a variety of uniqued data structures at the MC layer, including symbols, sections, etc. As such, this is the class that you interact with to create symbols and sections. This class can not be subclassed.

The `MCSymbol` class

The MCSymbol class represents a symbol (aka label) in the assembly file. There are two interesting kinds of symbols: assembler temporary symbols, and normal symbols. Assembler temporary symbols are used and processed by the assembler but are discarded when the object file is produced. The distinction is usually represented by adding a prefix to the label, for example "L" labels are assembler temporary labels in MachO.

MCSymbols are created by MCContext and uniqued there. This means that MCSymbols can be compared for pointer equivalence to find out if they are the same symbol. Note that pointer inequality does not guarantee the labels will end up at different addresses though. It's perfectly legal to output something like this to the .s file:

```
foo:
bar:
  .byte 4
```

In this case, both the foo and bar symbols will have the same address.

The `MCSection` class

The `MCSection` class represents an object-file specific section. It is subclassed by object file specific implementations (e.g. `MCSectionMachO`, `MCSectionCOFF`, `MCSectionELF`) and these are created and uniqued by MCContext. The MCStreamer has a notion of the current section, which can be changed with the SwitchToSection method (which corresponds to a ".section" directive in a .s file).

The `MCInst` class

The `MCInst` class is a target-independent representation of an instruction. It is a simple class (much more so than MachineInstr) that holds a target-specific opcode and a vector of MCOperands. MCOperand, in turn, is a simple discriminated union of three cases: 1) a simple immediate, 2) a target register ID, 3) a symbolic expression (e.g. "`Lfoo-Lbar+42`") as an MCExpr.

MCInst is the common currency used to represent machine instructions at the MC layer. It is the type used by the instruction encoder, the instruction printer, and the type generated by the assembly parser and disassembler.

4.6.5 Target-independent code generation algorithms

This section documents the phases described in the high-level design of the code generator. It explains how they work and some of the rationale behind their design.

Instruction Selection

Instruction Selection is the process of translating LLVM code presented to the code generator into target-specific machine instructions. There are several well-known ways to do this in the literature. LLVM uses a SelectionDAG based instruction selector.

Portions of the DAG instruction selector are generated from the target description (`*.td`) files. Our goal is for the entire instruction selector to be generated from these `.td` files, though currently there are still things that require custom C++ code.

Introduction to SelectionDAGs

The SelectionDAG provides an abstraction for code representation in a way that is amenable to instruction selection using automatic techniques (e.g. dynamic-programming based optimal pattern matching selectors). It is also well-suited to other phases of code generation; in particular, instruction scheduling (SelectionDAG's are very close to scheduling DAGs post-selection). Additionally, the SelectionDAG provides a host representation where a large variety of very-low-level (but target-independent) optimizations may be performed; ones which require extensive information about the instructions efficiently supported by the target.

The SelectionDAG is a Directed-Acyclic-Graph whose nodes are instances of the `SDNode` class. The primary payload of the `SDNode` is its operation code (Opcode) that indicates what operation the node performs and the operands to the operation. The various operation node types are described at the top of the `include/llvm/CodeGen/SelectionDAGNodes.h` file.

Although most operations define a single value, each node in the graph may define multiple values. For example, a combined div/rem operation will define both the dividend and the remainder. Many other situations require multiple values as well. Each node also has some number of operands, which are edges to the node defining the used value. Because nodes may define multiple values, edges are represented by instances of the `SDValue` class, which is a `<SDNode, unsigned>` pair, indicating the node and result value being used, respectively. Each value produced by an `SDNode` has an associated `MVT` (Machine Value Type) indicating what the type of the value is.

SelectionDAGs contain two different kinds of values: those that represent data flow and those that represent control flow dependencies. Data values are simple edges with an integer or floating point value type. Control edges are represented as "chain" edges which are of type `MVT::Other`. These edges provide an ordering between nodes that have side effects (such as loads, stores, calls, returns, etc). All nodes that have side effects should take a token chain as input and produce a new one as output. By convention, token chain inputs are always operand #0, and chain results are always the last value produced by an operation. However, after instruction selection, the machine nodes have their chain after the instruction's operands, and may be followed by glue nodes.

A SelectionDAG has designated "Entry" and "Root" nodes. The Entry node is always a marker node with an Opcode of `ISD::EntryToken`. The Root node is the final side-effecting node in the token chain. For example, in a single basic block function it would be the return node.

One important concept for SelectionDAGs is the notion of a "legal" vs. "illegal" DAG. A legal DAG for a target is one that only uses supported operations and supported types. On a 32-bit PowerPC, for example, a DAG with a value of type i1, i8, i16, or i64 would be illegal, as would a DAG that uses a SREM or UREM operation. The legalize types and legalize operations phases are responsible for turning an illegal DAG into a legal DAG.

SelectionDAG Instruction Selection Process

SelectionDAG-based instruction selection consists of the following steps:

1. Build initial DAG — This stage performs a simple translation from the input LLVM code to an illegal SelectionDAG.
2. Optimize SelectionDAG — This stage performs simple optimizations on the SelectionDAG to simplify it, and recognize meta instructions (like rotates and `div`/`rem` pairs) for targets that support these meta operations. This makes the resultant code more efficient and the select instructions from DAG phase (below) simpler.
3. Legalize SelectionDAG Types — This stage transforms SelectionDAG nodes to eliminate any types that are unsupported on the target.
4. Optimize SelectionDAG — The SelectionDAG optimizer is run to clean up redundancies exposed by type legalization.
5. Legalize SelectionDAG Ops — This stage transforms SelectionDAG nodes to eliminate any operations that are unsupported on the target.

6. Optimize SelectionDAG — The SelectionDAG optimizer is run to eliminate inefficiencies introduced by operation legalization.
7. Select instructions from DAG — Finally, the target instruction selector matches the DAG operations to target instructions. This process translates the target-independent input DAG into another DAG of target instructions.
8. SelectionDAG Scheduling and Formation — The last phase assigns a linear order to the instructions in the target-instruction DAG and emits them into the MachineFunction being compiled. This step uses traditional prepass scheduling techniques.

After all of these steps are complete, the SelectionDAG is destroyed and the rest of the code generation passes are run.

One great way to visualize what is going on here is to take advantage of a few LLC command line options. The following options pop up a window displaying the SelectionDAG at specific times (if you only get errors printed to the console while using this, you probably need to configure your system to add support for it).

- `-view-dag-combine1-dags` displays the DAG after being built, before the first optimization pass.
- `-view-legalize-dags` displays the DAG before Legalization.
- `-view-dag-combine2-dags` displays the DAG before the second optimization pass.
- `-view-isel-dags` displays the DAG before the Select phase.
- `-view-sched-dags` displays the DAG before Scheduling.

The `-view-sunit-dags` displays the Scheduler's dependency graph. This graph is based on the final SelectionDAG, with nodes that must be scheduled together bundled into a single scheduling-unit node, and with immediate operands and other nodes that aren't relevant for scheduling omitted.

Initial SelectionDAG Construction

The initial SelectionDAG is navely peephole expanded from the LLVM input by the `SelectionDAGBuilder` class. The intent of this pass is to expose as much low-level, target-specific details to the SelectionDAG as possible. This pass is mostly hard-coded (e.g. an LLVM `add` turns into an `SDNode add` while a `getelementptr` is expanded into the obvious arithmetic). This pass requires target-specific hooks to lower calls, returns, varargs, etc. For these features, the TargetLowering interface is used.

SelectionDAG LegalizeTypes Phase

The Legalize phase is in charge of converting a DAG to only use the types that are natively supported by the target.

There are two main ways of converting values of unsupported scalar types to values of supported types: converting small types to larger types ("promoting"), and breaking up large integer types into smaller ones ("expanding"). For example, a target might require that all f32 values are promoted to f64 and that all i1/i8/i16 values are promoted to i32. The same target might require that all i64 values be expanded into pairs of i32 values. These changes can insert sign and zero extensions as needed to make sure that the final code has the same behavior as the input.

There are two main ways of converting values of unsupported vector types to value of supported types: splitting vector types, multiple times if necessary, until a legal type is found, and extending vector types by adding elements to the end to round them out to legal types ("widening"). If a vector gets split all the way down to single-element parts with no supported vector type being found, the elements are converted to scalars ("scalarizing").

A target implementation tells the legalizer which types are supported (and which register class to use for them) by calling the `addRegisterClass` method in its `TargetLowering` constructor.

SelectionDAG Legalize Phase

The Legalize phase is in charge of converting a DAG to only use the operations that are natively supported by the target.

Targets often have weird constraints, such as not supporting every operation on every supported datatype (e.g. X86 does not support byte conditional moves and PowerPC does not support sign-extending loads from a 16-bit memory location). Legalize takes care of this by open-coding another sequence of operations to emulate the operation ("expansion"), by promoting one type to a larger type that supports the operation ("promotion"), or by using a target-specific hook to implement the legalization ("custom").

A target implementation tells the legalizer which operations are not supported (and which of the above three actions to take) by calling the `setOperationAction` method in its `TargetLowering` constructor.

Prior to the existence of the Legalize passes, we required that every target selector supported and handled every operator and type even if they are not natively supported. The introduction of the Legalize phases allows all of the canonicalization patterns to be shared across targets, and makes it very easy to optimize the canonicalized code because it is still in the form of a DAG.

SelectionDAG Optimization Phase: the DAG Combiner

The SelectionDAG optimization phase is run multiple times for code generation, immediately after the DAG is built and once after each legalization. The first run of the pass allows the initial code to be cleaned up (e.g. performing optimizations that depend on knowing that the operators have restricted type inputs). Subsequent runs of the pass clean up the messy code generated by the Legalize passes, which allows Legalize to be very simple (it can focus on making code legal instead of focusing on generating *good* and legal code).

One important class of optimizations performed is optimizing inserted sign and zero extension instructions. We currently use ad-hoc techniques, but could move to more rigorous techniques in the future. Here are some good papers on the subject:

"Widening integer arithmetic" Kevin Redwine and Norman Ramsey International Conference on Compiler Construction (CC) 2004

"Effective sign extension elimination" Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation.

SelectionDAG Select Phase

The Select phase is the bulk of the target-specific code for instruction selection. This phase takes a legal SelectionDAG as input, pattern matches the instructions supported by the target to this DAG, and produces a new DAG of target code. For example, consider the following LLVM fragment:

```
%t1 = fadd float %W, %X
%t2 = fmul float %t1, %Y
%t3 = fadd float %t2, %Z
```

This LLVM code corresponds to a SelectionDAG that looks basically like this:

```
(fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z)
```

If a target supports floating point multiply-and-add (FMA) operations, one of the adds can be merged with the multiply. On the PowerPC, for example, the output of the instruction selector might look like this DAG:

```
(FMADDS (FADDS W, X), Y, Z)
```

The FMADDS instruction is a ternary instruction that multiplies its first two operands and adds the third (as single-precision floating-point numbers). The FADDS instruction is a simple binary single-precision add instruction. To perform this pattern match, the PowerPC backend includes the following instruction definitions:

```
def FMADDS : AForm_1<59, 29,
                     (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRC, F4RC:$FRB),
                     "fmadds $FRT, $FRA, $FRC, $FRB",
                     [(set F4RC:$FRT, (fadd (fmul F4RC:$FRA, F4RC:$FRC),
                                            F4RC:$FRB))]>;
def FADDS : AForm_2<59, 21,
                     (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRB),
                     "fadds $FRT, $FRA, $FRB",
                     [(set F4RC:$FRT, (fadd F4RC:$FRA, F4RC:$FRB))]>;
```

The highlighted portion of the instruction definitions indicates the pattern used to match the instructions. The DAG operators (like fmul/fadd) are defined in the include/llvm/Target/TargetSelectionDAG.td file. "F4RC" is the register class of the input and result values.

The TableGen DAG instruction selector generator reads the instruction patterns in the .td file and automatically builds parts of the pattern matching code for your target. It has the following strengths:

- At compiler-compiler time, it analyzes your instruction patterns and tells you if your patterns make sense or not.
- It can handle arbitrary constraints on operands for the pattern match. In particular, it is straight-forward to say things like "match any immediate that is a 13-bit sign-extended value". For examples, see the immSExt16 and related tblgen classes in the PowerPC backend.
- It knows several important identities for the patterns defined. For example, it knows that addition is commutative, so it allows the FMADDS pattern above to match "(fadd X, (fmul Y, Z))" as well as "(fadd (fmul X, Y), Z)", without the target author having to specially handle this case.
- It has a full-featured type-inferencing system. In particular, you should rarely have to explicitly tell the system what type parts of your patterns are. In the FMADDS case above, we didn't have to tell tblgen that all of the nodes in the pattern are of type 'f32'. It was able to infer and propagate this knowledge from the fact that F4RC has type 'f32'.
- Targets can define their own (and rely on built-in) "pattern fragments". Pattern fragments are chunks of reusable patterns that get inlined into your patterns during compiler-compiler time. For example, the integer "(not x)" operation is actually defined as a pattern fragment that expands as "(xor x, -1)", since the SelectionDAG does not have a native 'not' operation. Targets can define their own short-hand fragments as they see fit. See the definition of 'not' and 'ineg' for examples.
- In addition to instructions, targets can specify arbitrary patterns that map to one or more instructions using the 'Pat' class. For example, the PowerPC has no way to load an arbitrary integer immediate into a register in one instruction. To tell tblgen how to do this, it defines:

  ```
  // Arbitrary immediate support.  Implement in terms of LIS/ORI.
  def : Pat<(i32 imm:$imm),
            (ORI (LIS (HI16 imm:$imm)), (LO16 imm:$imm))>;
  ```

  If none of the single-instruction patterns for loading an immediate into a register match, this will be used. This rule says "match an arbitrary i32 immediate, turning it into an ORI ('or a 16-bit immediate') and an LIS ('load 16-bit immediate, where the immediate is shifted to the left 16 bits') instruction". To make this work, the LO16/HI16 node transformations are used to manipulate the input immediate (in this case, take the high or low 16-bits of the immediate).
- When using the 'Pat' class to map a pattern to an instruction that has one or more complex operands (like e.g. X86 addressing mode), the pattern may either specify the operand as a whole using a ComplexPattern, or else it may specify the components of the complex operand separately. The latter is done e.g. for pre-increment instructions by the PowerPC back end:

```
def STWU  : DForm_1<37, (outs ptr_rc:$ea_res), (ins GPRC:$rS, memri:$dst),
                "stwu $rS, $dst", LdStStoreUpd, []>,
                RegConstraint<"$dst.reg = $ea_res">, NoEncode<"$ea_res">;

def : Pat<(pre_store GPRC:$rS, ptr_rc:$ptrreg, iaddroff:$ptroff),
          (STWU GPRC:$rS, iaddroff:$ptroff, ptr_rc:$ptrreg)>;
```

Here, the pair of `ptroff` and `ptrreg` operands is matched onto the complex operand `dst` of class `memri` in the `STWU` instruction.

- While the system does automate a lot, it still allows you to write custom C++ code to match special cases if there is something that is hard to express.

While it has many strengths, the system currently has some limitations, primarily because it is a work in progress and is not yet finished:

- Overall, there is no way to define or match SelectionDAG nodes that define multiple values (e.g. `SMUL_LOHI`, `LOAD`, `CALL`, etc). This is the biggest reason that you currently still *have to* write custom C++ code for your instruction selector.
- There is no great way to support matching complex addressing modes yet. In the future, we will extend pattern fragments to allow them to define multiple values (e.g. the four operands of the X86 addressing mode, which are currently matched with custom C++ code). In addition, we'll extend fragments so that a fragment can match multiple different patterns.
- We don't automatically infer flags like `isStore`/`isLoad` yet.
- We don't automatically generate the set of supported registers and operations for the Legalizer yet.
- We don't have a way of tying in custom legalized nodes yet.

Despite these limitations, the instruction selector generator is still quite useful for most of the binary and logical operations in typical instruction sets. If you run into any problems or can't figure out how to do something, please let Chris know!

SelectionDAG Scheduling and Formation Phase

The scheduling phase takes the DAG of target instructions from the selection phase and assigns an order. The scheduler can pick an order depending on various constraints of the machines (i.e. order for minimal register pressure or try to cover instruction latencies). Once an order is established, the DAG is converted to a list of MachineInstrs and the SelectionDAG is destroyed.

Note that this phase is logically separate from the instruction selection phase, but is tied to it closely in the code because it operates on SelectionDAGs.

Future directions for the SelectionDAG

1. Optional function-at-a-time selection.
2. Auto-generate entire selector from `.td` file.

SSA-based Machine Code Optimizations

To Be Written

Live Intervals

Live Intervals are the ranges (intervals) where a variable is *live*. They are used by some register allocator passes to determine if two or more virtual registers which require the same physical register are live at the same point in the program (i.e., they conflict). When this situation occurs, one virtual register must be *spilled*.

Live Variable Analysis

The first step in determining the live intervals of variables is to calculate the set of registers that are immediately dead after the instruction (i.e., the instruction calculates the value, but it is never used) and the set of registers that are used by the instruction, but are never used after the instruction (i.e., they are killed). Live variable information is computed for each *virtual* register and *register allocatable* physical register in the function. This is done in a very efficient manner because it uses SSA to sparsely compute lifetime information for virtual registers (which are in SSA form) and only has to track physical registers within a block. Before register allocation, LLVM can assume that physical registers are only live within a single basic block. This allows it to do a single, local analysis to resolve physical register lifetimes within each basic block. If a physical register is not register allocatable (e.g., a stack pointer or condition codes), it is not tracked.

Physical registers may be live in to or out of a function. Live in values are typically arguments in registers. Live out values are typically return values in registers. Live in values are marked as such, and are given a dummy "defining" instruction during live intervals analysis. If the last basic block of a function is a `return`, then it's marked as using all live out values in the function.

`PHI` nodes need to be handled specially, because the calculation of the live variable information from a depth first traversal of the CFG of the function won't guarantee that a virtual register used by the `PHI` node is defined before it's used. When a `PHI` node is encountered, only the definition is handled, because the uses will be handled in other basic blocks.

For each `PHI` node of the current basic block, we simulate an assignment at the end of the current basic block and traverse the successor basic blocks. If a successor basic block has a `PHI` node and one of the `PHI` node's operands is coming from the current basic block, then the variable is marked as *alive* within the current basic block and all of its predecessor basic blocks, until the basic block with the defining instruction is encountered.

Live Intervals Analysis

We now have the information available to perform the live intervals analysis and build the live intervals themselves. We start off by numbering the basic blocks and machine instructions. We then handle the "live-in" values. These are in physical registers, so the physical register is assumed to be killed by the end of the basic block. Live intervals for virtual registers are computed for some ordering of the machine instructions `[1, N]`. A live interval is an interval `[i, j)`, where `1 >= i >= j > N`, for which a variable is live.

Note: More to come...

Register Allocation

The *Register Allocation problem* consists in mapping a program P_v, that can use an unbounded number of virtual registers, to a program P_p that contains a finite (possibly small) number of physical registers. Each target architecture has a different number of physical registers. If the number of physical registers is not enough to accommodate all the virtual registers, some of them will have to be mapped into memory. These virtuals are called *spilled virtuals*.

How registers are represented in LLVM

In LLVM, physical registers are denoted by integer numbers that normally range from 1 to 1023. To see how this numbering is defined for a particular architecture, you can read the `GenRegisterNames.inc` file for that architecture. For instance, by inspecting `lib/Target/X86/X86GenRegisterInfo.inc` we see that the 32-bit register `EAX` is denoted by 43, and the MMX register `MM0` is mapped to 65.

Some architectures contain registers that share the same physical location. A notable example is the X86 platform. For instance, in the X86 architecture, the registers `EAX`, `AX` and `AL` share the first eight bits. These physical registers are marked as *aliased* in LLVM. Given a particular architecture, you can check which registers are aliased by inspecting its `RegisterInfo.td` file. Moreover, the class `MCRegAliasIterator` enumerates all the physical registers aliased to a register.

Physical registers, in LLVM, are grouped in *Register Classes*. Elements in the same register class are functionally equivalent, and can be interchangeably used. Each virtual register can only be mapped to physical registers of a particular class. For instance, in the X86 architecture, some virtuals can only be allocated to 8 bit registers. A register class is described by `TargetRegisterClass` objects. To discover if a virtual register is compatible with a given physical, this code can be used:

```
bool RegMapping_Fer::compatible_class(MachineFunction &mf,
                                      unsigned v_reg,
                                      unsigned p_reg) {
  assert(TargetRegisterInfo::isPhysicalRegister(p_reg) &&
         "Target register must be physical");
  const TargetRegisterClass *trc = mf.getRegInfo().getRegClass(v_reg);
  return trc->contains(p_reg);
}
```

Sometimes, mostly for debugging purposes, it is useful to change the number of physical registers available in the target architecture. This must be done statically, inside the `TargetRegsterInfo.td` file. Just `grep` for `RegisterClass`, the last parameter of which is a list of registers. Just commenting some out is one simple way to avoid them being used. A more polite way is to explicitly exclude some registers from the *allocation order*. See the definition of the `GR8` register class in `lib/Target/X86/X86RegisterInfo.td` for an example of this.

Virtual registers are also denoted by integer numbers. Contrary to physical registers, different virtual registers never share the same number. Whereas physical registers are statically defined in a `TargetRegisterInfo.td` file and cannot be created by the application developer, that is not the case with virtual registers. In order to create new virtual registers, use the method `MachineRegisterInfo::createVirtualRegister()`. This method will return a new virtual register. Use an `IndexedMap<Foo, VirtReg2IndexFunctor>` to hold information per virtual register. If you need to enumerate all virtual registers, use the function `TargetRegisterInfo::index2VirtReg()` to find the virtual register numbers:

```
for (unsigned i = 0, e = MRI->getNumVirtRegs(); i != e; ++i) {
  unsigned VirtReg = TargetRegisterInfo::index2VirtReg(i);
  stuff(VirtReg);
}
```

Before register allocation, the operands of an instruction are mostly virtual registers, although physical registers may also be used. In order to check if a given machine operand is a register, use the boolean function `MachineOperand::isRegister()`. To obtain the integer code of a register, use `MachineOperand::getReg()`. An instruction may define or use a register. For instance, `ADD reg:1026 := reg:1025 reg:1024` defines the registers 1024, and uses registers 1025 and 1026. Given a register operand, the method `MachineOperand::isUse()` informs if that register is being used by the instruction. The method `MachineOperand::isDef()` informs if that registers is being defined.

We will call physical registers present in the LLVM bitcode before register allocation *pre-colored registers*. Pre-colored registers are used in many different situations, for instance, to pass parameters of functions calls, and

to store results of particular instructions. There are two types of pre-colored registers: the ones *implicitly* defined, and those *explicitly* defined. Explicitly defined registers are normal operands, and can be accessed with `MachineInstr::getOperand(int)::getReg()`. In order to check which registers are implicitly defined by an instruction, use the `TargetInstrInfo::get(opcode)::ImplicitDefs`, where `opcode` is the opcode of the target instruction. One important difference between explicit and implicit physical registers is that the latter are defined statically for each instruction, whereas the former may vary depending on the program being compiled. For example, an instruction that represents a function call will always implicitly define or use the same set of physical registers. To read the registers implicitly used by an instruction, use `TargetInstrInfo::get(opcode)::ImplicitUses`. Pre-colored registers impose constraints on any register allocation algorithm. The register allocator must make sure that none of them are overwritten by the values of virtual registers while still alive.

Mapping virtual registers to physical registers

There are two ways to map virtual registers to physical registers (or to memory slots). The first way, that we will call *direct mapping*, is based on the use of methods of the classes `TargetRegisterInfo`, and `MachineOperand`. The second way, that we will call *indirect mapping*, relies on the `VirtRegMap` class in order to insert loads and stores sending and getting values to and from memory.

The direct mapping provides more flexibility to the developer of the register allocator; however, it is more error prone, and demands more implementation work. Basically, the programmer will have to specify where load and store instructions should be inserted in the target function being compiled in order to get and store values in memory. To assign a physical register to a virtual register present in a given operand, use `MachineOperand::setReg(p_reg)`. To insert a store instruction, use `TargetInstrInfo::storeRegToStackSlot(...)`, and to insert a load instruction, use `TargetInstrInfo::loadRegFromStackSlot`.

The indirect mapping shields the application developer from the complexities of inserting load and store instructions. In order to map a virtual register to a physical one, use `VirtRegMap::assignVirt2Phys(vreg, preg)`. In order to map a certain virtual register to memory, use `VirtRegMap::assignVirt2StackSlot(vreg)`. This method will return the stack slot where `vreg`'s value will be located. If it is necessary to map another virtual register to the same stack slot, use `VirtRegMap::assignVirt2StackSlot(vreg, stack_location)`. One important point to consider when using the indirect mapping, is that even if a virtual register is mapped to memory, it still needs to be mapped to a physical register. This physical register is the location where the virtual register is supposed to be found before being stored or after being reloaded.

If the indirect strategy is used, after all the virtual registers have been mapped to physical registers or stack slots, it is necessary to use a spiller object to place load and store instructions in the code. Every virtual that has been mapped to a stack slot will be stored to memory after been defined and will be loaded before being used. The implementation of the spiller tries to recycle load/store instructions, avoiding unnecessary instructions. For an example of how to invoke the spiller, see `RegAllocLinearScan::runOnMachineFunction` in `lib/CodeGen/RegAllocLinearScan.cpp`.

Handling two address instructions

With very rare exceptions (e.g., function calls), the LLVM machine code instructions are three address instructions. That is, each instruction is expected to define at most one register, and to use at most two registers. However, some architectures use two address instructions. In this case, the defined register is also one of the used register. For instance, an instruction such as `ADD %EAX, %EBX`, in X86 is actually equivalent to `%EAX = %EAX + %EBX`.

In order to produce correct code, LLVM must convert three address instructions that represent two address instructions into true two address instructions. LLVM provides the pass `TwoAddressInstructionPass` for this specific purpose. It must be run before register allocation takes place. After its execution, the resulting code may no longer be in SSA form. This happens, for instance, in situations where an instruction such as `%a = ADD %b %c` is converted to two instructions such as:

```
%a = MOVE %b
%a = ADD %a %c
```

Notice that, internally, the second instruction is represented as `ADD %a[def/use] %c`. I.e., the register operand `%a` is both used and defined by the instruction.

The SSA deconstruction phase

An important transformation that happens during register allocation is called the *SSA Deconstruction Phase*. The SSA form simplifies many analyses that are performed on the control flow graph of programs. However, traditional instruction sets do not implement PHI instructions. Thus, in order to generate executable code, compilers must replace PHI instructions with other instructions that preserve their semantics.

There are many ways in which PHI instructions can safely be removed from the target code. The most traditional PHI deconstruction algorithm replaces PHI instructions with copy instructions. That is the strategy adopted by LLVM. The SSA deconstruction algorithm is implemented in `lib/CodeGen/PHIElimination.cpp`. In order to invoke this pass, the identifier `PHIEliminationID` must be marked as required in the code of the register allocator.

Instruction folding

Instruction folding is an optimization performed during register allocation that removes unnecessary copy instructions. For instance, a sequence of instructions such as:

```
%EBX = LOAD %mem_address
%EAX = COPY %EBX
```

can be safely substituted by the single instruction:

```
%EAX = LOAD %mem_address
```

Instructions can be folded with the `TargetRegisterInfo::foldMemoryOperand(...)` method. Care must be taken when folding instructions; a folded instruction can be quite different from the original instruction. See `LiveIntervals::addIntervalsForSpills` in `lib/CodeGen/LiveIntervalAnalysis.cpp` for an example of its use.

Built in register allocators

The LLVM infrastructure provides the application developer with three different register allocators:

- *Fast* — This register allocator is the default for debug builds. It allocates registers on a basic block level, attempting to keep values in registers and reusing registers as appropriate.
- *Basic* — This is an incremental approach to register allocation. Live ranges are assigned to registers one at a time in an order that is driven by heuristics. Since code can be rewritten on-the-fly during allocation, this framework allows interesting allocators to be developed as extensions. It is not itself a production register allocator but is a potentially useful stand-alone mode for triaging bugs and as a performance baseline.
- *Greedy — The default allocator*. This is a highly tuned implementation of the *Basic* allocator that incorporates global live range splitting. This allocator works hard to minimize the cost of spill code.
- *PBQP* — A Partitioned Boolean Quadratic Programming (PBQP) based register allocator. This allocator works by constructing a PBQP problem representing the register allocation problem under consideration, solving this using a PBQP solver, and mapping the solution back to a register assignment.

The type of register allocator used in `llc` can be chosen with the command line option `-regalloc=...`:

```
$ llc -regalloc=linearscan file.bc -o ln.s
$ llc -regalloc=fast file.bc -o fa.s
$ llc -regalloc=pbqp file.bc -o pbqp.s
```

Prolog/Epilog Code Insertion

Compact Unwind

Throwing an exception requires *unwinding* out of a function. The information on how to unwind a given function is traditionally expressed in DWARF unwind (a.k.a. frame) info. But that format was originally developed for debuggers to backtrace, and each Frame Description Entry (FDE) requires ~20-30 bytes per function. There is also the cost of mapping from an address in a function to the corresponding FDE at runtime. An alternative unwind encoding is called *compact unwind* and requires just 4-bytes per function.

The compact unwind encoding is a 32-bit value, which is encoded in an architecture-specific way. It specifies which registers to restore and from where, and how to unwind out of the function. When the linker creates a final linked image, it will create a `__TEXT,__unwind_info` section. This section is a small and fast way for the runtime to access unwind info for any given function. If we emit compact unwind info for the function, that compact unwind info will be encoded in the `__TEXT,__unwind_info` section. If we emit DWARF unwind info, the `__TEXT,__unwind_info` section will contain the offset of the FDE in the `__TEXT,__eh_frame` section in the final linked image.

For X86, there are three modes for the compact unwind encoding:

Function with a Frame Pointer (‘‘EBP‘‘ or ‘‘RBP‘‘) `EBP/RBP`-based frame, where `EBP/RBP` is pushed onto the stack immediately after the return address, then `ESP/RSP` is moved to `EBP/RBP`. Thus to unwind, `ESP/RSP` is restored with the current `EBP/RBP` value, then `EBP/RBP` is restored by popping the stack, and the return is done by popping the stack once more into the PC. All non-volatile registers that need to be restored must have been saved in a small range on the stack that starts `EBP-4` to `EBP-1020` (`RBP-8` to `RBP-1020`). The offset (divided by 4 in 32-bit mode and 8 in 64-bit mode) is encoded in bits 16-23 (mask: `0x00FF0000`). The registers saved are encoded in bits 0-14 (mask: `0x00007FFF`) as five 3-bit entries from the following table:

Compact Number	i386 Register	x86-64 Register
1	EBX	RBX
2	ECX	R12
3	EDX	R13
4	EDI	R14
5	ESI	R15
6	EBP	RBP

Frameless with a Small Constant Stack Size (‘‘EBP‘‘ or ‘‘RBP‘‘ is not used as a frame pointer) To return, a constant (encoded in the compact unwind encoding) is added to the `ESP/RSP`. Then the return is done by popping the stack into the PC. All non-volatile registers that need to be restored must have been saved on the stack immediately after the return address. The stack size (divided by 4 in 32-bit mode and 8 in 64-bit mode) is encoded in bits 16-23 (mask: `0x00FF0000`). There is a maximum stack size of 1024 bytes in 32-bit mode and 2048 in 64-bit mode. The number of registers saved is encoded in bits 9-12 (mask: `0x00001C00`). Bits 0-9 (mask: `0x000003FF`) contain which registers were saved and their order. (See the `encodeCompactUnwindRegistersWithoutFrame()` function in `lib/Target/X86FrameLowering.cpp` for the encoding algorithm.)

Frameless with a Large Constant Stack Size (‘‘EBP‘‘ or ‘‘RBP‘‘ is not used as a frame pointer) This case is like the “Frameless with a Small Constant Stack Size” case, but the stack size is too large to encode in the compact unwind encoding. Instead it requires that the function contains “`subl $nnnnnn, %esp`” in its prolog. The compact encoding contains the offset to the `$nnnnnn` value in the function in bits 9-12 (mask: `0x00001C00`).

Late Machine Code Optimizations

Note: To Be Written

Code Emission

The code emission step of code generation is responsible for lowering from the code generator abstractions (like MachineFunction, MachineInstr, etc) down to the abstractions used by the MC layer (MCInst, MCStreamer, etc). This is done with a combination of several different classes: the (misnamed) target-independent AsmPrinter class, target-specific subclasses of AsmPrinter (such as SparcAsmPrinter), and the TargetLoweringObjectFile class.

Since the MC layer works at the level of abstraction of object files, it doesn't have a notion of functions, global variables etc. Instead, it thinks about labels, directives, and instructions. A key class used at this time is the MCStreamer class. This is an abstract API that is implemented in different ways (e.g. to output a .s file, output an ELF .o file, etc) that is effectively an "assembler API". MCStreamer has one method per directive, such as EmitLabel, EmitSymbolAttribute, SwitchSection, etc, which directly correspond to assembly level directives.

If you are interested in implementing a code generator for a target, there are three important things that you have to implement for your target:

1. First, you need a subclass of AsmPrinter for your target. This class implements the general lowering process converting MachineFunction's into MC label constructs. The AsmPrinter base class provides a number of useful methods and routines, and also allows you to override the lowering process in some important ways. You should get much of the lowering for free if you are implementing an ELF, COFF, or MachO target, because the TargetLoweringObjectFile class implements much of the common logic.

2. Second, you need to implement an instruction printer for your target. The instruction printer takes an MCInst and renders it to a raw_ostream as text. Most of this is automatically generated from the .td file (when you specify something like "`add $dst, $src1, $src2`" in the instructions), but you need to implement routines to print operands.

3. Third, you need to implement code that lowers a MachineInstr to an MCInst, usually implemented in "<target>MCInstLower.cpp". This lowering process is often target specific, and is responsible for turning jump table entries, constant pool indices, global variable addresses, etc into MCLabels as appropriate. This translation layer is also responsible for expanding pseudo ops used by the code generator into the actual machine instructions they correspond to. The MCInsts that are generated by this are fed into the instruction printer or the encoder.

Finally, at your choosing, you can also implement an subclass of MCCodeEmitter which lowers MCInst's into machine code bytes and relocations. This is important if you want to support direct .o file emission, or would like to implement an assembler for your target.

VLIW Packetizer

In a Very Long Instruction Word (VLIW) architecture, the compiler is responsible for mapping instructions to functional-units available on the architecture. To that end, the compiler creates groups of instructions called *packets* or *bundles*. The VLIW packetizer in LLVM is a target-independent mechanism to enable the packetization of machine instructions.

Mapping from instructions to functional units

Instructions in a VLIW target can typically be mapped to multiple functional units. During the process of packetizing, the compiler must be able to reason about whether an instruction can be added to a packet. This decision can be complex since the compiler has to examine all possible mappings of instructions to functional units. Therefore to

alleviate compilation-time complexity, the VLIW packetizer parses the instruction classes of a target and generates tables at compiler build time. These tables can then be queried by the provided machine-independent API to determine if an instruction can be accommodated in a packet.

How the packetization tables are generated and used

The packetizer reads instruction classes from a target's itineraries and creates a deterministic finite automaton (DFA) to represent the state of a packet. A DFA consists of three major elements: inputs, states, and transitions. The set of inputs for the generated DFA represents the instruction being added to a packet. The states represent the possible consumption of functional units by instructions in a packet. In the DFA, transitions from one state to another occur on the addition of an instruction to an existing packet. If there is a legal mapping of functional units to instructions, then the DFA contains a corresponding transition. The absence of a transition indicates that a legal mapping does not exist and that the instruction cannot be added to the packet.

To generate tables for a VLIW target, add *Target*GenDFAPacketizer.inc as a target to the Makefile in the target directory. The exported API provides three functions: `DFAPacketizer::clearResources()`, `DFAPacketizer::reserveResources(MachineInstr *MI)`, and `DFAPacketizer::canReserveResources(MachineInstr *MI)`. These functions allow a target packetizer to add an instruction to an existing packet and to check whether an instruction can be added to a packet. See `llvm/CodeGen/DFAPacketizer.h` for more information.

4.6.6 Implementing a Native Assembler

Though you're probably reading this because you want to write or maintain a compiler backend, LLVM also fully supports building a native assembler. We've tried hard to automate the generation of the assembler from the .td files (in particular the instruction syntax and encodings), which means that a large part of the manual and repetitive data entry can be factored and shared with the compiler.

Instruction Parsing

Note: To Be Written

Instruction Alias Processing

Once the instruction is parsed, it enters the MatchInstructionImpl function. The MatchInstructionImpl function performs alias processing and then does actual matching.

Alias processing is the phase that canonicalizes different lexical forms of the same instructions down to one representation. There are several different kinds of alias that are possible to implement and they are listed below in the order that they are processed (which is in order from simplest/weakest to most complex/powerful). Generally you want to use the first alias mechanism that meets the needs of your instruction, because it will allow a more concise description.

Mnemonic Aliases

The first phase of alias processing is simple instruction mnemonic remapping for classes of instructions which are allowed with two different mnemonics. This phase is a simple and unconditionally remapping from one input mnemonic to one output mnemonic. It isn't possible for this form of alias to look at the operands at all, so the remapping must apply for all forms of a given mnemonic. Mnemonic aliases are defined simply, for example X86 has:

```
def : MnemonicAlias<"cbw",     "cbtw">;
def : MnemonicAlias<"smovq",   "movsq">;
def : MnemonicAlias<"fldcww",  "fldcw">;
def : MnemonicAlias<"fucompi", "fucomip">;
def : MnemonicAlias<"ud2a",    "ud2">;
```

... and many others. With a MnemonicAlias definition, the mnemonic is remapped simply and directly. Though MnemonicAlias's can't look at any aspect of the instruction (such as the operands) they can depend on global modes (the same ones supported by the matcher), through a Requires clause:

```
def : MnemonicAlias<"pushf", "pushfq">, Requires<[In64BitMode]>;
def : MnemonicAlias<"pushf", "pushfl">, Requires<[In32BitMode]>;
```

In this example, the mnemonic gets mapped into a different one depending on the current instruction set.

Instruction Aliases

The most general phase of alias processing occurs while matching is happening: it provides new forms for the matcher to match along with a specific instruction to generate. An instruction alias has two parts: the string to match and the instruction to generate. For example:

```
def : InstAlias<"movsx $src, $dst", (MOVSX16rr8W GR16:$dst, GR8  :$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX16rm8W GR16:$dst, i8mem:$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX32rr8  GR32:$dst, GR8  :$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX32rr16 GR32:$dst, GR16 :$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX64rr8  GR64:$dst, GR8  :$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX64rr16 GR64:$dst, GR16 :$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX64rr32 GR64:$dst, GR32 :$src)>;
```

This shows a powerful example of the instruction aliases, matching the same mnemonic in multiple different ways depending on what operands are present in the assembly. The result of instruction aliases can include operands in a different order than the destination instruction, and can use an input multiple times, for example:

```
def : InstAlias<"clrb $reg", (XOR8rr  GR8 :$reg, GR8 :$reg)>;
def : InstAlias<"clrw $reg", (XOR16rr GR16:$reg, GR16:$reg)>;
def : InstAlias<"clrl $reg", (XOR32rr GR32:$reg, GR32:$reg)>;
def : InstAlias<"clrq $reg", (XOR64rr GR64:$reg, GR64:$reg)>;
```

This example also shows that tied operands are only listed once. In the X86 backend, XOR8rr has two input GR8's and one output GR8 (where an input is tied to the output). InstAliases take a flattened operand list without duplicates for tied operands. The result of an instruction alias can also use immediates and fixed physical registers which are added as simple immediate operands in the result, for example:

```
// Fixed Immediate operand.
def : InstAlias<"aad", (AAD8i8 10)>;

// Fixed register operand.
def : InstAlias<"fcomi", (COM_FIr ST1)>;

// Simple alias.
def : InstAlias<"fcomi $reg", (COM_FIr RST:$reg)>;
```

Instruction aliases can also have a Requires clause to make them subtarget specific.

If the back-end supports it, the instruction printer can automatically emit the alias rather than what's being aliased. It typically leads to better, more readable code. If it's better to print out what's being aliased, then pass a '0' as the third parameter to the InstAlias definition.

Instruction Matching

Note: To Be Written

4.6.7 Target-specific Implementation Notes

This section of the document explains features or design decisions that are specific to the code generator for a particular target. First we start with a table that summarizes what features are supported by each target.

Target Feature Matrix

Note that this table does not include the C backend or Cpp backends, since they do not use the target independent code generator infrastructure. It also doesn't list features that are not supported fully by any target yet. It considers a feature to be supported if at least one subtarget supports it. A feature being supported means that it is useful and works for most cases, it does not indicate that there are zero known bugs in the implementation. Here is the key:

Here is the table:

Is Generally Reliable

This box indicates whether the target is considered to be production quality. This indicates that the target has been used as a static compiler to compile large amounts of code by a variety of different people and is in continuous use.

Assembly Parser

This box indicates whether the target supports parsing target specific .s files by implementing the MCAsmParser interface. This is required for llvm-mc to be able to act as a native assembler and is required for inline assembly support in the native .o file writer.

Disassembler

This box indicates whether the target supports the MCDisassembler API for disassembling machine opcode bytes into MCInst's.

Inline Asm

This box indicates whether the target supports most popular inline assembly constraints and modifiers.

JIT Support

This box indicates whether the target supports the JIT compiler through the ExecutionEngine interface. The ARM backend has basic support for integer code in ARM codegen mode, but lacks NEON and full Thumb support.

.o File Writing

This box indicates whether the target supports writing .o files (e.g. MachO, ELF, and/or COFF) files directly from the target. Note that the target also must include an assembly parser and general inline assembly support for full inline assembly support in the .o writer.

Targets that don't support this feature can obviously still write out .o files, they just rely on having an external assembler to translate from a .s file to a .o file (as is the case for many C compilers).

Tail Calls

This box indicates whether the target supports guaranteed tail calls. These are calls marked "tail" and use the fastcc calling convention. Please see the tail call section for more details.

Segmented Stacks

This box indicates whether the target supports segmented stacks. This replaces the traditional large C stack with many linked segments. It is compatible with the gcc implementation used by the Go front end. Basic support exists on the X86 backend. Currently vararg doesn't work and the object files are not marked the way the gold linker expects, but simple Go programs can be built by dragonegg.

Tail call optimization

Tail call optimization, callee reusing the stack of the caller, is currently supported on x86/x86-64 and PowerPC. It is performed if:

- Caller and callee have the calling convention `fastcc`, `cc 10` (GHC calling convention) or `cc 11` (HiPE calling convention).
- The call is a tail call - in tail position (ret immediately follows call and ret uses value of call or is void).
- Option `-tailcallopt` is enabled.
- Platform-specific constraints are met.

x86/x86-64 constraints:

- No variable argument lists are used.
- On x86-64 when generating GOT/PIC code only module-local calls (visibility = hidden or protected) are supported.

PowerPC constraints:

- No variable argument lists are used.
- No byval parameters are used.
- On ppc32/64 GOT/PIC only module-local calls (visibility = hidden or protected) are supported.

Example:

Call as `llc -tailcallopt test.ll`.

```
declare fastcc i32 @tailcallee(i32 inreg %a1, i32 inreg %a2, i32 %a3, i32 %a4)

define fastcc i32 @tailcaller(i32 %in1, i32 %in2) {
  %l1 = add i32 %in1, %in2
  %tmp = tail call fastcc i32 @tailcallee(i32 %in1 inreg, i32 %in2 inreg, i32 %in1, i32 %l1)
```

```
  ret i32 %tmp
}
```

Implications of `-tailcallopt`:

To support tail call optimization in situations where the callee has more arguments than the caller a 'callee pops arguments' convention is used. This currently causes each `fastcc` call that is not tail call optimized (because one or more of above constraints are not met) to be followed by a readjustment of the stack. So performance might be worse in such cases.

Sibling call optimization

Sibling call optimization is a restricted form of tail call optimization. Unlike tail call optimization described in the previous section, it can be performed automatically on any tail calls when `-tailcallopt` option is not specified.

Sibling call optimization is currently performed on x86/x86-64 when the following constraints are met:

- Caller and callee have the same calling convention. It can be either `c` or `fastcc`.
- The call is a tail call - in tail position (ret immediately follows call and ret uses value of call or is void).
- Caller and callee have matching return type or the callee result is not used.
- If any of the callee arguments are being passed in stack, they must be available in caller's own incoming argument stack and the frame offsets must be the same.

Example:

```
declare i32 @bar(i32, i32)

define i32 @foo(i32 %a, i32 %b, i32 %c) {
entry:
  %0 = tail call i32 @bar(i32 %a, i32 %b)
  ret i32 %0
}
```

The X86 backend

The X86 code generator lives in the `lib/Target/X86` directory. This code generator is capable of targeting a variety of x86-32 and x86-64 processors, and includes support for ISA extensions such as MMX and SSE.

X86 Target Triples supported

The following are the known target triples that are supported by the X86 backend. This is not an exhaustive list, and it would be useful to add those that people test.

- **i686-pc-linux-gnu** — Linux
- **i386-unknown-freebsd5.3** — FreeBSD 5.3
- **i686-pc-cygwin** — Cygwin on Win32
- **i686-pc-mingw32** — MingW on Win32
- **i386-pc-mingw32msvc** — MingW crosscompiler on Linux
- **i686-apple-darwin*** — Apple Darwin on X86
- **x86_64-unknown-linux-gnu** — Linux

X86 Calling Conventions supported

The following target-specific calling conventions are known to backend:

- **x86_StdCall** — stdcall calling convention seen on Microsoft Windows platform (CC ID = 64).
- **x86_FastCall** — fastcall calling convention seen on Microsoft Windows platform (CC ID = 65).
- **x86_ThisCall** — Similar to X86_StdCall. Passes first argument in ECX, others via stack. Callee is responsible for stack cleaning. This convention is used by MSVC by default for methods in its ABI (CC ID = 70).

Representing X86 addressing modes in MachineInstrs

The x86 has a very flexible way of accessing memory. It is capable of forming memory addresses of the following expression directly in integer instructions (which use ModR/M addressing):

```
SegmentReg: Base + [1,2,4,8] * IndexReg + Disp32
```

In order to represent this, LLVM tracks no less than 5 operands for each memory operand of this form. This means that the "load" form of '`mov`' has the following `MachineOperand`s in this order:

```
Index:        0     |    1        2       3           4          5
Meaning:   DestReg, | BaseReg,  Scale, IndexReg, Displacement Segment
OperandTy: VirtReg, | VirtReg, UnsImm, VirtReg,   SignExtImm  PhysReg
```

Stores, and all other instructions, treat the four memory operands in the same way and in the same order. If the segment register is unspecified (regno = 0), then no segment override is generated. "Lea" operations do not have a segment register specified, so they only have 4 operands for their memory reference.

X86 address spaces supported

x86 has a feature which provides the ability to perform loads and stores to different address spaces via the x86 segment registers. A segment override prefix byte on an instruction causes the instruction's memory access to go to the specified segment. LLVM address space 0 is the default address space, which includes the stack, and any unqualified memory accesses in a program. Address spaces 1-255 are currently reserved for user-defined code. The GS-segment is represented by address space 256, while the FS-segment is represented by address space 257. Other x86 segments have yet to be allocated address space numbers.

While these address spaces may seem similar to TLS via the `thread_local` keyword, and often use the same underlying hardware, there are some fundamental differences.

The `thread_local` keyword applies to global variables and specifies that they are to be allocated in thread-local memory. There are no type qualifiers involved, and these variables can be pointed to with normal pointers and accessed with normal loads and stores. The `thread_local` keyword is target-independent at the LLVM IR level (though LLVM doesn't yet have implementations of it for some configurations)

Special address spaces, in contrast, apply to static types. Every load and store has a particular address space in its address operand type, and this is what determines which address space is accessed. LLVM ignores these special address space qualifiers on global variables, and does not provide a way to directly allocate storage in them. At the LLVM IR level, the behavior of these special address spaces depends in part on the underlying OS or runtime environment, and they are specific to x86 (and LLVM doesn't yet handle them correctly in some cases).

Some operating systems and runtime environments use (or may in the future use) the FS/GS-segment registers for various low-level purposes, so care should be taken when considering them.

Instruction naming

An instruction name consists of the base name, a default operand size, and a a character per operand with an optional special size. For example:

```
ADD8rr      -> add, 8-bit register, 8-bit register
IMUL16rmi   -> imul, 16-bit register, 16-bit memory, 16-bit immediate
IMUL16rmi8  -> imul, 16-bit register, 16-bit memory, 8-bit immediate
MOVSX32rm16 -> movsx, 32-bit register, 16-bit memory
```

The PowerPC backend

The PowerPC code generator lives in the lib/Target/PowerPC directory. The code generation is retargetable to several variations or *subtargets* of the PowerPC ISA; including ppc32, ppc64 and altivec.

LLVM PowerPC ABI

LLVM follows the AIX PowerPC ABI, with two deviations. LLVM uses a PC relative (PIC) or static addressing for accessing global values, so no TOC (r2) is used. Second, r31 is used as a frame pointer to allow dynamic growth of a stack frame. LLVM takes advantage of having no TOC to provide space to save the frame pointer in the PowerPC linkage area of the caller frame. Other details of PowerPC ABI can be found at PowerPC ABI. Note: This link describes the 32 bit ABI. The 64 bit ABI is similar except space for GPRs are 8 bytes wide (not 4) and r13 is reserved for system use.

Frame Layout

The size of a PowerPC frame is usually fixed for the duration of a function's invocation. Since the frame is fixed size, all references into the frame can be accessed via fixed offsets from the stack pointer. The exception to this is when dynamic alloca or variable sized arrays are present, then a base pointer (r31) is used as a proxy for the stack pointer and stack pointer is free to grow or shrink. A base pointer is also used if llvm-gcc is not passed the -fomit-frame-pointer flag. The stack pointer is always aligned to 16 bytes, so that space allocated for altivec vectors will be properly aligned.

An invocation frame is laid out as follows (low memory at top):

The *linkage* area is used by a callee to save special registers prior to allocating its own frame. Only three entries are relevant to LLVM. The first entry is the previous stack pointer (sp), aka link. This allows probing tools like gdb or exception handlers to quickly scan the frames in the stack. A function epilog can also use the link to pop the frame from the stack. The third entry in the linkage area is used to save the return address from the lr register. Finally, as mentioned above, the last entry is used to save the previous frame pointer (r31.) The entries in the linkage area are the size of a GPR, thus the linkage area is 24 bytes long in 32 bit mode and 48 bytes in 64 bit mode.

32 bit linkage area:

64 bit linkage area:

The *parameter area* is used to store arguments being passed to a callee function. Following the PowerPC ABI, the first few arguments are actually passed in registers, with the space in the parameter area unused. However, if there are not enough registers or the callee is a thunk or vararg function, these register arguments can be spilled into the parameter area. Thus, the parameter area must be large enough to store all the parameters for the largest call sequence made by the caller. The size must also be minimally large enough to spill registers r3-r10. This allows callees blind to the call signature, such as thunks and vararg functions, enough space to cache the argument registers. Therefore, the parameter area is minimally 32 bytes (64 bytes in 64 bit mode.) Also note that since the parameter area is a fixed offset from the top of the frame, that a callee can access its spilt arguments using fixed offsets from the stack pointer (or base pointer.)

Combining the information about the linkage, parameter areas and alignment. A stack frame is minimally 64 bytes in 32 bit mode and 128 bytes in 64 bit mode.

The *dynamic area* starts out as size zero. If a function uses dynamic alloca then space is added to the stack, the linkage and parameter areas are shifted to top of stack, and the new space is available immediately below the linkage and parameter areas. The cost of shifting the linkage and parameter areas is minor since only the link value needs to be copied. The link value can be easily fetched by adding the original frame size to the base pointer. Note that allocations in the dynamic space need to observe 16 byte alignment.

The *locals area* is where the llvm compiler reserves space for local variables.

The *saved registers area* is where the llvm compiler spills callee saved registers on entry to the callee.

Prolog/Epilog

The llvm prolog and epilog are the same as described in the PowerPC ABI, with the following exceptions. Callee saved registers are spilled after the frame is created. This allows the llvm epilog/prolog support to be common with other targets. The base pointer callee saved register r31 is saved in the TOC slot of linkage area. This simplifies allocation of space for the base pointer and makes it convenient to locate programatically and during debugging.

Dynamic Allocation

Note: TODO - More to come.

The NVPTX backend

The NVPTX code generator under lib/Target/NVPTX is an open-source version of the NVIDIA NVPTX code generator for LLVM. It is contributed by NVIDIA and is a port of the code generator used in the CUDA compiler (nvcc). It targets the PTX 3.0/3.1 ISA and can target any compute capability greater than or equal to 2.0 (Fermi).

This target is of production quality and should be completely compatible with the official NVIDIA toolchain.

Code Generator Options:

4.7 Exception Handling in LLVM

4.7.1 Introduction

This document is the central repository for all information pertaining to exception handling in LLVM. It describes the format that LLVM exception handling information takes, which is useful for those interested in creating front-ends or dealing directly with the information. Further, this document provides specific examples of what exception handling information is used for in C and C++.

Itanium ABI Zero-cost Exception Handling

Exception handling for most programming languages is designed to recover from conditions that rarely occur during general use of an application. To that end, exception handling should not interfere with the main flow of an application's algorithm by performing checkpointing tasks, such as saving the current pc or register state.

The Itanium ABI Exception Handling Specification defines a methodology for providing outlying data in the form of exception tables without inlining speculative exception handling code in the flow of an application's main algorithm. Thus, the specification is said to add "zero-cost" to the normal execution of an application.

A more complete description of the Itanium ABI exception handling runtime support of can be found at Itanium C++ ABI: Exception Handling. A description of the exception frame format can be found at Exception Frames, with details of the DWARF 4 specification at DWARF 4 Standard. A description for the C++ exception table formats can be found at Exception Handling Tables.

Setjmp/Longjmp Exception Handling

Setjmp/Longjmp (SJLJ) based exception handling uses LLVM intrinsics llvm.eh.sjlj.setjmp and llvm.eh.sjlj.longjmp to handle control flow for exception handling.

For each function which does exception processing — be it `try`/`catch` blocks or cleanups — that function registers itself on a global frame list. When exceptions are unwinding, the runtime uses this list to identify which functions need processing.

Landing pad selection is encoded in the call site entry of the function context. The runtime returns to the function via llvm.eh.sjlj.longjmp, where a switch table transfers control to the appropriate landing pad based on the index stored in the function context.

In contrast to DWARF exception handling, which encodes exception regions and frame information in out-of-line tables, SJLJ exception handling builds and removes the unwind frame context at runtime. This results in faster exception handling at the expense of slower execution when no exceptions are thrown. As exceptions are, by their nature, intended for uncommon code paths, DWARF exception handling is generally preferred to SJLJ.

Overview

When an exception is thrown in LLVM code, the runtime does its best to find a handler suited to processing the circumstance.

The runtime first attempts to find an *exception frame* corresponding to the function where the exception was thrown. If the programming language supports exception handling (e.g. C++), the exception frame contains a reference to an exception table describing how to process the exception. If the language does not support exception handling (e.g. C), or if the exception needs to be forwarded to a prior activation, the exception frame contains information about how to unwind the current activation and restore the state of the prior activation. This process is repeated until the exception is handled. If the exception is not handled and no activations remain, then the application is terminated with an appropriate error message.

Because different programming languages have different behaviors when handling exceptions, the exception handling ABI provides a mechanism for supplying *personalities*. An exception handling personality is defined by way of a *personality function* (e.g. `__gxx_personality_v0` in C++), which receives the context of the exception, an *exception structure* containing the exception object type and value, and a reference to the exception table for the current function. The personality function for the current compile unit is specified in a *common exception frame*.

The organization of an exception table is language dependent. For C++, an exception table is organized as a series of code ranges defining what to do if an exception occurs in that range. Typically, the information associated with a range defines which types of exception objects (using C++ *type info*) that are handled in that range, and an associated action that should take place. Actions typically pass control to a *landing pad*.

A landing pad corresponds roughly to the code found in the `catch` portion of a `try`/`catch` sequence. When execution resumes at a landing pad, it receives an *exception structure* and a *selector value* corresponding to the *type* of exception thrown. The selector is then used to determine which *catch* should actually process the exception.

4.7.2 LLVM Code Generation

From a C++ developer's perspective, exceptions are defined in terms of the `throw` and `try`/`catch` statements. In this section we will describe the implementation of LLVM exception handling in terms of C++ examples.

Throw

Languages that support exception handling typically provide a `throw` operation to initiate the exception process. Internally, a `throw` operation breaks down into two steps.

1. A request is made to allocate exception space for an exception structure. This structure needs to survive beyond the current activation. This structure will contain the type and value of the object being thrown.
2. A call is made to the runtime to raise the exception, passing the exception structure as an argument.

In C++, the allocation of the exception structure is done by the `__cxa_allocate_exception` runtime function. The exception raising is handled by `__cxa_throw`. The type of the exception is represented using a C++ RTTI structure.

Try/Catch

A call within the scope of a *try* statement can potentially raise an exception. In those circumstances, the LLVM C++ front-end replaces the call with an `invoke` instruction. Unlike a call, the `invoke` has two potential continuation points:

1. where to continue when the call succeeds as per normal, and
2. where to continue if the call raises an exception, either by a throw or the unwinding of a throw

The term used to define the place where an `invoke` continues after an exception is called a *landing pad*. LLVM landing pads are conceptually alternative function entry points where an exception structure reference and a type info index are passed in as arguments. The landing pad saves the exception structure reference and then proceeds to select the catch block that corresponds to the type info of the exception object.

The LLVM *'landingpad' Instruction* is used to convey information about the landing pad to the back end. For C++, the `landingpad` instruction returns a pointer and integer pair corresponding to the pointer to the *exception structure* and the *selector value* respectively.

The `landingpad` instruction takes a reference to the personality function to be used for this `try`/`catch` sequence. The remainder of the instruction is a list of *cleanup*, *catch*, and *filter* clauses. The exception is tested against the clauses sequentially from first to last. The clauses have the following meanings:

- `catch <type> @ExcType`
  - This clause means that the landingpad block should be entered if the exception being thrown is of type `@ExcType` or a subtype of `@ExcType`. For C++, `@ExcType` is a pointer to the `std::type_info` object (an RTTI object) representing the C++ exception type.
  - If `@ExcType` is `null`, any exception matches, so the landingpad should always be entered. This is used for C++ catch-all blocks ("`catch (...)`").
  - When this clause is matched, the selector value will be equal to the value returned by "`@llvm.eh.typeid.for(i8* @ExcType)`". This will always be a positive value.
- `filter <type> [<type> @ExcType1, ..., <type> @ExcTypeN]`
  - This clause means that the landingpad should be entered if the exception being thrown does *not* match any of the types in the list (which, for C++, are again specified as `std::type_info` pointers).
  - C++ front-ends use this to implement C++ exception specifications, such as "`void foo() throw (ExcType1, ..., ExcTypeN) { ... }`".
  - When this clause is matched, the selector value will be negative.
  - The array argument to `filter` may be empty; for example, "`[0 x i8**] undef`". This means that the landingpad should always be entered. (Note that such a `filter` would not be equivalent to "`catch i8* null`", because `filter` and `catch` produce negative and positive selector values respectively.)
- `cleanup`
  - This clause means that the landingpad should always be entered.
  - C++ front-ends use this for calling objects' destructors.
  - When this clause is matched, the selector value will be zero.
  - The runtime may treat "`cleanup`" differently from "`catch <type> null`".

    In C++, if an unhandled exception occurs, the language runtime will call `std::terminate()`, but it is implementation-defined whether the runtime unwinds the stack and calls object destructors first. For example, the GNU C++ unwinder does not call object destructors when an unhandled exception occurs. The reason for this is to improve debuggability: it ensures that `std::terminate()` is called from the context of the `throw`, so that this context is not lost by unwinding the stack. A runtime will typically

implement this by searching for a matching non-`cleanup` clause, and aborting if it does not find one, before entering any landingpad blocks.

Once the landing pad has the type info selector, the code branches to the code for the first catch. The catch then checks the value of the type info selector against the index of type info for that catch. Since the type info index is not known until all the type infos have been gathered in the backend, the catch code must call the llvm.eh.typeid.for intrinsic to determine the index for a given type info. If the catch fails to match the selector then control is passed on to the next catch.

Finally, the entry and exit of catch code is bracketed with calls to `__cxa_begin_catch` and `__cxa_end_catch`.

- `__cxa_begin_catch` takes an exception structure reference as an argument and returns the value of the exception object.
- `__cxa_end_catch` takes no arguments. This function:
  1. Locates the most recently caught exception and decrements its handler count,
  2. Removes the exception from the *caught* stack if the handler count goes to zero, and
  3. Destroys the exception if the handler count goes to zero and the exception was not re-thrown by throw.

Note: a rethrow from within the catch may replace this call with a `__cxa_rethrow`.

Cleanups

A cleanup is extra code which needs to be run as part of unwinding a scope. C++ destructors are a typical example, but other languages and language extensions provide a variety of different kinds of cleanups. In general, a landing pad may need to run arbitrary amounts of cleanup code before actually entering a catch block. To indicate the presence of cleanups, a *'landingpad' Instruction* should have a *cleanup* clause. Otherwise, the unwinder will not stop at the landing pad if there are no catches or filters that require it to.

Note: Do not allow a new exception to propagate out of the execution of a cleanup. This can corrupt the internal state of the unwinder. Different languages describe different high-level semantics for these situations: for example, C++ requires that the process be terminated, whereas Ada cancels both exceptions and throws a third.

When all cleanups are finished, if the exception is not handled by the current function, resume unwinding by calling the resume instruction, passing in the result of the `landingpad` instruction for the original landing pad.

Throw Filters

C++ allows the specification of which exception types may be thrown from a function. To represent this, a top level landing pad may exist to filter out invalid types. To express this in LLVM code the *'landingpad' Instruction* will have a filter clause. The clause consists of an array of type infos. `landingpad` will return a negative value if the exception does not match any of the type infos. If no match is found then a call to `__cxa_call_unexpected` should be made, otherwise `_Unwind_Resume`. Each of these functions requires a reference to the exception structure. Note that the most general form of a `landingpad` instruction can have any number of catch, cleanup, and filter clauses (though having more than one cleanup is pointless). The LLVM C++ front-end can generate such `landingpad` instructions due to inlining creating nested exception handling scopes.

Restrictions

The unwinder delegates the decision of whether to stop in a call frame to that call frame's language-specific personality function. Not all unwinders guarantee that they will stop to perform cleanups. For example, the GNU C++ unwinder

doesn't do so unless the exception is actually caught somewhere further up the stack.

In order for inlining to behave correctly, landing pads must be prepared to handle selector results that they did not originally advertise. Suppose that a function catches exceptions of type `A`, and it's inlined into a function that catches exceptions of type `B`. The inliner will update the `landingpad` instruction for the inlined landing pad to include the fact that `B` is also caught. If that landing pad assumes that it will only be entered to catch an `A`, it's in for a rude awakening. Consequently, landing pads must test for the selector results they understand and then resume exception propagation with the resume instruction if none of the conditions match.

4.7.3 Exception Handling Intrinsics

In addition to the `landingpad` and `resume` instructions, LLVM uses several intrinsic functions (name prefixed with `llvm.eh`) to provide exception handling information at various points in generated code.

llvm.eh.typeid.for

```
i32 @llvm.eh.typeid.for(i8* %type_info)
```

This intrinsic returns the type info index in the exception table of the current function. This value can be used to compare against the result of `landingpad` instruction. The single argument is a reference to a type info.

Uses of this intrinsic are generated by the C++ front-end.

SJLJ Intrinsics

The `llvm.eh.sjlj` intrinsics are used internally within LLVM's backend. Uses of them are generated by the backend's `SjLjEHPrepare` pass.

llvm.eh.sjlj.setjmp

```
i32 @llvm.eh.sjlj.setjmp(i8* %setjmp_buf)
```

For SJLJ based exception handling, this intrinsic forces register saving for the current function and stores the address of the following instruction for use as a destination address by llvm.eh.sjlj.longjmp. The buffer format and the overall functioning of this intrinsic is compatible with the GCC `__builtin_setjmp` implementation allowing code built with the clang and GCC to interoperate.

The single parameter is a pointer to a five word buffer in which the calling context is saved. The front end places the frame pointer in the first word, and the target implementation of this intrinsic should place the destination address for a llvm.eh.sjlj.longjmp in the second word. The following three words are available for use in a target-specific manner.

llvm.eh.sjlj.longjmp

```
void @llvm.eh.sjlj.longjmp(i8* %setjmp_buf)
```

For SJLJ based exception handling, the `llvm.eh.sjlj.longjmp` intrinsic is used to implement `__builtin_longjmp()`. The single parameter is a pointer to a buffer populated by llvm.eh.sjlj.setjmp. The frame pointer and stack pointer are restored from the buffer, then control is transferred to the destination address.

llvm.eh.sjlj.lsda

```
i8* @llvm.eh.sjlj.lsda()
```

For SJLJ based exception handling, the `llvm.eh.sjlj.lsda` intrinsic returns the address of the Language Specific Data Area (LSDA) for the current function. The SJLJ front-end code stores this address in the exception handling function context for use by the runtime.

llvm.eh.sjlj.callsite

```
void @llvm.eh.sjlj.callsite(i32 %call_site_num)
```

For SJLJ based exception handling, the `llvm.eh.sjlj.callsite` intrinsic identifies the callsite value associated with the following `invoke` instruction. This is used to ensure that landing pad entries in the LSDA are generated in matching order.

4.7.4 Asm Table Formats

There are two tables that are used by the exception handling runtime to determine which actions should be taken when an exception is thrown.

Exception Handling Frame

An exception handling frame `eh_frame` is very similar to the unwind frame used by DWARF debug info. The frame contains all the information necessary to tear down the current frame and restore the state of the prior frame. There is an exception handling frame for each function in a compile unit, plus a common exception handling frame that defines information common to all functions in the unit.

Exception Tables

An exception table contains information about what actions to take when an exception is thrown in a particular part of a function's code. There is one exception table per function, except leaf functions and functions that have calls only to non-throwing functions. They do not need an exception table.

4.8 LLVM Link Time Optimization: Design and Implementation

4.8.1 Description

LLVM features powerful intermodular optimizations which can be used at link time. Link Time Optimization (LTO) is another name for intermodular optimization when performed during the link stage. This document describes the interface and design between the LTO optimizer and the linker.

4.8.2 Design Philosophy

The LLVM Link Time Optimizer provides complete transparency, while doing intermodular optimization, in the compiler tool chain. Its main goal is to let the developer take advantage of intermodular optimizations without making any significant changes to the developer's makefiles or build system. This is achieved through tight integration with the linker. In this model, the linker treates LLVM bitcode files like native object files and allows mixing and matching among them. The linker uses libLTO, a shared object, to handle LLVM bitcode files. This tight integration between the linker and LLVM optimizer helps to do optimizations that are not possible in other models. The linker input allows the optimizer to avoid relying on conservative escape analysis.

Example of link time optimization

The following example illustrates the advantages of LTO's integrated approach and clean interface. This example requires a system linker which supports LTO through the interface described in this document. Here, clang transparently invokes system linker.

- Input source file `a.c` is compiled into LLVM bitcode form.
- Input source file `main.c` is compiled into native object code.

```
--- a.h ---
extern int foo1(void);
extern void foo2(void);
extern void foo4(void);

--- a.c ---
#include "a.h"

static signed int i = 0;

void foo2(void) {
  i = -1;
}
```

```
static int foo3() {
  foo4();
  return 10;
}

int foo1(void) {
  int data = 0;

  if (i < 0)
    data = foo3();

  data = data + 42;
  return data;
}

--- main.c ---
#include <stdio.h>
#include "a.h"

void foo4(void) {
  printf("Hi\n");
}

int main() {
  return foo1();
}
```

To compile, run:

```
% clang -emit-llvm -c a.c -o a.o    # <-- a.o is LLVM bitcode file
% clang -c main.c -o main.o         # <-- main.o is native object file
% clang a.o main.o -o main          # <-- standard link command without modifications
```

- In this example, the linker recognizes that `foo2()` is an externally visible symbol defined in LLVM bitcode file. The linker completes its usual symbol resolution pass and finds that `foo2()` is not used anywhere. This information is used by the LLVM optimizer and it removes `foo2()`.
- As soon as `foo2()` is removed, the optimizer recognizes that condition `i < 0` is always false, which means `foo3()` is never used. Hence, the optimizer also removes `foo3()`.
- And this in turn, enables linker to remove `foo4()`.

This example illustrates the advantage of tight integration with the linker. Here, the optimizer can not remove `foo3()` without the linker's input.

Alternative Approaches

Compiler driver invokes link time optimizer separately. In this model the link time optimizer is not able to take advantage of information collected during the linker's normal symbol resolution phase. In the above example, the optimizer can not remove `foo2()` without the linker's input because it is externally visible. This in turn prohibits the optimizer from removing `foo3()`.

Use separate tool to collect symbol information from all object files. In this model, a new, separate, tool or library replicates the linker's capability to collect information for link time optimization. Not only is this code duplication difficult to justify, but it also has several other disadvantages. For example, the linking semantics and the features provided by the linker on various platform are not unique. This means, this new tool needs to support all such features and platforms in one super tool or a separate tool per platform is required. This increases

maintenance cost for link time optimizer significantly, which is not necessary. This approach also requires staying synchronized with linker developements on various platforms, which is not the main focus of the link time optimizer. Finally, this approach increases end user's build time due to the duplication of work done by this separate tool and the linker itself.

4.8.3 Multi-phase communication between `libLTO` and linker

The linker collects information about symbol defininitions and uses in various link objects which is more accurate than any information collected by other tools during typical build cycles. The linker collects this information by looking at the definitions and uses of symbols in native .o files and using symbol visibility information. The linker also uses user-supplied information, such as a list of exported symbols. LLVM optimizer collects control flow information, data flow information and knows much more about program structure from the optimizer's point of view. Our goal is to take advantage of tight integration between the linker and the optimizer by sharing this information during various linking phases.

Phase 1 : Read LLVM Bitcode Files

The linker first reads all object files in natural order and collects symbol information. This includes native object files as well as LLVM bitcode files. To minimize the cost to the linker in the case that all .o files are native object files, the linker only calls `lto_module_create()` when a supplied object file is found to not be a native object file. If `lto_module_create()` returns that the file is an LLVM bitcode file, the linker then iterates over the module using `lto_module_get_symbol_name()` and `lto_module_get_symbol_attribute()` to get all symbols defined and referenced. This information is added to the linker's global symbol table.

The lto* functions are all implemented in a shared object libLTO. This allows the LLVM LTO code to be updated independently of the linker tool. On platforms that support it, the shared object is lazily loaded.

Phase 2 : Symbol Resolution

In this stage, the linker resolves symbols using global symbol table. It may report undefined symbol errors, read archive members, replace weak symbols, etc. The linker is able to do this seamlessly even though it does not know the exact content of input LLVM bitcode files. If dead code stripping is enabled then the linker collects the list of live symbols.

Phase 3 : Optimize Bitcode Files

After symbol resolution, the linker tells the LTO shared object which symbols are needed by native object files. In the example above, the linker reports that only `foo1()` is used by native object files using `lto_codegen_add_must_preserve_symbol()`. Next the linker invokes the LLVM optimizer and code generators using `lto_codegen_compile()` which returns a native object file creating by merging the LLVM bitcode files and applying various optimization passes.

Phase 4 : Symbol Resolution after optimization

In this phase, the linker reads optimized a native object file and updates the internal global symbol table to reflect any changes. The linker also collects information about any changes in use of external symbols by LLVM bitcode files. In the example above, the linker notes that `foo4()` is not used any more. If dead code stripping is enabled then the linker refreshes the live symbol information appropriately and performs dead code stripping.

After this phase, the linker continues linking as if it never saw LLVM bitcode files.

4.8.4 libLTO

libLTO is a shared object that is part of the LLVM tools, and is intended for use by a linker. libLTO provides an abstract C interface to use the LLVM interprocedural optimizer without exposing details of LLVM's internals. The intention is to keep the interface as stable as possible even when the LLVM optimizer continues to evolve. It should even be possible for a completely different compilation technology to provide a different libLTO that works with their object files and the standard linker tool.

lto_module_t

A non-native object file is handled via an lto_module_t. The following functions allow the linker to check if a file (on disk or in a memory buffer) is a file which libLTO can process:

```
lto_module_is_object_file(const char*)
lto_module_is_object_file_for_target(const char*, const char*)
lto_module_is_object_file_in_memory(const void*, size_t)
lto_module_is_object_file_in_memory_for_target(const void*, size_t, const char*)
```

If the object file can be processed by libLTO, the linker creates a lto_module_t by using one of:

```
lto_module_create(const char*)
lto_module_create_from_memory(const void*, size_t)
```

and when done, the handle is released via

```
lto_module_dispose(lto_module_t)
```

The linker can introspect the non-native object file by getting the number of symbols and getting the name and attributes of each symbol via:

```
lto_module_get_num_symbols(lto_module_t)
lto_module_get_symbol_name(lto_module_t, unsigned int)
lto_module_get_symbol_attribute(lto_module_t, unsigned int)
```

The attributes of a symbol include the alignment, visibility, and kind.

lto_code_gen_t

Once the linker has loaded each non-native object files into an lto_module_t, it can request libLTO to process them all and generate a native object file. This is done in a couple of steps. First, a code generator is created with:

```
lto_codegen_create()
```

Then, each non-native object file is added to the code generator with:

```
lto_codegen_add_module(lto_code_gen_t, lto_module_t)
```

The linker then has the option of setting some codegen options. Whether or not to generate DWARF debug info is set with:

```
lto_codegen_set_debug_model(lto_code_gen_t)
```

Which kind of position independence is set with:

```
lto_codegen_set_pic_model(lto_code_gen_t)
```

And each symbol that is referenced by a native object file or otherwise must not be optimized away is set with:

```
lto_codegen_add_must_preserve_symbol(lto_code_gen_t, const char*)
```

After all these settings are done, the linker requests that a native object file be created from the modules with the settings using:

```
lto_codegen_compile(lto_code_gen_t, size*)
```

which returns a pointer to a buffer containing the generated native object file. The linker then parses that and links it with the rest of the native object files.

4.9 Segmented Stacks in LLVM

4.9.1 Introduction

Segmented stack allows stack space to be allocated incrementally than as a monolithic chunk (of some worst case size) at thread initialization. This is done by allocating stack blocks (henceforth called *stacklets*) and linking them into a doubly linked list. The function prologue is responsible for checking if the current stacklet has enough space for the function to execute; and if not, call into the libgcc runtime to allocate more stack space. Segmented stacks are enabled with the `"split-stack"` attribute on LLVM functions.

The runtime functionality is already there in libgcc.

4.9.2 Implementation Details

Allocating Stacklets

As mentioned above, the function prologue checks if the current stacklet has enough space. The current approach is to use a slot in the TCB to store the current stack limit (minus the amount of space needed to allocate a new block) - this slot's offset is again dictated by `libgcc`. The generated assembly looks like this on x86-64:

```
  leaq     -8(%rsp), %r10
  cmpq     %fs:112,  %r10
  jg       .LBB0_2

  # More stack space needs to be allocated
  movabsq  $8, %r10   # The amount of space needed
  movabsq  $0, %r11   # The total size of arguments passed on stack
  callq    __morestack
  ret                 # The reason for this extra return is explained below
.LBB0_2:
  # Usual prologue continues here
```

The size of function arguments on the stack needs to be passed to `__morestack` (this function is implemented in `libgcc`) since that number of bytes has to be copied from the previous stacklet to the current one. This is so that SP (and FP) relative addressing of function arguments work as expected.

The unusual `ret` is needed to have the function which made a call to `__morestack` return correctly. `__morestack`, instead of returning, calls into `.LBB0_2`. This is possible since both, the size of the `ret` instruction and the PC of call to `__morestack` are known. When the function body returns, control is transferred back to `__morestack`. `__morestack` then de-allocates the new stacklet, restores the correct SP value, and does a second return, which returns control to the correct caller.

Variable Sized Allocas

The section on allocating stacklets automatically assumes that every stack frame will be of fixed size. However, LLVM allows the use of the `llvm.alloca` intrinsic to allocate dynamically sized blocks of memory on the stack. When faced with such a variable-sized alloca, code is generated to:

- Check if the current stacklet has enough space. If yes, just bump the SP, like in the normal case.
- If not, generate a call to `libgcc`, which allocates the memory from the heap.

The memory allocated from the heap is linked into a list in the current stacklet, and freed along with the same. This prevents a memory leak.

4.10 TableGen Fundamentals

4.10.1 Moved

The TableGen fundamentals documentation has moved to a directory on its own and is now available at *TableGen*. Please, change your links to that page.

4.11 TableGen

4.11.1 TableGen BackEnds

Introduction

TableGen backends are at the core of TableGen's functionality. The source files provide the semantics to a generated (in memory) structure, but it's up to the backend to print this out in a way that is meaningful to the user (normally a C program including a file or a textual list of warnings, options and error messages).

TableGen is used by both LLVM and Clang with very different goals. LLVM uses it as a way to automate the generation of massive amounts of information regarding instructions, schedules, cores and architecture features. Some backends generate output that is consumed by more than one source file, so they need to be created in a way that is easy to use pre-processor tricks. Some backends can also print C code structures, so that they can be directly included as-is.

Clang, on the other hand, uses it mainly for diagnostic messages (errors, warnings, tips) and attributes, so more on the textual end of the scale.

LLVM BackEnds

> **Warning:** This document is raw. Each section below needs three sub-sections: description of its purpose with a list of users, output generated from generic input, and finally why it needed a new backend (in case there's something similar).

Overall, each backend will take the same TableGen file type and transform into similar output for different targets/uses. There is an implicit contract between the TableGen files, the back-ends and their users.

For instance, a global contract is that each back-end produces macro-guarded sections. Based on whether the file is included by a header or a source file, or even in which context of each file the include is being used, you have todefine a macro just before including it, to get the right output:

```
#define GET_REGINFO_TARGET_DESC
#include "ARMGenRegisterInfo.inc"
```

And just part of the generated file would be included. This is useful if you need the same information in multiple formats (instantiation, initialization, getter/setter functions, etc) from the same source TableGen file without having to re-compile the TableGen file multiple times.

Sometimes, multiple macros might be defined before the same include file to output multiple blocks:

```
#define GET_REGISTER_MATCHER
#define GET_SUBTARGET_FEATURE_NAME
#define GET_MATCHER_IMPLEMENTATION
#include "ARMGenAsmMatcher.inc"
```

The macros will be undef'd automatically as they're used, in the include file.

On all LLVM back-ends, the `llvm-tblgen` binary will be executed on the root TableGen file `<Target>.td`, which should include all others. This guarantees that all information needed is accessible, and that no duplication is needed in the TbleGen files.

CodeEmitter

Purpose: CodeEmitterGen uses the descriptions of instructions and their fields to construct an automated code emitter: a function that, given a MachineInstr, returns the (currently, 32-bit unsigned) value of the instruction.

Output: C++ code, implementing the target's CodeEmitter class by overriding the virtual functions as `<Target>CodeEmitter::function()`.

Usage: Used to include directly at the end of `<Target>MCCodeEmitter.cpp`.

RegisterInfo

Purpose: This tablegen backend is responsible for emitting a description of a target register file for a code generator. It uses instances of the Register, RegisterAliases, and RegisterClass classes to gather this information.

Output: C++ code with enums and structures representing the register mappings, properties, masks, etc.

Usage: Both on `<Target>BaseRegisterInfo` and `<Target>MCTargetDesc` (headers and source files) with macros defining in which they are for declaration vs. initialization issues.

InstrInfo

Purpose: This tablegen backend is responsible for emitting a description of the target instruction set for the code generator. (what are the differences from CodeEmitter?)

Output: C++ code with enums and structures representing the register mappings, properties, masks, etc.

Usage: Both on `<Target>BaseInstrInfo` and `<Target>MCTargetDesc` (headers and source files) with macros defining in which they are for declaration vs.

AsmWriter

Purpose: Emits an assembly printer for the current target.

Output: Implementation of `<Target>InstPrinter::printInstruction()`, among other things.

Usage: Included directly into `InstPrinter/<Target>InstPrinter.cpp`.

AsmMatcher

Purpose: Emits a target specifier matcher for converting parsed assembly operands in the MCInst structures. It also emits a matcher for custom operand parsing. Extensive documentation is written on the `AsmMatcherEmitter.cpp` file.

Output: Assembler parsers' matcher functions, declarations, etc.

Usage: Used in back-ends' `AsmParser/<Target>AsmParser.cpp` for building the AsmParser class.

Disassembler

Purpose: Contains disassembler table emitters for various architectures. Extensive documentation is written on the `DisassemblerEmitter.cpp` file.

Output: Decoding tables, static decoding functions, etc.

Usage: Directly included in `Disassembler/<Target>Disassembler.cpp` to cater for all default decodings, after all hand-made ones.

PseudoLowering

Purpose: Generate pseudo instruction lowering.

Output: Implements `ARMAsmPrinter::emitPseudoExpansionLowering()`.

Usage: Included directly into `<Target>AsmPrinter.cpp`.

CallingConv

Purpose: Responsible for emitting descriptions of the calling conventions supported by this target.

Output: Implement static functions to deal with calling conventions chained by matching styles, returning false on no match.

Usage: Used in ISelLowering and FastIsel as function pointers to implementation returned by a CC sellection function.

DAGISel

Purpose: Generate a DAG instruction selector.

Output: Creates huge functions for automating DAG selection.

Usage: Included in `<Target>ISelDAGToDAG.cpp` inside the target's implementation of `SelectionDAGISel`.

DFAPacketizer

Purpose: This class parses the Schedule.td file and produces an API that can be used to reason about whether an instruction can be added to a packet on a VLIW architecture. The class internally generates a deterministic finite automaton (DFA) that models all possible mappings of machine instructions to functional units as instructions are added to a packet.

Output: Scheduling tables for GPU back-ends (Hexagon, AMD).

Usage: Included directly on `<Target>InstrInfo.cpp`.

FastISel

Purpose: This tablegen backend emits code for use by the "fast" instruction selection algorithm. See the comments at the top of lib/CodeGen/SelectionDAG/FastISel.cpp for background. This file scans through the target's tablegen instruction-info files and extracts instructions with obvious-looking patterns, and it emits code to look up these instructions by type and operator.

Output: Generates `Predicate` and `FastEmit` methods.

Usage: Implements private methods of the targets' implementation of `FastISel` class.

Subtarget

Purpose: Generate subtarget enumerations.

Output: Enums, globals, local tables for sub-target information.

Usage: Populates `<Target>Subtarget` and `MCTargetDesc/<Target>MCTargetDesc` files (both headers and source).

Intrinsic

Purpose: Generate (target) intrinsic information.

OptParserDefs

Purpose: Print enum values for a class.

CTags

Purpose: This tablegen backend emits an index of definitions in ctags(1) format. A helper script, utils/TableGen/tdtags, provides an easier-to-use interface; run 'tdtags -H' for documentation.

Clang BackEnds

ClangAttrClasses

Purpose: Creates Attrs.inc, which contains semantic attribute class declarations for any attribute in `Attr.td` that has not set `ASTNode = 0`. This file is included as part of `Attr.h`.

ClangAttrParserStringSwitches

Purpose: Creates AttrParserStringSwitches.inc, which contains StringSwitch::Case statements for parser-related string switches. Each switch is given its own macro (such as `CLANG_ATTR_ARG_CONTEXT_LIST`, or `CLANG_ATTR_IDENTIFIER_ARG_LIST`), which is expected to be defined before including AttrParserStringSwitches.inc, and undefined after.

ClangAttrImpl

Purpose: Creates AttrImpl.inc, which contains semantic attribute class definitions for any attribute in `Attr.td` that has not set `ASTNode = 0`. This file is included as part of `AttrImpl.cpp`.

ClangAttrList

Purpose: Creates AttrList.inc, which is used when a list of semantic attribute identifiers is required. For instance, `AttrKinds.h` includes this file to generate the list of `attr::Kind` enumeration values. This list is separated out into multiple categories: attributes, inheritable attributes, and inheritable parameter attributes. This categorization happens automatically based on information in `Attr.td` and is used to implement the `classof` functionality required for `dyn_cast` and similar APIs.

ClangAttrPCHRead

Purpose: Creates AttrPCHRead.inc, which is used to deserialize attributes in the `ASTReader::ReadAttributes` function.

ClangAttrPCHWrite

Purpose: Creates AttrPCHWrite.inc, which is used to serialize attributes in the `ASTWriter::WriteAttributes` function.

ClangAttrSpellings

Purpose: Creates AttrSpellings.inc, which is used to implement the `__has_attribute` feature test macro.

ClangAttrSpellingListIndex

Purpose: Creates AttrSpellingListIndex.inc, which is used to map parsed attribute spellings (including which syntax or scope was used) to an attribute spelling list index. These spelling list index values are internal implementation details exposed via `AttributeList::getAttributeSpellingListIndex`.

ClangAttrVisitor

Purpose: Creates AttrVisitor.inc, which is used when implementing recursive AST visitors.

ClangAttrTemplateInstantiate

Purpose: Creates AttrTemplateInstantiate.inc, which implements the `instantiateTemplateAttribute` function, used when instantiating a template that requires an attribute to be cloned.

ClangAttrParsedAttrList

Purpose: Creates AttrParsedAttrList.inc, which is used to generate the `AttributeList::Kind` parsed attribute enumeration.

ClangAttrParsedAttrImpl

Purpose: Creates AttrParsedAttrImpl.inc, which is used by `AttributeList.cpp` to implement several functions on the `AttributeList` class. This functionality is implemented via the `AttrInfoMap ParsedAttrInfo` array, which contains one element per parsed attribute object.

ClangAttrParsedAttrKinds

Purpose: Creates AttrParsedAttrKinds.inc, which is used to implement the `AttributeList::getKind` function, mapping a string (and syntax) to a parsed attribute `AttributeList::Kind` enumeration.

ClangAttrDump

Purpose: Creates AttrDump.inc, which dumps information about an attribute. It is used to implement `ASTDumper::dumpAttr`.

ClangDiagsDefs

Generate Clang diagnostics definitions.

ClangDiagGroups

Generate Clang diagnostic groups.

ClangDiagsIndexName

Generate Clang diagnostic name index.

ClangCommentNodes

Generate Clang AST comment nodes.

ClangDeclNodes

Generate Clang AST declaration nodes.

ClangStmtNodes

Generate Clang AST statement nodes.

ClangSACheckers

Generate Clang Static Analyzer checkers.

ClangCommentHTMLTags

Generate efficient matchers for HTML tag names that are used in documentation comments.

ClangCommentHTMLTagsProperties

Generate efficient matchers for HTML tag properties.

ClangCommentHTMLNamedCharacterReferences

Generate function to translate named character references to UTF-8 sequences.

ClangCommentCommandInfo

Generate command properties for commands that are used in documentation comments.

ClangCommentCommandList

Generate list of commands that are used in documentation comments.

ArmNeon

Generate arm_neon.h for clang.

ArmNeonSema

Generate ARM NEON sema support for clang.

ArmNeonTest

Generate ARM NEON tests for clang.

AttrDocs

Purpose: Creates `AttributeReference.rst` from `AttrDocs.td`, and is used for documenting user-facing attributes.

How to write a back-end

TODO.

Until we get a step-by-step HowTo for writing TableGen backends, you can at least grab the boilerplate (build system, new files, etc.) from Clang's r173931.

TODO: How they work, how to write one. This section should not contain details about any particular backend, except maybe `-print-enums` as an example. This should highlight the APIs in `TableGen/Record.h`.

4.11.2 TableGen Language Reference

Warning: This document is extremely rough. If you find something lacking, please fix it, file a documentation bug, or ask about it on llvmdev.

Introduction

This document is meant to be a normative spec about the TableGen language in and of itself (i.e. how to understand a given construct in terms of how it affects the final set of records represented by the TableGen file). If you are unsure if this document is really what you are looking for, please read the *introduction to TableGen* first.

Notation

The lexical and syntax notation used here is intended to imitate Python's. In particular, for lexical definitions, the productions operate at the character level and there is no implied whitespace between elements. The syntax definitions operate at the token level, so there is implied whitespace between tokens.

Lexical Analysis

TableGen supports BCPL (`// ...`) and nestable C-style (`/* ... */`) comments.

The following is a listing of the basic punctuation tokens:

```
- + [ ] { } ( ) < > : ; .  = ? #
```

Numeric literals take one of the following forms:

```
TokInteger      ::=   DecimalInteger | HexInteger | BinInteger
DecimalInteger  ::=   ["+" | "-"] ("0"..."9")+
HexInteger      ::=   "0x" ("0"..."9" | "a"..."f" | "A"..."F")+
BinInteger      ::=   "0b" ("0" | "1")+
```

One aspect to note is that the `DecimalInteger` token *includes* the + or −, as opposed to having + and − be unary operators as most languages do.

Also note that `BinInteger` creates a value of type `bits<n>` (where `n` is the number of bits). This will implicitly convert to integers when needed.

TableGen has identifier-like tokens:

```
ualpha          ::=   "a"..."z" | "A"..."Z" | "_"
TokIdentifier   ::=   ("0"..."9")* ualpha (ualpha | "0"..."9")*
TokVarName      ::=   "$" ualpha (ualpha | "0"..."9")*
```

Note that unlike most languages, TableGen allows `TokIdentifier` to begin with a number. In case of ambiguity, a token will be interpreted as a numeric literal rather than an identifier.

TableGen also has two string-like literals:

```
TokString         ::=   '"' <non-'"' characters and C-like escapes> '"'
TokCodeFragment   ::=   "[{" <shortest text not containing "}]"> "}]"
```

`TokCodeFragment` is essentially a multiline string literal delimited by `[{` and `}]`.

Note: The current implementation accepts the following C-like escapes:

```
\\ \' \" \t \n
```

TableGen also has the following keywords:

```
bit   bits      class   code         dag
def   foreach   defm    field        in
int   let       list    multiclass   string
```

TableGen also has "bang operators" which have a wide variety of meanings:

```
BangOperator  ::=   one of
                  !eq !if !head !tail !con
                  !add !shl !sra !srl !and
                  !cast !empty !subst !foreach !listconcat !strconcat
```

Syntax

TableGen has an `include` mechanism. It does not play a role in the syntax per se, since it is lexically replaced with the contents of the included file.

```
IncludeDirective  ::=   "include" TokString
```

TableGen's top-level production consists of "objects".

```
TableGenFile  ::=   Object*
Object        ::=   Class | Def | Defm | Let | MultiClass | Foreach
```

classes

```
Class  ::=   "class" TokIdentifier [TemplateArgList] ObjectBody
```

A `class` declaration creates a record which other records can inherit from. A class can be parametrized by a list of "template arguments", whose values can be used in the class body.

A given class can only be defined once. A `class` declaration is considered to define the class if any of the following is true:

1. The `TemplateArgList` is present.
2. The `Body` in the `ObjectBody` is present and is not empty.
3. The `BaseClassList` in the `ObjectBody` is present.

You can declare an empty class by giving and empty `TemplateArgList` and an empty `ObjectBody`. This can serve as a restricted form of forward declaration: note that records deriving from the forward-declared class will inherit no fields from it since the record expansion is done when the record is parsed.

```
TemplateArgList  ::=   "<" Declaration ("," Declaration)* ">"
```

Declarations

The declaration syntax is pretty much what you would expect as a C++ programmer.

```
Declaration  ::=   Type TokIdentifier ["=" Value]
```

It assigns the value to the identifer.

Types

```
Type     ::=   "string" | "code" | "bit" | "int" | "dag"
             | "bits" "<" TokInteger ">"
             | "list" "<" Type ">"
             | ClassID
ClassID  ::=   TokIdentifier
```

Both `string` and `code` correspond to the string type; the difference is purely to indicate programmer intention.

The `ClassID` must identify a class that has been previously declared or defined.

Values

```
Value        ::=   SimpleValue ValueSuffix*
ValueSuffix  ::=   "{" RangeList "}"
                 | "[" RangeList "]"
                 | "."  TokIdentifier
RangeList    ::=   RangePiece ("," RangePiece)*
RangePiece   ::=   TokInteger
                 | TokInteger "-" TokInteger
                 | TokInteger TokInteger
```

The peculiar last form of `RangePiece` is due to the fact that the "-" is included in the `TokInteger`, hence `1-5` gets lexed as two consecutive `TokInteger`'s, with values `1` and `-5`, instead of "1", "-", and "5". The `RangeList` can be thought of as specifying "list slice" in some contexts.

`SimpleValue` has a number of forms:

```
SimpleValue  ::=   TokIdentifier
```

The value will be the variable referenced by the identifier. It can be one of:

- name of a `def`, such as the use of `Bar` in:

```
def Bar : SomeClass {
  int X = 5;
}

def Foo {
  SomeClass Baz = Bar;
}
```

- value local to a `def`, such as the use of `Bar` in:

```
def Foo {
  int Bar = 5;
  int Baz = Bar;
}
```

- a template arg of a `class`, such as the use of `Bar` in:

```
class Foo<int Bar> {
  int Baz = Bar;
}
```

- value local to a `multiclass`, such as the use of `Bar` in:

```
multiclass Foo {
  int Bar = 5;
  int Baz = Bar;
}
```

- a template arg to a `multiclass`, such as the use of `Bar` in:

```
multiclass Foo<int Bar> {
  int Baz = Bar;
}
```

```
SimpleValue  ::=   TokInteger
```

This represents the numeric value of the integer.

```
SimpleValue  ::=   TokString+
```

Multiple adjacent string literals are concatenated like in C/C++. The value is the concatenation of the strings.

```
SimpleValue  ::=   TokCodeFragment
```

The value is the string value of the code fragment.

```
SimpleValue  ::=   "?"
```

`?` represents an "unset" initializer.

```
SimpleValue  ::=   "{" ValueList "}"
ValueList    ::=   [ValueListNE]
ValueListNE  ::=   Value ("," Value)*
```

This represents a sequence of bits, as would be used to initialize a `bits<n>` field (where `n` is the number of bits).

```
SimpleValue  ::=   ClassID "<" ValueListNE ">"
```

This generates a new anonymous record definition (as would be created by an unnamed `def` inheriting from the given class with the given template arguments) and the value is the value of that record definition.

```
SimpleValue  ::=   "[" ValueList "]" ["<" Type ">"]
```

A list initializer. The optional `Type` can be used to indicate a specific element type, otherwise the element type will be deduced from the given values.

```
SimpleValue  ::=   "(" DagArg DagArgList ")"
DagArgList   ::=   DagArg ("," DagArg)*
DagArg       ::=   Value [":"  TokVarName] | TokVarName
```

The initial `DagArg` is called the "operator" of the dag.

```
SimpleValue  ::=   BangOperator ["<" Type ">"] "(" ValueListNE ")"
```

Bodies

```
ObjectBody       ::=   BaseClassList Body
BaseClassList    ::=   [":"  BaseClassListNE]
BaseClassListNE  ::=   SubClassRef (","  SubClassRef)*
SubClassRef      ::=   (ClassID | MultiClassID) ["<" ValueList ">"]
DefmID           ::=   TokIdentifier
```

The version with the `MultiClassID` is only valid in the `BaseClassList` of a `defm`. The `MultiClassID` should be the name of a `multiclass`.

It is after parsing the base class list that the "let stack" is applied.

```
Body      ::=   ";" | "{" BodyList "}"
BodyList  ::=   BodyItem*
BodyItem  ::=   Declaration ";"
              | "let" TokIdentifier [RangeList] "=" Value ";"
```

The `let` form allows overriding the value of an inherited field.

def

```
Def  ::=   "def" TokIdentifier ObjectBody
```

Defines a record whose name is given by the `TokIdentifier`. The fields of the record are inherited from the base classes and defined in the body.

Special handling occurs if this `def` appears inside a `multiclass` or a `foreach`.

defm

```
Defm  ::=   "defm" TokIdentifier ":"  BaseClassListNE ";"
```

Note that in the `BaseClassList`, all of the `multiclass`'s must precede any `class`'s that appear.

foreach

```
Foreach  ::=   "foreach" Declaration "in" "{" Object* "}"
             | "foreach" Declaration "in" Object
```

The value assigned to the variable in the declaration is iterated over and the object or object list is reevaluated with the variable set at each iterated value.

Top-Level `let`

```
Let      ::=   "let" LetList "in" "{" Object* "}"
             | "let" LetList "in" Object
LetList  ::=   LetItem ("," LetItem)*
LetItem  ::=   TokIdentifier [RangeList] "=" Value
```

This is effectively equivalent to `let` inside the body of a record except that it applies to multiple records at a time. The bindings are applied at the end of parsing the base classes of a record.

`multiclass`

```
MultiClass          ::=   "multiclass" TokIdentifier [TemplateArgList]
                          [":"  BaseMultiClassList] "{" MultiClassObject+ "}"
BaseMultiClassList  ::=   MultiClassID ("," MultiClassID)*
MultiClassID        ::=   TokIdentifier
MultiClassObject    ::=   Def | Defm | Let | Foreach
```

4.11.3 TableGen Language Introduction

Warning: This document is extremely rough. If you find something lacking, please fix it, file a documentation bug, or ask about it on llvmdev.

Introduction

This document is not meant to be a normative spec about the TableGen language in and of itself (i.e. how to understand a given construct in terms of how it affects the final set of records represented by the TableGen file). For the formal language specification, see *TableGen Language Reference*.

TableGen syntax

TableGen doesn't care about the meaning of data (that is up to the backend to define), but it does care about syntax, and it enforces a simple type system. This section describes the syntax and the constructs allowed in a TableGen file.

TableGen primitives

TableGen comments TableGen supports C++ style "`//`" comments, which run to the end of the line, and it also supports **nestable** "`/* */`" comments.

The TableGen type system TableGen files are strongly typed, in a simple (but complete) type-system. These types are used to perform automatic conversions, check for errors, and to help interface designers constrain the input that they allow. Every value definition is required to have an associated type.

TableGen supports a mixture of very low-level types (such as `bit`) and very high-level types (such as `dag`). This flexibility is what allows it to describe a wide range of information conveniently and compactly. The TableGen types are:

`bit` A 'bit' is a boolean value that can hold either 0 or 1.

`int` The 'int' type represents a simple 32-bit integer value, such as 5.

`string` The 'string' type represents an ordered sequence of characters of arbitrary length.

`bits<n>` A 'bits' type is an arbitrary, but fixed, size integer that is broken up into individual bits. This type is useful because it can handle some bits being defined while others are undefined.

`list<ty>` This type represents a list whose elements are some other type. The contained type is arbitrary: it can even be another list type.

Class type Specifying a class name in a type context means that the defined value must be a subclass of the specified class. This is useful in conjunction with the `list` type, for example, to constrain the elements of the list to a common base class (e.g., a `list<Register>` can only contain definitions derived from the "`Register`" class).

`dag` This type represents a nestable directed graph of elements.

To date, these types have been sufficient for describing things that TableGen has been used for, but it is straight-forward to extend this list if needed.

TableGen values and expressions TableGen allows for a pretty reasonable number of different expression forms when building up values. These forms allow the TableGen file to be written in a natural syntax and flavor for the application. The current expression forms supported include:

`?` uninitialized field

`0b1001011` binary integer value. Note that this is sized by the number of bits given and will not be silently extended/truncated.

`07654321` octal integer value (indicated by a leading 0)

`7` decimal integer value

`0x7F` hexadecimal integer value

`"foo"` string value

`[{ ... }]` usually called a "code fragment", but is just a multiline string literal

[X, Y, Z]<type> list value. <type> is the type of the list element and is usually optional. In rare cases, TableGen is unable to deduce the element type in which case the user must specify it explicitly.

{ a, b, 0b10 } initializer for a "bits<4>" value. 1-bit from "a", 1-bit from "b", 2-bits from 0b10.

value value reference

value{17} access to one bit of a value

value{15-17} access to multiple bits of a value

DEF reference to a record definition

CLASS<val list> reference to a new anonymous definition of CLASS with the specified template arguments.

X.Y reference to the subfield of a value

list[4-7,17,2-3] A slice of the 'list' list, including elements 4,5,6,7,17,2, and 3 from it. Elements may be included multiple times.

`foreach <var> = [ <list> ] in { <body> }`

foreach <var> = [<list>] in <def> Replicate <body> or <def>, replacing instances of <var> with each value in <list>. <var> is scoped at the level of the `foreach` loop and must not conflict with any other object introduced in <body> or <def>. Currently only `defs` are expanded within <body>.

`foreach <var> = 0-15 in ...`

foreach <var> = {0-15,32-47} in ... Loop over ranges of integers. The braces are required for multiple ranges.

(DEF a, b) a dag value. The first element is required to be a record definition, the remaining elements in the list may be arbitrary other values, including nested `'dag'` values.

!listconcat(a, b, ...) A list value that is the result of concatenating the 'a' and 'b' lists. The lists must have the same element type. More than two arguments are accepted with the result being the concatenation of all the lists given.

!strconcat(a, b, ...) A string value that is the result of concatenating the 'a' and 'b' strings. More than two arguments are accepted with the result being the concatenation of all the strings given.

str1#str2 "#" (paste) is a shorthand for !strconcat. It may concatenate things that are not quoted strings, in which case an implicit !cast<string> is done on the operand of the paste.

!cast<type>(a) A symbol of type *type* obtained by looking up the string 'a' in the symbol table. If the type of 'a' does not match *type*, TableGen aborts with an error. !cast<string> is a special case in that the argument must be an object defined by a 'def' construct.

!subst(a, b, c) If 'a' and 'b' are of string type or are symbol references, substitute 'b' for 'a' in 'c.' This operation is analogous to $(subst) in GNU make.

!foreach(a, b, c) For each member 'b' of dag or list 'a' apply operator 'c.' 'b' is a dummy variable that should be declared as a member variable of an instantiated class. This operation is analogous to $(foreach) in GNU make.

!head(a) The first element of list 'a.'

!tail(a) The 2nd-N elements of list 'a.'

!empty(a) An integer {0,1} indicating whether list 'a' is empty.

!if(a,b,c) 'b' if the result of 'int' or 'bit' operator 'a' is nonzero, 'c' otherwise.

!eq(a,b) 'bit 1' if string a is equal to string b, 0 otherwise. This only operates on string, int and bit objects. Use !cast<string> to compare other types of objects.

!shl(a,b) !srl(a,b) !sra(a,b) !add(a,b) !and(a,b) The usual binary and arithmetic operators.

Note that all of the values have rules specifying how they convert to values for different types. These rules allow you to assign a value like "7" to a "bits<4>" value, for example.

Classes and definitions

As mentioned in the *introduction*, classes and definitions (collectively known as 'records') in TableGen are the main high-level unit of information that TableGen collects. Records are defined with a def or class keyword, the record name, and an optional list of "template arguments". If the record has superclasses, they are specified as a comma separated list that starts with a colon character (":"). If value definitions or let expressions are needed for the class, they are enclosed in curly braces ("{}"); otherwise, the record ends with a semicolon.

Here is a simple TableGen file:

```
class C { bit V = 1; }
def X : C;
def Y : C {
  string Greeting = "hello";
}
```

This example defines two definitions, X and Y, both of which derive from the C class. Because of this, they both get the V bit value. The Y definition also gets the Greeting member as well.

In general, classes are useful for collecting together the commonality between a group of records and isolating it in a single place. Also, classes permit the specification of default values for their subclasses, allowing the subclasses to override them as they wish.

Value definitions Value definitions define named entries in records. A value must be defined before it can be referred to as the operand for another value definition or before the value is reset with a let expression. A value is defined by specifying a TableGen type and a name. If an initial value is available, it may be specified after the type with an equal sign. Value definitions require terminating semicolons.

'let' expressions A record-level let expression is used to change the value of a value definition in a record. This is primarily useful when a superclass defines a value that a derived class or definition wants to override. Let expressions consist of the 'let' keyword followed by a value name, an equal sign ("="), and a new value. For example, a new class could be added to the example above, redefining the V field for all of its subclasses:

```
class D : C { let V = 0; }
def Z : D;
```

In this case, the Z definition will have a zero value for its V value, despite the fact that it derives (indirectly) from the C class, because the D class overrode its value.

Class template arguments TableGen permits the definition of parameterized classes as well as normal concrete classes. Parameterized TableGen classes specify a list of variable bindings (which may optionally have defaults) that are bound when used. Here is a simple example:

```
class FPFormat<bits<3> val> {
  bits<3> Value = val;
}
def NotFP      : FPFormat<0>;
def ZeroArgFP  : FPFormat<1>;
def OneArgFP   : FPFormat<2>;
def OneArgFPRW : FPFormat<3>;
```

```
def TwoArgFP   : FPFormat<4>;
def CompareFP  : FPFormat<5>;
def CondMovFP  : FPFormat<6>;
def SpecialFP  : FPFormat<7>;
```

In this case, template arguments are used as a space efficient way to specify a list of "enumeration values", each with a "`Value`" field set to the specified integer.

The more esoteric forms of TableGen expressions are useful in conjunction with template arguments. As an example:

```
class ModRefVal<bits<2> val> {
  bits<2> Value = val;
}

def None   : ModRefVal<0>;
def Mod    : ModRefVal<1>;
def Ref    : ModRefVal<2>;
def ModRef : ModRefVal<3>;

class Value<ModRefVal MR> {
  // Decode some information into a more convenient format, while providing
  // a nice interface to the user of the "Value" class.
  bit isMod = MR.Value{0};
  bit isRef = MR.Value{1};

  // other stuff...
}

// Example uses
def bork : Value<Mod>;
def zork : Value<Ref>;
def hork : Value<ModRef>;
```

This is obviously a contrived example, but it shows how template arguments can be used to decouple the interface provided to the user of the class from the actual internal data representation expected by the class. In this case, running `llvm-tblgen` on the example prints the following definitions:

```
def bork {      // Value
  bit isMod = 1;
  bit isRef = 0;
}
def hork {      // Value
  bit isMod = 1;
  bit isRef = 1;
}
def zork {      // Value
  bit isMod = 0;
  bit isRef = 1;
}
```

This shows that TableGen was able to dig into the argument and extract a piece of information that was requested by the designer of the "Value" class. For more realistic examples, please see existing users of TableGen, such as the X86 backend.

Multiclass definitions and instances While classes with template arguments are a good way to factor commonality between two instances of a definition, multiclasses allow a convenient notation for defining multiple definitions at once (instances of implicitly constructed classes). For example, consider an 3-address instruction set whose instructions

come in two forms: "`reg = reg op reg`" and "`reg = reg op imm`" (e.g. SPARC). In this case, you'd like to specify in one place that this commonality exists, then in a separate place indicate what all the ops are.

Here is an example TableGen fragment that shows this idea:

```
def ops;
def GPR;
def Imm;
class inst<int opc, string asmstr, dag operandlist>;

multiclass ri_inst<int opc, string asmstr> {
  def _rr : inst<opc, !strconcat(asmstr, " $dst, $src1, $src2"),
                 (ops GPR:$dst, GPR:$src1, GPR:$src2)>;
  def _ri : inst<opc, !strconcat(asmstr, " $dst, $src1, $src2"),
                 (ops GPR:$dst, GPR:$src1, Imm:$src2)>;
}

// Instantiations of the ri_inst multiclass.
defm ADD : ri_inst<0b111, "add">;
defm SUB : ri_inst<0b101, "sub">;
defm MUL : ri_inst<0b100, "mul">;
...
```

The name of the resultant definitions has the multidef fragment names appended to them, so this defines `ADD_rr`, `ADD_ri`, `SUB_rr`, etc. A defm may inherit from multiple multiclasses, instantiating definitions from each multiclass. Using a multiclass this way is exactly equivalent to instantiating the classes multiple times yourself, e.g. by writing:

```
def ops;
def GPR;
def Imm;
class inst<int opc, string asmstr, dag operandlist>;

class rrinst<int opc, string asmstr>
  : inst<opc, !strconcat(asmstr, " $dst, $src1, $src2"),
         (ops GPR:$dst, GPR:$src1, GPR:$src2)>;

class riinst<int opc, string asmstr>
  : inst<opc, !strconcat(asmstr, " $dst, $src1, $src2"),
         (ops GPR:$dst, GPR:$src1, Imm:$src2)>;

// Instantiations of the ri_inst multiclass.
def ADD_rr : rrinst<0b111, "add">;
def ADD_ri : riinst<0b111, "add">;
def SUB_rr : rrinst<0b101, "sub">;
def SUB_ri : riinst<0b101, "sub">;
def MUL_rr : rrinst<0b100, "mul">;
def MUL_ri : riinst<0b100, "mul">;
...
```

A `defm` can also be used inside a multiclass providing several levels of multiclass instantiations.

```
class Instruction<bits<4> opc, string Name> {
  bits<4> opcode = opc;
  string name = Name;
}

multiclass basic_r<bits<4> opc> {
  def rr : Instruction<opc, "rr">;
  def rm : Instruction<opc, "rm">;
}
```

```
multiclass basic_s<bits<4> opc> {
  defm SS : basic_r<opc>;
  defm SD : basic_r<opc>;
  def X : Instruction<opc, "x">;
}

multiclass basic_p<bits<4> opc> {
  defm PS : basic_r<opc>;
  defm PD : basic_r<opc>;
  def Y : Instruction<opc, "y">;
}

defm ADD : basic_s<0xf>, basic_p<0xf>;
...

// Results
def ADDPDrm { ...
def ADDPDrr { ...
def ADDPSrm { ...
def ADDPSrr { ...
def ADDSDrm { ...
def ADDSDrr { ...
def ADDY { ...
def ADDX { ...
```

`defm` declarations can inherit from classes too, the rule to follow is that the class list must start after the last multiclass, and there must be at least one multiclass before them.

```
class XD { bits<4> Prefix = 11; }
class XS { bits<4> Prefix = 12; }

class I<bits<4> op> {
  bits<4> opcode = op;
}

multiclass R {
  def rr : I<4>;
  def rm : I<2>;
}

multiclass Y {
  defm SS : R, XD;
  defm SD : R, XS;
}

defm Instr : Y;

// Results
def InstrSDrm {
  bits<4> opcode = { 0, 0, 1, 0 };
  bits<4> Prefix = { 1, 1, 0, 0 };
}
...
def InstrSSrr {
  bits<4> opcode = { 0, 1, 0, 0 };
  bits<4> Prefix = { 1, 0, 1, 1 };
}
```

File scope entities

File inclusion TableGen supports the ‘`include`‘ token, which textually substitutes the specified file in place of the include directive. The filename should be specified as a double quoted string immediately after the ‘`include`‘ keyword. Example:

```
include "foo.td"
```

‘let’ expressions “Let” expressions at file scope are similar to “let” expressions within a record, except they can specify a value binding for multiple records at a time, and may be useful in certain other cases. File-scope let expressions are really just another way that TableGen allows the end-user to factor out commonality from the records.

File-scope “let” expressions take a comma-separated list of bindings to apply, and one or more records to bind the values in. Here are some examples:

```
let isTerminator = 1, isReturn = 1, isBarrier = 1, hasCtrlDep = 1 in
  def RET : I<0xC3, RawFrm, (outs), (ins), "ret", [(X86retflag 0)]>;

let isCall = 1 in
  // All calls clobber the non-callee saved registers...
  let Defs = [EAX, ECX, EDX, FP0, FP1, FP2, FP3, FP4, FP5, FP6, ST0,
              MM0, MM1, MM2, MM3, MM4, MM5, MM6, MM7,
              XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6, XMM7, EFLAGS] in {
    def CALLpcrel32 : Ii32<0xE8, RawFrm, (outs), (ins i32imm:$dst,variable_ops),
                           "call\t${dst:call}", []>;
    def CALL32r     : I<0xFF, MRM2r, (outs), (ins GR32:$dst, variable_ops),
                        "call\t{*}$dst", [(X86call GR32:$dst)]>;
    def CALL32m     : I<0xFF, MRM2m, (outs), (ins i32mem:$dst, variable_ops),
                        "call\t{*}$dst", []>;
  }
```

File-scope “let” expressions are often useful when a couple of definitions need to be added to several records, and the records do not otherwise need to be opened, as in the case with the `CALL*` instructions above.

It’s also possible to use “let” expressions inside multiclasses, providing more ways to factor out commonality from the records, specially if using several levels of multiclass instantiations. This also avoids the need of using “let” expressions within subsequent records inside a multiclass.

```
multiclass basic_r<bits<4> opc> {
  let Predicates = [HasSSE2] in {
    def rr : Instruction<opc, "rr">;
    def rm : Instruction<opc, "rm">;
  }
  let Predicates = [HasSSE3] in
    def rx : Instruction<opc, "rx">;
}

multiclass basic_ss<bits<4> opc> {
  let IsDouble = 0 in
    defm SS : basic_r<opc>;

  let IsDouble = 1 in
    defm SD : basic_r<opc>;
}

defm ADD : basic_ss<0xf>;
```

Looping TableGen supports the '`foreach`' block, which textually replicates the loop body, substituting iterator values for iterator references in the body. Example:

```
foreach i = [0, 1, 2, 3] in {
  def R#i : Register<...>;
  def F#i : Register<...>;
}
```

This will create objects `R0`, `R1`, `R2` and `R3`. `foreach` blocks may be nested. If there is only one item in the body the braces may be elided:

```
foreach i = [0, 1, 2, 3] in
  def R#i : Register<...>;
```

Code Generator backend info

Expressions used by code generator to describe instructions and isel patterns:

`(implicit a)` an implicitly defined physical register. This tells the dag instruction selection emitter the input pattern's extra definitions matches implicit physical register definitions.

4.11.4 TableGen Deficiencies

Introduction

Despite being very generic, TableGen has some deficiencies that have been pointed out numerous times. The common theme is that, while TableGen allows you to build Domain-Specific-Languages, the final languages that you create lack the power of other DSLs, which in turn increase considerably the size and complexity of TableGen files.

At the same time, TableGen allows you to create virtually any meaning of the basic concepts via custom-made back-ends, which can pervert the original design and make it very hard for newcomers to understand it.

There are some in favour of extending the semantics even more, but making sure back-ends adhere to strict rules. Others suggesting we should move to more powerful DSLs designed with specific purposes, or even re-using existing DSLs.

Known Problems

TODO: Add here frequently asked questions about why TableGen doesn't do what you want, how it might, and how we could extend/restrict it to be more use friendly.

4.11.5 Introduction

TableGen's purpose is to help a human develop and maintain records of domain-specific information. Because there may be a large number of these records, it is specifically designed to allow writing flexible descriptions and for common features of these records to be factored out. This reduces the amount of duplication in the description, reduces the chance of error, and makes it easier to structure domain specific information.

The core part of TableGen parses a file, instantiates the declarations, and hands the result off to a domain-specific backend for processing.

The current major users of TableGen are *The LLVM Target-Independent Code Generator* and the Clang diagnostics and attributes.

Note that if you work on TableGen much, and use emacs or vim, that you can find an emacs "TableGen mode" and a vim language file in the `llvm/utils/emacs` and `llvm/utils/vim` directories of your LLVM distribution, respectively.

4.11.6 The TableGen program

TableGen files are interpreted by the TableGen program: *llvm-tblgen* available on your build directory under *bin*. It is not installed in the system (or where your sysroot is set to), since it has no use beyond LLVM's build process.

Running TableGen

TableGen runs just like any other LLVM tool. The first (optional) argument specifies the file to read. If a filename is not specified, `llvm-tblgen` reads from standard input.

To be useful, one of the backends must be used. These backends are selectable on the command line (type '`llvm-tblgen -help`' for a list). For example, to get a list of all of the definitions that subclass a particular type (which can be useful for building up an enum list of these records), use the `-print-enums` option:

```
$ llvm-tblgen X86.td -print-enums -class=Register
AH, AL, AX, BH, BL, BP, BPL, BX, CH, CL, CX, DH, DI, DIL, DL, DX, EAX, EBP, EBX,
ECX, EDI, EDX, EFLAGS, EIP, ESI, ESP, FP0, FP1, FP2, FP3, FP4, FP5, FP6, IP,
MM0, MM1, MM2, MM3, MM4, MM5, MM6, MM7, R10, R10B, R10D, R10W, R11, R11B, R11D,
R11W, R12, R12B, R12D, R12W, R13, R13B, R13D, R13W, R14, R14B, R14D, R14W, R15,
R15B, R15D, R15W, R8, R8B, R8D, R8W, R9, R9B, R9D, R9W, RAX, RBP, RBX, RCX, RDI,
RDX, RIP, RSI, RSP, SI, SIL, SP, SPL, ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7,
XMM0, XMM1, XMM10, XMM11, XMM12, XMM13, XMM14, XMM15, XMM2, XMM3, XMM4, XMM5,
XMM6, XMM7, XMM8, XMM9,

$ llvm-tblgen X86.td -print-enums -class=Instruction
ABS_F, ABS_Fp32, ABS_Fp64, ABS_Fp80, ADC32mi, ADC32mi8, ADC32mr, ADC32ri,
ADC32ri8, ADC32rm, ADC32rr, ADC64mi32, ADC64mi8, ADC64mr, ADC64ri32, ADC64ri8,
ADC64rm, ADC64rr, ADD16mi, ADD16mi8, ADD16mr, ADD16ri, ADD16ri8, ADD16rm,
ADD16rr, ADD32mi, ADD32mi8, ADD32mr, ADD32ri, ADD32ri8, ADD32rm, ADD32rr,
ADD64mi32, ADD64mi8, ADD64mr, ADD64ri32, ...
```

The default backend prints out all of the records.

If you plan to use TableGen, you will most likely have to write a backend that extracts the information specific to what you need and formats it in the appropriate way.

Example

With no other arguments, *llvm-tblgen* parses the specified file and prints out all of the classes, then all of the definitions. This is a good way to see what the various definitions expand to fully. Running this on the `X86.td` file prints this (at the time of this writing):

```
...
def ADD32rr {   // Instruction X86Inst I
  string Namespace = "X86";
  dag OutOperandList = (outs GR32:$dst);
```

```
  dag InOperandList = (ins GR32:$src1, GR32:$src2);
  string AsmString = "add{l}\t{$src2, $dst|$dst, $src2}";
  list<dag> Pattern = [(set GR32:$dst, (add GR32:$src1, GR32:$src2))];
  list<Register> Uses = [];
  list<Register> Defs = [EFLAGS];
  list<Predicate> Predicates = [];
  int CodeSize = 3;
  int AddedComplexity = 0;
  bit isReturn = 0;
  bit isBranch = 0;
  bit isIndirectBranch = 0;
  bit isBarrier = 0;
  bit isCall = 0;
  bit canFoldAsLoad = 0;
  bit mayLoad = 0;
  bit mayStore = 0;
  bit isImplicitDef = 0;
  bit isConvertibleToThreeAddress = 1;
  bit isCommutable = 1;
  bit isTerminator = 0;
  bit isReMaterializable = 0;
  bit isPredicable = 0;
  bit hasDelaySlot = 0;
  bit usesCustomInserter = 0;
  bit hasCtrlDep = 0;
  bit isNotDuplicable = 0;
  bit hasSideEffects = 0;
  bit neverHasSideEffects = 0;
  InstrItinClass Itinerary = NoItinerary;
  string Constraints = "";
  string DisableEncoding = "";
  bits<8> Opcode = { 0, 0, 0, 0, 0, 0, 0, 1 };
  Format Form = MRMDestReg;
  bits<6> FormBits = { 0, 0, 0, 0, 1, 1 };
  ImmType ImmT = NoImm;
  bits<3> ImmTypeBits = { 0, 0, 0 };
  bit hasOpSizePrefix = 0;
  bit hasAdSizePrefix = 0;
  bits<4> Prefix = { 0, 0, 0, 0 };
  bit hasREX_WPrefix = 0;
  FPFormat FPForm = ?;
  bits<3> FPFormBits = { 0, 0, 0 };
}
...
```

This definition corresponds to the 32-bit register-register `add` instruction of the x86 architecture. `def ADD32rr` defines a record named `ADD32rr`, and the comment at the end of the line indicates the superclasses of the definition. The body of the record contains all of the data that TableGen assembled for the record, indicating that the instruction is part of the "X86" namespace, the pattern indicating how the instruction is selected by the code generator, that it is a two-address instruction, has a particular encoding, etc. The contents and semantics of the information in the record are specific to the needs of the X86 backend, and are only shown as an example.

As you can see, a lot of information is needed for every instruction supported by the code generator, and specifying it all manually would be unmaintainable, prone to bugs, and tiring to do in the first place. Because we are using TableGen, all of the information was derived from the following definition:

```
let Defs = [EFLAGS],
    isCommutable = 1,                   // X = ADD Y,Z --> X = ADD Z,Y
```

```
    isConvertibleToThreeAddress = 1 in // Can transform into LEA.
def ADD32rr  : I<0x01, MRMDestReg, (outs GR32:$dst),
                                   (ins GR32:$src1, GR32:$src2),
                 "add{l}\t{$src2, $dst|$dst, $src2}",
                 [(set GR32:$dst, (add GR32:$src1, GR32:$src2))]>;
```

This definition makes use of the custom class `I` (extended from the custom class `X86Inst`), which is defined in the X86-specific TableGen file, to factor out the common features that instructions of its class share. A key feature of TableGen is that it allows the end-user to define the abstractions they prefer to use when describing their information.

Each `def` record has a special entry called "NAME". This is the name of the record ("`ADD32rr`" above). In the general case `def` names can be formed from various kinds of string processing expressions and `NAME` resolves to the final value obtained after resolving all of those expressions. The user may refer to `NAME` anywhere she desires to use the ultimate name of the `def`. `NAME` should not be defined anywhere else in user code to avoid conflicts.

4.11.7 Syntax

TableGen has a syntax that is loosely based on C++ templates, with built-in types and specification. In addition, TableGen's syntax introduces some automation concepts like multiclass, foreach, let, etc.

Basic concepts

TableGen files consist of two key parts: 'classes' and 'definitions', both of which are considered 'records'.

TableGen records have a unique name, a list of values, and a list of superclasses. The list of values is the main data that TableGen builds for each record; it is this that holds the domain specific information for the application. The interpretation of this data is left to a specific backend, but the structure and format rules are taken care of and are fixed by TableGen.

TableGen definitions are the concrete form of 'records'. These generally do not have any undefined values, and are marked with the '`def`' keyword.

```
def FeatureFPARMv8 : SubtargetFeature<"fp-armv8", "HasFPARMv8", "true",
                                      "Enable ARMv8 FP">;
```

In this example, FeatureFPARMv8 is `SubtargetFeature` record initialised with some values. The names of the classes are defined via the keyword *class* either on the same file or some other included. Most target TableGen files include the generic ones in `include/llvm/Target`.

TableGen classes are abstract records that are used to build and describe other records. These classes allow the end-user to build abstractions for either the domain they are targeting (such as "Register", "RegisterClass", and "Instruction" in the LLVM code generator) or for the implementor to help factor out common properties of records (such as "FPInst", which is used to represent floating point instructions in the X86 backend). TableGen keeps track of all of the classes that are used to build up a definition, so the backend can find all definitions of a particular class, such as "Instruction".

```
class ProcNoItin<string Name, list<SubtargetFeature> Features>
      : Processor<Name, NoItineraries, Features>;
```

Here, the class ProcNoItin, receiving parameters *Name* of type *string* and a list of target features is specializing the class Processor by passing the arguments down as well as hard-coding NoItineraries.

TableGen multiclasses are groups of abstract records that are instantiated all at once. Each instantiation can result in multiple TableGen definitions. If a multiclass inherits from another multiclass, the definitions in the sub-multiclass become part of the current multiclass, as if they were declared in the current multiclass.

```
multiclass ro_signed_pats<string T, string Rm, dag Base, dag Offset, dag Extend,
                          dag address, ValueType sty> {
def : Pat<(i32 (!cast<SDNode>("sextload" # sty) address)),
          (!cast<Instruction>("LDRS" # T # "w_" # Rm # "_RegOffset")
            Base, Offset, Extend)>;

def : Pat<(i64 (!cast<SDNode>("sextload" # sty) address)),
          (!cast<Instruction>("LDRS" # T # "x_" # Rm # "_RegOffset")
            Base, Offset, Extend)>;
}

defm : ro_signed_pats<"B", Rm, Base, Offset, Extend,
                      !foreach(decls.pattern, address,
                               !subst(SHIFT, imm_eq0, decls.pattern)),
                      i8>;
```

See the *TableGen Language Introduction* for more generic information on the usage of the language, and the *TableGen Language Reference* for more in-depth description of the formal language specification.

4.11.8 TableGen backends

TableGen files have no real meaning without a back-end. The default operation of running `llvm-tblgen` is to print the information in a textual format, but that's only useful for debugging of the TableGen files themselves. The power in TableGen is, however, to interpret the source files into an internal representation that can be generated into anything you want.

Current usage of TableGen is to create huge include files with tables that you can either include directly (if the output is in the language you're coding), or be used in pre-processing via macros surrounding the include of the file.

Direct output can be used if the back-end already prints a table in C format or if the output is just a list of strings (for error and warning messages). Pre-processed output should be used if the same information needs to be used in different contexts (like Instruction names), so your back-end should print a meta-information list that can be shaped into different compile-time formats.

See the TableGen BackEnds for more information.

4.11.9 TableGen Deficiencies

Despite being very generic, TableGen has some deficiencies that have been pointed out numerous times. The common theme is that, while TableGen allows you to build Domain-Specific-Languages, the final languages that you create lack the power of other DSLs, which in turn increase considerably the size and complexity of TableGen files.

At the same time, TableGen allows you to create virtually any meaning of the basic concepts via custom-made back-ends, which can pervert the original design and make it very hard for newcomers to understand the evil TableGen file.

There are some in favour of extending the semantics even more, but making sure back-ends adhere to strict rules. Others are suggesting we should move to less, more powerful DSLs designed with specific purposes, or even re-using existing DSLs.

Either way, this is a discussion that will likely span across several years, if not decades. You can read more in the TableGen Deficiencies document.

4.12 Debugging JIT-ed Code With GDB

4.12.1 Background

Without special runtime support, debugging dynamically generated code with GDB (as well as most debuggers) can be quite painful. Debuggers generally read debug information from the object file of the code, but for JITed code, there is no such file to look for.

In order to communicate the necessary debug info to GDB, an interface for registering JITed code with debuggers has been designed and implemented for GDB and LLVM MCJIT. At a high level, whenever MCJIT generates new machine code, it does so in an in-memory object file that contains the debug information in DWARF format. MCJIT then adds this in-memory object file to a global list of dynamically generated object files and calls a special function (`__jit_debug_register_code`) marked noinline that GDB knows about. When GDB attaches to a process, it puts a breakpoint in this function and loads all of the object files in the global list. When MCJIT calls the registration function, GDB catches the breakpoint signal, loads the new object file from the inferior's memory, and resumes the execution. In this way, GDB can get the necessary debug information.

4.12.2 GDB Version

In order to debug code JIT-ed by LLVM, you need GDB 7.0 or newer, which is available on most modern distributions of Linux. The version of GDB that Apple ships with Xcode has been frozen at 6.3 for a while. LLDB may be a better option for debugging JIT-ed code on Mac OS X.

4.12.3 Debugging MCJIT-ed code

The emerging MCJIT component of LLVM allows full debugging of JIT-ed code with GDB. This is due to MCJIT's ability to use the MC emitter to provide full DWARF debugging information to GDB.

Note that lli has to be passed the `-use-mcjit` flag to JIT the code with MCJIT instead of the old JIT.

Example

Consider the following C code (with line numbers added to make the example easier to follow):

```
1   int compute_factorial(int n)
2   {
3       if (n <= 1)
4           return 1;
5
6       int f = n;
7       while (--n > 1)
8           f *= n;
9       return f;
10  }
11
12
13  int main(int argc, char** argv)
14  {
15      if (argc < 2)
16          return -1;
17      char firstletter = argv[1][0];
18      int result = compute_factorial(firstletter - '0');
19
```

```
20      // Returned result is clipped at 255...
21      return result;
22  }
```

Here is a sample command line session that shows how to build and run this code via `lli` inside GDB:

```
$ $BINPATH/clang -cc1 -O0 -g -emit-llvm showdebug.c
$ gdb --quiet --args $BINPATH/lli -use-mcjit showdebug.ll 5
Reading symbols from $BINPATH/lli...done.
(gdb) b showdebug.c:6
No source file named showdebug.c.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (showdebug.c:6) pending.
(gdb) r
Starting program: $BINPATH/lli -use-mcjit showdebug.ll 5
[Thread debugging using libthread_db enabled]

Breakpoint 1, compute_factorial (n=5) at showdebug.c:6
6         int f = n;
(gdb) p n
$1 = 5
(gdb) p f
$2 = 0
(gdb) n
7         while (--n > 1)
(gdb) p f
$3 = 5
(gdb) b showdebug.c:9
Breakpoint 2 at 0x7ffff7ed404c: file showdebug.c, line 9.
(gdb) c
Continuing.

Breakpoint 2, compute_factorial (n=1) at showdebug.c:9
9         return f;
(gdb) p f
$4 = 120
(gdb) bt
#0  compute_factorial (n=1) at showdebug.c:9
#1  0x00007ffff7ed40a9 in main (argc=2, argv=0x16677e0) at showdebug.c:18
#2  0x3500000001652748 in ?? ()
#3  0x00000000016677e0 in ?? ()
#4  0x0000000000000002 in ?? ()
#5  0x0000000000d953b3 in llvm::MCJIT::runFunction (this=0x16151f0, F=0x1603020, ArgValues=...) at /
#6  0x0000000000dc8872 in llvm::ExecutionEngine::runFunctionAsMain (this=0x16151f0, Fn=0x1603020, arg
    at /home/ebenders_test/llvm_svn_rw/lib/ExecutionEngine/ExecutionEngine.cpp:397
#7  0x000000000059c583 in main (argc=4, argv=0x7fffffffe018, envp=0x7fffffffe040) at /home/ebenders_t
(gdb) finish
Run till exit from #0  compute_factorial (n=1) at showdebug.c:9
0x00007ffff7ed40a9 in main (argc=2, argv=0x16677e0) at showdebug.c:18
18        int result = compute_factorial(firstletter - '0');
Value returned is $5 = 120
(gdb) p result
$6 = 23406408
(gdb) n
21        return result;
(gdb) p result
$7 = 120
(gdb) c
```

```
Continuing.

Program exited with code 0170.
(gdb)
```

4.13 The LLVM gold plugin

4.13.1 Introduction

Building with link time optimization requires cooperation from the system linker. LTO support on Linux systems requires that you use the gold linker which supports LTO via plugins. This is the same mechanism used by the GCC LTO project.

The LLVM gold plugin implements the gold plugin interface on top of *libLTO*. The same plugin can also be used by other tools such as `ar` and `nm`.

4.13.2 How to build it

You need to have gold with plugin support and build the LLVMgold plugin. Check whether you have gold running `/usr/bin/ld -v`. It will report "GNU gold" or else "GNU ld" if not. If you have gold, check for plugin support by running `/usr/bin/ld -plugin`. If it complains "missing argument" then you have plugin support. If not, such as an "unknown option" error then you will either need to build gold or install a version with plugin support.

- Download, configure and build gold with plugin support:

  ```
  $ git clone --depth 1 git://sourceware.org/git/binutils-gdb.git binutils
  $ mkdir build
  $ cd build
  $ ../binutils/configure --enable-gold --enable-plugins --disable-werror
  $ make all-gold
  ```

  That should leave you with `build/gold/ld-new` which supports the `-plugin` option. Running `make` will additionally build `build/binutils/ar` and `nm-new` binaries supporting plugins.

- Build the LLVMgold plugin: Configure LLVM with `--with-binutils-include=/path/to/binutils/include` and run `make`.

4.13.3 Usage

The linker takes a `-plugin` option that points to the path of the plugin `.so` file. To find out what link command `gcc` would run in a given situation, run `gcc -v [...]` and look for the line where it runs `collect2`. Replace that with `ld-new -plugin /path/to/LLVMgold.so` to test it out. Once you're ready to switch to using gold, backup your existing `/usr/bin/ld` then replace it with `ld-new`.

You should produce bitcode files from `clang` with the option `-flto`. This flag will also cause `clang` to look for the gold plugin in the `lib` directory under its prefix and pass the `-plugin` option to `ld`. It will not look for an alternate linker, which is why you need gold to be the installed system linker in your path.

`ar` and `nm` also accept the `-plugin` option and it's possible to to install `LLVMgold.so` to `/usr/lib/bfd-plugins` for a seamless setup. If you built your own gold, be sure to install the `ar` and `nm-new` you built to `/usr/bin`.

Example of link time optimization

The following example shows a worked example of the gold plugin mixing LLVM bitcode and native code.

```
--- a.c ---
#include <stdio.h>

extern void foo1(void);
extern void foo4(void);

void foo2(void) {
  printf("Foo2\n");
}

void foo3(void) {
  foo4();
}

int main(void) {
  foo1();
}

--- b.c ---
#include <stdio.h>

extern void foo2(void);

void foo1(void) {
  foo2();
}

void foo4(void) {
  printf("Foo4");
}

--- command lines ---
$ clang -flto a.c -c -o a.o      # <-- a.o is LLVM bitcode file
$ ar q a.a a.o                   # <-- a.a is an archive with LLVM bitcode
$ clang b.c -c -o b.o            # <-- b.o is native object file
$ clang -flto a.a b.o -o main    # <-- link with LLVMgold plugin
```

Gold informs the plugin that foo3 is never referenced outside the IR, leading LLVM to delete that function. However, unlike in the *libLTO example* gold does not currently eliminate foo4.

4.13.4 Quickstart for using LTO with autotooled projects

Once your system `ld`, `ar`, and `nm` all support LLVM bitcode, everything is in place for an easy to use LTO build of autotooled projects:

- Follow the instructions *on how to build LLVMgold.so*.
- Install the newly built binutils to `$PREFIX`
- Copy `Release/lib/LLVMgold.so` to `$PREFIX/lib/bfd-plugins/`
- Set environment variables (`$PREFIX` is where you installed clang and binutils):

```
export CC="$PREFIX/bin/clang -flto"
export CXX="$PREFIX/bin/clang++ -flto"
```

```
export AR="$PREFIX/bin/ar"
export NM="$PREFIX/bin/nm"
export RANLIB=/bin/true #ranlib is not needed, and doesn't support .bc files in .a
```

- Or you can just set your path:

```
export PATH="$PREFIX/bin:$PATH"
export CC="clang -flto"
export CXX="clang++ -flto"
export RANLIB=/bin/true
```

- Configure and build the project as usual:

```
% ./configure && make && make check
```

The environment variable settings may work for non-autotooled projects too, but you may need to set the `LD` environment variable as well.

4.13.5 Licensing

Gold is licensed under the GPLv3. LLVMgold uses the interface file `plugin-api.h` from gold which means that the resulting `LLVMgold.so` binary is also GPLv3. This can still be used to link non-GPLv3 programs just as much as gold could without the plugin.

4.14 LLVM's Optional Rich Disassembly Output

4.14.1 Introduction

LLVM's default disassembly output is raw text. To allow consumers more ability to introspect the instructions' textual representation or to reformat for a more user friendly display there is an optional rich disassembly output.

This optional output is sufficient to reference into individual portions of the instruction text. This is intended for clients like disassemblers, list file generators, and pretty-printers, which need more than the raw instructions and the ability to print them.

To provide this functionality the assembly text is marked up with annotations. The markup is simple enough in syntax to be robust even in the case of version mismatches between consumers and producers. That is, the syntax generally does not carry semantics beyond "this text has an annotation," so consumers can simply ignore annotations they do not understand or do not care about.

After calling `LLVMCreateDisasm()` to create a disassembler context the optional output is enable with this call:

```
LLVMSetDisasmOptions(DC, LLVMDisassembler_Option_UseMarkup);
```

Then subsequent calls to `LLVMDisasmInstruction()` will return output strings with the marked up annotations.

4.14.2 Instruction Annotations

Contextual markups

Annoated assembly display will supply contextual markup to help clients more efficiently implement things like pretty printers. Most markup will be target independent, so clients can effectively provide good display without any target specific knowledge.

Annotated assembly goes through the normal instruction printer, but optionally includes contextual tags on portions of the instruction string. An annotation is any '<' '>' delimited section of text(1).

```
annotation: '<' tag-name tag-modifier-list ':' annotated-text '>'
tag-name: identifier
tag-modifier-list: comma delimited identifier list
```

The tag-name is an identifier which gives the type of the annotation. For the first pass, this will be very simple, with memory references, registers, and immediates having the tag names "mem", "reg", and "imm", respectively.

The tag-modifier-list is typically additional target-specific context, such as register class.

Clients should accept and ignore any tag-names or tag-modifiers they do not understand, allowing the annotations to grow in richness without breaking older clients.

For example, a possible annotation of an ARM load of a stack-relative location might be annotated as:

```
ldr <reg gpr:r0>, <mem regoffset:[<reg gpr:sp>, <imm:#4>]>
```

1: For assembly dialects in which '<' and/or '>' are legal tokens, a literal token is escaped by following immediately with a repeat of the character. For example, a literal '<' character is output as '<<' in an annotated assembly string.

C API Details

The intended consumers of this information use the C API, therefore the new C API function for the disassembler will be added to provide an option to produce disassembled instructions with annotations, `LLVMSetDisasmOptions()` and the `LLVMDisassembler_Option_UseMarkup` option (see above).

4.15 System Library

4.15.1 Abstract

This document provides some details on LLVM's System Library, located in the source at `lib/System` and `include/llvm/System`. The library's purpose is to shield LLVM from the differences between operating systems for the few services LLVM needs from the operating system. Much of LLVM is written using portability features of standard C++. However, in a few areas, system dependent facilities are needed and the System Library is the wrapper around those system calls.

By centralizing LLVM's use of operating system interfaces, we make it possible for the LLVM tool chain and runtime libraries to be more easily ported to new platforms since (theoretically) only `lib/System` needs to be ported. This library also unclutters the rest of LLVM from #ifdef use and special cases for specific operating systems. Such uses are replaced with simple calls to the interfaces provided in `include/llvm/System`.

Note that the System Library is not intended to be a complete operating system wrapper (such as the Adaptive Communications Environment (ACE) or Apache Portable Runtime (APR)), but only provides the functionality necessary to support LLVM.

The System Library was written by Reid Spencer who formulated the design based on similar work originating from the eXtensible Programming System (XPS). Several people helped with the effort; especially, Jeff Cohen and Henrik Bach on the Win32 port.

4.15.2 Keeping LLVM Portable

In order to keep LLVM portable, LLVM developers should adhere to a set of portability rules associated with the System Library. Adherence to these rules should help the System Library achieve its goal of shielding LLVM from the variations in operating system interfaces and doing so efficiently. The following sections define the rules needed to fulfill this objective.

Don't Include System Headers

Except in `lib/System`, no LLVM source code should directly `#include` a system header. Care has been taken to remove all such `#includes` from LLVM while `lib/System` was being developed. Specifically this means that header files like "`unistd.h`", "`windows.h`", "`stdio.h`", and "`string.h`" are forbidden to be included by LLVM source code outside the implementation of `lib/System`.

To obtain system-dependent functionality, existing interfaces to the system found in `include/llvm/System` should be used. If an appropriate interface is not available, it should be added to `include/llvm/System` and implemented in `lib/System` for all supported platforms.

Don't Expose System Headers

The System Library must shield LLVM from **all** system headers. To obtain system level functionality, LLVM source must `#include "llvm/System/Thing.h"` and nothing else. This means that `Thing.h` cannot expose any system header files. This protects LLVM from accidentally using system specific functionality and only allows it via the `lib/System` interface.

Use Standard C Headers

The **standard** C headers (the ones beginning with "c") are allowed to be exposed through the `lib/System` interface. These headers and the things they declare are considered to be platform agnostic. LLVM source files may include them directly or obtain their inclusion through `lib/System` interfaces.

Use Standard C++ Headers

The **standard** C++ headers from the standard C++ library and standard template library may be exposed through the `lib/System` interface. These headers and the things they declare are considered to be platform agnostic. LLVM source files may include them or obtain their inclusion through `lib/System` interfaces.

High Level Interface

The entry points specified in the interface of `lib/System` must be aimed at completing some reasonably high level task needed by LLVM. We do not want to simply wrap each operating system call. It would be preferable to wrap several operating system calls that are always used in conjunction with one another by LLVM.

For example, consider what is needed to execute a program, wait for it to complete, and return its result code. On Unix, this involves the following operating system calls: `getenv`, `fork`, `execve`, and `wait`. The correct thing for `lib/System` to provide is a function, say `ExecuteProgramAndWait`, that implements the functionality completely. what we don't want is wrappers for the operating system calls involved.

There must **not** be a one-to-one relationship between operating system calls and the System library's interface. Any such interface function will be suspicious.

No Unused Functionality

There must be no functionality specified in the interface of `lib/System` that isn't actually used by LLVM. We're not writing a general purpose operating system wrapper here, just enough to satisfy LLVM's needs. And, LLVM doesn't need much. This design goal aims to keep the `lib/System` interface small and understandable which should foster its actual use and adoption.

No Duplicate Implementations

The implementation of a function for a given platform must be written exactly once. This implies that it must be possible to apply a function's implementation to multiple operating systems if those operating systems can share the same implementation. This rule applies to the set of operating systems supported for a given class of operating system (e.g. Unix, Win32).

No Virtual Methods

The System Library interfaces can be called quite frequently by LLVM. In order to make those calls as efficient as possible, we discourage the use of virtual methods. There is no need to use inheritance for implementation differences, it just adds complexity. The `#include` mechanism works just fine.

No Exposed Functions

Any functions defined by system libraries (i.e. not defined by `lib/System`) must not be exposed through the `lib/System` interface, even if the header file for that function is not exposed. This prevents inadvertent use of system specific functionality.

For example, the `stat` system call is notorious for having variations in the data it provides. `lib/System` must not declare `stat` nor allow it to be declared. Instead it should provide its own interface to discovering information about files and directories. Those interfaces may be implemented in terms of `stat` but that is strictly an implementation detail. The interface provided by the System Library must be implemented on all platforms (even those without `stat`).

No Exposed Data

Any data defined by system libraries (i.e. not defined by `lib/System`) must not be exposed through the `lib/System` interface, even if the header file for that function is not exposed. As with functions, this prevents inadvertent use of data that might not exist on all platforms.

Minimize Soft Errors

Operating system interfaces will generally provide error results for every little thing that could go wrong. In almost all cases, you can divide these error results into two groups: normal/good/soft and abnormal/bad/hard. That is, some of the errors are simply information like "file not found", "insufficient privileges", etc. while other errors are much harder like "out of space", "bad disk sector", or "system call interrupted". We'll call the first group "*soft*" errors and the second group "*hard*" errors.

`lib/System` must always attempt to minimize soft errors. This is a design requirement because the minimization of soft errors can affect the granularity and the nature of the interface. In general, if you find that you're wanting to throw soft errors, you must review the granularity of the interface because it is likely you're trying to implement something

that is too low level. The rule of thumb is to provide interface functions that **can't** fail, except when faced with hard errors.

For a trivial example, suppose we wanted to add an "`OpenFileForWriting`" function. For many operating systems, if the file doesn't exist, attempting to open the file will produce an error. However, `lib/System` should not simply throw that error if it occurs because its a soft error. The problem is that the interface function, `OpenFileForWriting` is too low level. It should be `OpenOrCreateFileForWriting`. In the case of the soft "doesn't exist" error, this function would just create it and then open it for writing.

This design principle needs to be maintained in `lib/System` because it avoids the propagation of soft error handling throughout the rest of LLVM. Hard errors will generally just cause a termination for an LLVM tool so don't be bashful about throwing them.

Rules of thumb:

1. Don't throw soft errors, only hard errors.
2. If you're tempted to throw a soft error, re-think the interface.
3. Handle internally the most common normal/good/soft error conditions so the rest of LLVM doesn't have to.

No throw Specifications

None of the `lib/System` interface functions may be declared with C++ `throw()` specifications on them. This requirement makes sure that the compiler does not insert additional exception handling code into the interface functions. This is a performance consideration: `lib/System` functions are at the bottom of many call chains and as such can be frequently called. We need them to be as efficient as possible. However, no routines in the system library should actually throw exceptions.

Code Organization

Implementations of the System Library interface are separated by their general class of operating system. Currently only Unix and Win32 classes are defined but more could be added for other operating system classifications. To distinguish which implementation to compile, the code in `lib/System` uses the `LLVM_ON_UNIX` and `LLVM_ON_WIN32` `#defines` provided via configure through the `llvm/Config/config.h` file. Each source file in `lib/System`, after implementing the generic (operating system independent) functionality needs to include the correct implementation using a set of `#if defined(LLVM_ON_XYZ)` directives. For example, if we had `lib/System/File.cpp`, we'd expect to see in that file:

```
#if defined(LLVM_ON_UNIX)
#include "Unix/File.cpp"
#endif
#if defined(LLVM_ON_WIN32)
#include "Win32/File.cpp"
#endif
```

The implementation in `lib/System/Unix/File.cpp` should handle all Unix variants. The implementation in `lib/System/Win32/File.cpp` should handle all Win32 variants. What this does is quickly differentiate the basic class of operating system that will provide the implementation. The specific details for a given platform must still be determined through the use of `#ifdef`.

Consistent Semantics

The implementation of a `lib/System` interface can vary drastically between platforms. That's okay as long as the end result of the interface function is the same. For example, a function to create a directory is pretty straight forward on all operating system. System V IPC on the other hand isn't even supported on all platforms. Instead of "supporting"

System V IPC, `lib/System` should provide an interface to the basic concept of inter-process communications. The implementations might use System V IPC if that was available or named pipes, or whatever gets the job done effectively for a given operating system. In all cases, the interface and the implementation must be semantically consistent.

4.16 Source Level Debugging with LLVM

4.16.1 Introduction

This document is the central repository for all information pertaining to debug information in LLVM. It describes the *actual format that the LLVM debug information takes*, which is useful for those interested in creating front-ends or dealing directly with the information. Further, this document provides specific examples of what debug information for C/C++ looks like.

Philosophy behind LLVM debugging information

The idea of the LLVM debugging information is to capture how the important pieces of the source-language's Abstract Syntax Tree map onto LLVM code. Several design aspects have shaped the solution that appears here. The important ones are:

- Debugging information should have very little impact on the rest of the compiler. No transformations, analyses, or code generators should need to be modified because of debugging information.
- LLVM optimizations should interact in *well-defined and easily described ways* with the debugging information.
- Because LLVM is designed to support arbitrary programming languages, LLVM-to-LLVM tools should not need to know anything about the semantics of the source-level-language.
- Source-level languages are often **widely** different from one another. LLVM should not put any restrictions of the flavor of the source-language, and the debugging information should work with any language.
- With code generator support, it should be possible to use an LLVM compiler to compile a program to native machine code and standard debugging formats. This allows compatibility with traditional machine-code level debuggers, like GDB or DBX.

The approach used by the LLVM implementation is to use a small set of *intrinsic functions* to define a mapping between LLVM program objects and the source-level objects. The description of the source-level program is maintained in LLVM metadata in an *implementation-defined format* (the C/C++ front-end currently uses working draft 7 of the DWARF 3 standard).

When a program is being debugged, a debugger interacts with the user and turns the stored debug information into source-language specific information. As such, a debugger must be aware of the source-language, and is thus tied to a specific language or family of languages.

Debug information consumers

The role of debug information is to provide meta information normally stripped away during the compilation process. This meta information provides an LLVM user a relationship between generated code and the original program source code.

Currently, debug information is consumed by DwarfDebug to produce dwarf information used by the gdb debugger. Other targets could use the same information to produce stabs or other debug forms.

It would also be reasonable to use debug information to feed profiling tools for analysis of generated code, or, tools for reconstructing the original source from generated code.

TODO - expound a bit more.

Debugging optimized code

An extremely high priority of LLVM debugging information is to make it interact well with optimizations and analysis. In particular, the LLVM debug information provides the following guarantees:

- LLVM debug information **always provides information to accurately read the source-level state of the program**, regardless of which LLVM optimizations have been run, and without any modification to the optimizations themselves. However, some optimizations may impact the ability to modify the current state of the program with a debugger, such as setting program variables, or calling functions that have been deleted.
- As desired, LLVM optimizations can be upgraded to be aware of the LLVM debugging information, allowing them to update the debugging information as they perform aggressive optimizations. This means that, with effort, the LLVM optimizers could optimize debug code just as well as non-debug code.

- LLVM debug information does not prevent optimizations from happening (for example inlining, basic block reordering/merging/cleanup, tail duplication, etc).
- LLVM debug information is automatically optimized along with the rest of the program, using existing facilities. For example, duplicate information is automatically merged by the linker, and unused information is automatically removed.

Basically, the debug information allows you to compile a program with "`-O0 -g`" and get full debug information, allowing you to arbitrarily modify the program as it executes from a debugger. Compiling a program with "`-O3 -g`" gives you full debug information that is always available and accurate for reading (e.g., you get accurate stack traces despite tail call elimination and inlining), but you might lose the ability to modify the program and call functions where were optimized out of the program, or inlined away completely.

LLVM test suite provides a framework to test optimizer's handling of debugging information. It can be run like this:

```
% cd llvm/projects/test-suite/MultiSource/Benchmarks  # or some other level
% make TEST=dbgopt
```

This will test impact of debugging information on optimization passes. If debugging information influences optimization passes then it will be reported as a failure. See *LLVM Testing Infrastructure Guide* for more information on LLVM test infrastructure and how to run various tests.

4.16.2 Debugging information format

LLVM debugging information has been carefully designed to make it possible for the optimizer to optimize the program and debugging information without necessarily having to know anything about debugging information. In particular, the use of metadata avoids duplicated debugging information from the beginning, and the global dead code elimination pass automatically deletes debugging information for a function if it decides to delete the function.

To do this, most of the debugging information (descriptors for types, variables, functions, source files, etc) is inserted by the language front-end in the form of LLVM metadata.

Debug information is designed to be agnostic about the target debugger and debugging information representation (e.g. DWARF/Stabs/etc). It uses a generic pass to decode the information that represents variables, types, functions, namespaces, etc: this allows for arbitrary source-language semantics and type-systems to be used, as long as there is a module written for the target debugger to interpret the information.

To provide basic functionality, the LLVM debugger does have to make some assumptions about the source-level language being debugged, though it keeps these to a minimum. The only common features that the LLVM debugger assumes exist are *source files*, and *program objects*. These abstract objects are used by a debugger to form stack traces, show information about local variables, etc.

This section of the documentation first describes the representation aspects common to any source-language. *C/C++ front-end specific debug information* describes the data layout conventions used by the C and C++ front-ends.

Debug information descriptors

In consideration of the complexity and volume of debug information, LLVM provides a specification for well formed debug descriptors.

Consumers of LLVM debug information expect the descriptors for program objects to start in a canonical format, but the descriptors can include additional information appended at the end that is source-language specific. All debugging information objects start with a tag to indicate what type of object it is. The source-language is allowed to define its own objects, by using unreserved tag numbers. We recommend using with tags in the range 0x1000 through 0x2000 (there is a defined `enum DW_TAG_user_base = 0x1000`.)

The fields of debug descriptors used internally by LLVM are restricted to only the simple data types `i32`, `i1`, `float`, `double`, `mdstring` and `mdnode`.

```
!1 = metadata !{
  i32,   ;; A tag
  ...
}
```

Most of the string and integer fields in descriptors are packed into a single, null-separated `mdstring`. The first field of the header is always an `i32` containing the DWARF tag value identifying the content of the descriptor.

For clarity of definition in this document, these header fields are described below split inside an imaginary `DIHeader` construct. This is invalid assembly syntax. In valid IR, these fields are stringified and concatenated, separated by `\00`.

The details of the various descriptors follow.

Compile unit descriptors

```
!0 = metadata !{
  DIHeader(
    i32,       ;; Tag = 17 (DW_TAG_compile_unit)
    i32,       ;; DWARF language identifier (ex. DW_LANG_C89)
    mdstring,  ;; Producer (ex. "4.0.1 LLVM (LLVM research group)")
    i1,        ;; True if this is optimized.
    mdstring,  ;; Flags
    i32,       ;; Runtime version
    mdstring,  ;; Split debug filename
    i32        ;; Debug info emission kind (1 = Full Debug Info, 2 = Line Tables Only)
  ),
  metadata,  ;; Source directory (including trailing slash) & file pair
  metadata,  ;; List of enums types
  metadata,  ;; List of retained types
  metadata,  ;; List of subprograms
  metadata,  ;; List of global variables
  metadata   ;; List of imported entities
}
```

These descriptors contain a source language ID for the file (we use the DWARF 3.0 ID numbers, such as `DW_LANG_C89`, `DW_LANG_C_plus_plus`, `DW_LANG_Cobol74`, etc), a reference to a metadata node containing a pair of strings for the source file name and the working directory, as well as an identifier string for the compiler that produced it.

Compile unit descriptors provide the root context for objects declared in a specific compilation unit. File descriptors are defined using this context. These descriptors are collected by a named metadata `!llvm.dbg.cu`. They keep track of subprograms, global variables, type information, and imported entities (declarations and namespaces).

File descriptors

```
!0 = metadata !{
  DIHeader(
    i32       ;; Tag = 41 (DW_TAG_file_type)
  ),
  metadata  ;; Source directory (including trailing slash) & file pair
}
```

These descriptors contain information for a file. Global variables and top level functions would be defined using this context. File descriptors also provide context for source line correspondence.

Each input file is encoded as a separate file descriptor in LLVM debugging information output.

Global variable descriptors

```
!1 = metadata !{
  DIHeader(
    i32,      ;; Tag = 52 (DW_TAG_variable)
    mdstring, ;; Name
    mdstring, ;; Display name (fully qualified C++ name)
    mdstring, ;; MIPS linkage name (for C++)
    i32,      ;; Line number where defined
    i1,       ;; True if the global is local to compile unit (static)
    i1        ;; True if the global is defined in the compile unit (not extern)
  ),
  metadata, ;; Reference to context descriptor
  metadata, ;; Reference to file where defined
  metadata, ;; Reference to type descriptor
  {}*,      ;; Reference to the global variable
  metadata, ;; The static member declaration, if any
}
```

These descriptors provide debug information about global variables. They provide details such as name, type and where the variable is defined. All global variables are collected inside the named metadata `!llvm.dbg.cu`.

Subprogram descriptors

```
!2 = metadata !{
  DIHeader(
    i32,      ;; Tag = 46 (DW_TAG_subprogram)
    mdstring, ;; Name
    mdstring, ;; Display name (fully qualified C++ name)
    mdstring, ;; MIPS linkage name (for C++)
    i32,      ;; Line number where defined
    i1,       ;; True if the global is local to compile unit (static)
    i1,       ;; True if the global is defined in the compile unit (not extern)
    i32,      ;; Virtuality, e.g. dwarf::DW_VIRTUALITY__virtual
    i32,      ;; Index into a virtual function
    i32,      ;; Flags - Artificial, Private, Protected, Explicit, Prototyped.
    i1,       ;; isOptimized
    i32       ;; Line number where the scope of the subprogram begins
  ),
  metadata, ;; Source directory (including trailing slash) & file pair
  metadata, ;; Reference to context descriptor
  metadata, ;; Reference to type descriptor
  metadata, ;; indicates which base type contains the vtable pointer for the
            ;; derived class
  {}*,      ;; Reference to the LLVM function
  metadata, ;; Lists function template parameters
  metadata, ;; Function declaration descriptor
  metadata  ;; List of function variables
}
```

These descriptors provide debug information about functions, methods and subprograms. They provide details such as name, return types and the source location where the subprogram is defined.

Block descriptors

```
!3 = metadata !{
  DIHeader(
    i32,      ;; Tag = 11 (DW_TAG_lexical_block)
    i32,      ;; Line number
    i32,      ;; Column number
    i32       ;; Unique ID to identify blocks from a template function
  ),
  metadata, ;; Source directory (including trailing slash) & file pair
  metadata  ;; Reference to context descriptor
}
```

This descriptor provides debug information about nested blocks within a subprogram. The line number and column numbers are used to dinstinguish two lexical blocks at same depth.

```
!3 = metadata !{
  DIHeader(
    i32,      ;; Tag = 11 (DW_TAG_lexical_block)
    i32       ;; DWARF path discriminator value
  ),
  metadata, ;; Source directory (including trailing slash) & file pair
  metadata  ;; Reference to the scope we're annotating with a file change
}
```

This descriptor provides a wrapper around a lexical scope to handle file changes in the middle of a lexical block.

Basic type descriptors

```
!4 = metadata !{
  DIHeader(
    i32,      ;; Tag = 36 (DW_TAG_base_type)
    mdstring, ;; Name (may be "" for anonymous types)
    i32,      ;; Line number where defined (may be 0)
    i64,      ;; Size in bits
    i64,      ;; Alignment in bits
    i64,      ;; Offset in bits
    i32,      ;; Flags
    i32       ;; DWARF type encoding
  ),
  metadata, ;; Source directory (including trailing slash) & file pair (may be null)
  metadata  ;; Reference to context
}
```

These descriptors define primitive types used in the code. Example `int`, `bool` and `float`. The context provides the scope of the type, which is usually the top level. Since basic types are not usually user defined the context and line number can be left as NULL and 0. The size, alignment and offset are expressed in bits and can be 64 bit values. The alignment is used to round the offset when embedded in a *composite type* (example to keep float doubles on 64 bit boundaries). The offset is the bit offset if embedded in a *composite type*.

The type encoding provides the details of the type. The values are typically one of the following:

```
DW_ATE_address       = 1
DW_ATE_boolean       = 2
DW_ATE_float         = 4
DW_ATE_signed        = 5
DW_ATE_signed_char   = 6
```

```
DW_ATE_unsigned      = 7
DW_ATE_unsigned_char = 8
```

Derived type descriptors

```
!5 = metadata !{
  DIHeader(
    i32,      ;; Tag (see below)
    mdstring, ;; Name (may be "" for anonymous types)
    i32,      ;; Line number where defined (may be 0)
    i64,      ;; Size in bits
    i64,      ;; Alignment in bits
    i64,      ;; Offset in bits
    i32       ;; Flags to encode attributes, e.g. private
  ),
  metadata, ;; Source directory (including trailing slash) & file pair (may be null)
  metadata, ;; Reference to context
  metadata, ;; Reference to type derived from
  metadata  ;; (optional) Objective C property node
}
```

These descriptors are used to define types derived from other types. The value of the tag varies depending on the meaning. The following are possible tag values:

```
DW_TAG_formal_parameter   = 5
DW_TAG_member             = 13
DW_TAG_pointer_type       = 15
DW_TAG_reference_type     = 16
DW_TAG_typedef            = 22
DW_TAG_ptr_to_member_type = 31
DW_TAG_const_type         = 38
DW_TAG_volatile_type      = 53
DW_TAG_restrict_type      = 55
```

`DW_TAG_member` is used to define a member of a *composite type* or *subprogram*. The type of the member is the *derived type*. `DW_TAG_formal_parameter` is used to define a member which is a formal argument of a subprogram.

`DW_TAG_typedef` is used to provide a name for the derived type.

`DW_TAG_pointer_type`, `DW_TAG_reference_type`, `DW_TAG_const_type`, `DW_TAG_volatile_type` and `DW_TAG_restrict_type` are used to qualify the *derived type*.

Derived type location can be determined from the context and line number. The size, alignment and offset are expressed in bits and can be 64 bit values. The alignment is used to round the offset when embedded in a *composite type* (example to keep float doubles on 64 bit boundaries.) The offset is the bit offset if embedded in a *composite type*.

Note that the `void *` type is expressed as a type derived from NULL.

Composite type descriptors

```
!6 = metadata !{
  DIHeader(
    i32,      ;; Tag (see below)
    mdstring, ;; Name (may be "" for anonymous types)
    i32,      ;; Line number where defined (may be 0)
```

```
    i64,      ;; Size in bits
    i64,      ;; Alignment in bits
    i64,      ;; Offset in bits
    i32,      ;; Flags
    i32       ;; Runtime languages
  ),
  metadata, ;; Source directory (including trailing slash) & file pair (may be null)
  metadata, ;; Reference to context
  metadata, ;; Reference to type derived from
  metadata, ;; Reference to array of member descriptors
  metadata, ;; Base type containing the vtable pointer for this type
  metadata, ;; Template parameters
  mdstring  ;; A unique identifier for type uniquing purpose (may be null)
}
```

These descriptors are used to define types that are composed of 0 or more elements. The value of the tag varies depending on the meaning. The following are possible tag values:

```
DW_TAG_array_type       = 1
DW_TAG_enumeration_type = 4
DW_TAG_structure_type   = 19
DW_TAG_union_type       = 23
DW_TAG_subroutine_type  = 21
DW_TAG_inheritance      = 28
```

The vector flag indicates that an array type is a native packed vector.

The members of array types (tag = `DW_TAG_array_type`) are *subrange descriptors*, each representing the range of subscripts at that level of indexing.

The members of enumeration types (tag = `DW_TAG_enumeration_type`) are *enumerator descriptors*, each representing the definition of enumeration value for the set. All enumeration type descriptors are collected inside the named metadata `!llvm.dbg.cu`.

The members of structure (tag = `DW_TAG_structure_type`) or union (tag = `DW_TAG_union_type`) types are any one of the *basic*, *derived* or *composite* type descriptors, each representing a field member of the structure or union.

For C++ classes (tag = `DW_TAG_structure_type`), member descriptors provide information about base classes, static members and member functions. If a member is a *derived type descriptor* and has a tag of `DW_TAG_inheritance`, then the type represents a base class. If the member of is a *global variable descriptor* then it represents a static member. And, if the member is a *subprogram descriptor* then it represents a member function. For static members and member functions, `getName()` returns the members link or the C++ mangled name. `getDisplayName()` the simplied version of the name.

The first member of subroutine (tag = `DW_TAG_subroutine_type`) type elements is the return type for the subroutine. The remaining elements are the formal arguments to the subroutine.

Composite type location can be determined from the context and line number. The size, alignment and offset are expressed in bits and can be 64 bit values. The alignment is used to round the offset when embedded in a *composite type* (as an example, to keep float doubles on 64 bit boundaries). The offset is the bit offset if embedded in a *composite type*.

Subrange descriptors

```
!42 = metadata !{
  DIHeader(
    i32,      ;; Tag = 33 (DW_TAG_subrange_type)
    i64,      ;; Low value
```

```
    i64       ;; High value
  )
}
```

These descriptors are used to define ranges of array subscripts for an array *composite type*. The low value defines the lower bounds typically zero for C/C++. The high value is the upper bounds. Values are 64 bit. `High - Low + 1` is the size of the array. If `Low > High` the array bounds are not included in generated debugging information.

Enumerator descriptors

```
!6 = metadata !{
  DIHeader(
    i32,      ;; Tag = 40 (DW_TAG_enumerator)
    mdstring, ;; Name
    i64       ;; Value
  )
}
```

These descriptors are used to define members of an enumeration *composite type*, it associates the name to the value.

Local variables

```
!7 = metadata !{
  DIHeader(
    i32,      ;; Tag (see below)
    mdstring, ;; Name
    i32,      ;; 24 bit - Line number where defined
              ;; 8 bit - Argument number. 1 indicates 1st argument.
    i32       ;; flags
  ),
  metadata, ;; Context
  metadata, ;; Reference to file where defined
  metadata, ;; Reference to the type descriptor
  metadata  ;; (optional) Reference to inline location
}
```

These descriptors are used to define variables local to a sub program. The value of the tag depends on the usage of the variable:

```
DW_TAG_auto_variable   = 256
DW_TAG_arg_variable    = 257
```

An auto variable is any variable declared in the body of the function. An argument variable is any variable that appears as a formal argument to the function.

The context is either the subprogram or block where the variable is defined. Name the source variable name. Context and line indicate where the variable was defined. Type descriptor defines the declared type of the variable.

Complex Expressions

```
!8 = metadata !{
  i32,      ;; DW_TAG_expression
  ...
}
```

Complex expressions describe variable storage locations in terms of prefix-notated DWARF expressions. Currently the only supported operators are `DW_OP_plus`, `DW_OP_deref`, and `DW_OP_piece`.

The `DW_OP_piece` operator is used for (typically larger aggregate) variables that are fragmented across several locations. It takes two i32 arguments, an offset and a size in bytes to describe which piece of the variable is at this location.

Debugger intrinsic functions

LLVM uses several intrinsic functions (name prefixed with "`llvm.dbg`") to provide debug information at various points in generated code.

`llvm.dbg.declare`

```
void %llvm.dbg.declare(metadata, metadata)
```

This intrinsic provides information about a local element (e.g., variable). The first argument is metadata holding the alloca for the variable. The second argument is metadata containing a description of the variable.

`llvm.dbg.value`

```
void %llvm.dbg.value(metadata, i64, metadata)
```

This intrinsic provides information when a user source variable is set to a new value. The first argument is the new value (wrapped as metadata). The second argument is the offset in the user source variable where the new value is written. The third argument is metadata containing a description of the user source variable.

4.16.3 Object lifetimes and scoping

In many languages, the local variables in functions can have their lifetimes or scopes limited to a subset of a function. In the C family of languages, for example, variables are only live (readable and writable) within the source block that they are defined in. In functional languages, values are only readable after they have been defined. Though this is a very obvious concept, it is non-trivial to model in LLVM, because it has no notion of scoping in this sense, and does not want to be tied to a language's scoping rules.

In order to handle this, the LLVM debug format uses the metadata attached to llvm instructions to encode line number and scoping information. Consider the following C fragment, for example:

```
1.  void foo() {
2.    int X = 21;
3.    int Y = 22;
4.    {
5.      int Z = 23;
6.      Z = X;
7.    }
8.    X = Y;
9.  }
```

Compiled to LLVM, this function would be represented like this:

```
define void @foo() #0 {
entry:
 %X = alloca i32, align 4
```

```
  %Y = alloca i32, align 4
  %Z = alloca i32, align 4
  call void @llvm.dbg.declare(metadata !{i32* %X}, metadata !10), !dbg !12
    ; [debug line = 2:7] [debug variable = X]
  store i32 21, i32* %X, align 4, !dbg !12
  call void @llvm.dbg.declare(metadata !{i32* %Y}, metadata !13), !dbg !14
    ; [debug line = 3:7] [debug variable = Y]
  store i32 22, i32* %Y, align 4, !dbg !14
  call void @llvm.dbg.declare(metadata !{i32* %Z}, metadata !15), !dbg !17
    ; [debug line = 5:9] [debug variable = Z]
  store i32 23, i32* %Z, align 4, !dbg !17
  %0 = load i32* %X, align 4, !dbg !18
    [debug line = 6:5]
  store i32 %0, i32* %Z, align 4, !dbg !18
  %1 = load i32* %Y, align 4, !dbg !19
    [debug line = 8:3]
  store i32 %1, i32* %X, align 4, !dbg !19
  ret void, !dbg !20
}

; Function Attrs: nounwind readnone
declare void @llvm.dbg.declare(metadata, metadata) #1

attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false"
  "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"
  "no-infs-fp-math"="false" "no-nans-fp-math"="false"
  "stack-protector-buffer-size"="8" "unsafe-fp-math"="false"
  "use-soft-float"="false" }
attributes #1 = { nounwind readnone }

!llvm.dbg.cu = !{!0}
!llvm.module.flags = !{!8}
!llvm.ident = !{!9}

!0 = metadata !{i32 786449, metadata !1, i32 12,
                metadata !"clang version 3.4 (trunk 193128) (llvm/trunk 193139)",
                i1 false, metadata !"", i32 0, metadata !2, metadata !2, metadata !3,
                metadata !2, metadata !2, metadata !""} ; [ DW_TAG_compile_unit ] \
                  [/private/tmp/foo.c] \
                  [DW_LANG_C99]
!1 = metadata !{metadata !"t.c", metadata !"/private/tmp"}
!2 = metadata !{i32 0}
!3 = metadata !{metadata !4}
!4 = metadata !{i32 786478, metadata !1, metadata !5, metadata !"foo",
                metadata !"foo", metadata !"", i32 1, metadata !6,
                i1 false, i1 true, i32 0, i32 0, null, i32 0, i1 false,
                void ()* @foo, null, null, metadata !2, i32 1}
                ; [ DW_TAG_subprogram ] [line 1] [def] [foo]
!5 = metadata !{i32 786473, metadata !1}  ; [ DW_TAG_file_type ] \
                  [/private/tmp/t.c]
!6 = metadata !{i32 786453, i32 0, null, metadata !"", i32 0, i64 0, i64 0,
                i64 0, i32 0, null, metadata !7, i32 0, null, null, null}
                ; [ DW_TAG_subroutine_type ] \
                  [line 0, size 0, align 0, offset 0] [from ]
!7 = metadata !{null}
!8 = metadata !{i32 2, metadata !"Dwarf Version", i32 2}
!9 = metadata !{metadata !"clang version 3.4 (trunk 193128) (llvm/trunk 193139)"}
!10 = metadata !{i32 786688, metadata !4, metadata !"X", metadata !5, i32 2,
```

```
                  metadata !11, i32 0, i32 0} ; [ DW_TAG_auto_variable ] [X] \
                    [line 2]
!11 = metadata !{i32 786468, null, null, metadata !"int", i32 0, i64 32,
                  i64 32, i64 0, i32 0, i32 5} ; [ DW_TAG_base_type ] [int] \
                    [line 0, size 32, align 32, offset 0, enc DW_ATE_signed]
!12 = metadata !{i32 2, i32 0, metadata !4, null}
!13 = metadata !{i32 786688, metadata !4, metadata !"Y", metadata !5, i32 3,
                  metadata !11, i32 0, i32 0} ; [ DW_TAG_auto_variable ] [Y] \
                    [line 3]
!14 = metadata !{i32 3, i32 0, metadata !4, null}
!15 = metadata !{i32 786688, metadata !16, metadata !"Z", metadata !5, i32 5,
                  metadata !11, i32 0, i32 0} ; [ DW_TAG_auto_variable ] [Z] \
                    [line 5]
!16 = metadata !{i32 786443, metadata !1, metadata !4, i32 4, i32 0, i32 0} \
                  ; [ DW_TAG_lexical_block ] [/private/tmp/t.c]
!17 = metadata !{i32 5, i32 0, metadata !16, null}
!18 = metadata !{i32 6, i32 0, metadata !16, null}
!19 = metadata !{i32 8, i32 0, metadata !4, null} ; [ DW_TAG_imported_declaration ]
!20 = metadata !{i32 9, i32 0, metadata !4, null}
```

This example illustrates a few important details about LLVM debugging information. In particular, it shows how the `llvm.dbg.declare` intrinsic and location information, which are attached to an instruction, are applied together to allow a debugger to analyze the relationship between statements, variable definitions, and the code used to implement the function.

```
call void @llvm.dbg.declare(metadata !{i32* %X}, metadata !10), !dbg !12
  ; [debug line = 2:7] [debug variable = X]
```

The first intrinsic `%llvm.dbg.declare` encodes debugging information for the variable `X`. The metadata `!dbg !12` attached to the intrinsic provides scope information for the variable `X`.

```
!12 = metadata !{i32 2, i32 0, metadata !4, null}
!4 = metadata !{i32 786478, metadata !1, metadata !5, metadata !"foo",
                metadata !"foo", metadata !"", i32 1, metadata !6,
                i1 false, i1 true, i32 0, i32 0, null, i32 0, i1 false,
                void ()* @foo, null, null, metadata !2, i32 1}
                  ; [ DW_TAG_subprogram ] [line 1] [def] [foo]
```

Here `!12` is metadata providing location information. It has four fields: line number, column number, scope, and original scope. The original scope represents inline location if this instruction is inlined inside a caller, and is null otherwise. In this example, scope is encoded by `!4`, a *subprogram descriptor*. This way the location information attached to the intrinsics indicates that the variable `X` is declared at line number 2 at a function level scope in function `foo`.

Now lets take another example.

```
call void @llvm.dbg.declare(metadata !{i32* %Z}, metadata !15), !dbg !17
  ; [debug line = 5:9] [debug variable = Z]
```

The third intrinsic `%llvm.dbg.declare` encodes debugging information for variable `Z`. The metadata `!dbg !17` attached to the intrinsic provides scope information for the variable `Z`.

```
!16 = metadata !{i32 786443, metadata !1, metadata !4, i32 4, i32 0, i32 0} \
                  ; [ DW_TAG_lexical_block ] [/private/tmp/t.c]
!17 = metadata !{i32 5, i32 0, metadata !16, null}
```

Here `!15` indicates that `Z` is declared at line number 5 and column number 0 inside of lexical scope `!16`. The lexical scope itself resides inside of subprogram `!4` described above.

The scope information attached with each instruction provides a straightforward way to find instructions covered by a scope.

4.16.4 C/C++ front-end specific debug information

The C and C++ front-ends represent information about the program in a format that is effectively identical to DWARF 3.0 in terms of information content. This allows code generators to trivially support native debuggers by generating standard dwarf information, and contains enough information for non-dwarf targets to translate it as needed.

This section describes the forms used to represent C and C++ programs. Other languages could pattern themselves after this (which itself is tuned to representing programs in the same way that DWARF 3 does), or they could choose to provide completely different forms if they don't fit into the DWARF model. As support for debugging information gets added to the various LLVM source-language front-ends, the information used should be documented here.

The following sections provide examples of a few C/C++ constructs and the debug information that would best describe those constructs. The canonical references are the `DIDescriptor` classes defined in `include/llvm/IR/DebugInfo.h` and the implementations of the helper functions in `lib/IR/DIBuilder.cpp`.

C/C++ source file information

`llvm::Instruction` provides easy access to metadata attached with an instruction. One can extract line number information encoded in LLVM IR using `Instruction::getMetadata()` and `DILocation::getLineNumber()`.

```
if (MDNode *N = I->getMetadata("dbg")) {  // Here I is an LLVM instruction
  DILocation Loc(N);                      // DILocation is in DebugInfo.h
  unsigned Line = Loc.getLineNumber();
  StringRef File = Loc.getFilename();
  StringRef Dir = Loc.getDirectory();
}
```

C/C++ global variable information

Given an integer global variable declared as follows:

```
int MyGlobal = 100;
```

a C/C++ front-end would generate the following descriptors:

```
;;
;; Define the global itself.
;;
@MyGlobal = global i32 100, align 4
...
;;
;; List of debug info of globals
;;
!llvm.dbg.cu = !{!0}

;; Define the compile unit.
!0 = metadata !{
  ; Header(
  ;   i32 17,                           ;; Tag
  ;   i32 0,                            ;; Context
```

```
  ;   i32 4,                              ;; Language
  ;   metadata !"clang version 3.6.0 ", ;; Producer
  ;   i1 false,                           ;; "isOptimized"?
  ;   metadata !"",                       ;; Flags
  ;   i32 0,                              ;; Runtime Version
  ;   "",                                 ;; Split debug filename
  ;   1                                   ;; Full debug info
  ; )
  metadata !"0x11\0012\00clang version 3.6.0 \000\00\000\00\001",
  metadata !1,                            ;; File
  metadata !2,                            ;; Enum Types
  metadata !2,                            ;; Retained Types
  metadata !2,                            ;; Subprograms
  metadata !3,                            ;; Global Variables
  metadata !2                             ;; Imported entities
} ; [ DW_TAG_compile_unit ]

;; The file/directory pair.
!1 = metadata !{
  metadata !"foo.c",                                  ;; Filename
  metadata !"/Users/dexonsmith/data/llvm/debug-info" ;; Directory
}

;; An empty array.
!2 = metadata !{}

;; The Array of Global Variables
!3 = metadata !{
  metadata !4
}

;;
;; Define the global variable itself.
;;
!4 = metadata !{
  ; Header(
  ;   i32 52,                         ;; Tag
  ;   metadata !"MyGlobal",           ;; Name
  ;   metadata !"MyGlobal",           ;; Display Name
  ;   metadata !"",                   ;; Linkage Name
  ;   i32 1,                          ;; Line
  ;   i32 0,                          ;; IsLocalToUnit
  ;   i32 1                           ;; IsDefinition
  ; )
  metadata !"0x34\00MyGlobal\00MyGlobal\00\001\000\001",
  null,                               ;; Unused
  metadata !5,                        ;; File
  metadata !6,                        ;; Type
  i32* @MyGlobal,                     ;; LLVM-IR Value
  null                                ;; Static member declaration
} ; [ DW_TAG_variable ]

;;
;; Define the file
;;
!5 = metadata !{
  ; Header(
  ;   i32 41             ;; Tag
```

```
  ; )
  metadata !"0x29",
  metadata !1             ;; File/directory pair
} ; [ DW_TAG_file_type ]

;;
;; Define the type
;;
!6 = metadata !{
  ; Header(
  ;   i32 36,                       ;; Tag
  ;   metadata !"int",              ;; Name
  ;   i32 0,                        ;; Line
  ;   i64 32,                       ;; Size in Bits
  ;   i64 32,                       ;; Align in Bits
  ;   i64 0,                        ;; Offset
  ;   i32 0,                        ;; Flags
  ;   i32 5                         ;; Encoding
  ; )
  metadata !"0x24\00int\000\0032\0032\000\000\005",
  null,                             ;; Unused
  null                              ;; Unused
} ; [ DW_TAG_base_type ]
```

C/C++ function information

Given a function declared as follows:

```
int main(int argc, char *argv[]) {
  return 0;
}
```

a C/C++ front-end would generate the following descriptors:

```
;;
;; Define the anchor for subprograms.
;;
!6 = metadata !{
  ; Header(
  ;   i32 46,               ;; Tag
  ;   metadata !"main",     ;; Name
  ;   metadata !"main",     ;; Display name
  ;   metadata !"",         ;; Linkage name
  ;   i32 1,                ;; Line number
  ;   i1 false,             ;; Is local
  ;   i1 true,              ;; Is definition
  ;   i32 0,                ;; Virtuality attribute, e.g. pure virtual function
  ;   i32 0,                ;; Index into virtual table for C++ methods
  ;   i32 256,              ;; Flags
  ;   i1 0,                 ;; True if this function is optimized
  ;   1                     ;; Line number of the opening '{' of the function
  ; )
  metadata !"0x2e\00main\00main\00\001\000\001\000\000\00256\000\001",
  metadata !1,              ;; File
  metadata !5,              ;; Context
  metadata !6,              ;; Type
  null,                     ;; Containing type
  i32 (i32, i8**)* @main, ;; Pointer to llvm::Function
```

```
  null,                   ;; Function template parameters
  null,                   ;; Function declaration
  metadata !2             ;; List of function variables (emitted when optimizing)
}

;;
;; Define the subprogram itself.
;;
define i32 @main(i32 %argc, i8** %argv) {
...
}
```

4.16.5 Debugging information format

Debugging Information Extension for Objective C Properties

Introduction

Objective C provides a simpler way to declare and define accessor methods using declared properties. The language provides features to declare a property and to let compiler synthesize accessor methods.

The debugger lets developer inspect Objective C interfaces and their instance variables and class variables. However, the debugger does not know anything about the properties defined in Objective C interfaces. The debugger consumes information generated by compiler in DWARF format. The format does not support encoding of Objective C properties. This proposal describes DWARF extensions to encode Objective C properties, which the debugger can use to let developers inspect Objective C properties.

Proposal

Objective C properties exist separately from class members. A property can be defined only by "setter" and "getter" selectors, and be calculated anew on each access. Or a property can just be a direct access to some declared ivar. Finally it can have an ivar "automatically synthesized" for it by the compiler, in which case the property can be referred to in user code directly using the standard C dereference syntax as well as through the property "dot" syntax, but there is no entry in the `@interface` declaration corresponding to this ivar.

To facilitate debugging, these properties we will add a new DWARF TAG into the `DW_TAG_structure_type` definition for the class to hold the description of a given property, and a set of DWARF attributes that provide said description. The property tag will also contain the name and declared type of the property.

If there is a related ivar, there will also be a DWARF property attribute placed in the `DW_TAG_member` DIE for that ivar referring back to the property TAG for that property. And in the case where the compiler synthesizes the ivar directly, the compiler is expected to generate a `DW_TAG_member` for that ivar (with the `DW_AT_artificial` set to 1), whose name will be the name used to access this ivar directly in code, and with the property attribute pointing back to the property it is backing.

The following examples will serve as illustration for our discussion:

```
@interface I1 {
  int n2;
}

@property int p1;
@property int p2;
@end
```

```
@implementation I1
@synthesize p1;
@synthesize p2 = n2;
@end
```

This produces the following DWARF (this is a "pseudo dwarfdump" output):

```
0x00000100:  TAG_structure_type [7] *
               AT_APPLE_runtime_class( 0x10 )
               AT_name( "I1" )
               AT_decl_file( "Objc_Property.m" )
               AT_decl_line( 3 )

0x00000110    TAG_APPLE_property
                AT_name ( "p1" )
                AT_type ( {0x00000150} ( int ) )

0x00000120:   TAG_APPLE_property
                AT_name ( "p2" )
                AT_type ( {0x00000150} ( int ) )

0x00000130:   TAG_member [8]
                AT_name( "_p1" )
                AT_APPLE_property ( {0x00000110} "p1" )
                AT_type( {0x00000150} ( int ) )
                AT_artificial ( 0x1 )

0x00000140:    TAG_member [8]
                 AT_name( "n2" )
                 AT_APPLE_property ( {0x00000120} "p2" )
                 AT_type( {0x00000150} ( int ) )

0x00000150:  AT_type( ( int ) )
```

Note, the current convention is that the name of the ivar for an auto-synthesized property is the name of the property from which it derives with an underscore prepended, as is shown in the example. But we actually don't need to know this convention, since we are given the name of the ivar directly.

Also, it is common practice in ObjC to have different property declarations in the @interface and @implementation - e.g. to provide a read-only property in the interface,and a read-write interface in the implementation. In that case, the compiler should emit whichever property declaration will be in force in the current translation unit.

Developers can decorate a property with attributes which are encoded using `DW_AT_APPLE_property_attribute`.

```
@property (readonly, nonatomic) int pr;

TAG_APPLE_property [8]
  AT_name( "pr" )
  AT_type ( {0x00000147} (int) )
  AT_APPLE_property_attribute (DW_APPLE_PROPERTY_readonly, DW_APPLE_PROPERTY_nonatomic)
```

The setter and getter method names are attached to the property using `DW_AT_APPLE_property_setter` and `DW_AT_APPLE_property_getter` attributes.

```
@interface I1
@property (setter=myOwnP3Setter:) int p3;
-(void)myOwnP3Setter:(int)a;
@end
```

```
@implementation I1
@synthesize p3;
-(void)myOwnP3Setter:(int)a{ }
@end
```

The DWARF for this would be:

```
0x000003bd: TAG_structure_type [7] *
              AT_APPLE_runtime_class( 0x10 )
              AT_name( "I1" )
              AT_decl_file( "Objc_Property.m" )
              AT_decl_line( 3 )

0x000003cd      TAG_APPLE_property
                  AT_name ( "p3" )
                  AT_APPLE_property_setter ( "myOwnP3Setter:" )
                  AT_type( {0x00000147} ( int ) )

0x000003f3:     TAG_member [8]
                  AT_name( "_p3" )
                  AT_type ( {0x00000147} ( int ) )
                  AT_APPLE_property ( {0x000003cd} )
                  AT_artificial ( 0x1 )
```

New DWARF Tags

TAG	Value
DW_TAG_APPLE_property	0x4200

New DWARF Attributes

Attribute	Value	Classes
DW_AT_APPLE_property	0x3fed	Reference
DW_AT_APPLE_property_getter	0x3fe9	String
DW_AT_APPLE_property_setter	0x3fea	String
DW_AT_APPLE_property_attribute	0x3feb	Constant

New DWARF Constants

Name	Value
DW_APPLE_PROPERTY_readonly	0x01
DW_APPLE_PROPERTY_getter	0x02
DW_APPLE_PROPERTY_assign	0x04
DW_APPLE_PROPERTY_readwrite	0x08
DW_APPLE_PROPERTY_retain	0x10
DW_APPLE_PROPERTY_copy	0x20
DW_APPLE_PROPERTY_nonatomic	0x40
DW_APPLE_PROPERTY_setter	0x80
DW_APPLE_PROPERTY_atomic	0x100
DW_APPLE_PROPERTY_weak	0x200
DW_APPLE_PROPERTY_strong	0x400
DW_APPLE_PROPERTY_unsafe_unretained	0x800

Name Accelerator Tables

Introduction

The "`.debug_pubnames`" and "`.debug_pubtypes`" formats are not what a debugger needs. The "`pub`" in the section name indicates that the entries in the table are publicly visible names only. This means no static or hidden functions show up in the "`.debug_pubnames`". No static variables or private class variables are in the "`.debug_pubtypes`". Many compilers add different things to these tables, so we can't rely upon the contents between gcc, icc, or clang.

The typical query given by users tends not to match up with the contents of these tables. For example, the DWARF spec states that "In the case of the name of a function member or static data member of a C++ structure, class or union, the name presented in the "`.debug_pubnames`" section is not the simple name given by the `DW_AT_name attribute` of the referenced debugging information entry, but rather the fully qualified name of the data or function member." So the only names in these tables for complex C++ entries is a fully qualified name. Debugger users tend not to enter their search strings as "`a::b::c(int,const Foo&) const`", but rather as "`c`", "`b::c`" , or "`a::b::c`". So the name entered in the name table must be demangled in order to chop it up appropriately and additional names must be manually entered into the table to make it effective as a name lookup table for debuggers to se.

All debuggers currently ignore the "`.debug_pubnames`" table as a result of its inconsistent and useless public-only name content making it a waste of space in the object file. These tables, when they are written to disk, are not sorted in any way, leaving every debugger to do its own parsing and sorting. These tables also include an inlined copy of the string values in the table itself making the tables much larger than they need to be on disk, especially for large C++ programs.

Can't we just fix the sections by adding all of the names we need to this table? No, because that is not what the tables are defined to contain and we won't know the difference between the old bad tables and the new good tables. At best we could make our own renamed sections that contain all of the data we need.

These tables are also insufficient for what a debugger like LLDB needs. LLDB uses clang for its expression parsing where LLDB acts as a PCH. LLDB is then often asked to look for type "`foo`" or namespace "`bar`", or list items in namespace "`baz`". Namespaces are not included in the pubnames or pubtypes tables. Since clang asks a lot of questions when it is parsing an expression, we need to be very fast when looking up names, as it happens a lot. Having new accelerator tables that are optimized for very quick lookups will benefit this type of debugging experience greatly.

We would like to generate name lookup tables that can be mapped into memory from disk, and used as is, with little or no up-front parsing. We would also be able to control the exact content of these different tables so they contain

exactly what we need. The Name Accelerator Tables were designed to fix these issues. In order to solve these issues we need to:

- Have a format that can be mapped into memory from disk and used as is
- Lookups should be very fast
- Extensible table format so these tables can be made by many producers
- Contain all of the names needed for typical lookups out of the box
- Strict rules for the contents of tables

Table size is important and the accelerator table format should allow the reuse of strings from common string tables so the strings for the names are not duplicated. We also want to make sure the table is ready to be used as-is by simply mapping the table into memory with minimal header parsing.

The name lookups need to be fast and optimized for the kinds of lookups that debuggers tend to do. Optimally we would like to touch as few parts of the mapped table as possible when doing a name lookup and be able to quickly find the name entry we are looking for, or discover there are no matches. In the case of debuggers we optimized for lookups that fail most of the time.

Each table that is defined should have strict rules on exactly what is in the accelerator tables and documented so clients can rely on the content.

Hash Tables

Standard Hash Tables Typical hash tables have a header, buckets, and each bucket points to the bucket contents:

```
.------------.
|  HEADER    |
|------------|
|  BUCKETS   |
|------------|
|  DATA      |
`------------'
```

The BUCKETS are an array of offsets to DATA for each hash:

```
.------------.
| 0x00001000 | BUCKETS[0]
| 0x00002000 | BUCKETS[1]
| 0x00002200 | BUCKETS[2]
| 0x000034f0 | BUCKETS[3]
|            | ...
| 0xXXXXXXXX | BUCKETS[n_buckets]
'------------'
```

So for `bucket[3]` in the example above, we have an offset into the table 0x000034f0 which points to a chain of entries for the bucket. Each bucket must contain a next pointer, full 32 bit hash value, the string itself, and the data for the current string value.

```
            .------------.
0x000034f0: | 0x00003500 | next pointer
            | 0x12345678 | 32 bit hash
            | "erase"    | string value
            | data[n]    | HashData for this bucket
            |------------|
0x00003500: | 0x00003550 | next pointer
            | 0x29273623 | 32 bit hash
            | "dump"     | string value
```

```
            | data[n]    | HashData for this bucket
            |------------|
0x00003550: | 0x00000000 | next pointer
            | 0x82638293 | 32 bit hash
            | "main"     | string value
            | data[n]    | HashData for this bucket
            `------------'
```

The problem with this layout for debuggers is that we need to optimize for the negative lookup case where the symbol we're searching for is not present. So if we were to lookup "`printf`" in the table above, we would make a 32 hash for "`printf`", it might match `bucket[3]`. We would need to go to the offset 0x000034f0 and start looking to see if our 32 bit hash matches. To do so, we need to read the next pointer, then read the hash, compare it, and skip to the next bucket. Each time we are skipping many bytes in memory and touching new cache pages just to do the compare on the full 32 bit hash. All of these accesses then tell us that we didn't have a match.

Name Hash Tables To solve the issues mentioned above we have structured the hash tables a bit differently: a header, buckets, an array of all unique 32 bit hash values, followed by an array of hash value data offsets, one for each hash value, then the data for all hash values:

```
.-------------.
|  HEADER     |
|-------------|
|  BUCKETS    |
|-------------|
|  HASHES     |
|-------------|
|  OFFSETS    |
|-------------|
|  DATA       |
`-------------'
```

The `BUCKETS` in the name tables are an index into the `HASHES` array. By making all of the full 32 bit hash values contiguous in memory, we allow ourselves to efficiently check for a match while touching as little memory as possible. Most often checking the 32 bit hash values is as far as the lookup goes. If it does match, it usually is a match with no collisions. So for a table with "`n_buckets`" buckets, and "`n_hashes`" unique 32 bit hash values, we can clarify the contents of the `BUCKETS`, `HASHES` and `OFFSETS` as:

```
.-------------------------.
|  HEADER.magic           | uint32_t
|  HEADER.version         | uint16_t
|  HEADER.hash_function   | uint16_t
|  HEADER.bucket_count    | uint32_t
|  HEADER.hashes_count    | uint32_t
|  HEADER.header_data_len | uint32_t
|  HEADER_DATA            | HeaderData
|-------------------------|
|  BUCKETS                | uint32_t[n_buckets] // 32 bit hash indexes
|-------------------------|
|  HASHES                 | uint32_t[n_hashes] // 32 bit hash values
|-------------------------|
|  OFFSETS                | uint32_t[n_hashes] // 32 bit offsets to hash value data
|-------------------------|
|  ALL HASH DATA          |
`-------------------------'
```

So taking the exact same data from the standard hash example above we end up with:

```
            .------------.
            | HEADER     |
            |------------|
            |          0 | BUCKETS[0]
            |          2 | BUCKETS[1]
            |          5 | BUCKETS[2]
            |          6 | BUCKETS[3]
            |            | ...
            |        ... | BUCKETS[n_buckets]
            |------------|
            | 0x........ | HASHES[0]
            | 0x........ | HASHES[1]
            | 0x........ | HASHES[2]
            | 0x........ | HASHES[3]
            | 0x........ | HASHES[4]
            | 0x........ | HASHES[5]
            | 0x12345678 | HASHES[6]    hash for BUCKETS[3]
            | 0x29273623 | HASHES[7]    hash for BUCKETS[3]
            | 0x82638293 | HASHES[8]    hash for BUCKETS[3]
            | 0x........ | HASHES[9]
            | 0x........ | HASHES[10]
            | 0x........ | HASHES[11]
            | 0x........ | HASHES[12]
            | 0x........ | HASHES[13]
            | 0x........ | HASHES[n_hashes]
            |------------|
            | 0x........ | OFFSETS[0]
            | 0x........ | OFFSETS[1]
            | 0x........ | OFFSETS[2]
            | 0x........ | OFFSETS[3]
            | 0x........ | OFFSETS[4]
            | 0x........ | OFFSETS[5]
            | 0x000034f0 | OFFSETS[6]   offset for BUCKETS[3]
            | 0x00003500 | OFFSETS[7]   offset for BUCKETS[3]
            | 0x00003550 | OFFSETS[8]   offset for BUCKETS[3]
            | 0x........ | OFFSETS[9]
            | 0x........ | OFFSETS[10]
            | 0x........ | OFFSETS[11]
            | 0x........ | OFFSETS[12]
            | 0x........ | OFFSETS[13]
            | 0x........ | OFFSETS[n_hashes]
            |------------|
            |            |
            |            |
            |            |
            |            |
            |            |
            |------------|
0x000034f0: | 0x00001203 | .debug_str ("erase")
            | 0x00000004 | A 32 bit array count - number of HashData with name "erase"
            | 0x........ | HashData[0]
            | 0x........ | HashData[1]
            | 0x........ | HashData[2]
            | 0x........ | HashData[3]
            | 0x00000000 | String offset into .debug_str (terminate data for hash)
            |------------|
0x00003500: | 0x00001203 | String offset into .debug_str ("collision")
            | 0x00000002 | A 32 bit array count - number of HashData with name "collision"
```

```
            | 0x........ | HashData[0]
            | 0x........ | HashData[1]
            | 0x00001203 | String offset into .debug_str ("dump")
            | 0x00000003 | A 32 bit array count - number of HashData with name "dump"
            | 0x........ | HashData[0]
            | 0x........ | HashData[1]
            | 0x........ | HashData[2]
            | 0x00000000 | String offset into .debug_str (terminate data for hash)
            |------------|
0x00003550: | 0x00001203 | String offset into .debug_str ("main")
            | 0x00000009 | A 32 bit array count - number of HashData with name "main"
            | 0x........ | HashData[0]
            | 0x........ | HashData[1]
            | 0x........ | HashData[2]
            | 0x........ | HashData[3]
            | 0x........ | HashData[4]
            | 0x........ | HashData[5]
            | 0x........ | HashData[6]
            | 0x........ | HashData[7]
            | 0x........ | HashData[8]
            | 0x00000000 | String offset into .debug_str (terminate data for hash)
            `------------'
```

So we still have all of the same data, we just organize it more efficiently for debugger lookup. If we repeat the same "`printf`" lookup from above, we would hash "`printf`" and find it matches `BUCKETS[3]` by taking the 32 bit hash value and modulo it by `n_buckets`. `BUCKETS[3]` contains "6" which is the index into the `HASHES` table. We would then compare any consecutive 32 bit hashes values in the `HASHES` array as long as the hashes would be in `BUCKETS[3]`. We do this by verifying that each subsequent hash value modulo `n_buckets` is still 3. In the case of a failed lookup we would access the memory for `BUCKETS[3]`, and then compare a few consecutive 32 bit hashes before we know that we have no match. We don't end up marching through multiple words of memory and we really keep the number of processor data cache lines being accessed as small as possible.

The string hash that is used for these lookup tables is the Daniel J. Bernstein hash which is also used in the ELF `GNU_HASH` sections. It is a very good hash for all kinds of names in programs with very few hash collisions.

Empty buckets are designated by using an invalid hash index of `UINT32_MAX`.

Details

These name hash tables are designed to be generic where specializations of the table get to define additional data that goes into the header ("`HeaderData`"), how the string value is stored ("`KeyType`") and the content of the data for each hash value.

Header Layout The header has a fixed part, and the specialized part. The exact format of the header is:

```
struct Header
{
  uint32_t   magic;           // 'HASH' magic value to allow endian detection
  uint16_t   version;         // Version number
  uint16_t   hash_function;   // The hash function enumeration that was used
  uint32_t   bucket_count;    // The number of buckets in this hash table
  uint32_t   hashes_count;    // The total number of unique hash values and hash data offsets in this
  uint32_t   header_data_len; // The bytes to skip to get to the hash indexes (buckets) for correct a
                              // Specifically the length of the following HeaderData field - this doe
                              // include the size of the preceding fields
```

```
  HeaderData header_data;      // Implementation specific header data
};
```

The header starts with a 32 bit "magic" value which must be 'HASH' encoded as an ASCII integer. This allows the detection of the start of the hash table and also allows the table's byte order to be determined so the table can be correctly extracted. The "magic" value is followed by a 16 bit version number which allows the table to be revised and modified in the future. The current version number is 1. hash_function is a uint16_t enumeration that specifies which hash function was used to produce this table. The current values for the hash function enumerations include:

```
enum HashFunctionType
{
  eHashFunctionDJB = 0u, // Daniel J Bernstein hash function
};
```

bucket_count is a 32 bit unsigned integer that represents how many buckets are in the BUCKETS array. hashes_count is the number of unique 32 bit hash values that are in the HASHES array, and is the same number of offsets are contained in the OFFSETS array. header_data_len specifies the size in bytes of the HeaderData that is filled in by specialized versions of this table.

Fixed Lookup The header is followed by the buckets, hashes, offsets, and hash value data.

```
struct FixedTable
{
  uint32_t buckets[Header.bucket_count];  // An array of hash indexes into the "hashes[]" array belo
  uint32_t hashes [Header.hashes_count];  // Every unique 32 bit hash for the entire table is in thi
  uint32_t offsets[Header.hashes_count];  // An offset that corresponds to each item in the "hashes[
};
```

buckets is an array of 32 bit indexes into the hashes array. The hashes array contains all of the 32 bit hash values for all names in the hash table. Each hash in the hashes table has an offset in the offsets array that points to the data for the hash value.

This table setup makes it very easy to repurpose these tables to contain different data, while keeping the lookup mechanism the same for all tables. This layout also makes it possible to save the table to disk and map it in later and do very efficient name lookups with little or no parsing.

DWARF lookup tables can be implemented in a variety of ways and can store a lot of information for each name. We want to make the DWARF tables extensible and able to store the data efficiently so we have used some of the DWARF features that enable efficient data storage to define exactly what kind of data we store for each name.

The HeaderData contains a definition of the contents of each HashData chunk. We might want to store an offset to all of the debug information entries (DIEs) for each name. To keep things extensible, we create a list of items, or Atoms, that are contained in the data for each name. First comes the type of the data in each atom:

```
enum AtomType
{
  eAtomTypeNULL       = 0u,
  eAtomTypeDIEOffset  = 1u,   // DIE offset, check form for encoding
  eAtomTypeCUOffset   = 2u,   // DIE offset of the compiler unit header that contains the item in qu
  eAtomTypeTag        = 3u,   // DW_TAG_xxx value, should be encoded as DW_FORM_data1 (if no tags ex
  eAtomTypeNameFlags  = 4u,   // Flags from enum NameFlags
  eAtomTypeTypeFlags  = 5u,   // Flags from enum TypeFlags
};
```

The enumeration values and their meanings are:

```
eAtomTypeNULL       - a termination atom that specifies the end of the atom list
eAtomTypeDIEOffset  - an offset into the .debug_info section for the DWARF DIE for this name
eAtomTypeCUOffset   - an offset into the .debug_info section for the CU that contains the DIE
eAtomTypeDIETag     - The DW_TAG_XXX enumeration value so you don't have to parse the DWARF to see wh
eAtomTypeNameFlags  - Flags for functions and global variables (isFunction, isInlined, isExternal...)
eAtomTypeTypeFlags  - Flags for types (isCXXClass, isObjCClass, ...)
```

Then we allow each atom type to define the atom type and how the data for each atom type data is encoded:

```
struct Atom
{
  uint16_t type;  // AtomType enum value
  uint16_t form;  // DWARF DW_FORM_XXX defines
};
```

The `form` type above is from the DWARF specification and defines the exact encoding of the data for the Atom type. See the DWARF specification for the `DW_FORM_` definitions.

```
struct HeaderData
{
  uint32_t die_offset_base;
  uint32_t atom_count;
  Atoms    atoms[atom_count0];
};
```

`HeaderData` defines the base DIE offset that should be added to any atoms that are encoded using the `DW_FORM_ref1`, `DW_FORM_ref2`, `DW_FORM_ref4`, `DW_FORM_ref8` or `DW_FORM_ref_udata`. It also defines what is contained in each `HashData` object – `Atom.form` tells us how large each field will be in the `HashData` and the `Atom.type` tells us how this data should be interpreted.

For the current implementations of the "`.apple_names`" (all functions + globals), the "`.apple_types`" (names of all types that are defined), and the "`.apple_namespaces`" (all namespaces), we currently set the `Atom` array to be:

```
HeaderData.atom_count = 1;
HeaderData.atoms[0].type = eAtomTypeDIEOffset;
HeaderData.atoms[0].form = DW_FORM_data4;
```

This defines the contents to be the DIE offset (eAtomTypeDIEOffset) that is encoded as a 32 bit value (DW_FORM_data4). This allows a single name to have multiple matching DIEs in a single file, which could come up with an inlined function for instance. Future tables could include more information about the DIE such as flags indicating if the DIE is a function, method, block, or inlined.

The KeyType for the DWARF table is a 32 bit string table offset into the ".debug_str" table. The ".debug_str" is the string table for the DWARF which may already contain copies of all of the strings. This helps make sure, with help from the compiler, that we reuse the strings between all of the DWARF sections and keeps the hash table size down. Another benefit to having the compiler generate all strings as DW_FORM_strp in the debug info, is that DWARF parsing can be made much faster.

After a lookup is made, we get an offset into the hash data. The hash data needs to be able to deal with 32 bit hash collisions, so the chunk of data at the offset in the hash data consists of a triple:

```
uint32_t str_offset
uint32_t hash_data_count
HashData[hash_data_count]
```

If "str_offset" is zero, then the bucket contents are done. 99.9% of the hash data chunks contain a single item (no 32 bit hash collision):

```
.-------------.
| 0x00001023 | uint32_t KeyType (.debug_str[0x0001023] => "main")
| 0x00000004 | uint32_t HashData count
| 0x........ | uint32_t HashData[0] DIE offset
| 0x........ | uint32_t HashData[1] DIE offset
| 0x........ | uint32_t HashData[2] DIE offset
| 0x........ | uint32_t HashData[3] DIE offset
| 0x00000000 | uint32_t KeyType (end of hash chain)
`-------------'
```

If there are collisions, you will have multiple valid string offsets:

```
.-------------.
| 0x00001023 | uint32_t KeyType (.debug_str[0x0001023] => "main")
| 0x00000004 | uint32_t HashData count
| 0x........ | uint32_t HashData[0] DIE offset
| 0x........ | uint32_t HashData[1] DIE offset
| 0x........ | uint32_t HashData[2] DIE offset
| 0x........ | uint32_t HashData[3] DIE offset
| 0x00002023 | uint32_t KeyType (.debug_str[0x0002023] => "print")
| 0x00000002 | uint32_t HashData count
| 0x........ | uint32_t HashData[0] DIE offset
| 0x........ | uint32_t HashData[1] DIE offset
| 0x00000000 | uint32_t KeyType (end of hash chain)
`-------------'
```

Current testing with real world C++ binaries has shown that there is around 1 32 bit hash collision per 100,000 name entries.

Contents

As we said, we want to strictly define exactly what is included in the different tables. For DWARF, we have 3 tables: "`.apple_names`", "`.apple_types`", and "`.apple_namespaces`".

"`.apple_names`" sections should contain an entry for each DWARF DIE whose `DW_TAG` is a `DW_TAG_label`, `DW_TAG_inlined_subroutine`, or `DW_TAG_subprogram` that has address attributes: `DW_AT_low_pc`, `DW_AT_high_pc`, `DW_AT_ranges` or `DW_AT_entry_pc`. It also contains `DW_TAG_variable` DIEs that have a `DW_OP_addr` in the location (global and static variables). All global and static variables should be included, including those scoped within functions and classes. For example using the following code:

```
static int var = 0;

void f ()
{
  static int var = 0;
}
```

Both of the static `var` variables would be included in the table. All functions should emit both their full names and their basenames. For C or C++, the full name is the mangled name (if available) which is usually in the `DW_AT_MIPS_linkage_name` attribute, and the `DW_AT_name` contains the function basename. If global or static variables have a mangled name in a `DW_AT_MIPS_linkage_name` attribute, this should be emitted along with the simple name found in the `DW_AT_name` attribute.

"`.apple_types`" sections should contain an entry for each DWARF DIE whose tag is one of:

- DW_TAG_array_type
- DW_TAG_class_type

- DW_TAG_enumeration_type
- DW_TAG_pointer_type
- DW_TAG_reference_type
- DW_TAG_string_type
- DW_TAG_structure_type
- DW_TAG_subroutine_type
- DW_TAG_typedef
- DW_TAG_union_type
- DW_TAG_ptr_to_member_type
- DW_TAG_set_type
- DW_TAG_subrange_type
- DW_TAG_base_type
- DW_TAG_const_type
- DW_TAG_constant
- DW_TAG_file_type
- DW_TAG_namelist
- DW_TAG_packed_type
- DW_TAG_volatile_type
- DW_TAG_restrict_type
- DW_TAG_interface_type
- DW_TAG_unspecified_type
- DW_TAG_shared_type

Only entries with a `DW_AT_name` attribute are included, and the entry must not be a forward declaration (`DW_AT_declaration` attribute with a non-zero value). For example, using the following code:

```
int main ()
{
  int *b = 0;
  return *b;
}
```

We get a few type DIEs:

```
0x00000067:     TAG_base_type [5]
                AT_encoding( DW_ATE_signed )
                AT_name( "int" )
                AT_byte_size( 0x04 )

0x0000006e:     TAG_pointer_type [6]
                AT_type( {0x00000067} ( int ) )
                AT_byte_size( 0x08 )
```

The DW_TAG_pointer_type is not included because it does not have a `DW_AT_name`.

"`.apple_namespaces`" section should contain all `DW_TAG_namespace` DIEs. If we run into a namespace that has no name this is an anonymous namespace, and the name should be output as "`(anonymous namespace)`"

(without the quotes). Why? This matches the output of the `abi::cxa_demangle()` that is in the standard C++ library that demangles mangled names.

Language Extensions and File Format Changes

Objective-C Extensions "`.apple_objc`" section should contain all `DW_TAG_subprogram` DIEs for an Objective-C class. The name used in the hash table is the name of the Objective-C class itself. If the Objective-C class has a category, then an entry is made for both the class name without the category, and for the class name with the category. So if we have a DIE at offset 0x1234 with a name of method "`-[NSString(my_additions) stringWithSpecialString:]`", we would add an entry for "`NSString`" that points to DIE 0x1234, and an entry for "`NSString(my_additions)`" that points to 0x1234. This allows us to quickly track down all Objective-C methods for an Objective-C class when doing expressions. It is needed because of the dynamic nature of Objective-C where anyone can add methods to a class. The DWARF for Objective-C methods is also emitted differently from C++ classes where the methods are not usually contained in the class definition, they are scattered about across one or more compile units. Categories can also be defined in different shared libraries. So we need to be able to quickly find all of the methods and class functions given the Objective-C class name, or quickly find all methods and class functions for a class + category name. This table does not contain any selector names, it just maps Objective-C class names (or class names + category) to all of the methods and class functions. The selectors are added as function basenames in the "`.debug_names`" section.

In the "`.apple_names`" section for Objective-C functions, the full name is the entire function name with the brackets ("`-[NSString stringWithCString:]`") and the basename is the selector only ("`stringWithCString:`").

Mach-O Changes The sections names for the apple hash tables are for non-mach-o files. For mach-o files, the sections should be contained in the `__DWARF` segment with names as follows:

- "`.apple_names`" -> "`__apple_names`"
- "`.apple_types`" -> "`__apple_types`"
- "`.apple_namespaces`" -> "`__apple_namespac`" (16 character limit)
- "`.apple_objc`" -> "`__apple_objc`"

4.17 Auto-Vectorization in LLVM

LLVM has two vectorizers: The *Loop Vectorizer*, which operates on Loops, and the *SLP Vectorizer*. These vectorizers focus on different optimization opportunities and use different techniques. The SLP vectorizer merges multiple scalars that are found in the code into vectors while the Loop Vectorizer widens instructions in loops to operate on multiple consecutive iterations.

Both the Loop Vectorizer and the SLP Vectorizer are enabled by default.

4.17.1 The Loop Vectorizer

Usage

The Loop Vectorizer is enabled by default, but it can be disabled through clang using the command line flag:

```
$ clang ... -fno-vectorize  file.c
```

Command line flags

The loop vectorizer uses a cost model to decide on the optimal vectorization factor and unroll factor. However, users of the vectorizer can force the vectorizer to use specific values. Both 'clang' and 'opt' support the flags below.

Users can control the vectorization SIMD width using the command line flag "-force-vector-width".

```
$ clang  -mllvm -force-vector-width=8 ...
$ opt -loop-vectorize -force-vector-width=8 ...
```

Users can control the unroll factor using the command line flag "-force-vector-unroll"

```
$ clang  -mllvm -force-vector-unroll=2 ...
$ opt -loop-vectorize -force-vector-unroll=2 ...
```

Pragma loop hint directives

The `#pragma clang loop` directive allows loop vectorization hints to be specified for the subsequent for, while, do-while, or c++11 range-based for loop. The directive allows vectorization and interleaving to be enabled or disabled. Vector width as well as interleave count can also be manually specified. The following example explicitly enables vectorization and interleaving:

```
#pragma clang loop vectorize(enable) interleave(enable)
while(...) {
  ...
}
```

The following example implicitly enables vectorization and interleaving by specifying a vector width and interleaving count:

```
#pragma clang loop vectorize_width(2) interleave_count(2)
for(...) {
  ...
}
```

See the Clang language extensions for details.

Diagnostics

Many loops cannot be vectorized including loops with complicated control flow, unvectorizable types, and unvectorizable calls. The loop vectorizer generates optimization remarks which can be queried using command line options to identify and diagnose loops that are skipped by the loop-vectorizer.

Optimization remarks are enabled using:

`-Rpass=loop-vectorize` identifies loops that were successfully vectorized.

`-Rpass-missed=loop-vectorize` identifies loops that failed vectorization and indicates if vectorization was specified.

`-Rpass-analysis=loop-vectorize` identifies the statements that caused vectorization to fail.

Consider the following loop:

```
#pragma clang loop vectorize(enable)
for (int i = 0; i < Length; i++) {
  switch(A[i]) {
  case 0: A[i] = i*2; break;
  case 1: A[i] = i;   break;
  default: A[i] = 0;
  }
}
```

The command line `-Rpass-missed=loop-vectorized` prints the remark:

```
no_switch.cpp:4:5: remark: loop not vectorized: vectorization is explicitly enabled [-Rpass-missed=lo
```

And the command line `-Rpass-analysis=loop-vectorize` indicates that the switch statement cannot be vectorized.

```
no_switch.cpp:4:5: remark: loop not vectorized: loop contains a switch statement [-Rpass-analysis=loo
  switch(A[i]) {
  ^
```

To ensure line and column numbers are produced include the command line options `-gline-tables-only` and `-gcolumn-info`. See the Clang user manual for details

Features

The LLVM Loop Vectorizer has a number of features that allow it to vectorize complex loops.

Loops with unknown trip count

The Loop Vectorizer supports loops with an unknown trip count. In the loop below, the iteration `start` and `finish` points are unknown, and the Loop Vectorizer has a mechanism to vectorize loops that do not start at zero. In this example, 'n' may not be a multiple of the vector width, and the vectorizer has to execute the last few iterations as scalar code. Keeping a scalar copy of the loop increases the code size.

```
void bar(float *A, float* B, float K, int start, int end) {
  for (int i = start; i < end; ++i)
    A[i] *= B[i] + K;
}
```

Runtime Checks of Pointers

In the example below, if the pointers A and B point to consecutive addresses, then it is illegal to vectorize the code because some elements of A will be written before they are read from array B.

Some programmers use the 'restrict' keyword to notify the compiler that the pointers are disjointed, but in our example, the Loop Vectorizer has no way of knowing that the pointers A and B are unique. The Loop Vectorizer handles this loop by placing code that checks, at runtime, if the arrays A and B point to disjointed memory locations. If arrays A and B overlap, then the scalar version of the loop is executed.

```
void bar(float *A, float* B, float K, int n) {
  for (int i = 0; i < n; ++i)
    A[i] *= B[i] + K;
}
```

Reductions

In this example the `sum` variable is used by consecutive iterations of the loop. Normally, this would prevent vectorization, but the vectorizer can detect that 'sum' is a reduction variable. The variable 'sum' becomes a vector of integers, and at the end of the loop the elements of the array are added together to create the correct result. We support a number of different reduction operations, such as addition, multiplication, XOR, AND and OR.

```
int foo(int *A, int *B, int n) {
  unsigned sum = 0;
  for (int i = 0; i < n; ++i)
    sum += A[i] + 5;
  return sum;
}
```

We support floating point reduction operations when *-ffast-math* is used.

Inductions

In this example the value of the induction variable `i` is saved into an array. The Loop Vectorizer knows to vectorize induction variables.

```
void bar(float *A, float* B, float K, int n) {
  for (int i = 0; i < n; ++i)
    A[i] = i;
}
```

If Conversion

The Loop Vectorizer is able to "flatten" the IF statement in the code and generate a single stream of instructions. The Loop Vectorizer supports any control flow in the innermost loop. The innermost loop may contain complex nesting of IFs, ELSEs and even GOTOs.

```
int foo(int *A, int *B, int n) {
  unsigned sum = 0;
  for (int i = 0; i < n; ++i)
    if (A[i] > B[i])
      sum += A[i] + 5;
  return sum;
}
```

Pointer Induction Variables

This example uses the "accumulate" function of the standard c++ library. This loop uses C++ iterators, which are pointers, and not integer indices. The Loop Vectorizer detects pointer induction variables and can vectorize this loop. This feature is important because many C++ programs use iterators.

```
int baz(int *A, int n) {
  return std::accumulate(A, A + n, 0);
}
```

Reverse Iterators

The Loop Vectorizer can vectorize loops that count backwards.

```
int foo(int *A, int *B, int n) {
  for (int i = n; i > 0; --i)
    A[i] +=1;
}
```

Scatter / Gather

The Loop Vectorizer can vectorize code that becomes a sequence of scalar instructions that scatter/gathers memory.

```
int foo(int * A, int * B, int n) {
  for (intptr_t i = 0; i < n; ++i)
      A[i] += B[i * 4];
}
```

In many situations the cost model will inform LLVM that this is not beneficial and LLVM will only vectorize such code if forced with "-mllvm -force-vector-width=#".

Vectorization of Mixed Types

The Loop Vectorizer can vectorize programs with mixed types. The Vectorizer cost model can estimate the cost of the type conversion and decide if vectorization is profitable.

```
int foo(int *A, char *B, int n, int k) {
  for (int i = 0; i < n; ++i)
    A[i] += 4 * B[i];
}
```

Global Structures Alias Analysis

Access to global structures can also be vectorized, with alias analysis being used to make sure accesses don't alias. Run-time checks can also be added on pointer access to structure members.

Many variations are supported, but some that rely on undefined behaviour being ignored (as other compilers do) are still being left un-vectorized.

```
struct { int A[100], K, B[100]; } Foo;

int foo() {
  for (int i = 0; i < 100; ++i)
    Foo.A[i] = Foo.B[i] + 100;
}
```

Vectorization of function calls

The Loop Vectorize can vectorize intrinsic math functions. See the table below for a list of these functions.

pow	exp	exp2
sin	cos	sqrt
log	log2	log10
fabs	floor	ceil
fma	trunc	nearbyint
		fmuladd

The loop vectorizer knows about special instructions on the target and will vectorize a loop containing a function call that maps to the instructions. For example, the loop below will be vectorized on Intel x86 if the SSE4.1 roundps instruction is available.

```
void foo(float *f) {
  for (int i = 0; i != 1024; ++i)
    f[i] = floorf(f[i]);
}
```

Partial unrolling during vectorization

Modern processors feature multiple execution units, and only programs that contain a high degree of parallelism can fully utilize the entire width of the machine. The Loop Vectorizer increases the instruction level parallelism (ILP) by performing partial-unrolling of loops.

In the example below the entire array is accumulated into the variable 'sum'. This is inefficient because only a single execution port can be used by the processor. By unrolling the code the Loop Vectorizer allows two or more execution ports to be used simultaneously.

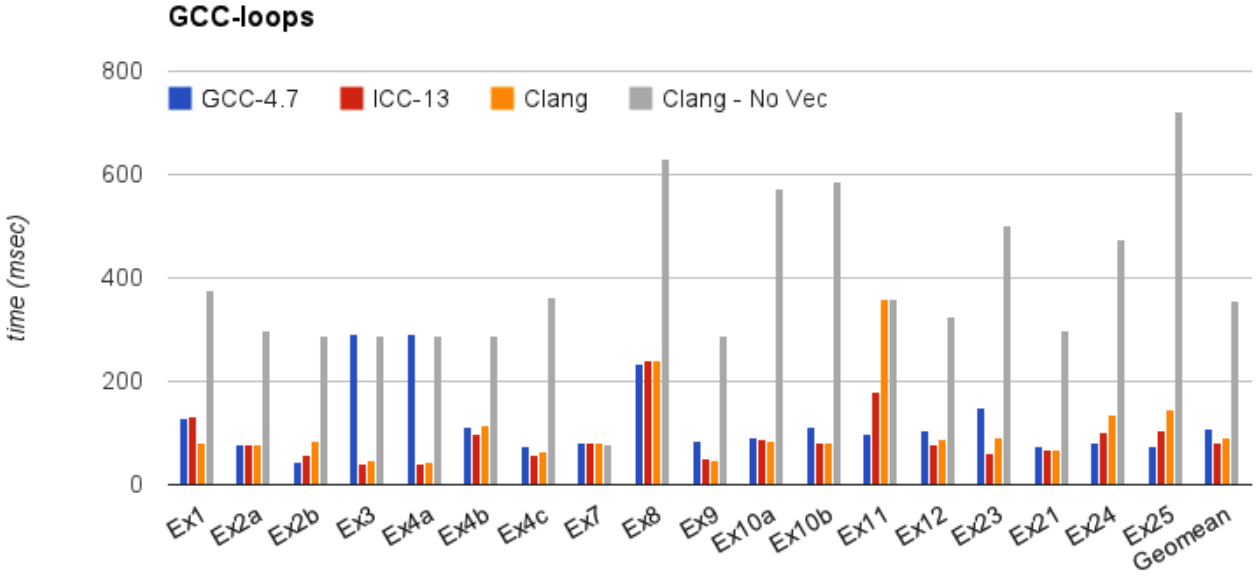
```
int foo(int *A, int *B, int n) {
  unsigned sum = 0;
  for (int i = 0; i < n; ++i)
      sum += A[i];
  return sum;
}
```

The Loop Vectorizer uses a cost model to decide when it is profitable to unroll loops. The decision to unroll the loop depends on the register pressure and the generated code size.

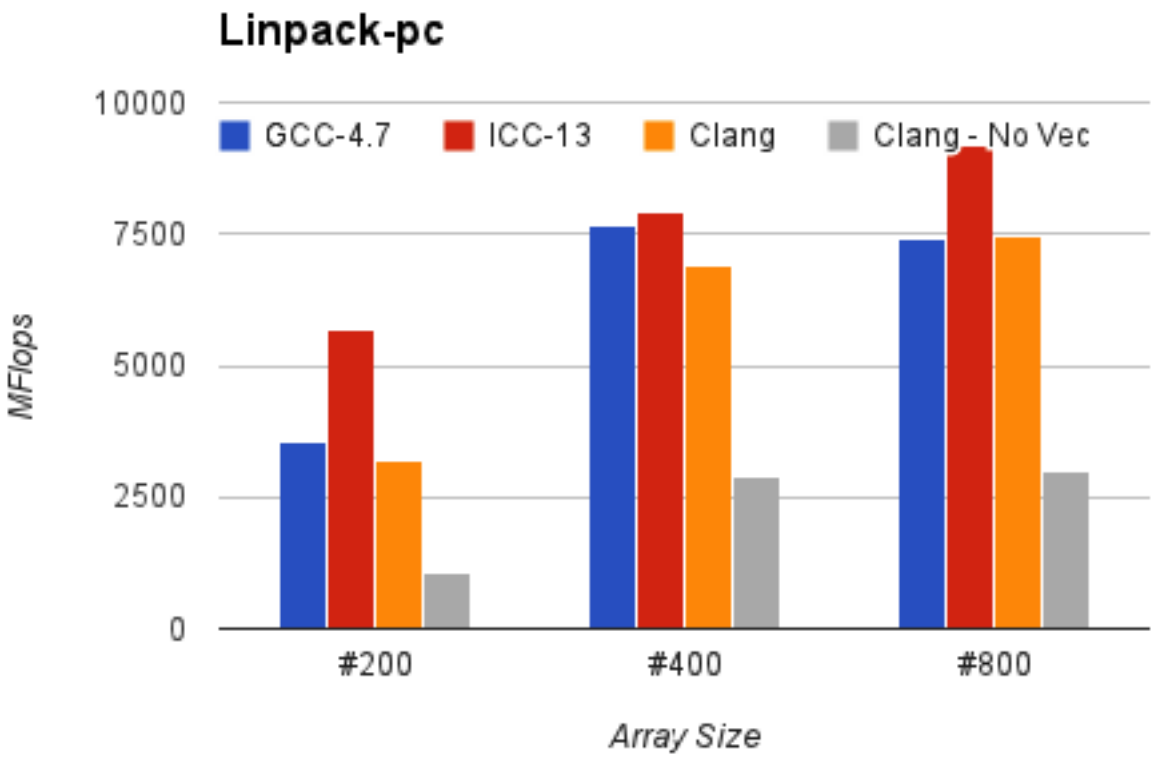
Performance

This section shows the the execution time of Clang on a simple benchmark: gcc-loops. This benchmarks is a collection of loops from the GCC autovectorization page by Dorit Nuzman.

The chart below compares GCC-4.7, ICC-13, and Clang-SVN with and without loop vectorization at -O3, tuned for "corei7-avx", running on a Sandybridge iMac. The Y-axis shows the time in msec. Lower is better. The last column shows the geomean of all the kernels.



And Linpack-pc with the same configuration. Result is Mflops, higher is better.



4.17.2 The SLP Vectorizer

Details

The goal of SLP vectorization (a.k.a. superword-level parallelism) is to combine similar independent instructions into vector instructions. Memory accesses, arithmetic operations, comparison operations, PHI-nodes, can all be vectorized using this technique.

For example, the following function performs very similar operations on its inputs (a1, b1) and (a2, b2). The basic-block vectorizer may combine these into vector operations.

```
void foo(int a1, int a2, int b1, int b2, int *A) {
  A[0] = a1*(a1 + b1)/b1 + 50*b1/a1;
  A[1] = a2*(a2 + b2)/b2 + 50*b2/a2;
}
```

The SLP-vectorizer processes the code bottom-up, across basic blocks, in search of scalars to combine.

Usage

The SLP Vectorizer is enabled by default, but it can be disabled through clang using the command line flag:

```
$ clang -fno-slp-vectorize file.c
```

LLVM has a second basic block vectorization phase which is more compile-time intensive (The BB vectorizer). This optimization can be enabled through clang using the command line flag:

```
$ clang -fslp-vectorize-aggressive file.c
```

4.18 Writing an LLVM Backend

4.18.1 How To Use Instruction Mappings

Introduction

This document contains information about adding instruction mapping support for a target. The motivation behind this feature comes from the need to switch between different instruction formats during various optimizations. One approach could be to use switch cases which list all the instructions along with formats they can transition to. However, it has large maintenance overhead because of the hardcoded instruction names. Also, whenever a new instruction is added in the .td files, all the relevant switch cases should be modified accordingly. Instead, the same functionality could be achieved with TableGen and some support from the .td files for a fraction of maintenance cost.

`InstrMapping` Class Overview

TableGen uses relationship models to map instructions with each other. These models are described using `InstrMapping` class as a base. Each model sets various fields of the `InstrMapping` class such that they can uniquely describe all the instructions using that model. TableGen parses all the relation models and uses the information to construct relation tables which relate instructions with each other. These tables are emitted in the `XXXInstrInfo.inc` file along with the functions to query them. Following is the definition of `InstrMapping` class definied in Target.td file:

```
class InstrMapping {
  // Used to reduce search space only to the instructions using this
  // relation model.
  string FilterClass;

  // List of fields/attributes that should be same for all the instructions in
  // a row of the relation table. Think of this as a set of properties shared
  // by all the instructions related by this relationship.
  list<string> RowFields = [];

  // List of fields/attributes that are same for all the instructions
  // in a column of the relation table.
  list<string> ColFields = [];

  // Values for the fields/attributes listed in 'ColFields' corresponding to
  // the key instruction. This is the instruction that will be transformed
  // using this relation model.
  list<string> KeyCol = [];

  // List of values for the fields/attributes listed in 'ColFields', one for
  // each column in the relation table. These are the instructions a key
  // instruction will be transformed into.
  list<list<string> > ValueCols = [];
}
```

Sample Example

Let's say that we want to have a function `int getPredOpcode(uint16_t Opcode, enum PredSense inPredSense)` which takes a non-predicated instruction and returns its predicated true or false form depending on some input flag, `inPredSense`. The first step in the process is to define a relationship model that relates predicated instructions to their non-predicated form by assigning appropriate values to the `InstrMapping` fields. For this relationship, non-predicated instructions are treated as key instruction since they are the one used to query the interface function.

```
def getPredOpcode : InstrMapping {
  // Choose a FilterClass that is used as a base class for all the
  // instructions modeling this relationship. This is done to reduce the
  // search space only to these set of instructions.
  let FilterClass = "PredRel";

  // Instructions with same values for all the fields in RowFields form a
  // row in the resulting relation table.
  // For example, if we want to relate 'ADD' (non-predicated) with 'Add_pt'
  // (predicated true) and 'Add_pf' (predicated false), then all 3
  // instructions need to have same value for BaseOpcode field. It can be any
  // unique value (Ex: XYZ) and should not be shared with any other
  // instruction not related to 'add'.
  let RowFields = ["BaseOpcode"];

  // List of attributes that can be used to define key and column instructions
  // for a relation. Key instruction is passed as an argument
  // to the function used for querying relation tables. Column instructions
  // are the instructions they (key) can transform into.
  //
  // Here, we choose 'PredSense' as ColFields since this is the unique
  // attribute of the key (non-predicated) and column (true/false)
  // instructions involved in this relationship model.
  let ColFields = ["PredSense"];

  // The key column contains non-predicated instructions.
  let KeyCol = ["none"];

  // Two value columns - first column contains instructions with
  // PredSense=true while second column has instructions with PredSense=false.
  let ValueCols = [["true"], ["false"]];
}
```

TableGen uses the above relationship model to emit relation table that maps non-predicated instructions with their predicated forms. It also outputs the interface function `int getPredOpcode(uint16_t Opcode, enum PredSense inPredSense)` to query the table. Here, Function `getPredOpcode` takes two arguments, opcode of the current instruction and PredSense of the desired instruction, and returns predicated form of the instruction, if found in the relation table. In order for an instruction to be added into the relation table, it needs to include relevant information in its definition. For example, consider following to be the current definitions of ADD, ADD_pt (true) and ADD_pf (false) instructions:

```
def ADD : ALU32_rr<(outs IntRegs:$dst), (ins IntRegs:$a, IntRegs:$b),
            "$dst = add($a, $b)",
            [(set (i32 IntRegs:$dst), (add (i32 IntRegs:$a),
                                           (i32 IntRegs:$b)))]>;

def ADD_Pt : ALU32_rr<(outs IntRegs:$dst),
                      (ins PredRegs:$p, IntRegs:$a, IntRegs:$b),
            "if ($p) $dst = add($a, $b)",
```

```
            []>;

def ADD_Pf : ALU32_rr<(outs IntRegs:$dst),
                      (ins PredRegs:$p, IntRegs:$a, IntRegs:$b),
            "if (!$p) $dst = add($a, $b)",
            []>;
```

In this step, we modify these instructions to include the information required by the relationship model, <tt>getPredOpcode</tt>, so that they can be related.

```
def ADD : PredRel, ALU32_rr<(outs IntRegs:$dst), (ins IntRegs:$a, IntRegs:$b),
            "$dst = add($a, $b)",
            [(set (i32 IntRegs:$dst), (add (i32 IntRegs:$a),
                                           (i32 IntRegs:$b)))]> {
  let BaseOpcode = "ADD";
  let PredSense = "none";
}

def ADD_Pt : PredRel, ALU32_rr<(outs IntRegs:$dst),
                      (ins PredRegs:$p, IntRegs:$a, IntRegs:$b),
            "if ($p) $dst = add($a, $b)",
            []> {
  let BaseOpcode = "ADD";
  let PredSense = "true";
}

def ADD_Pf : PredRel, ALU32_rr<(outs IntRegs:$dst),
                      (ins PredRegs:$p, IntRegs:$a, IntRegs:$b),
            "if (!$p) $dst = add($a, $b)",
            []> {
  let BaseOpcode = "ADD";
  let PredSense = "false";
}
```

Please note that all the above instructions use `PredRel` as a base class. This is extremely important since TableGen uses it as a filter for selecting instructions for `getPredOpcode` model. Any instruction not derived from `PredRel` is excluded from the analysis. `BaseOpcode` is another important field. Since it's selected as a `RowFields` of the model, it is required to have the same value for all 3 instructions in order to be related. Next, `PredSense` is used to determine their column positions by comparing its value with `KeyCol` and `ValueCols`. If an instruction sets its `PredSense` value to something not used in the relation model, it will not be assigned a column in the relation table.

4.18.2 Introduction

This document describes techniques for writing compiler backends that convert the LLVM Intermediate Representation (IR) to code for a specified machine or other languages. Code intended for a specific machine can take the form of either assembly code or binary code (usable for a JIT compiler).

The backend of LLVM features a target-independent code generator that may create output for several types of target CPUs — including X86, PowerPC, ARM, and SPARC. The backend may also be used to generate code targeted at SPUs of the Cell processor or GPUs to support the execution of compute kernels.

The document focuses on existing examples found in subdirectories of `llvm/lib/Target` in a downloaded LLVM release. In particular, this document focuses on the example of creating a static compiler (one that emits text assembly) for a SPARC target, because SPARC has fairly standard characteristics, such as a RISC instruction set and straightforward calling conventions.

Audience

The audience for this document is anyone who needs to write an LLVM backend to generate code for a specific hardware or software target.

Prerequisite Reading

These essential documents must be read before reading this document:

- LLVM Language Reference Manual — a reference manual for the LLVM assembly language.
- *The LLVM Target-Independent Code Generator* — a guide to the components (classes and code generation algorithms) for translating the LLVM internal representation into machine code for a specified target. Pay particular attention to the descriptions of code generation stages: Instruction Selection, Scheduling and Formation, SSA-based Optimization, Register Allocation, Prolog/Epilog Code Insertion, Late Machine Code Optimizations, and Code Emission.
- *TableGen* — a document that describes the TableGen (`tblgen`) application that manages domain-specific information to support LLVM code generation. TableGen processes input from a target description file (`.td` suffix) and generates C++ code that can be used for code generation.
- *Writing an LLVM Pass* — The assembly printer is a `FunctionPass`, as are several `SelectionDAG` processing steps.

To follow the SPARC examples in this document, have a copy of The SPARC Architecture Manual, Version 8 for reference. For details about the ARM instruction set, refer to the ARM Architecture Reference Manual. For more about the GNU Assembler format (`GAS`), see Using As, especially for the assembly printer. "Using As" contains a list of target machine dependent features.

Basic Steps

To write a compiler backend for LLVM that converts the LLVM IR to code for a specified target (machine or other language), follow these steps:

- Create a subclass of the `TargetMachine` class that describes characteristics of your target machine. Copy existing examples of specific `TargetMachine` class and header files; for example, start with `SparcTargetMachine.cpp` and `SparcTargetMachine.h`, but change the file names for your target. Similarly, change code that references "`Sparc`" to reference your target.
- Describe the register set of the target. Use TableGen to generate code for register definition, register aliases, and register classes from a target-specific `RegisterInfo.td` input file. You should also write additional code for a subclass of the `TargetRegisterInfo` class that represents the class register file data used for register allocation and also describes the interactions between registers.
- Describe the instruction set of the target. Use TableGen to generate code for target-specific instructions from target-specific versions of `TargetInstrFormats.td` and `TargetInstrInfo.td`. You should write additional code for a subclass of the `TargetInstrInfo` class to represent machine instructions supported by the target machine.
- Describe the selection and conversion of the LLVM IR from a Directed Acyclic Graph (DAG) representation of instructions to native target-specific instructions. Use TableGen to generate code that matches patterns and selects instructions based on additional information in a target-specific version of `TargetInstrInfo.td`. Write code for `XXXISelDAGToDAG.cpp`, where `XXX` identifies the specific target, to perform pattern matching and DAG-to-DAG instruction selection. Also write code in `XXXISelLowering.cpp` to replace or remove operations and data types that are not supported natively in a SelectionDAG.
- Write code for an assembly printer that converts LLVM IR to a GAS format for your target machine. You should add assembly strings to the instructions defined in your target-specific version of `TargetInstrInfo.td`. You should also write code for a subclass of `AsmPrinter` that performs the LLVM-to-assembly conversion and a trivial subclass of `TargetAsmInfo`.
- Optionally, add support for subtargets (i.e., variants with different capabilities). You should also write code for a subclass of the `TargetSubtarget` class, which allows you to use the `-mcpu=` and `-mattr=` command-line options.

- Optionally, add JIT support and create a machine code emitter (subclass of `TargetJITInfo`) that is used to emit binary code directly into memory.

In the `.cpp` and `.h` files, initially stub up these methods and then implement them later. Initially, you may not know which private members that the class will need and which components will need to be subclassed.

Preliminaries

To actually create your compiler backend, you need to create and modify a few files. The absolute minimum is discussed here. But to actually use the LLVM target-independent code generator, you must perform the steps described in the *LLVM Target-Independent Code Generator* document.

First, you should create a subdirectory under `lib/Target` to hold all the files related to your target. If your target is called "Dummy", create the directory `lib/Target/Dummy`.

In this new directory, create a `Makefile`. It is easiest to copy a `Makefile` of another target and modify it. It should at least contain the `LEVEL`, `LIBRARYNAME` and `TARGET` variables, and then include `$(LEVEL)/Makefile.common`. The library can be named `LLVMDummy` (for example, see the MIPS target). Alternatively, you can split the library into `LLVMDummyCodeGen` and `LLVMDummyAsmPrinter`, the latter of which should be implemented in a subdirectory below `lib/Target/Dummy` (for example, see the PowerPC target).

Note that these two naming schemes are hardcoded into `llvm-config`. Using any other naming scheme will confuse `llvm-config` and produce a lot of (seemingly unrelated) linker errors when linking `llc`.

To make your target actually do something, you need to implement a subclass of `TargetMachine`. This implementation should typically be in the file `lib/Target/DummyTargetMachine.cpp`, but any file in the `lib/Target` directory will be built and should work. To use LLVM's target independent code generator, you should do what all current machine backends do: create a subclass of `LLVMTargetMachine`. (To create a target from scratch, create a subclass of `TargetMachine`.)

To get LLVM to actually build and link your target, you need to add it to the `TARGETS_TO_BUILD` variable. To do this, you modify the configure script to know about your target when parsing the `--enable-targets` option. Search the configure script for `TARGETS_TO_BUILD`, add your target to the lists there (some creativity required), and then reconfigure. Alternatively, you can change `autotools/configure.ac` and regenerate configure by running `./autoconf/AutoRegen.sh`.

4.18.3 Target Machine

`LLVMTargetMachine` is designed as a base class for targets implemented with the LLVM target-independent code generator. The `LLVMTargetMachine` class should be specialized by a concrete target class that implements the various virtual methods. `LLVMTargetMachine` is defined as a subclass of `TargetMachine` in `include/llvm/Target/TargetMachine.h`. The `TargetMachine` class implementation (`TargetMachine.cpp`) also processes numerous command-line options.

To create a concrete target-specific subclass of `LLVMTargetMachine`, start by copying an existing `TargetMachine` class and header. You should name the files that you create to reflect your specific target. For instance, for the SPARC target, name the files `SparcTargetMachine.h` and `SparcTargetMachine.cpp`.

For a target machine `XXX`, the implementation of `XXXTargetMachine` must have access methods to obtain objects that represent target components. These methods are named `get*Info`, and are intended to obtain the instruction set (`getInstrInfo`), register set (`getRegisterInfo`), stack frame layout (`getFrameInfo`), and similar information. `XXXTargetMachine` must also implement the `getDataLayout` method to access an object with target-specific data characteristics, such as data type size and alignment requirements.

For instance, for the SPARC target, the header file `SparcTargetMachine.h` declares prototypes for several `get*Info` and `getDataLayout` methods that simply return a class member.

```
namespace llvm {

class Module;

class SparcTargetMachine : public LLVMTargetMachine {
  const DataLayout DataLayout;       // Calculates type size & alignment
  SparcSubtarget Subtarget;
  SparcInstrInfo InstrInfo;
  TargetFrameInfo FrameInfo;

protected:
  virtual const TargetAsmInfo *createTargetAsmInfo() const;

public:
  SparcTargetMachine(const Module &M, const std::string &FS);

  virtual const SparcInstrInfo *getInstrInfo() const {return &InstrInfo; }
  virtual const TargetFrameInfo *getFrameInfo() const {return &FrameInfo; }
  virtual const TargetSubtarget *getSubtargetImpl() const{return &Subtarget; }
  virtual const TargetRegisterInfo *getRegisterInfo() const {
    return &InstrInfo.getRegisterInfo();
  }
  virtual const DataLayout *getDataLayout() const { return &DataLayout; }
  static unsigned getModuleMatchQuality(const Module &M);

  // Pass Pipeline Configuration
  virtual bool addInstSelector(PassManagerBase &PM, bool Fast);
  virtual bool addPreEmitPass(PassManagerBase &PM, bool Fast);
};

} // end namespace llvm
```

- `getInstrInfo()`
- `getRegisterInfo()`
- `getFrameInfo()`
- `getDataLayout()`
- `getSubtargetImpl()`

For some targets, you also need to support the following methods:

- `getTargetLowering()`
- `getJITInfo()`

Some architectures, such as GPUs, do not support jumping to an arbitrary program location and implement branching using masked execution and loop using special instructions around the loop body. In order to avoid CFG modifications that introduce irreducible control flow not handled by such hardware, a target must call *setRequiresStructuredCFG(true)* when being initialized.

In addition, the `XXXTargetMachine` constructor should specify a `TargetDescription` string that determines the data layout for the target machine, including characteristics such as pointer size, alignment, and endianness. For example, the constructor for `SparcTargetMachine` contains the following:

```
SparcTargetMachine::SparcTargetMachine(const Module &M, const std::string &FS)
  : DataLayout("E-p:32:32-f128:128:128"),
    Subtarget(M, FS), InstrInfo(Subtarget),
```

```
    FrameInfo(TargetFrameInfo::StackGrowsDown, 8, 0) {
}
```

Hyphens separate portions of the `TargetDescription` string.

- An upper-case "`E`" in the string indicates a big-endian target data model. A lower-case "`e`" indicates little-endian.
- "`p:`" is followed by pointer information: size, ABI alignment, and preferred alignment. If only two figures follow "`p:`", then the first value is pointer size, and the second value is both ABI and preferred alignment.
- Then a letter for numeric type alignment: "`i`", "`f`", "`v`", or "`a`" (corresponding to integer, floating point, vector, or aggregate). "`i`", "`v`", or "`a`" are followed by ABI alignment and preferred alignment. "`f`" is followed by three values: the first indicates the size of a long double, then ABI alignment, and then ABI preferred alignment.

4.18.4 Target Registration

You must also register your target with the `TargetRegistry`, which is what other LLVM tools use to be able to lookup and use your target at runtime. The `TargetRegistry` can be used directly, but for most targets there are helper templates which should take care of the work for you.

All targets should declare a global `Target` object which is used to represent the target during registration. Then, in the target's `TargetInfo` library, the target should define that object and use the `RegisterTarget` template to register the target. For example, the Sparc registration code looks like this:

```
Target llvm::TheSparcTarget;

extern "C" void LLVMInitializeSparcTargetInfo() {
  RegisterTarget<Triple::sparc, /*HasJIT=*/false>
    X(TheSparcTarget, "sparc", "Sparc");
}
```

This allows the `TargetRegistry` to look up the target by name or by target triple. In addition, most targets will also register additional features which are available in separate libraries. These registration steps are separate, because some clients may wish to only link in some parts of the target — the JIT code generator does not require the use of the assembler printer, for example. Here is an example of registering the Sparc assembly printer:

```
extern "C" void LLVMInitializeSparcAsmPrinter() {
  RegisterAsmPrinter<SparcAsmPrinter> X(TheSparcTarget);
}
```

For more information, see "llvm/Target/TargetRegistry.h".

4.18.5 Register Set and Register Classes

You should describe a concrete target-specific class that represents the register file of a target machine. This class is called `XXXRegisterInfo` (where `XXX` identifies the target) and represents the class register file data that is used for register allocation. It also describes the interactions between registers.

You also need to define register classes to categorize related registers. A register class should be added for groups of registers that are all treated the same way for some instruction. Typical examples are register classes for integer, floating-point, or vector registers. A register allocator allows an instruction to use any register in a specified register class to perform the instruction in a similar manner. Register classes allocate virtual registers to instructions from these sets, and register classes let the target-independent register allocator automatically choose the actual registers.

Much of the code for registers, including register definition, register aliases, and register classes, is generated by TableGen from `XXXRegisterInfo.td` input files and placed in `XXXGenRegisterInfo.h.inc` and

XXXGenRegisterInfo.inc output files. Some of the code in the implementation of XXXRegisterInfo requires hand-coding.

Defining a Register

The XXXRegisterInfo.td file typically starts with register definitions for a target machine. The Register class (specified in Target.td) is used to define an object for each register. The specified string n becomes the Name of the register. The basic Register object does not have any subregisters and does not specify any aliases.

```
class Register<string n> {
  string Namespace = "";
  string AsmName = n;
  string Name = n;
  int SpillSize = 0;
  int SpillAlignment = 0;
  list<Register> Aliases = [];
  list<Register> SubRegs = [];
  list<int> DwarfNumbers = [];
}
```

For example, in the X86RegisterInfo.td file, there are register definitions that utilize the Register class, such as:

```
def AL : Register<"AL">, DwarfRegNum<[0, 0, 0]>;
```

This defines the register AL and assigns it values (with DwarfRegNum) that are used by gcc, gdb, or a debug information writer to identify a register. For register AL, DwarfRegNum takes an array of 3 values representing 3 different modes: the first element is for X86-64, the second for exception handling (EH) on X86-32, and the third is generic. -1 is a special Dwarf number that indicates the gcc number is undefined, and -2 indicates the register number is invalid for this mode.

From the previously described line in the X86RegisterInfo.td file, TableGen generates this code in the X86GenRegisterInfo.inc file:

```
static const unsigned GR8[] = { X86::AL, ... };

const unsigned AL_AliasSet[] = { X86::AX, X86::EAX, X86::RAX, 0 };

const TargetRegisterDesc RegisterDescriptors[] = {
  ...
{ "AL", "AL", AL_AliasSet, Empty_SubRegsSet, Empty_SubRegsSet, AL_SuperRegsSet }, ...
```

From the register info file, TableGen generates a TargetRegisterDesc object for each register. TargetRegisterDesc is defined in include/llvm/Target/TargetRegisterInfo.h with the following fields:

```
struct TargetRegisterDesc {
  const char     *AsmName;      // Assembly language name for the register
  const char     *Name;         // Printable name for the reg (for debugging)
  const unsigned *AliasSet;     // Register Alias Set
  const unsigned *SubRegs;      // Sub-register set
  const unsigned *ImmSubRegs;   // Immediate sub-register set
  const unsigned *SuperRegs;    // Super-register set
};
```

TableGen uses the entire target description file (.td) to determine text names for the register (in the AsmName and Name fields of TargetRegisterDesc) and the relationships of other registers to the defined register (in the other

TargetRegisterDesc fields). In this example, other definitions establish the registers "AX", "EAX", and "RAX" as aliases for one another, so TableGen generates a null-terminated array (AL_AliasSet) for this register alias set.

The Register class is commonly used as a base class for more complex classes. In Target.td, the Register class is the base for the RegisterWithSubRegs class that is used to define registers that need to specify subregisters in the SubRegs list, as shown here:

```
class RegisterWithSubRegs<string n, list<Register> subregs> : Register<n> {
  let SubRegs = subregs;
}
```

In SparcRegisterInfo.td, additional register classes are defined for SPARC: a Register subclass, SparcReg, and further subclasses: Ri, Rf, and Rd. SPARC registers are identified by 5-bit ID numbers, which is a feature common to these subclasses. Note the use of "let" expressions to override values that are initially defined in a superclass (such as SubRegs field in the Rd class).

```
class SparcReg<string n> : Register<n> {
  field bits<5> Num;
  let Namespace = "SP";
}
// Ri - 32-bit integer registers
class Ri<bits<5> num, string n> :
SparcReg<n> {
  let Num = num;
}
// Rf - 32-bit floating-point registers
class Rf<bits<5> num, string n> :
SparcReg<n> {
  let Num = num;
}
// Rd - Slots in the FP register file for 64-bit floating-point values.
class Rd<bits<5> num, string n, list<Register> subregs> : SparcReg<n> {
  let Num = num;
  let SubRegs = subregs;
}
```

In the SparcRegisterInfo.td file, there are register definitions that utilize these subclasses of Register, such as:

```
def G0 : Ri< 0, "G0">, DwarfRegNum<[0]>;
def G1 : Ri< 1, "G1">, DwarfRegNum<[1]>;
...
def F0 : Rf< 0, "F0">, DwarfRegNum<[32]>;
def F1 : Rf< 1, "F1">, DwarfRegNum<[33]>;
...
def D0 : Rd< 0, "F0", [F0, F1]>, DwarfRegNum<[32]>;
def D1 : Rd< 2, "F2", [F2, F3]>, DwarfRegNum<[34]>;
```

The last two registers shown above (D0 and D1) are double-precision floating-point registers that are aliases for pairs of single-precision floating-point sub-registers. In addition to aliases, the sub-register and super-register relationships of the defined register are in fields of a register's TargetRegisterDesc.

Defining a Register Class

The RegisterClass class (specified in Target.td) is used to define an object that represents a group of related registers and also defines the default allocation order of the registers. A target description file XXXRegisterInfo.td that uses Target.td can construct register classes using the following class:

```
class RegisterClass<string namespace,
list<ValueType> regTypes, int alignment, dag regList> {
  string Namespace = namespace;
  list<ValueType> RegTypes = regTypes;
  int Size = 0;  // spill size, in bits; zero lets tblgen pick the size
  int Alignment = alignment;

  // CopyCost is the cost of copying a value between two registers
  // default value 1 means a single instruction
  // A negative value means copying is extremely expensive or impossible
  int CopyCost = 1;
  dag MemberList = regList;

  // for register classes that are subregisters of this class
  list<RegisterClass> SubRegClassList = [];

  code MethodProtos = [{}];  // to insert arbitrary code
  code MethodBodies = [{}];
}
```

To define a `RegisterClass`, use the following 4 arguments:

- The first argument of the definition is the name of the namespace.
- The second argument is a list of `ValueType` register type values that are defined in `include/llvm/CodeGen/ValueTypes.td`. Defined values include integer types (such as `i16`, `i32`, and `i1` for Boolean), floating-point types (`f32`, `f64`), and vector types (for example, `v8i16` for an `8 x i16` vector). All registers in a `RegisterClass` must have the same `ValueType`, but some registers may store vector data in different configurations. For example a register that can process a 128-bit vector may be able to handle 16 8-bit integer elements, 8 16-bit integers, 4 32-bit integers, and so on.
- The third argument of the `RegisterClass` definition specifies the alignment required of the registers when they are stored or loaded to memory.
- The final argument, `regList`, specifies which registers are in this class. If an alternative allocation order method is not specified, then `regList` also defines the order of allocation used by the register allocator. Besides simply listing registers with `(add R0, R1, ...)`, more advanced set operators are available. See `include/llvm/Target/Target.td` for more information.

In `SparcRegisterInfo.td`, three `RegisterClass` objects are defined: `FPRegs`, `DFPRegs`, and `IntRegs`. For all three register classes, the first argument defines the namespace with the string "`SP`". `FPRegs` defines a group of 32 single-precision floating-point registers (`F0` to `F31`); `DFPRegs` defines a group of 16 double-precision registers (`D0-D15`).

```
// F0, F1, F2, ..., F31
def FPRegs : RegisterClass<"SP", [f32], 32, (sequence "F%u", 0, 31)>;

def DFPRegs : RegisterClass<"SP", [f64], 64,
                            (add D0, D1, D2, D3, D4, D5, D6, D7, D8,
                                 D9, D10, D11, D12, D13, D14, D15)>;

def IntRegs : RegisterClass<"SP", [i32], 32,
    (add L0, L1, L2, L3, L4, L5, L6, L7,
         I0, I1, I2, I3, I4, I5,
         O0, O1, O2, O3, O4, O5, O7,
         G1,
         // Non-allocatable regs:
         G2, G3, G4,
         O6,        // stack ptr
```

```
        I6,         // frame ptr
        I7,         // return address
        G0,         // constant zero
        G5, G6, G7 // reserved for kernel
    )>;
```

Using `SparcRegisterInfo.td` with TableGen generates several output files that are intended for inclusion in other source code that you write. `SparcRegisterInfo.td` generates `SparcGenRegisterInfo.h.inc`, which should be included in the header file for the implementation of the SPARC register implementation that you write (`SparcRegisterInfo.h`). In `SparcGenRegisterInfo.h.inc` a new structure is defined called `SparcGenRegisterInfo` that uses `TargetRegisterInfo` as its base. It also specifies types, based upon the defined register classes: `DFPRegsClass`, `FPRegsClass`, and `IntRegsClass`.

`SparcRegisterInfo.td` also generates `SparcGenRegisterInfo.inc`, which is included at the bottom of `SparcRegisterInfo.cpp`, the SPARC register implementation. The code below shows only the generated integer registers and associated register classes. The order of registers in `IntRegs` reflects the order in the definition of `IntRegs` in the target description file.

```
// IntRegs Register Class...
static const unsigned IntRegs[] = {
  SP::L0, SP::L1, SP::L2, SP::L3, SP::L4, SP::L5,
  SP::L6, SP::L7, SP::I0, SP::I1, SP::I2, SP::I3,
  SP::I4, SP::I5, SP::O0, SP::O1, SP::O2, SP::O3,
  SP::O4, SP::O5, SP::O7, SP::G1, SP::G2, SP::G3,
  SP::G4, SP::O6, SP::I6, SP::I7, SP::G0, SP::G5,
  SP::G6, SP::G7,
};

// IntRegsVTs Register Class Value Types...
static const MVT::ValueType IntRegsVTs[] = {
  MVT::i32, MVT::Other
};

namespace SP {   // Register class instances
  DFPRegsClass    DFPRegsRegClass;
  FPRegsClass     FPRegsRegClass;
  IntRegsClass    IntRegsRegClass;
...
  // IntRegs Sub-register Classess...
  static const TargetRegisterClass* const IntRegsSubRegClasses [] = {
    NULL
  };
...
  // IntRegs Super-register Classess...
  static const TargetRegisterClass* const IntRegsSuperRegClasses [] = {
    NULL
  };
...
  // IntRegs Register Class sub-classes...
  static const TargetRegisterClass* const IntRegsSubclasses [] = {
    NULL
  };
...
  // IntRegs Register Class super-classes...
  static const TargetRegisterClass* const IntRegsSuperclasses [] = {
    NULL
  };
```

```
  IntRegsClass::IntRegsClass() : TargetRegisterClass(IntRegsRegClassID,
    IntRegsVTs, IntRegsSubclasses, IntRegsSuperclasses, IntRegsSubRegClasses,
    IntRegsSuperRegClasses, 4, 4, 1, IntRegs, IntRegs + 32) {}
}
```

The register allocators will avoid using reserved registers, and callee saved registers are not used until all the volatile registers have been used. That is usually good enough, but in some cases it may be necessary to provide custom allocation orders.

Implement a subclass of `TargetRegisterInfo`

The final step is to hand code portions of `XXXRegisterInfo`, which implements the interface described in `TargetRegisterInfo.h` (see *The TargetRegisterInfo class*). These functions return `0`, `NULL`, or `false`, unless overridden. Here is a list of functions that are overridden for the SPARC implementation in `SparcRegisterInfo.cpp`:

- `getCalleeSavedRegs` — Returns a list of callee-saved registers in the order of the desired callee-save stack frame offset.
- `getReservedRegs` — Returns a bitset indexed by physical register numbers, indicating if a particular register is unavailable.
- `hasFP` — Return a Boolean indicating if a function should have a dedicated frame pointer register.
- `eliminateCallFramePseudoInstr` — If call frame setup or destroy pseudo instructions are used, this can be called to eliminate them.
- `eliminateFrameIndex` — Eliminate abstract frame indices from instructions that may use them.
- `emitPrologue` — Insert prologue code into the function.
- `emitEpilogue` — Insert epilogue code into the function.

4.18.6 Instruction Set

During the early stages of code generation, the LLVM IR code is converted to a `SelectionDAG` with nodes that are instances of the `SDNode` class containing target instructions. An `SDNode` has an opcode, operands, type requirements, and operation properties. For example, is an operation commutative, does an operation load from memory. The various operation node types are described in the `include/llvm/CodeGen/SelectionDAGNodes.h` file (values of the `NodeType` enum in the `ISD` namespace).

TableGen uses the following target description (`.td`) input files to generate much of the code for instruction definition:

- `Target.td` — Where the `Instruction`, `Operand`, `InstrInfo`, and other fundamental classes are defined.
- `TargetSelectionDAG.td` — Used by `SelectionDAG` instruction selection generators, contains `SDTC*` classes (selection DAG type constraint), definitions of `SelectionDAG` nodes (such as `imm`, `cond`, `bb`, `add`, `fadd`, `sub`), and pattern support (`Pattern`, `Pat`, `PatFrag`, `PatLeaf`, `ComplexPattern`.
- `XXXInstrFormats.td` — Patterns for definitions of target-specific instructions.
- `XXXInstrInfo.td` — Target-specific definitions of instruction templates, condition codes, and instructions of an instruction set. For architecture modifications, a different file name may be used. For example, for Pentium with SSE instruction, this file is `X86InstrSSE.td`, and for Pentium with MMX, this file is `X86InstrMMX.td`.

There is also a target-specific XXX.td file, where XXX is the name of the target. The XXX.td file includes the other .td input files, but its contents are only directly important for subtargets.

You should describe a concrete target-specific class XXXInstrInfo that represents machine instructions supported by a target machine. XXXInstrInfo contains an array of XXXInstrDescriptor objects, each of which describes one instruction. An instruction descriptor defines:

- Opcode mnemonic
- Number of operands
- List of implicit register definitions and uses
- Target-independent properties (such as memory access, is commutable)
- Target-specific flags

The Instruction class (defined in Target.td) is mostly used as a base for more complex instruction classes.

```
class Instruction {
  string Namespace = "";
  dag OutOperandList;    // A dag containing the MI def operand list.
  dag InOperandList;     // A dag containing the MI use operand list.
  string AsmString = ""; // The .s format to print the instruction with.
  list<dag> Pattern;     // Set to the DAG pattern for this instruction.
  list<Register> Uses = [];
  list<Register> Defs = [];
  list<Predicate> Predicates = [];  // predicates turned into isel match code
  ... remainder not shown for space ...
}
```

A SelectionDAG node (SDNode) should contain an object representing a target-specific instruction that is defined in XXXInstrInfo.td. The instruction objects should represent instructions from the architecture manual of the target machine (such as the SPARC Architecture Manual for the SPARC target).

A single instruction from the architecture manual is often modeled as multiple target instructions, depending upon its operands. For example, a manual might describe an add instruction that takes a register or an immediate operand. An LLVM target could model this with two instructions named ADDri and ADDrr.

You should define a class for each instruction category and define each opcode as a subclass of the category with appropriate parameters such as the fixed binary encoding of opcodes and extended opcodes. You should map the register bits to the bits of the instruction in which they are encoded (for the JIT). Also you should specify how the instruction should be printed when the automatic assembly printer is used.

As is described in the SPARC Architecture Manual, Version 8, there are three major 32-bit formats for instructions. Format 1 is only for the CALL instruction. Format 2 is for branch on condition codes and SETHI (set high bits of a register) instructions. Format 3 is for other instructions.

Each of these formats has corresponding classes in SparcInstrFormat.td. InstSP is a base class for other instruction classes. Additional base classes are specified for more precise formats: for example in SparcInstrFormat.td, F2_1 is for SETHI, and F2_2 is for branches. There are three other base classes: F3_1 for register/register operations, F3_2 for register/immediate operations, and F3_3 for floating-point operations. SparcInstrInfo.td also adds the base class Pseudo for synthetic SPARC instructions.

SparcInstrInfo.td largely consists of operand and instruction definitions for the SPARC target. In SparcInstrInfo.td, the following target description file entry, LDrr, defines the Load Integer instruction for a Word (the LD SPARC opcode) from a memory address to a register. The first parameter, the value 3 (11_2), is the operation value for this category of operation. The second parameter (000000_2) is the specific operation value for LD/Load Word. The third parameter is the output destination, which is a register operand and defined in the Register target description file (IntRegs).

```
def LDrr : F3_1 <3, 0b000000, (outs IntRegs:$dst), (ins MEMrr:$addr),
                 "ld [$addr], $dst",
                 [(set i32:$dst, (load ADDRrr:$addr))]>;
```

The fourth parameter is the input source, which uses the address operand `MEMrr` that is defined earlier in `SparcInstrInfo.td`:

```
def MEMrr : Operand<i32> {
  let PrintMethod = "printMemOperand";
  let MIOperandInfo = (ops IntRegs, IntRegs);
}
```

The fifth parameter is a string that is used by the assembly printer and can be left as an empty string until the assembly printer interface is implemented. The sixth and final parameter is the pattern used to match the instruction during the SelectionDAG Select Phase described in *The LLVM Target-Independent Code Generator*. This parameter is detailed in the next section, *Instruction Selector*.

Instruction class definitions are not overloaded for different operand types, so separate versions of instructions are needed for register, memory, or immediate value operands. For example, to perform a Load Integer instruction for a Word from an immediate operand to a register, the following instruction class is defined:

```
def LDri : F3_2 <3, 0b000000, (outs IntRegs:$dst), (ins MEMri:$addr),
                 "ld [$addr], $dst",
                 [(set i32:$dst, (load ADDRri:$addr))]>;
```

Writing these definitions for so many similar instructions can involve a lot of cut and paste. In `.td` files, the `multiclass` directive enables the creation of templates to define several instruction classes at once (using the `defm` directive). For example in `SparcInstrInfo.td`, the `multiclass` pattern `F3_12` is defined to create 2 instruction classes each time `F3_12` is invoked:

```
multiclass F3_12 <string OpcStr, bits<6> Op3Val, SDNode OpNode> {
  def rr  : F3_1 <2, Op3Val,
                  (outs IntRegs:$dst), (ins IntRegs:$b, IntRegs:$c),
                  !strconcat(OpcStr, " $b, $c, $dst"),
                  [(set i32:$dst, (OpNode i32:$b, i32:$c))]>;
  def ri  : F3_2 <2, Op3Val,
                  (outs IntRegs:$dst), (ins IntRegs:$b, i32imm:$c),
                  !strconcat(OpcStr, " $b, $c, $dst"),
                  [(set i32:$dst, (OpNode i32:$b, simm13:$c))]>;
}
```

So when the `defm` directive is used for the `XOR` and `ADD` instructions, as seen below, it creates four instruction objects: `XORrr`, `XORri`, `ADDrr`, and `ADDri`.

```
defm XOR   : F3_12<"xor", 0b000011, xor>;
defm ADD   : F3_12<"add", 0b000000, add>;
```

`SparcInstrInfo.td` also includes definitions for condition codes that are referenced by branch instructions. The following definitions in `SparcInstrInfo.td` indicate the bit location of the SPARC condition code. For example, the 10th bit represents the "greater than" condition for integers, and the 22nd bit represents the "greater than" condition for floats.

```
def ICC_NE  : ICC_VAL< 9>;  // Not Equal
def ICC_E   : ICC_VAL< 1>;  // Equal
def ICC_G   : ICC_VAL<10>;  // Greater
...
def FCC_U   : FCC_VAL<23>;  // Unordered
def FCC_G   : FCC_VAL<22>;  // Greater
```

```
def FCC_UG  : FCC_VAL<21>;  // Unordered or Greater
...
```

(Note that `Sparc.h` also defines enums that correspond to the same SPARC condition codes. Care must be taken to ensure the values in `Sparc.h` correspond to the values in `SparcInstrInfo.td`. I.e., `SPCC::ICC_NE = 9`, `SPCC::FCC_U = 23` and so on.)

Instruction Operand Mapping

The code generator backend maps instruction operands to fields in the instruction. Operands are assigned to unbound fields in the instruction in the order they are defined. Fields are bound when they are assigned a value. For example, the Sparc target defines the `XNORrr` instruction as a `F3_1` format instruction having three operands.

```
def XNORrr  : F3_1<2, 0b000111,
                   (outs IntRegs:$dst), (ins IntRegs:$b, IntRegs:$c),
                   "xnor $b, $c, $dst",
                   [(set i32:$dst, (not (xor i32:$b, i32:$c)))]>;
```

The instruction templates in `SparcInstrFormats.td` show the base class for `F3_1` is `InstSP`.

```
class InstSP<dag outs, dag ins, string asmstr, list<dag> pattern> : Instruction {
  field bits<32> Inst;
  let Namespace = "SP";
  bits<2> op;
  let Inst{31-30} = op;
  dag OutOperandList = outs;
  dag InOperandList = ins;
  let AsmString   = asmstr;
  let Pattern = pattern;
}
```

`InstSP` leaves the `op` field unbound.

```
class F3<dag outs, dag ins, string asmstr, list<dag> pattern>
    : InstSP<outs, ins, asmstr, pattern> {
  bits<5> rd;
  bits<6> op3;
  bits<5> rs1;
  let op{1} = 1;   // Op = 2 or 3
  let Inst{29-25} = rd;
  let Inst{24-19} = op3;
  let Inst{18-14} = rs1;
}
```

`F3` binds the `op` field and defines the `rd`, `op3`, and `rs1` fields. `F3` format instructions will bind the operands `rd`, `op3`, and `rs1` fields.

```
class F3_1<bits<2> opVal, bits<6> op3val, dag outs, dag ins,
           string asmstr, list<dag> pattern> : F3<outs, ins, asmstr, pattern> {
  bits<8> asi = 0; // asi not currently used
  bits<5> rs2;
  let op         = opVal;
  let op3        = op3val;
  let Inst{13}   = 0;     // i field = 0
  let Inst{12-5} = asi;   // address space identifier
  let Inst{4-0}  = rs2;
}
```

F3_1 binds the op3 field and defines the rs2 fields. F3_1 format instructions will bind the operands to the rd, rs1, and rs2 fields. This results in the XNORrr instruction binding $dst, $b, and $c operands to the rd, rs1, and rs2 fields respectively.

Instruction Operand Name Mapping

TableGen will also generate a function called getNamedOperandIdx() which can be used to look up an operand's index in a MachineInstr based on its TableGen name. Setting the UseNamedOperandTable bit in an instruction's TableGen definition will add all of its operands to an enumeration in the llvm::XXX:OpName namespace and also add an entry for it into the OperandMap table, which can be queried using getNamedOperandIdx()

```
int DstIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::dst); // => 0
int BIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::b);     // => 1
int CIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::c);     // => 2
int DIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::d);     // => -1

...
```

The entries in the OpName enum are taken verbatim from the TableGen definitions, so operands with lowercase names will have lower case entries in the enum.

To include the getNamedOperandIdx() function in your backend, you will need to define a few preprocessor macros in XXXInstrInfo.cpp and XXXInstrInfo.h. For example:

XXXInstrInfo.cpp:

```
#define GET_INSTRINFO_NAMED_OPS // For getNamedOperandIdx() function
#include "XXXGenInstrInfo.inc"
```

XXXInstrInfo.h:

```
#define GET_INSTRINFO_OPERAND_ENUM // For OpName enum
#include "XXXGenInstrInfo.inc"

namespace XXX {
  int16_t getNamedOperandIdx(uint16_t Opcode, uint16_t NamedIndex);
} // End namespace XXX
```

Instruction Operand Types

TableGen will also generate an enumeration consisting of all named Operand types defined in the backend, in the llvm::XXX::OpTypes namespace. Some common immediate Operand types (for instance i8, i32, i64, f32, f64) are defined for all targets in include/llvm/Target/Target.td, and are available in each Target's OpTypes enum. Also, only named Operand types appear in the enumeration: anonymous types are ignored. For example, the X86 backend defines brtarget and brtarget8, both instances of the TableGen Operand class, which represent branch target operands:

```
def brtarget : Operand<OtherVT>;
def brtarget8 : Operand<OtherVT>;
```

This results in:

```
namespace X86 {
namespace OpTypes {
enum OperandType {
  ...
  brtarget,
```

```
  brtarget8,
  ...
  i32imm,
  i64imm,
  ...
  OPERAND_TYPE_LIST_END
} // End namespace OpTypes
} // End namespace X86
```

In typical TableGen fashion, to use the enum, you will need to define a preprocessor macro:

```
#define GET_INSTRINFO_OPERAND_TYPES_ENUM // For OpTypes enum
#include "XXXGenInstrInfo.inc"
```

Instruction Scheduling

Instruction itineraries can be queried using MCDesc::getSchedClass(). The value can be named by an enumemation in llvm::XXX::Sched namespace generated by TableGen in XXXGenInstrInfo.inc. The name of the schedule classes are the same as provided in XXXSchedule.td plus a default NoItinerary class.

Instruction Relation Mapping

This TableGen feature is used to relate instructions with each other. It is particularly useful when you have multiple instruction formats and need to switch between them after instruction selection. This entire feature is driven by relation models which can be defined in `XXXInstrInfo.td` files according to the target-specific instruction set. Relation models are defined using `InstrMapping` class as a base. TableGen parses all the models and generates instruction relation maps using the specified information. Relation maps are emitted as tables in the `XXXGenInstrInfo.inc` file along with the functions to query them. For the detailed information on how to use this feature, please refer to *How To Use Instruction Mappings*.

Implement a subclass of `TargetInstrInfo`

The final step is to hand code portions of `XXXInstrInfo`, which implements the interface described in `TargetInstrInfo.h` (see *The TargetInstrInfo class*). These functions return `0` or a Boolean or they assert, unless overridden. Here's a list of functions that are overridden for the SPARC implementation in `SparcInstrInfo.cpp`:

- `isLoadFromStackSlot` — If the specified machine instruction is a direct load from a stack slot, return the register number of the destination and the `FrameIndex` of the stack slot.
- `isStoreToStackSlot` — If the specified machine instruction is a direct store to a stack slot, return the register number of the destination and the `FrameIndex` of the stack slot.
- `copyPhysReg` — Copy values between a pair of physical registers.
- `storeRegToStackSlot` — Store a register value to a stack slot.
- `loadRegFromStackSlot` — Load a register value from a stack slot.
- `storeRegToAddr` — Store a register value to memory.
- `loadRegFromAddr` — Load a register value from memory.
- `foldMemoryOperand` — Attempt to combine instructions of any load or store instruction for the specified operand(s).

Branch Folding and If Conversion

Performance can be improved by combining instructions or by eliminating instructions that are never reached. The `AnalyzeBranch` method in `XXXInstrInfo` may be implemented to examine conditional instructions and remove unnecessary instructions. `AnalyzeBranch` looks at the end of a machine basic block (MBB) for opportunities for improvement, such as branch folding and if conversion. The `BranchFolder` and `IfConverter` machine function passes (see the source files `BranchFolding.cpp` and `IfConversion.cpp` in the `lib/CodeGen` directory) call `AnalyzeBranch` to improve the control flow graph that represents the instructions.

Several implementations of `AnalyzeBranch` (for ARM, Alpha, and X86) can be examined as models for your own `AnalyzeBranch` implementation. Since SPARC does not implement a useful `AnalyzeBranch`, the ARM target implementation is shown below.

`AnalyzeBranch` returns a Boolean value and takes four parameters:

- `MachineBasicBlock &MBB` — The incoming block to be examined.
- `MachineBasicBlock *&TBB` — A destination block that is returned. For a conditional branch that evaluates to true, `TBB` is the destination.
- `MachineBasicBlock *&FBB` — For a conditional branch that evaluates to false, `FBB` is returned as the destination.
- `std::vector<MachineOperand> &Cond` — List of operands to evaluate a condition for a conditional branch.

In the simplest case, if a block ends without a branch, then it falls through to the successor block. No destination blocks are specified for either `TBB` or `FBB`, so both parameters return `NULL`. The start of the `AnalyzeBranch` (see code below for the ARM target) shows the function parameters and the code for the simplest case.

```
bool ARMInstrInfo::AnalyzeBranch(MachineBasicBlock &MBB,
                                 MachineBasicBlock *&TBB,
                                 MachineBasicBlock *&FBB,
                                 std::vector<MachineOperand> &Cond) const
{
  MachineBasicBlock::iterator I = MBB.end();
  if (I == MBB.begin() || !isUnpredicatedTerminator(--I))
    return false;
```

If a block ends with a single unconditional branch instruction, then `AnalyzeBranch` (shown below) should return the destination of that branch in the `TBB` parameter.

```
if (LastOpc == ARM::B || LastOpc == ARM::tB) {
  TBB = LastInst->getOperand(0).getMBB();
  return false;
}
```

If a block ends with two unconditional branches, then the second branch is never reached. In that situation, as shown below, remove the last branch instruction and return the penultimate branch in the `TBB` parameter.

```
if ((SecondLastOpc == ARM::B || SecondLastOpc == ARM::tB) &&
    (LastOpc == ARM::B || LastOpc == ARM::tB)) {
  TBB = SecondLastInst->getOperand(0).getMBB();
  I = LastInst;
  I->eraseFromParent();
  return false;
}
```

A block may end with a single conditional branch instruction that falls through to successor block if the condition evaluates to false. In that case, `AnalyzeBranch` (shown below) should return the destination of that conditional branch in the `TBB` parameter and a list of operands in the `Cond` parameter to evaluate the condition.

```
if (LastOpc == ARM::Bcc || LastOpc == ARM::tBcc) {
  // Block ends with fall-through condbranch.
  TBB = LastInst->getOperand(0).getMBB();
  Cond.push_back(LastInst->getOperand(1));
  Cond.push_back(LastInst->getOperand(2));
  return false;
}
```

If a block ends with both a conditional branch and an ensuing unconditional branch, then `AnalyzeBranch` (shown below) should return the conditional branch destination (assuming it corresponds to a conditional evaluation of "`true`") in the `TBB` parameter and the unconditional branch destination in the `FBB` (corresponding to a conditional evaluation of "`false`"). A list of operands to evaluate the condition should be returned in the `Cond` parameter.

```
unsigned SecondLastOpc = SecondLastInst->getOpcode();

if ((SecondLastOpc == ARM::Bcc && LastOpc == ARM::B) ||
    (SecondLastOpc == ARM::tBcc && LastOpc == ARM::tB)) {
  TBB =  SecondLastInst->getOperand(0).getMBB();
  Cond.push_back(SecondLastInst->getOperand(1));
  Cond.push_back(SecondLastInst->getOperand(2));
  FBB = LastInst->getOperand(0).getMBB();
  return false;
}
```

For the last two cases (ending with a single conditional branch or ending with one conditional and one unconditional branch), the operands returned in the `Cond` parameter can be passed to methods of other instructions to create new branches or perform other operations. An implementation of `AnalyzeBranch` requires the helper methods `RemoveBranch` and `InsertBranch` to manage subsequent operations.

`AnalyzeBranch` should return false indicating success in most circumstances. `AnalyzeBranch` should only return true when the method is stumped about what to do, for example, if a block has three terminating branches. `AnalyzeBranch` may return true if it encounters a terminator it cannot handle, such as an indirect branch.

4.18.7 Instruction Selector

LLVM uses a `SelectionDAG` to represent LLVM IR instructions, and nodes of the `SelectionDAG` ideally represent native target instructions. During code generation, instruction selection passes are performed to convert non-native DAG instructions into native target-specific instructions. The pass described in `XXXISelDAGToDAG.cpp` is used to match patterns and perform DAG-to-DAG instruction selection. Optionally, a pass may be defined (in `XXXBranchSelector.cpp`) to perform similar DAG-to-DAG operations for branch instructions. Later, the code in `XXXISelLowering.cpp` replaces or removes operations and data types not supported natively (legalizes) in a `SelectionDAG`.

TableGen generates code for instruction selection using the following target description input files:

- `XXXInstrInfo.td` — Contains definitions of instructions in a target-specific instruction set, generates `XXXGenDAGISel.inc`, which is included in `XXXISelDAGToDAG.cpp`.
- `XXXCallingConv.td` — Contains the calling and return value conventions for the target architecture, and it generates `XXXGenCallingConv.inc`, which is included in `XXXISelLowering.cpp`.

The implementation of an instruction selection pass must include a header that declares the `FunctionPass` class or a subclass of `FunctionPass`. In `XXXTargetMachine.cpp`, a Pass Manager (PM) should add each instruction selection pass into the queue of passes to run.

The LLVM static compiler (`llc`) is an excellent tool for visualizing the contents of DAGs. To display the `SelectionDAG` before or after specific processing phases, use the command line options for `llc`, described at *SelectionDAG Instruction Selection Process*.

To describe instruction selector behavior, you should add patterns for lowering LLVM code into a `SelectionDAG` as the last parameter of the instruction definitions in `XXXInstrInfo.td`. For example, in `SparcInstrInfo.td`, this entry defines a register store operation, and the last parameter describes a pattern with the store DAG operator.

```
def STrr  : F3_1< 3, 0b000100, (outs), (ins MEMrr:$addr, IntRegs:$src),
                  "st $src, [$addr]", [(store i32:$src, ADDRrr:$addr)]>;
```

`ADDRrr` is a memory mode that is also defined in `SparcInstrInfo.td`:

```
def ADDRrr : ComplexPattern<i32, 2, "SelectADDRrr", [], []>;
```

The definition of `ADDRrr` refers to `SelectADDRrr`, which is a function defined in an implementation of the Instructor Selector (such as `SparcISelDAGToDAG.cpp`).

In `lib/Target/TargetSelectionDAG.td`, the DAG operator for store is defined below:

```
def store : PatFrag<(ops node:$val, node:$ptr),
                    (st node:$val, node:$ptr), [{
  if (StoreSDNode *ST = dyn_cast<StoreSDNode>(N))
    return !ST->isTruncatingStore() &&
           ST->getAddressingMode() == ISD::UNINDEXED;
  return false;
}]>;
```

`XXXInstrInfo.td` also generates (in `XXXGenDAGISel.inc`) the `SelectCode` method that is used to call the appropriate processing method for an instruction. In this example, `SelectCode` calls `Select_ISD_STORE` for the `ISD::STORE` opcode.

```
SDNode *SelectCode(SDValue N) {
  ...
  MVT::ValueType NVT = N.getNode()->getValueType(0);
  switch (N.getOpcode()) {
  case ISD::STORE: {
    switch (NVT) {
    default:
      return Select_ISD_STORE(N);
      break;
    }
    break;
  }
  ...
```

The pattern for `STrr` is matched, so elsewhere in `XXXGenDAGISel.inc`, code for `STrr` is created for `Select_ISD_STORE`. The `Emit_22` method is also generated in `XXXGenDAGISel.inc` to complete the processing of this instruction.

```
SDNode *Select_ISD_STORE(const SDValue &N) {
  SDValue Chain = N.getOperand(0);
  if (Predicate_store(N.getNode())) {
    SDValue N1 = N.getOperand(1);
    SDValue N2 = N.getOperand(2);
    SDValue CPTmp0;
    SDValue CPTmp1;

    // Pattern: (st:void i32:i32:$src,
    //           ADDRrr:i32:$addr)<<P:Predicate_store>>
    // Emits: (STrr:void ADDRrr:i32:$addr, IntRegs:i32:$src)
    // Pattern complexity = 13  cost = 1  size = 0
    if (SelectADDRrr(N, N2, CPTmp0, CPTmp1) &&
        N1.getNode()->getValueType(0) == MVT::i32 &&
```

```
        N2.getNode()->getValueType(0) == MVT::i32) {
      return Emit_22(N, SP::STrr, CPTmp0, CPTmp1);
    }
...
```

The SelectionDAG Legalize Phase

The Legalize phase converts a DAG to use types and operations that are natively supported by the target. For natively unsupported types and operations, you need to add code to the target-specific `XXXTargetLowering` implementation to convert unsupported types and operations to supported ones.

In the constructor for the `XXXTargetLowering` class, first use the `addRegisterClass` method to specify which types are supported and which register classes are associated with them. The code for the register classes are generated by TableGen from `XXXRegisterInfo.td` and placed in `XXXGenRegisterInfo.h.inc`. For example, the implementation of the constructor for the SparcTargetLowering class (in `SparcISelLowering.cpp`) starts with the following code:

```
addRegisterClass(MVT::i32, SP::IntRegsRegisterClass);
addRegisterClass(MVT::f32, SP::FPRegsRegisterClass);
addRegisterClass(MVT::f64, SP::DFPRegsRegisterClass);
```

You should examine the node types in the `ISD` namespace (`include/llvm/CodeGen/SelectionDAGNodes.h`) and determine which operations the target natively supports. For operations that do **not** have native support, add a callback to the constructor for the `XXXTargetLowering` class, so the instruction selection process knows what to do. The `TargetLowering` class callback methods (declared in `llvm/Target/TargetLowering.h`) are:

- `setOperationAction` — General operation.
- `setLoadExtAction` — Load with extension.
- `setTruncStoreAction` — Truncating store.
- `setIndexedLoadAction` — Indexed load.
- `setIndexedStoreAction` — Indexed store.
- `setConvertAction` — Type conversion.
- `setCondCodeAction` — Support for a given condition code.

Note: on older releases, `setLoadXAction` is used instead of `setLoadExtAction`. Also, on older releases, `setCondCodeAction` may not be supported. Examine your release to see what methods are specifically supported.

These callbacks are used to determine that an operation does or does not work with a specified type (or types). And in all cases, the third parameter is a `LegalAction` type enum value: `Promote`, `Expand`, `Custom`, or `Legal`. `SparcISelLowering.cpp` contains examples of all four `LegalAction` values.

Promote

For an operation without native support for a given type, the specified type may be promoted to a larger type that is supported. For example, SPARC does not support a sign-extending load for Boolean values (`i1` type), so in `SparcISelLowering.cpp` the third parameter below, `Promote`, changes `i1` type values to a large type before loading.

```
setLoadExtAction(ISD::SEXTLOAD, MVT::i1, Promote);
```

Expand

For a type without native support, a value may need to be broken down further, rather than promoted. For an operation without native support, a combination of other operations may be used to similar effect. In SPARC, the floating-point sine and cosine trig operations are supported by expansion to other operations, as indicated by the third parameter, `Expand`, to `setOperationAction`:

```
setOperationAction(ISD::FSIN, MVT::f32, Expand);
setOperationAction(ISD::FCOS, MVT::f32, Expand);
```

Custom

For some operations, simple type promotion or operation expansion may be insufficient. In some cases, a special intrinsic function must be implemented.

For example, a constant value may require special treatment, or an operation may require spilling and restoring registers in the stack and working with register allocators.

As seen in `SparcISelLowering.cpp` code below, to perform a type conversion from a floating point value to a signed integer, first the `setOperationAction` should be called with `Custom` as the third parameter:

```
setOperationAction(ISD::FP_TO_SINT, MVT::i32, Custom);
```

In the `LowerOperation` method, for each `Custom` operation, a case statement should be added to indicate what function to call. In the following code, an `FP_TO_SINT` opcode will call the `LowerFP_TO_SINT` method:

```
SDValue SparcTargetLowering::LowerOperation(SDValue Op, SelectionDAG &DAG) {
  switch (Op.getOpcode()) {
  case ISD::FP_TO_SINT: return LowerFP_TO_SINT(Op, DAG);
  ...
  }
}
```

Finally, the `LowerFP_TO_SINT` method is implemented, using an FP register to convert the floating-point value to an integer.

```
static SDValue LowerFP_TO_SINT(SDValue Op, SelectionDAG &DAG) {
  assert(Op.getValueType() == MVT::i32);
  Op = DAG.getNode(SPISD::FTOI, MVT::f32, Op.getOperand(0));
  return DAG.getNode(ISD::BITCAST, MVT::i32, Op);
}
```

Legal

The `Legal` `LegalizeAction` enum value simply indicates that an operation **is** natively supported. `Legal` represents the default condition, so it is rarely used. In `SparcISelLowering.cpp`, the action for `CTPOP` (an operation to count the bits set in an integer) is natively supported only for SPARC v9. The following code enables the `Expand` conversion technique for non-v9 SPARC implementations.

```
setOperationAction(ISD::CTPOP, MVT::i32, Expand);
...
if (TM.getSubtarget<SparcSubtarget>().isV9())
  setOperationAction(ISD::CTPOP, MVT::i32, Legal);
```

Calling Conventions

To support target-specific calling conventions, XXXGenCallingConv.td uses interfaces (such as CCIfType and CCAssignToReg) that are defined in lib/Target/TargetCallingConv.td. TableGen can take the target descriptor file XXXGenCallingConv.td and generate the header file XXXGenCallingConv.inc, which is typically included in XXXISelLowering.cpp. You can use the interfaces in TargetCallingConv.td to specify:

- The order of parameter allocation.
- Where parameters and return values are placed (that is, on the stack or in registers).
- Which registers may be used.
- Whether the caller or callee unwinds the stack.

The following example demonstrates the use of the CCIfType and CCAssignToReg interfaces. If the CCIfType predicate is true (that is, if the current argument is of type f32 or f64), then the action is performed. In this case, the CCAssignToReg action assigns the argument value to the first available register: either R0 or R1.

```
CCIfType<[f32,f64], CCAssignToReg<[R0, R1]>>
```

SparcCallingConv.td contains definitions for a target-specific return-value calling convention (RetCC_Sparc32) and a basic 32-bit C calling convention (CC_Sparc32). The definition of RetCC_Sparc32 (shown below) indicates which registers are used for specified scalar return types. A single-precision float is returned to register F0, and a double-precision float goes to register D0. A 32-bit integer is returned in register I0 or I1.

```
def RetCC_Sparc32 : CallingConv<[
  CCIfType<[i32], CCAssignToReg<[I0, I1]>>,
  CCIfType<[f32], CCAssignToReg<[F0]>>,
  CCIfType<[f64], CCAssignToReg<[D0]>>
]>;
```

The definition of CC_Sparc32 in SparcCallingConv.td introduces CCAssignToStack, which assigns the value to a stack slot with the specified size and alignment. In the example below, the first parameter, 4, indicates the size of the slot, and the second parameter, also 4, indicates the stack alignment along 4-byte units. (Special cases: if size is zero, then the ABI size is used; if alignment is zero, then the ABI alignment is used.)

```
def CC_Sparc32 : CallingConv<[
  // All arguments get passed in integer registers if there is space.
  CCIfType<[i32, f32, f64], CCAssignToReg<[I0, I1, I2, I3, I4, I5]>>,
  CCAssignToStack<4, 4>
]>;
```

CCDelegateTo is another commonly used interface, which tries to find a specified sub-calling convention, and, if a match is found, it is invoked. In the following example (in X86CallingConv.td), the definition of RetCC_X86_32_C ends with CCDelegateTo. After the current value is assigned to the register ST0 or ST1, the RetCC_X86Common is invoked.

```
def RetCC_X86_32_C : CallingConv<[
  CCIfType<[f32], CCAssignToReg<[ST0, ST1]>>,
  CCIfType<[f64], CCAssignToReg<[ST0, ST1]>>,
  CCDelegateTo<RetCC_X86Common>
]>;
```

CCIfCC is an interface that attempts to match the given name to the current calling convention. If the name identifies the current calling convention, then a specified action is invoked. In the following example (in X86CallingConv.td), if the Fast calling convention is in use, then RetCC_X86_32_Fast is invoked. If the SSECall calling convention is in use, then RetCC_X86_32_SSE is invoked.

```
def RetCC_X86_32 : CallingConv<[
  CCIfCC<"CallingConv::Fast", CCDelegateTo<RetCC_X86_32_Fast>>,
  CCIfCC<"CallingConv::X86_SSECall", CCDelegateTo<RetCC_X86_32_SSE>>,
  CCDelegateTo<RetCC_X86_32_C>
]>;
```

Other calling convention interfaces include:

- `CCIf <predicate, action>` — If the predicate matches, apply the action.
- `CCIfInReg <action>` — If the argument is marked with the "`inreg`" attribute, then apply the action.
- `CCIfNest <action>` — If the argument is marked with the "`nest`" attribute, then apply the action.
- `CCIfNotVarArg <action>` — If the current function does not take a variable number of arguments, apply the action.
- `CCAssignToRegWithShadow <registerList, shadowList>` — similar to `CCAssignToReg`, but with a shadow list of registers.
- `CCPassByVal <size, align>` — Assign value to a stack slot with the minimum specified size and alignment.
- `CCPromoteToType <type>` — Promote the current value to the specified type.
- `CallingConv <[actions]>` — Define each calling convention that is supported.

4.18.8 Assembly Printer

During the code emission stage, the code generator may utilize an LLVM pass to produce assembly output. To do this, you want to implement the code for a printer that converts LLVM IR to a GAS-format assembly language for your target machine, using the following steps:

- Define all the assembly strings for your target, adding them to the instructions defined in the `XXXInstrInfo.td` file. (See *Instruction Set*.) TableGen will produce an output file (`XXXGenAsmWriter.inc`) with an implementation of the `printInstruction` method for the `XXXAsmPrinter` class.
- Write `XXXTargetAsmInfo.h`, which contains the bare-bones declaration of the `XXXTargetAsmInfo` class (a subclass of `TargetAsmInfo`).
- Write `XXXTargetAsmInfo.cpp`, which contains target-specific values for `TargetAsmInfo` properties and sometimes new implementations for methods.
- Write `XXXAsmPrinter.cpp`, which implements the `AsmPrinter` class that performs the LLVM-to-assembly conversion.

The code in `XXXTargetAsmInfo.h` is usually a trivial declaration of the `XXXTargetAsmInfo` class for use in `XXXTargetAsmInfo.cpp`. Similarly, `XXXTargetAsmInfo.cpp` usually has a few declarations of `XXXTargetAsmInfo` replacement values that override the default values in `TargetAsmInfo.cpp`. For example in `SparcTargetAsmInfo.cpp`:

```
SparcTargetAsmInfo::SparcTargetAsmInfo(const SparcTargetMachine &TM) {
  Data16bitsDirective = "\t.half\t";
  Data32bitsDirective = "\t.word\t";
  Data64bitsDirective = 0;  // .xword is only supported by V9.
  ZeroDirective = "\t.skip\t";
  CommentString = "!";
  ConstantPoolSection = "\t.section \".rodata\",#alloc\n";
}
```

The X86 assembly printer implementation (X86TargetAsmInfo) is an example where the target specific TargetAsmInfo class uses an overridden methods: ExpandInlineAsm.

A target-specific implementation of AsmPrinter is written in XXXAsmPrinter.cpp, which implements the AsmPrinter class that converts the LLVM to printable assembly. The implementation must include the following headers that have declarations for the AsmPrinter and MachineFunctionPass classes. The MachineFunctionPass is a subclass of FunctionPass.

```
#include "llvm/CodeGen/AsmPrinter.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
```

As a FunctionPass, AsmPrinter first calls doInitialization to set up the AsmPrinter. In SparcAsmPrinter, a Mangler object is instantiated to process variable names.

In XXXAsmPrinter.cpp, the runOnMachineFunction method (declared in MachineFunctionPass) must be implemented for XXXAsmPrinter. In MachineFunctionPass, the runOnFunction method invokes runOnMachineFunction. Target-specific implementations of runOnMachineFunction differ, but generally do the following to process each machine function:

- Call SetupMachineFunction to perform initialization.
- Call EmitConstantPool to print out (to the output stream) constants which have been spilled to memory.
- Call EmitJumpTableInfo to print out jump tables used by the current function.
- Print out the label for the current function.
- Print out the code for the function, including basic block labels and the assembly for the instruction (using printInstruction)

The XXXAsmPrinter implementation must also include the code generated by TableGen that is output in the XXXGenAsmWriter.inc file. The code in XXXGenAsmWriter.inc contains an implementation of the printInstruction method that may call these methods:

- printOperand
- printMemOperand
- printCCOperand (for conditional statements)
- printDataDirective
- printDeclare
- printImplicitDef
- printInlineAsm

The implementations of printDeclare, printImplicitDef, printInlineAsm, and printLabel in AsmPrinter.cpp are generally adequate for printing assembly and do not need to be overridden.

The printOperand method is implemented with a long switch/case statement for the type of operand: register, immediate, basic block, external symbol, global address, constant pool index, or jump table index. For an instruction with a memory address operand, the printMemOperand method should be implemented to generate the proper output. Similarly, printCCOperand should be used to print a conditional operand.

doFinalization should be overridden in XXXAsmPrinter, and it should be called to shut down the assembly printer. During doFinalization, global variables and constants are printed to output.

4.18.9 Subtarget Support

Subtarget support is used to inform the code generation process of instruction set variations for a given chip set. For example, the LLVM SPARC implementation provided covers three major versions of the SPARC microprocessor

architecture: Version 8 (V8, which is a 32-bit architecture), Version 9 (V9, a 64-bit architecture), and the UltraSPARC architecture. V8 has 16 double-precision floating-point registers that are also usable as either 32 single-precision or 8 quad-precision registers. V8 is also purely big-endian. V9 has 32 double-precision floating-point registers that are also usable as 16 quad-precision registers, but cannot be used as single-precision registers. The UltraSPARC architecture combines V9 with UltraSPARC Visual Instruction Set extensions.

If subtarget support is needed, you should implement a target-specific `XXXSubtarget` class for your architecture. This class should process the command-line options `-mcpu=` and `-mattr=`.

TableGen uses definitions in the `Target.td` and `Sparc.td` files to generate code in `SparcGenSubtarget.inc`. In `Target.td`, shown below, the `SubtargetFeature` interface is defined. The first 4 string parameters of the `SubtargetFeature` interface are a feature name, an attribute set by the feature, the value of the attribute, and a description of the feature. (The fifth parameter is a list of features whose presence is implied, and its default value is an empty array.)

```
class SubtargetFeature<string n, string a, string v, string d,
                       list<SubtargetFeature> i = []> {
  string Name = n;
  string Attribute = a;
  string Value = v;
  string Desc = d;
  list<SubtargetFeature> Implies = i;
}
```

In the `Sparc.td` file, the `SubtargetFeature` is used to define the following features.

```
def FeatureV9 : SubtargetFeature<"v9", "IsV9", "true",
                    "Enable SPARC-V9 instructions">;
def FeatureV8Deprecated : SubtargetFeature<"deprecated-v8",
                    "V8DeprecatedInsts", "true",
                    "Enable deprecated V8 instructions in V9 mode">;
def FeatureVIS : SubtargetFeature<"vis", "IsVIS", "true",
                    "Enable UltraSPARC Visual Instruction Set extensions">;
```

Elsewhere in `Sparc.td`, the `Proc` class is defined and then is used to define particular SPARC processor subtypes that may have the previously described features.

```
class Proc<string Name, list<SubtargetFeature> Features>
  : Processor<Name, NoItineraries, Features>;

def : Proc<"generic",         []>;
def : Proc<"v8",              []>;
def : Proc<"supersparc",      []>;
def : Proc<"sparclite",       []>;
def : Proc<"f934",            []>;
def : Proc<"hypersparc",      []>;
def : Proc<"sparclite86x",    []>;
def : Proc<"sparclet",        []>;
def : Proc<"tsc701",          []>;
def : Proc<"v9",              [FeatureV9]>;
def : Proc<"ultrasparc",      [FeatureV9, FeatureV8Deprecated]>;
def : Proc<"ultrasparc3",     [FeatureV9, FeatureV8Deprecated]>;
def : Proc<"ultrasparc3-vis", [FeatureV9, FeatureV8Deprecated, FeatureVIS]>;
```

From `Target.td` and `Sparc.td` files, the resulting `SparcGenSubtarget.inc` specifies enum values to identify the features, arrays of constants to represent the CPU features and CPU subtypes, and the `ParseSubtargetFeatures` method that parses the features string that sets specified subtarget options. The generated `SparcGenSubtarget.inc` file should be included in the `SparcSubtarget.cpp`. The target-specific implementation of the `XXXSubtarget` method should follow this pseudocode:

```
XXXSubtarget::XXXSubtarget(const Module &M, const std::string &FS) {
  // Set the default features
  // Determine default and user specified characteristics of the CPU
  // Call ParseSubtargetFeatures(FS, CPU) to parse the features string
  // Perform any additional operations
}
```

4.18.10 JIT Support

The implementation of a target machine optionally includes a Just-In-Time (JIT) code generator that emits machine code and auxiliary structures as binary output that can be written directly to memory. To do this, implement JIT code generation by performing the following steps:

- Write an `XXXCodeEmitter.cpp` file that contains a machine function pass that transforms target-machine instructions into relocatable machine code.
- Write an `XXXJITInfo.cpp` file that implements the JIT interfaces for target-specific code-generation activities, such as emitting machine code and stubs.
- Modify `XXXTargetMachine` so that it provides a `TargetJITInfo` object through its `getJITInfo` method.

There are several different approaches to writing the JIT support code. For instance, TableGen and target descriptor files may be used for creating a JIT code generator, but are not mandatory. For the Alpha and PowerPC target machines, TableGen is used to generate `XXXGenCodeEmitter.inc`, which contains the binary coding of machine instructions and the `getBinaryCodeForInstr` method to access those codes. Other JIT implementations do not.

Both `XXXJITInfo.cpp` and `XXXCodeEmitter.cpp` must include the `llvm/CodeGen/MachineCodeEmitter.h` header file that defines the `MachineCodeEmitter` class containing code for several callback functions that write data (in bytes, words, strings, etc.) to the output stream.

Machine Code Emitter

In `XXXCodeEmitter.cpp`, a target-specific of the `Emitter` class is implemented as a function pass (subclass of `MachineFunctionPass`). The target-specific implementation of `runOnMachineFunction` (invoked by `runOnFunction` in `MachineFunctionPass`) iterates through the `MachineBasicBlock` calls `emitInstruction` to process each instruction and emit binary code. `emitInstruction` is largely implemented with case statements on the instruction types defined in `XXXInstrInfo.h`. For example, in `X86CodeEmitter.cpp`, the `emitInstruction` method is built around the following `switch`/`case` statements:

```
switch (Desc->TSFlags & X86::FormMask) {
case X86II::Pseudo:  // for not yet implemented instructions
   ...               // or pseudo-instructions
   break;
case X86II::RawFrm:  // for instructions with a fixed opcode value
   ...
   break;
case X86II::AddRegFrm: // for instructions that have one register operand
   ...                 // added to their opcode
   break;
case X86II::MRMDestReg:// for instructions that use the Mod/RM byte
   ...                 // to specify a destination (register)
   break;
case X86II::MRMDestMem:// for instructions that use the Mod/RM byte
   ...                 // to specify a destination (memory)
```

```
   break;
case X86II::MRMSrcReg: // for instructions that use the Mod/RM byte
   ...                 // to specify a source (register)
   break;
case X86II::MRMSrcMem: // for instructions that use the Mod/RM byte
   ...                 // to specify a source (memory)
   break;
case X86II::MRM0r: case X86II::MRM1r:  // for instructions that operate on
case X86II::MRM2r: case X86II::MRM3r:  // a REGISTER r/m operand and
case X86II::MRM4r: case X86II::MRM5r:  // use the Mod/RM byte and a field
case X86II::MRM6r: case X86II::MRM7r:  // to hold extended opcode data
   ...
   break;
case X86II::MRM0m: case X86II::MRM1m:  // for instructions that operate on
case X86II::MRM2m: case X86II::MRM3m:  // a MEMORY r/m operand and
case X86II::MRM4m: case X86II::MRM5m:  // use the Mod/RM byte and a field
case X86II::MRM6m: case X86II::MRM7m:  // to hold extended opcode data
   ...
   break;
case X86II::MRMInitReg: // for instructions whose source and
   ...                  // destination are the same register
   break;
}
```

The implementations of these case statements often first emit the opcode and then get the operand(s). Then depending upon the operand, helper methods may be called to process the operand(s). For example, in `X86CodeEmitter.cpp`, for the `X86II::AddRegFrm` case, the first data emitted (by `emitByte`) is the opcode added to the register operand. Then an object representing the machine operand, `MO1`, is extracted. The helper methods such as `isImmediate`, `isGlobalAddress`, `isExternalSymbol`, `isConstantPoolIndex`, and `isJumpTableIndex` determine the operand type. (`X86CodeEmitter.cpp` also has private methods such as `emitConstant`, `emitGlobalAddress`, `emitExternalSymbolAddress`, `emitConstPoolAddress`, and `emitJumpTableAddress` that emit the data into the output stream.)

```
case X86II::AddRegFrm:
  MCE.emitByte(BaseOpcode + getX86RegNum(MI.getOperand(CurOp++).getReg()));

  if (CurOp != NumOps) {
    const MachineOperand &MO1 = MI.getOperand(CurOp++);
    unsigned Size = X86InstrInfo::sizeOfImm(Desc);
    if (MO1.isImmediate())
      emitConstant(MO1.getImm(), Size);
    else {
      unsigned rt = Is64BitMode ? X86::reloc_pcrel_word
        : (IsPIC ? X86::reloc_picrel_word : X86::reloc_absolute_word);
      if (Opcode == X86::MOV64ri)
        rt = X86::reloc_absolute_dword;  // FIXME: add X86II flag?
      if (MO1.isGlobalAddress()) {
        bool NeedStub = isa<Function>(MO1.getGlobal());
        bool isLazy = gvNeedsLazyPtr(MO1.getGlobal());
        emitGlobalAddress(MO1.getGlobal(), rt, MO1.getOffset(), 0,
                          NeedStub, isLazy);
      } else if (MO1.isExternalSymbol())
        emitExternalSymbolAddress(MO1.getSymbolName(), rt);
      else if (MO1.isConstantPoolIndex())
        emitConstPoolAddress(MO1.getIndex(), rt);
      else if (MO1.isJumpTableIndex())
        emitJumpTableAddress(MO1.getIndex(), rt);
    }
```

```
  }
  break;
```

In the previous example, `XXXCodeEmitter.cpp` uses the variable `rt`, which is a `RelocationType` enum that may be used to relocate addresses (for example, a global address with a PIC base offset). The `RelocationType` enum for that target is defined in the short target-specific `XXXRelocations.h` file. The `RelocationType` is used by the `relocate` method defined in `XXXJITInfo.cpp` to rewrite addresses for referenced global symbols.

For example, `X86Relocations.h` specifies the following relocation types for the X86 addresses. In all four cases, the relocated value is added to the value already in memory. For `reloc_pcrel_word` and `reloc_picrel_word`, there is an additional initial adjustment.

```
enum RelocationType {
  reloc_pcrel_word = 0,    // add reloc value after adjusting for the PC loc
  reloc_picrel_word = 1,   // add reloc value after adjusting for the PIC base
  reloc_absolute_word = 2, // absolute relocation; no additional adjustment
  reloc_absolute_dword = 3 // absolute relocation; no additional adjustment
};
```

Target JIT Info

`XXXJITInfo.cpp` implements the JIT interfaces for target-specific code-generation activities, such as emitting machine code and stubs. At minimum, a target-specific version of `XXXJITInfo` implements the following:

- `getLazyResolverFunction` — Initializes the JIT, gives the target a function that is used for compilation.
- `emitFunctionStub` — Returns a native function with a specified address for a callback function.
- `relocate` — Changes the addresses of referenced globals, based on relocation types.
- Callback function that are wrappers to a function stub that is used when the real target is not initially known.

`getLazyResolverFunction` is generally trivial to implement. It makes the incoming parameter as the global `JITCompilerFunction` and returns the callback function that will be used a function wrapper. For the Alpha target (in `AlphaJITInfo.cpp`), the `getLazyResolverFunction` implementation is simply:

```
TargetJITInfo::LazyResolverFn AlphaJITInfo::getLazyResolverFunction(
                                            JITCompilerFn F) {
  JITCompilerFunction = F;
  return AlphaCompilationCallback;
}
```

For the X86 target, the `getLazyResolverFunction` implementation is a little more complicated, because it returns a different callback function for processors with SSE instructions and XMM registers.

The callback function initially saves and later restores the callee register values, incoming arguments, and frame and return address. The callback function needs low-level access to the registers or stack, so it is typically implemented with assembler.

4.19 Accurate Garbage Collection with LLVM

4.19.1 Introduction

Garbage collection is a widely used technique that frees the programmer from having to know the lifetimes of heap objects, making software easier to produce and maintain. Many programming languages rely on garbage collection for automatic memory management. There are two primary forms of garbage collection: conservative and accurate.

Conservative garbage collection often does not require any special support from either the language or the compiler: it can handle non-type-safe programming languages (such as C/C++) and does not require any special information from the compiler. The Boehm collector is an example of a state-of-the-art conservative collector.

Accurate garbage collection requires the ability to identify all pointers in the program at run-time (which requires that the source-language be type-safe in most cases). Identifying pointers at run-time requires compiler support to locate all places that hold live pointer variables at run-time, including the *processor stack and registers*.

Conservative garbage collection is attractive because it does not require any special compiler support, but it does have problems. In particular, because the conservative garbage collector cannot *know* that a particular word in the machine is a pointer, it cannot move live objects in the heap (preventing the use of compacting and generational GC algorithms) and it can occasionally suffer from memory leaks due to integer values that happen to point to objects in the program. In addition, some aggressive compiler transformations can break conservative garbage collectors (though these seem rare in practice).

Accurate garbage collectors do not suffer from any of these problems, but they can suffer from degraded scalar optimization of the program. In particular, because the runtime must be able to identify and update all pointers active in the program, some optimizations are less effective. In practice, however, the locality and performance benefits of using aggressive garbage collection techniques dominates any low-level losses.

This document describes the mechanisms and interfaces provided by LLVM to support accurate garbage collection.

Goals and non-goals

LLVM's intermediate representation provides *garbage collection intrinsics* that offer support for a broad class of collector models. For instance, the intrinsics permit:

- semi-space collectors
- mark-sweep collectors
- generational collectors
- reference counting
- incremental collectors
- concurrent collectors
- cooperative collectors

We hope that the primitive support built into the LLVM IR is sufficient to support a broad class of garbage collected languages including Scheme, ML, Java, C#, Perl, Python, Lua, Ruby, other scripting languages, and more.

However, LLVM does not itself provide a garbage collector — this should be part of your language's runtime library. LLVM provides a framework for compile time *code generation plugins*. The role of these plugins is to generate code and data structures which conforms to the *binary interface* specified by the *runtime library*. This is similar to the relationship between LLVM and DWARF debugging info, for example. The difference primarily lies in the lack of an established standard in the domain of garbage collection — thus the plugins.

The aspects of the binary interface with which LLVM's GC support is concerned are:

- Creation of GC-safe points within code where collection is allowed to execute safely.
- Computation of the stack map. For each safe point in the code, object references within the stack frame must be identified so that the collector may traverse and perhaps update them.
- Write barriers when storing object references to the heap. These are commonly used to optimize incremental scans in generational collectors.
- Emission of read barriers when loading object references. These are useful for interoperating with concurrent collectors.

There are additional areas that LLVM does not directly address:

- Registration of global roots with the runtime.
- Registration of stack map entries with the runtime.
- The functions used by the program to allocate memory, trigger a collection, etc.
- Computation or compilation of type maps, or registration of them with the runtime. These are used to crawl the heap for object references.

In general, LLVM's support for GC does not include features which can be adequately addressed with other features of the IR and does not specify a particular binary interface. On the plus side, this means that you should be able to integrate LLVM with an existing runtime. On the other hand, it leaves a lot of work for the developer of a novel language. However, it's easy to get started quickly and scale up to a more sophisticated implementation as your compiler matures.

4.19.2 Getting started

Using a GC with LLVM implies many things, for example:

- Write a runtime library or find an existing one which implements a GC heap.
    1. Implement a memory allocator.
    2. Design a binary interface for the stack map, used to identify references within a stack frame on the machine stack.*
    3. Implement a stack crawler to discover functions on the call stack.*

4. Implement a registry for global roots.
5. Design a binary interface for type maps, used to identify references within heap objects.
6. Implement a collection routine bringing together all of the above.

- Emit compatible code from your compiler.
  - Initialization in the main function.
  - Use the `gc "..."` attribute to enable GC code generation (or `F.setGC("...")`).
  - Use `@llvm.gcroot` to mark stack roots.
  - Use `@llvm.gcread` and/or `@llvm.gcwrite` to manipulate GC references, if necessary.
  - Allocate memory using the GC allocation routine provided by the runtime library.
  - Generate type maps according to your runtime's binary interface.
- Write a compiler plugin to interface LLVM with the runtime library.*
  - Lower `@llvm.gcread` and `@llvm.gcwrite` to appropriate code sequences.*
  - Compile LLVM's stack map to the binary form expected by the runtime.
- Load the plugin into the compiler. Use `llc -load` or link the plugin statically with your language's compiler.*
- Link program executables with the runtime.

To help with several of these tasks (those indicated with a *), LLVM includes a highly portable, built-in ShadowStack code generator. It is compiled into `llc` and works even with the interpreter and C backends.

In your compiler

To turn the shadow stack on for your functions, first call:

```
F.setGC("shadow-stack");
```

for each function your compiler emits. Since the shadow stack is built into LLVM, you do not need to load a plugin.

Your compiler must also use `@llvm.gcroot` as documented. Don't forget to create a root for each intermediate value that is generated when evaluating an expression. In `h(f(), g())`, the result of `f()` could easily be collected if evaluating `g()` triggers a collection.

There's no need to use `@llvm.gcread` and `@llvm.gcwrite` over plain `load` and `store` for now. You will need them when switching to a more advanced GC.

In your runtime

The shadow stack doesn't imply a memory allocation algorithm. A semispace collector or building atop `malloc` are great places to start, and can be implemented with very little code.

When it comes time to collect, however, your runtime needs to traverse the stack roots, and for this it needs to integrate with the shadow stack. Luckily, doing so is very simple. (This code is heavily commented to help you understand the data structure, but there are only 20 lines of meaningful code.)

```
/// @brief The map for a single function's stack frame.  One of these is
///        compiled as constant data into the executable for each function.
///
/// Storage of metadata values is elided if the %metadata parameter to
/// @llvm.gcroot is null.
struct FrameMap {
```

```
  int32_t NumRoots;    //< Number of roots in stack frame.
  int32_t NumMeta;     //< Number of metadata entries.  May be < NumRoots.
  const void *Meta[0]; //< Metadata for each root.
};

/// @brief A link in the dynamic shadow stack.  One of these is embedded in
///        the stack frame of each function on the call stack.
struct StackEntry {
  StackEntry *Next;    //< Link to next stack entry (the caller's).
  const FrameMap *Map; //< Pointer to constant FrameMap.
  void *Roots[0];      //< Stack roots (in-place array).
};

/// @brief The head of the singly-linked list of StackEntries.  Functions push
///        and pop onto this in their prologue and epilogue.
///
/// Since there is only a global list, this technique is not threadsafe.
StackEntry *llvm_gc_root_chain;

/// @brief Calls Visitor(root, meta) for each GC root on the stack.
///        root and meta are exactly the values passed to
///        @llvm.gcroot.
///
/// Visitor could be a function to recursively mark live objects.  Or it
/// might copy them to another heap or generation.
///
/// @param Visitor A function to invoke for every GC root on the stack.
void visitGCRoots(void (*Visitor)(void **Root, const void *Meta)) {
  for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
    unsigned i = 0;

    // For roots [0, NumMeta), the metadata pointer is in the FrameMap.
    for (unsigned e = R->Map->NumMeta; i != e; ++i)
      Visitor(&R->Roots[i], R->Map->Meta[i]);

    // For roots [NumMeta, NumRoots), the metadata pointer is null.
    for (unsigned e = R->Map->NumRoots; i != e; ++i)
      Visitor(&R->Roots[i], NULL);
  }
}
```

About the shadow stack

Unlike many GC algorithms which rely on a cooperative code generator to compile stack maps, this algorithm carefully maintains a linked list of stack roots [*Henderson2002*]. This so-called "shadow stack" mirrors the machine stack. Maintaining this data structure is slower than using a stack map compiled into the executable as constant data, but has a significant portability advantage because it requires no special support from the target code generator, and does not require tricky platform-specific code to crawl the machine stack.

The tradeoff for this simplicity and portability is:

- High overhead per function call.
- Not thread-safe.

Still, it's an easy way to get started. After your compiler and runtime are up and running, writing a *plugin* will allow you to take advantage of *more advanced GC features* of LLVM in order to improve performance.

4.19.3 IR features

This section describes the garbage collection facilities provided by the *LLVM intermediate representation*. The exact behavior of these IR features is specified by the binary interface implemented by a *code generation plugin*, not by this document.

These facilities are limited to those strictly necessary; they are not intended to be a complete interface to any garbage collector. A program will need to interface with the GC library using the facilities provided by that program.

Specifying GC code generation: `gc "..."`

```
define ty @name(...) gc "name" { ...
```

The `gc` function attribute is used to specify the desired GC style to the compiler. Its programmatic equivalent is the `setGC` method of `Function`.

Setting `gc "name"` on a function triggers a search for a matching code generation plugin "*name*"; it is that plugin which defines the exact nature of the code generated to support GC. If none is found, the compiler will raise an error.

Specifying the GC style on a per-function basis allows LLVM to link together programs that use different garbage collection algorithms (or none at all).

Identifying GC roots on the stack: `llvm.gcroot`

```
void @llvm.gcroot(i8** %ptrloc, i8* %metadata)
```

The `llvm.gcroot` intrinsic is used to inform LLVM that a stack variable references an object on the heap and is to be tracked for garbage collection. The exact impact on generated code is specified by a *compiler plugin*. All calls to `llvm.gcroot` **must** reside inside the first basic block.

A compiler which uses mem2reg to raise imperative code using `alloca` into SSA form need only add a call to `@llvm.gcroot` for those variables which a pointers into the GC heap.

It is also important to mark intermediate values with `llvm.gcroot`. For example, consider `h(f(), g())`. Beware leaking the result of `f()` in the case that `g()` triggers a collection. Note, that stack variables must be initialized and marked with `llvm.gcroot` in function's prologue.

The first argument **must** be a value referring to an alloca instruction or a bitcast of an alloca. The second contains a pointer to metadata that should be associated with the pointer, and **must** be a constant or global value address. If your target collector uses tags, use a null pointer for metadata.

The `%metadata` argument can be used to avoid requiring heap objects to have 'isa' pointers or tag bits. [Appel89, Goldberg91, Tolmach94] If specified, its value will be tracked along with the location of the pointer in the stack frame.

Consider the following fragment of Java code:

```
{
  Object X;   // A null-initialized reference to an object
  ...
}
```

This block (which may be located in the middle of a function or in a loop nest), could be compiled to this LLVM code:

```
Entry:
   ;; In the entry block for the function, allocate the
   ;; stack space for X, which is an LLVM pointer.
   %X = alloca %Object*
```

```
    ;; Tell LLVM that the stack space is a stack root.
    ;; Java has type-tags on objects, so we pass null as metadata.
    %tmp = bitcast %Object** %X to i8**
    call void @llvm.gcroot(i8** %tmp, i8* null)
    ...

    ;; "CodeBlock" is the block corresponding to the start
    ;;  of the scope above.
CodeBlock:
    ;; Java null-initializes pointers.
    store %Object* null, %Object** %X

    ...

    ;; As the pointer goes out of scope, store a null value into
    ;; it, to indicate that the value is no longer live.
    store %Object* null, %Object** %X
    ...
```

Reading and writing references in the heap

Some collectors need to be informed when the mutator (the program that needs garbage collection) either reads a pointer from or writes a pointer to a field of a heap object. The code fragments inserted at these points are called *read barriers* and *write barriers*, respectively. The amount of code that needs to be executed is usually quite small and not on the critical path of any computation, so the overall performance impact of the barrier is tolerable.

Barriers often require access to the *object pointer* rather than the *derived pointer* (which is a pointer to the field within the object). Accordingly, these intrinsics take both pointers as separate arguments for completeness. In this snippet, `%object` is the object pointer, and `%derived` is the derived pointer:

```
;; An array type.
%class.Array = type { %class.Object, i32, [0 x %class.Object*] }
...

;; Load the object pointer from a gcroot.
%object = load %class.Array** %object_addr

;; Compute the derived pointer.
%derived = getelementptr %object, i32 0, i32 2, i32 %n
```

LLVM does not enforce this relationship between the object and derived pointer (although a *plugin* might). However, it would be an unusual collector that violated it.

The use of these intrinsics is naturally optional if the target GC does require the corresponding barrier. Such a GC plugin will replace the intrinsic calls with the corresponding `load` or `store` instruction if they are used.

Write barrier: `llvm.gcwrite`

```
void @llvm.gcwrite(i8* %value, i8* %object, i8** %derived)
```

For write barriers, LLVM provides the `llvm.gcwrite` intrinsic function. It has exactly the same semantics as a non-volatile `store` to the derived pointer (the third argument). The exact code generated is specified by a compiler *plugin*.

Many important algorithms require write barriers, including generational and concurrent collectors. Additionally, write barriers could be used to implement reference counting.

Read barrier: `llvm.gcread`

```
i8* @llvm.gcread(i8* %object, i8** %derived)
```

For read barriers, LLVM provides the `llvm.gcread` intrinsic function. It has exactly the same semantics as a non-volatile `load` from the derived pointer (the second argument). The exact code generated is specified by a *compiler plugin*.

Read barriers are needed by fewer algorithms than write barriers, and may have a greater performance impact since pointer reads are more frequent than writes.

4.19.4 Implementing a collector plugin

User code specifies which GC code generation to use with the `gc` function attribute or, equivalently, with the `setGC` method of `Function`.

To implement a GC plugin, it is necessary to subclass `llvm::GCStrategy`, which can be accomplished in a few lines of boilerplate code. LLVM's infrastructure provides access to several important algorithms. For an uncontroversial collector, all that remains may be to compile LLVM's computed stack map to assembly code (using the binary representation expected by the runtime library). This can be accomplished in about 100 lines of code.

This is not the appropriate place to implement a garbage collected heap or a garbage collector itself. That code should exist in the language's runtime library. The compiler plugin is responsible for generating code which conforms to the binary interface defined by library, most essentially the *stack map*.

To subclass `llvm::GCStrategy` and register it with the compiler:

```
// lib/MyGC/MyGC.cpp - Example LLVM GC plugin

#include "llvm/CodeGen/GCStrategy.h"
#include "llvm/CodeGen/GCMetadata.h"
#include "llvm/Support/Compiler.h"

using namespace llvm;

namespace {
  class LLVM_LIBRARY_VISIBILITY MyGC : public GCStrategy {
  public:
    MyGC() {}
  };

  GCRegistry::Add<MyGC>
  X("mygc", "My bespoke garbage collector.");
}
```

This boilerplate collector does nothing. More specifically:

- `llvm.gcread` calls are replaced with the corresponding `load` instruction.
- `llvm.gcwrite` calls are replaced with the corresponding `store` instruction.
- No safe points are added to the code.
- The stack map is not compiled into the executable.

Using the LLVM makefiles, this code can be compiled as a plugin using a simple makefile:

```
# lib/MyGC/Makefile

LEVEL := ../..
```

```
LIBRARYNAME = MyGC
LOADABLE_MODULE = 1

include $(LEVEL)/Makefile.common
```

Once the plugin is compiled, code using it may be compiled using `llc -load=MyGC.so` (though MyGC.so may have some other platform-specific extension):

```
$ cat sample.ll
define void @f() gc "mygc" {
entry:
  ret void
}
$ llvm-as < sample.ll | llc -load=MyGC.so
```

It is also possible to statically link the collector plugin into tools, such as a language-specific compiler front-end.

Overview of available features

`GCStrategy` provides a range of features through which a plugin may do useful work. Some of these are callbacks, some are algorithms that can be enabled, disabled, or customized. This matrix summarizes the supported (and planned) features and correlates them with the collection techniques which typically require them.

Algorithm	Done	Shadow stack	ref-count	mark-sweep	copy-ing	incre-mental	threaded	con-current
stack map	[U+2714]			[U+2718]	[U+2718]	[U+2718]	[U+2718]	[U+2718]
initialize roots	[U+2714]	[U+2718]	[U+2718]	[U+2718]	[U+2718]	[U+2718]	[U+2718]	[U+2718]
derived pointers	NO						**N***	**N***
custom lowering	[U+2714]							
gcroot	[U+2714]	[U+2718]	[U+2718]					
gcwrite	[U+2714]		[U+2718]			[U+2718]		[U+2718]
gcread	[U+2714]							[U+2718]
safe points								
in calls	[U+2714]			[U+2718]	[U+2718]	[U+2718]	[U+2718]	[U+2718]
before calls	[U+2714]						[U+2718]	[U+2718]
for loops	NO						**N**	**N**
before escape	[U+2714]						[U+2718]	[U+2718]
emit code at safe points	NO						**N**	**N**
output								
assembly	[U+2714]			[U+2718]	[U+2718]	[U+2718]	[U+2718]	[U+2718]
JIT	NO			**?**	**?**	**?**	**?**	**?**
obj	NO			**?**	**?**	**?**	**?**	**?**
live analysis	NO			**?**	**?**	**?**	**?**	**?**
register map	NO			**?**	**?**	**?**	**?**	**?**
* Derived pointers only pose a hasard to copying collections.								
? denotes a feature which could be utilized if available.								

To be clear, the collection techniques above are defined as:

Shadow Stack The mutator carefully maintains a linked list of stack roots.

Reference Counting The mutator maintains a reference count for each object and frees an object when its count falls to zero.

Mark-Sweep When the heap is exhausted, the collector marks reachable objects starting from the roots, then deallocates unreachable objects in a sweep phase.

Copying As reachability analysis proceeds, the collector copies objects from one heap area to another, compacting them in the process. Copying collectors enable highly efficient "bump pointer" allocation and can improve locality of reference.

Incremental (Including generational collectors.) Incremental collectors generally have all the properties of a copying collector (regardless of whether the mature heap is compacting), but bring the added complexity of requiring write barriers.

Threaded Denotes a multithreaded mutator; the collector must still stop the mutator ("stop the world") before beginning reachability analysis. Stopping a multithreaded mutator is a complicated problem. It generally requires highly platform-specific code in the runtime, and the production of carefully designed machine code at safe points.

Concurrent In this technique, the mutator and the collector run concurrently, with the goal of eliminating pause times. In a *cooperative* collector, the mutator further aids with collection should a pause occur, allowing collection to take advantage of multiprocessor hosts. The "stop the world" problem of threaded collectors is generally still present to a limited extent. Sophisticated marking algorithms are necessary. Read barriers may be necessary.

As the matrix indicates, LLVM's garbage collection infrastructure is already suitable for a wide variety of collectors, but does not currently extend to multithreaded programs. This will be added in the future as there is interest.

Computing stack maps

LLVM automatically computes a stack map. One of the most important features of a `GCStrategy` is to compile this information into the executable in the binary representation expected by the runtime library.

The stack map consists of the location and identity of each GC root in the each function in the module. For each root:

- `RootNum`: The index of the root.
- `StackOffset`: The offset of the object relative to the frame pointer.
- `RootMetadata`: The value passed as the `%metadata` parameter to the `@llvm.gcroot` intrinsic.

Also, for the function as a whole:

- **`getFrameSize()`: The overall size of the function's initial stack frame,** not accounting for any dynamic allocation.
- `roots_size()`: The count of roots in the function.

To access the stack map, use `GCFunctionMetadata::roots_begin()` and `-end()` from the *GCMetadataPrinter*:

```
for (iterator I = begin(), E = end(); I != E; ++I) {
  GCFunctionInfo *FI = *I;
  unsigned FrameSize = FI->getFrameSize();
  size_t RootCount = FI->roots_size();

  for (GCFunctionInfo::roots_iterator RI = FI->roots_begin(),
                                      RE = FI->roots_end();
                                      RI != RE; ++RI) {
    int RootNum = RI->Num;
    int RootStackOffset = RI->StackOffset;
    Constant *RootMetadata = RI->Metadata;
  }
}
```

If the `llvm.gcroot` intrinsic is eliminated before code generation by a custom lowering pass, LLVM will compute an empty stack map. This may be useful for collector plugins which implement reference counting or a shadow stack.

Initializing roots to null: `InitRoots`

```
MyGC::MyGC() {
  InitRoots = true;
}
```

When set, LLVM will automatically initialize each root to `null` upon entry to the function. This prevents the GC's sweep phase from visiting uninitialized pointers, which will almost certainly cause it to crash. This initialization occurs before custom lowering, so the two may be used together.

Since LLVM does not yet compute liveness information, there is no means of distinguishing an uninitialized stack root from an initialized one. Therefore, this feature should be used by all GC plugins. It is enabled by default.

Custom lowering of intrinsics: `CustomRoots`, `CustomReadBarriers`, and `CustomWriteBarriers`

For GCs which use barriers or unusual treatment of stack roots, these flags allow the collector to perform arbitrary transformations of the LLVM IR:

```
class MyGC : public GCStrategy {
public:
  MyGC() {
    CustomRoots = true;
    CustomReadBarriers = true;
    CustomWriteBarriers = true;
  }

  virtual bool initializeCustomLowering(Module &M);
  virtual bool performCustomLowering(Function &F);
};
```

If any of these flags are set, then LLVM suppresses its default lowering for the corresponding intrinsics and instead calls `performCustomLowering`.

LLVM's default action for each intrinsic is as follows:

- `llvm.gcroot`: Leave it alone. The code generator must see it or the stack map will not be computed.
- `llvm.gcread`: Substitute a `load` instruction.
- `llvm.gcwrite`: Substitute a `store` instruction.

If `CustomReadBarriers` or `CustomWriteBarriers` are specified, then `performCustomLowering` **must** eliminate the corresponding barriers.

`performCustomLowering` must comply with the same restrictions as *FunctionPass::runOnFunction* Likewise, `initializeCustomLowering` has the same semantics as *Pass::doInitialization(Module&)*

The following can be used as a template:

```
#include "llvm/IR/Module.h"
#include "llvm/IR/IntrinsicInst.h"

bool MyGC::initializeCustomLowering(Module &M) {
  return false;
}
```

```
bool MyGC::performCustomLowering(Function &F) {
  bool MadeChange = false;

  for (Function::iterator BB = F.begin(), E = F.end(); BB != E; ++BB)
    for (BasicBlock::iterator II = BB->begin(), E = BB->end(); II != E; )
      if (IntrinsicInst *CI = dyn_cast<IntrinsicInst>(II++))
        if (Function *F = CI->getCalledFunction())
          switch (F->getIntrinsicID()) {
          case Intrinsic::gcwrite:
            // Handle llvm.gcwrite.
            CI->eraseFromParent();
            MadeChange = true;
            break;
          case Intrinsic::gcread:
            // Handle llvm.gcread.
            CI->eraseFromParent();
            MadeChange = true;
            break;
          case Intrinsic::gcroot:
            // Handle llvm.gcroot.
            CI->eraseFromParent();
            MadeChange = true;
            break;
          }

  return MadeChange;
}
```

Generating safe points: `NeededSafePoints`

LLVM can compute four kinds of safe points:

```
namespace GC {
  /// PointKind - The type of a collector-safe point.
  ///
  enum PointKind {
    Loop,     //< Instr is a loop (backwards branch).
    Return,   //< Instr is a return instruction.
    PreCall,  //< Instr is a call instruction.
    PostCall  //< Instr is the return address of a call.
  };
}
```

A collector can request any combination of the four by setting the `NeededSafePoints` mask:

```
MyGC::MyGC()  {
  NeededSafePoints = 1 << GC::Loop
                   | 1 << GC::Return
                   | 1 << GC::PreCall
                   | 1 << GC::PostCall;
}
```

It can then use the following routines to access safe points.

```
for (iterator I = begin(), E = end(); I != E; ++I) {
  GCFunctionInfo *MD = *I;
  size_t PointCount = MD->size();
```

```
  for (GCFunctionInfo::iterator PI = MD->begin(),
                                PE = MD->end(); PI != PE; ++PI) {
    GC::PointKind PointKind = PI->Kind;
    unsigned PointNum = PI->Num;
  }
}
```

Almost every collector requires `PostCall` safe points, since these correspond to the moments when the function is suspended during a call to a subroutine.

Threaded programs generally require `Loop` safe points to guarantee that the application will reach a safe point within a bounded amount of time, even if it is executing a long-running loop which contains no function calls.

Threaded collectors may also require `Return` and `PreCall` safe points to implement "stop the world" techniques using self-modifying code, where it is important that the program not exit the function without reaching a safe point (because only the topmost function has been patched).

Emitting assembly code: `GCMetadataPrinter`

LLVM allows a plugin to print arbitrary assembly code before and after the rest of a module's assembly code. At the end of the module, the GC can compile the LLVM stack map into assembly code. (At the beginning, this information is not yet computed.)

Since AsmWriter and CodeGen are separate components of LLVM, a separate abstract base class and registry is provided for printing assembly code, the `GCMetadaPrinter` and `GCMetadataPrinterRegistry`. The AsmWriter will look for such a subclass if the `GCStrategy` sets `UsesMetadata`:

```
MyGC::MyGC() {
  UsesMetadata = true;
}
```

This separation allows JIT-only clients to be smaller.

Note that LLVM does not currently have analogous APIs to support code generation in the JIT, nor using the object writers.

```
// lib/MyGC/MyGCPrinter.cpp - Example LLVM GC printer

#include "llvm/CodeGen/GCMetadataPrinter.h"
#include "llvm/Support/Compiler.h"

using namespace llvm;

namespace {
  class LLVM_LIBRARY_VISIBILITY MyGCPrinter : public GCMetadataPrinter {
  public:
    virtual void beginAssembly(AsmPrinter &AP);

    virtual void finishAssembly(AsmPrinter &AP);
  };

  GCMetadataPrinterRegistry::Add<MyGCPrinter>
  X("mygc", "My bespoke garbage collector.");
}
```

The collector should use `AsmPrinter` to print portable assembly code. The collector itself contains the stack map for the entire module, and may access the `GCFunctionInfo` using its own `begin()` and `end()` methods. Here's a realistic example:

```
#include "llvm/CodeGen/AsmPrinter.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/Target/TargetAsmInfo.h"
#include "llvm/Target/TargetMachine.h"

void MyGCPrinter::beginAssembly(AsmPrinter &AP) {
  // Nothing to do.
}

void MyGCPrinter::finishAssembly(AsmPrinter &AP) {
  MCStreamer &OS = AP.OutStreamer;
  unsigned IntPtrSize = AP.TM.getSubtargetImpl()->getDataLayout()->getPointerSize();

  // Put this in the data section.
  OS.SwitchSection(AP.getObjFileLowering().getDataSection());

  // For each function...
  for (iterator FI = begin(), FE = end(); FI != FE; ++FI) {
    GCFunctionInfo &MD = **FI;

    // A compact GC layout. Emit this data structure:
    //
    // struct {
    //   int32_t PointCount;
    //   void *SafePointAddress[PointCount];
    //   int32_t StackFrameSize; // in words
    //   int32_t StackArity;
    //   int32_t LiveCount;
    //   int32_t LiveOffsets[LiveCount];
    // } __gcmap_<FUNCTIONNAME>;

    // Align to address width.
    AP.EmitAlignment(IntPtrSize == 4 ? 2 : 3);

    // Emit PointCount.
    OS.AddComment("safe point count");
    AP.EmitInt32(MD.size());

    // And each safe point...
    for (GCFunctionInfo::iterator PI = MD.begin(),
                                  PE = MD.end(); PI != PE; ++PI) {
      // Emit the address of the safe point.
      OS.AddComment("safe point address");
      MCSymbol *Label = PI->Label;
      AP.EmitLabelPlusOffset(Label/*Hi*/, 0/*Offset*/, 4/*Size*/);
    }

    // Stack information never change in safe points! Only print info from the
    // first call-site.
    GCFunctionInfo::iterator PI = MD.begin();

    // Emit the stack frame size.
    OS.AddComment("stack frame size (in words)");
    AP.EmitInt32(MD.getFrameSize() / IntPtrSize);

    // Emit stack arity, i.e. the number of stacked arguments.
    unsigned RegisteredArgs = IntPtrSize == 4 ? 5 : 6;
```

```
    unsigned StackArity = MD.getFunction().arg_size() > RegisteredArgs ?
                          MD.getFunction().arg_size() - RegisteredArgs : 0;
    OS.AddComment("stack arity");
    AP.EmitInt32(StackArity);

    // Emit the number of live roots in the function.
    OS.AddComment("live root count");
    AP.EmitInt32(MD.live_size(PI));

    // And for each live root...
    for (GCFunctionInfo::live_iterator LI = MD.live_begin(PI),
                                       LE = MD.live_end(PI);
                                       LI != LE; ++LI) {
      // Emit live root's offset within the stack frame.
      OS.AddComment("stack index (offset / wordsize)");
      AP.EmitInt32(LI->StackOffset);
    }
  }
}
```

4.19.5 References

[Appel89] Runtime Tags Aren't Necessary. Andrew W. Appel. Lisp and Symbolic Computation 19(7):703-705, July 1989. [Goldberg91] Tag-free garbage collection for strongly typed programming languages. Benjamin Goldberg. ACM SIGPLAN PLDI'91. [Tolmach94] Tag-free garbage collection using explicit type parameters. Andrew Tolmach. Proceedings of the 1994 ACM conference on LISP and functional programming. [Henderson2002] Accurate Garbage Collection in an Uncooperative Environment

4.20 Writing an LLVM Pass

4.20.1 Introduction — What is a pass?

The LLVM Pass Framework is an important part of the LLVM system, because LLVM passes are where most of the interesting parts of the compiler exist. Passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations, and they are, above all, a structuring technique for compiler code.

All LLVM passes are subclasses of the Pass class, which implement functionality by overriding virtual methods inherited from `Pass`. Depending on how your pass works, you should inherit from the *ModulePass* , *CallGraphSCCPass*, *FunctionPass* , or *LoopPass*, or *RegionPass*, or *BasicBlockPass* classes, which gives the system more information about what your pass does, and how it can be combined with other passes. One of the main features of the LLVM Pass Framework is that it schedules passes to run in an efficient way based on the constraints that your pass meets (which are indicated by which class they derive from).

We start by showing you how to construct a pass, everything from setting up the code, to compiling, loading, and executing it. After the basics are down, more advanced features are discussed.

4.20.2 Quick Start — Writing hello world

Here we describe how to write the "hello world" of passes. The "Hello" pass is designed to simply print out the name of non-external functions that exist in the program being compiled. It does not modify the program at all, it just inspects it. The source code and files for this pass are available in the LLVM source tree in the `lib/Transforms/Hello` directory.

Setting up the build environment

First, configure and build LLVM. This needs to be done directly inside the LLVM source tree rather than in a separate objects directory. Next, you need to create a new directory somewhere in the LLVM source base. For this example, we'll assume that you made `lib/Transforms/Hello`. Finally, you must set up a build script (`Makefile`) that will compile the source code for the new pass. To do this, copy the following into `Makefile`:

```
# Makefile for hello pass

# Path to top level of LLVM hierarchy
LEVEL = ../../..

# Name of the library to build
LIBRARYNAME = Hello

# Make the shared library become a loadable module so the tools can
# dlopen/dlsym on the resulting library.
LOADABLE_MODULE = 1

# Include the makefile implementation stuff
include $(LEVEL)/Makefile.common
```

This makefile specifies that all of the `.cpp` files in the current directory are to be compiled and linked together into a shared object `$(LEVEL)/Debug+Asserts/lib/Hello.so` that can be dynamically loaded by the **opt** or **bugpoint** tools via their *-load* options. If your operating system uses a suffix other than `.so` (such as Windows or Mac OS X), the appropriate extension will be used.

If you are used CMake to build LLVM, see *Developing LLVM passes out of source*.

Now that we have the build scripts set up, we just need to write the code for the pass itself.

Basic code required

Now that we have a way to compile our new pass, we just have to write it. Start out with:

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
```

Which are needed because we are writing a Pass, we are operating on Functions, and we will be doing some printing.

Next we have:

```
using namespace llvm;
```

... which is required because the functions from the include files live in the llvm namespace.

Next we have:

```
namespace {
```

... which starts out an anonymous namespace. Anonymous namespaces are to C++ what the "`static`" keyword is to C (at global scope). It makes the things declared inside of the anonymous namespace visible only to the current file. If you're not familiar with them, consult a decent C++ book for more information.

Next, we declare our pass itself:

```
struct Hello : public FunctionPass {
```

This declares a "`Hello`" class that is a subclass of *FunctionPass*. The different builtin pass subclasses are described in detail *later*, but for now, know that `FunctionPass` operates on a function at a time.

```
static char ID;
Hello() : FunctionPass(ID) {}
```

This declares pass identifier used by LLVM to identify pass. This allows LLVM to avoid using expensive C++ runtime information.

```
    virtual bool runOnFunction(Function &F) {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << "\n";
      return false;
    }
  }; // end of struct Hello
}  // end of anonymous namespace
```

We declare a *runOnFunction* method, which overrides an abstract virtual method inherited from *FunctionPass*. This is where we are supposed to do our thing, so we just print out our message with the name of each function.

```
char Hello::ID = 0;
```

We initialize pass ID here. LLVM uses ID's address to identify a pass, so initialization value is not important.

```
static RegisterPass<Hello> X("hello", "Hello World Pass",
                             false /* Only looks at CFG */,
                             false /* Analysis Pass */);
```

Lastly, we *register our class* `Hello`, giving it a command line argument "`hello`", and a name "Hello World Pass". The last two arguments describe its behavior: if a pass walks CFG without modifying it then the third argument is set to `true`; if a pass is an analysis pass, for example dominator tree pass, then `true` is supplied as the fourth argument.

As a whole, the `.cpp` file looks like:

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
  struct Hello : public FunctionPass {
    static char ID;
    Hello() : FunctionPass(ID) {}

    virtual bool runOnFunction(Function &F) {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass", false, false);
```

Now that it's all together, compile the file with a simple "`gmake`" command in the local directory and you should get a new file "`Debug+Asserts/lib/Hello.so`" under the top level directory of the LLVM source tree (not in the local directory). Note that everything in this file is contained in an anonymous namespace — this reflects the fact that passes are self contained units that do not need external interfaces (although they can have them) to be useful.

Running a pass with `opt`

Now that you have a brand new shiny shared object file, we can use the **opt** command to run an LLVM program through your pass. Because you registered your pass with `RegisterPass`, you will be able to use the **opt** tool to access it, once loaded.

To test it, follow the example at the end of the *Getting Started with the LLVM System* to compile "Hello World" to LLVM. We can now run the bitcode file (hello.bc) for the program through our transformation like this (or course, any bitcode file will work):

```
$ opt -load ../../../Debug+Asserts/lib/Hello.so -hello < hello.bc > /dev/null
Hello: __main
Hello: puts
Hello: main
```

The *-load* option specifies that **opt** should load your pass as a shared object, which makes "`-hello`" a valid command line argument (which is one reason you need to *register your pass*). Because the Hello pass does not modify the program in any interesting way, we just throw away the result of **opt** (sending it to `/dev/null`).

To see what happened to the other string you registered, try running **opt** with the *-help* option:

```
$ opt -load ../../../Debug+Asserts/lib/Hello.so -help
OVERVIEW: llvm .bc -> .bc modular optimizer

USAGE: opt [options] <input bitcode>

OPTIONS:
  Optimizations available:
...
    -globalopt                - Global Variable Optimizer
    -globalsmodref-aa         - Simple mod/ref analysis for globals
```

```
    -gvn                        - Global Value Numbering
    -hello                      - Hello World Pass
    -indvars                    - Induction Variable Simplification
    -inline                     - Function Integration/Inlining
...
```

The pass name gets added as the information string for your pass, giving some documentation to users of **opt**. Now that you have a working pass, you would go ahead and make it do the cool transformations you want. Once you get it all working and tested, it may become useful to find out how fast your pass is. The *PassManager* provides a nice command line option (*--time-passes*) that allows you to get information about the execution time of your pass along with the other passes you queue up. For example:

```
$ opt -load ../../../Debug+Asserts/lib/Hello.so -hello -time-passes < hello.bc > /dev/null
Hello: __main
Hello: puts
Hello: main
===============================================================================
                      ... Pass execution timing report ...
===============================================================================
  Total Execution Time: 0.02 seconds (0.0479059 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Pass Name ---
   0.0100 (100.0%)   0.0000 (  0.0%)   0.0100 ( 50.0%)   0.0402 ( 84.0%)  Bitcode Writer
   0.0000 (  0.0%)   0.0100 (100.0%)   0.0100 ( 50.0%)   0.0031 (  6.4%)  Dominator Set Construction
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0013 (  2.7%)  Module Verifier
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0033 (  6.9%)  Hello World Pass
   0.0100 (100.0%)   0.0100 (100.0%)   0.0200 (100.0%)   0.0479 (100.0%)  TOTAL
```

As you can see, our implementation above is pretty fast. The additional passes listed are automatically inserted by the **opt** tool to verify that the LLVM emitted by your pass is still valid and well formed LLVM, which hasn't been broken somehow.

Now that you have seen the basics of the mechanics behind passes, we can talk about some more details of how they work and how to use them.

4.20.3 Pass classes and requirements

One of the first things that you should do when designing a new pass is to decide what class you should subclass for your pass. The *Hello World* example uses the *FunctionPass* class for its implementation, but we did not discuss why or when this should occur. Here we talk about the classes available, from the most general to the most specific.

When choosing a superclass for your `Pass`, you should choose the **most specific** class possible, while still being able to meet the requirements listed. This gives the LLVM Pass Infrastructure information necessary to optimize how passes are run, so that the resultant compiler isn't unnecessarily slow.

The `ImmutablePass` class

The most plain and boring type of pass is the "ImmutablePass" class. This pass type is used for passes that do not have to be run, do not change state, and never need to be updated. This is not a normal type of transformation or analysis, but can provide information about the current compiler configuration.

Although this pass class is very infrequently used, it is important for providing information about the current target machine being compiled for, and other static information that can affect the various transformations.

`ImmutablePass`es never invalidate other transformations, are never invalidated, and are never "run".

The `ModulePass` class

The ModulePass class is the most general of all superclasses that you can use. Deriving from `ModulePass` indicates that your pass uses the entire program as a unit, referring to function bodies in no predictable order, or adding and removing functions. Because nothing is known about the behavior of `ModulePass` subclasses, no optimization can be done for their execution.

A module pass can use function level passes (e.g. dominators) using the `getAnalysis` interface `getAnalysis<DominatorTree>(llvm::Function *)` to provide the function to retrieve analysis result for, if the function pass does not require any module or immutable passes. Note that this can only be done for functions for which the analysis ran, e.g. in the case of dominators you should only ask for the `DominatorTree` for function definitions, not declarations.

To write a correct `ModulePass` subclass, derive from `ModulePass` and overload the `runOnModule` method with the following signature:

The `runOnModule` method

```
virtual bool runOnModule(Module &M) = 0;
```

The `runOnModule` method performs the interesting work of the pass. It should return `true` if the module was modified by the transformation and `false` otherwise.

The `CallGraphSCCPass` class

The CallGraphSCCPass is used by passes that need to traverse the program bottom-up on the call graph (callees before callers). Deriving from `CallGraphSCCPass` provides some mechanics for building and traversing the `CallGraph`, but also allows the system to optimize execution of `CallGraphSCCPass`es. If your pass meets the requirements outlined below, and doesn't meet the requirements of a *FunctionPass* or *BasicBlockPass*, you should derive from `CallGraphSCCPass`.

`TODO`: explain briefly what SCC, Tarjan's algo, and B-U mean.

To be explicit, CallGraphSCCPass subclasses are:

1. ... *not allowed* to inspect or modify any `Functions` other than those in the current SCC and the direct callers and direct callees of the SCC.
2. ... *required* to preserve the current `CallGraph` object, updating it to reflect any changes made to the program.
3. ... *not allowed* to add or remove SCC's from the current Module, though they may change the contents of an SCC.
4. ... *allowed* to add or remove global variables from the current Module.
5. ... *allowed* to maintain state across invocations of *runOnSCC* (including global data).

Implementing a `CallGraphSCCPass` is slightly tricky in some cases because it has to handle SCCs with more than one node in it. All of the virtual methods described below should return `true` if they modified the program, or `false` if they didn't.

The `doInitialization(CallGraph &)` method

```
virtual bool doInitialization(CallGraph &CG);
```

The `doInitialization` method is allowed to do most of the things that `CallGraphSCCPass`es are not allowed to do. They can add and remove functions, get pointers to functions, etc. The `doInitialization` method is designed to do simple initialization type of stuff that does not depend on the SCCs being processed. The `doInitialization` method call is not scheduled to overlap with any other pass executions (thus it should be very fast).

The `runOnSCC` method

```
virtual bool runOnSCC(CallGraphSCC &SCC) = 0;
```

The `runOnSCC` method performs the interesting work of the pass, and should return `true` if the module was modified by the transformation, `false` otherwise.

The `doFinalization(CallGraph &)` method

```
virtual bool doFinalization(CallGraph &CG);
```

The `doFinalization` method is an infrequently used method that is called when the pass framework has finished calling *runOnFunction* for every function in the program being compiled.

The `FunctionPass` class

In contrast to `ModulePass` subclasses, FunctionPass subclasses do have a predictable, local behavior that can be expected by the system. All `FunctionPass` execute on each function in the program independent of all of the other functions in the program. `FunctionPass`es do not require that they are executed in a particular order, and `FunctionPass`es do not modify external functions.

To be explicit, `FunctionPass` subclasses are not allowed to:

1. Inspect or modify a `Function` other than the one currently being processed.
2. Add or remove `Function`s from the current `Module`.
3. Add or remove global variables from the current `Module`.
4. Maintain state across invocations of:ref:*runOnFunction* <*writing-an-llvm-pass-runOnFunction*> (including global data).

Implementing a `FunctionPass` is usually straightforward (See the *Hello World* pass for example). `FunctionPass`es may overload three virtual methods to do their work. All of these methods should return `true` if they modified the program, or `false` if they didn't.

The `doInitialization(Module &)` method

```
virtual bool doInitialization(Module &M);
```

The `doInitialization` method is allowed to do most of the things that `FunctionPass`es are not allowed to do. They can add and remove functions, get pointers to functions, etc. The `doInitialization` method is designed to do simple initialization type of stuff that does not depend on the functions being processed. The `doInitialization` method call is not scheduled to overlap with any other pass executions (thus it should be very fast).

A good example of how this method should be used is the LowerAllocations pass. This pass converts `malloc` and `free` instructions into platform dependent `malloc()` and `free()` function calls. It uses the

`doInitialization` method to get a reference to the `malloc` and `free` functions that it needs, adding prototypes to the module if necessary.

The `runOnFunction` method

```
virtual bool runOnFunction(Function &F) = 0;
```

The `runOnFunction` method must be implemented by your subclass to do the transformation or analysis work of your pass. As usual, a `true` value should be returned if the function is modified.

The `doFinalization(Module &)` method

```
virtual bool doFinalization(Module &M);
```

The `doFinalization` method is an infrequently used method that is called when the pass framework has finished calling *runOnFunction* for every function in the program being compiled.

The `LoopPass` class

All `LoopPass` execute on each loop in the function independent of all of the other loops in the function. `LoopPass` processes loops in loop nest order such that outer most loop is processed last.

`LoopPass` subclasses are allowed to update loop nest using `LPPassManager` interface. Implementing a loop pass is usually straightforward. `LoopPass`es may overload three virtual methods to do their work. All these methods should return `true` if they modified the program, or `false` if they didn't.

The `doInitialization(Loop *, LPPassManager &)` method

```
virtual bool doInitialization(Loop *, LPPassManager &LPM);
```

The `doInitialization` method is designed to do simple initialization type of stuff that does not depend on the functions being processed. The `doInitialization` method call is not scheduled to overlap with any other pass executions (thus it should be very fast). `LPPassManager` interface should be used to access `Function` or `Module` level analysis information.

The `runOnLoop` method

```
virtual bool runOnLoop(Loop *, LPPassManager &LPM) = 0;
```

The `runOnLoop` method must be implemented by your subclass to do the transformation or analysis work of your pass. As usual, a `true` value should be returned if the function is modified. `LPPassManager` interface should be used to update loop nest.

The `doFinalization()` method

```
virtual bool doFinalization();
```

The `doFinalization` method is an infrequently used method that is called when the pass framework has finished calling *runOnLoop* for every loop in the program being compiled.

The `RegionPass` class

`RegionPass` is similar to *LoopPass*, but executes on each single entry single exit region in the function. `RegionPass` processes regions in nested order such that the outer most region is processed last.

`RegionPass` subclasses are allowed to update the region tree by using the `RGPassManager` interface. You may overload three virtual methods of `RegionPass` to implement your own region pass. All these methods should return `true` if they modified the program, or `false` if they did not.

The `doInitialization(Region *, RGPassManager &)` method

```
virtual bool doInitialization(Region *, RGPassManager &RGM);
```

The `doInitialization` method is designed to do simple initialization type of stuff that does not depend on the functions being processed. The `doInitialization` method call is not scheduled to overlap with any other pass executions (thus it should be very fast). `RPPassManager` interface should be used to access `Function` or `Module` level analysis information.

The `runOnRegion` method

```
virtual bool runOnRegion(Region *, RGPassManager &RGM) = 0;
```

The `runOnRegion` method must be implemented by your subclass to do the transformation or analysis work of your pass. As usual, a true value should be returned if the region is modified. `RGPassManager` interface should be used to update region tree.

The `doFinalization()` method

```
virtual bool doFinalization();
```

The `doFinalization` method is an infrequently used method that is called when the pass framework has finished calling *runOnRegion* for every region in the program being compiled.

The `BasicBlockPass` class

`BasicBlockPass`es are just like *FunctionPass's* , except that they must limit their scope of inspection and modification to a single basic block at a time. As such, they are **not** allowed to do any of the following:

1. Modify or inspect any basic blocks outside of the current one.
2. Maintain state across invocations of *runOnBasicBlock*.
3. Modify the control flow graph (by altering terminator instructions)
4. Any of the things forbidden for *FunctionPasses*.

`BasicBlockPass`es are useful for traditional local and "peephole" optimizations. They may override the same *doInitialization(Module &)* and *doFinalization(Module &)* methods that *FunctionPass's* have, but also have the following virtual methods that may also be implemented:

The `doInitialization(Function &)` method

```
virtual bool doInitialization(Function &F);
```

The `doInitialization` method is allowed to do most of the things that `BasicBlockPasses` are not allowed to do, but that `FunctionPasses` can. The `doInitialization` method is designed to do simple initialization that does not depend on the `BasicBlocks` being processed. The `doInitialization` method call is not scheduled to overlap with any other pass executions (thus it should be very fast).

The `runOnBasicBlock` method

```
virtual bool runOnBasicBlock(BasicBlock &BB) = 0;
```

Override this function to do the work of the `BasicBlockPass`. This function is not allowed to inspect or modify basic blocks other than the parameter, and are not allowed to modify the CFG. A `true` value must be returned if the basic block is modified.

The `doFinalization(Function &)` method

```
virtual bool doFinalization(Function &F);
```

The `doFinalization` method is an infrequently used method that is called when the pass framework has finished calling *runOnBasicBlock* for every `BasicBlock` in the program being compiled. This can be used to perform per-function finalization.

The `MachineFunctionPass` class

A `MachineFunctionPass` is a part of the LLVM code generator that executes on the machine-dependent representation of each LLVM function in the program.

Code generator passes are registered and initialized specially by `TargetMachine::addPassesToEmitFile` and similar routines, so they cannot generally be run from the **opt** or **bugpoint** commands.

A `MachineFunctionPass` is also a `FunctionPass`, so all the restrictions that apply to a `FunctionPass` also apply to it. `MachineFunctionPasses` also have additional restrictions. In particular, `MachineFunctionPasses` are not allowed to do any of the following:

1. Modify or create any LLVM IR `Instructions`, `BasicBlocks`, `Arguments`, `Functions`, `GlobalVariables`, `GlobalAliases`, or `Modules`.
2. Modify a `MachineFunction` other than the one currently being processed.
3. Maintain state across invocations of *runOnMachineFunction* (including global data).

The `runOnMachineFunction(MachineFunction &MF)` method

```
virtual bool runOnMachineFunction(MachineFunction &MF) = 0;
```

`runOnMachineFunction` can be considered the main entry point of a `MachineFunctionPass`; that is, you should override this method to do the work of your `MachineFunctionPass`.

The `runOnMachineFunction` method is called on every `MachineFunction` in a `Module`, so that the `MachineFunctionPass` may perform optimizations on the machine-dependent representation of the function. If you want to get at the LLVM `Function` for the `MachineFunction` you're working on, use

`MachineFunction`'s `getFunction()` accessor method — but remember, you may not modify the LLVM `Function` or its contents from a `MachineFunctionPass`.

Pass registration

In the *Hello World* example pass we illustrated how pass registration works, and discussed some of the reasons that it is used and what it does. Here we discuss how and why passes are registered.

As we saw above, passes are registered with the `RegisterPass` template. The template parameter is the name of the pass that is to be used on the command line to specify that the pass should be added to a program (for example, with **opt** or **bugpoint**). The first argument is the name of the pass, which is to be used for the *`-help`* output of programs, as well as for debug output generated by the *`--debug-pass`* option.

If you want your pass to be easily dumpable, you should implement the virtual print method:

The `print` method

```
virtual void print(llvm::raw_ostream &O, const Module *M) const;
```

The `print` method must be implemented by "analyses" in order to print a human readable version of the analysis results. This is useful for debugging an analysis itself, as well as for other people to figure out how an analysis works. Use the opt `-analyze` argument to invoke this method.

The `llvm::raw_ostream` parameter specifies the stream to write the results on, and the `Module` parameter gives a pointer to the top level module of the program that has been analyzed. Note however that this pointer may be `NULL` in certain circumstances (such as calling the `Pass::dump()` from a debugger), so it should only be used to enhance debug output, it should not be depended on.

Specifying interactions between passes

One of the main responsibilities of the `PassManager` is to make sure that passes interact with each other correctly. Because `PassManager` tries to *optimize the execution of passes* it must know how the passes interact with each other and what dependencies exist between the various passes. To track this, each pass can declare the set of passes that are required to be executed before the current pass, and the passes which are invalidated by the current pass.

Typically this functionality is used to require that analysis results are computed before your pass is run. Running arbitrary transformation passes can invalidate the computed analysis results, which is what the invalidation set specifies. If a pass does not implement the *getAnalysisUsage* method, it defaults to not having any prerequisite passes, and invalidating **all** other passes.

The `getAnalysisUsage` method

```
virtual void getAnalysisUsage(AnalysisUsage &Info) const;
```

By implementing the `getAnalysisUsage` method, the required and invalidated sets may be specified for your transformation. The implementation should fill in the AnalysisUsage object with information about which passes are required and not invalidated. To do this, a pass may call any of the following methods on the `AnalysisUsage` object:

The AnalysisUsage::addRequired<> and AnalysisUsage::addRequiredTransitive<> methods

If your pass requires a previous pass to be executed (an analysis for example), it can use one of these methods to arrange for it to be run before your pass. LLVM has many different types of analyses and passes that can be required, spanning the range from DominatorSet to BreakCriticalEdges. Requiring BreakCriticalEdges, for example, guarantees that there will be no critical edges in the CFG when your pass has been run.

Some analyses chain to other analyses to do their job. For example, an *AliasAnalysis* <*AliasAnalysis*> implementation is required to *chain* to other alias analysis passes. In cases where analyses chain, the addRequiredTransitive method should be used instead of the addRequired method. This informs the PassManager that the transitively required pass should be alive as long as the requiring pass is.

The AnalysisUsage::addPreserved<> method

One of the jobs of the PassManager is to optimize how and when analyses are run. In particular, it attempts to avoid recomputing data unless it needs to. For this reason, passes are allowed to declare that they preserve (i.e., they don't invalidate) an existing analysis if it's available. For example, a simple constant folding pass would not modify the CFG, so it can't possibly affect the results of dominator analysis. By default, all passes are assumed to invalidate all others.

The AnalysisUsage class provides several methods which are useful in certain circumstances that are related to addPreserved. In particular, the setPreservesAll method can be called to indicate that the pass does not modify the LLVM program at all (which is true for analyses), and the setPreservesCFG method can be used by transformations that change instructions in the program but do not modify the CFG or terminator instructions (note that this property is implicitly set for *BasicBlockPass*es).

addPreserved is particularly useful for transformations like BreakCriticalEdges. This pass knows how to update a small set of loop and dominator related analyses if they exist, so it can preserve them, despite the fact that it hacks on the CFG.

Example implementations of getAnalysisUsage

```
// This example modifies the program, but does not modify the CFG
void LICM::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesCFG();
  AU.addRequired<LoopInfo>();
}
```

The getAnalysis<> and getAnalysisIfAvailable<> methods

The Pass::getAnalysis<> method is automatically inherited by your class, providing you with access to the passes that you declared that you required with the *getAnalysisUsage* method. It takes a single template argument that specifies which pass class you want, and returns a reference to that pass. For example:

```
bool LICM::runOnFunction(Function &F) {
  LoopInfo &LI = getAnalysis<LoopInfo>();
  //...
}
```

This method call returns a reference to the pass desired. You may get a runtime assertion failure if you attempt to get an analysis that you did not declare as required in your *getAnalysisUsage* implementation. This method can be called by your run* method implementation, or by any other local method invoked by your run* method.

A module level pass can use function level analysis info using this interface. For example:

```
bool ModuleLevelPass::runOnModule(Module &M) {
  //...
  DominatorTree &DT = getAnalysis<DominatorTree>(Func);
  //...
}
```

In above example, `runOnFunction` for `DominatorTree` is called by pass manager before returning a reference to the desired pass.

If your pass is capable of updating analyses if they exist (e.g., `BreakCriticalEdges`, as described above), you can use the `getAnalysisIfAvailable` method, which returns a pointer to the analysis if it is active. For example:

```
if (DominatorSet *DS = getAnalysisIfAvailable<DominatorSet>()) {
  // A DominatorSet is active.  This code will update it.
}
```

Implementing Analysis Groups

Now that we understand the basics of how passes are defined, how they are used, and how they are required from other passes, it's time to get a little bit fancier. All of the pass relationships that we have seen so far are very simple: one pass depends on one other specific pass to be run before it can run. For many applications, this is great, for others, more flexibility is required.

In particular, some analyses are defined such that there is a single simple interface to the analysis results, but multiple ways of calculating them. Consider alias analysis for example. The most trivial alias analysis returns "may alias" for any alias query. The most sophisticated analysis a flow-sensitive, context-sensitive interprocedural analysis that can take a significant amount of time to execute (and obviously, there is a lot of room between these two extremes for other implementations). To cleanly support situations like this, the LLVM Pass Infrastructure supports the notion of Analysis Groups.

Analysis Group Concepts

An Analysis Group is a single simple interface that may be implemented by multiple different passes. Analysis Groups can be given human readable names just like passes, but unlike passes, they need not derive from the `Pass` class. An analysis group may have one or more implementations, one of which is the "default" implementation.

Analysis groups are used by client passes just like other passes are: the `AnalysisUsage::addRequired()` and `Pass::getAnalysis()` methods. In order to resolve this requirement, the *PassManager* scans the available passes to see if any implementations of the analysis group are available. If none is available, the default implementation is created for the pass to use. All standard rules for *interaction between passes* still apply.

Although *Pass Registration* is optional for normal passes, all analysis group implementations must be registered, and must use the *INITIALIZE_AG_PASS* template to join the implementation pool. Also, a default implementation of the interface **must** be registered with *RegisterAnalysisGroup*.

As a concrete example of an Analysis Group in action, consider the AliasAnalysis analysis group. The default implementation of the alias analysis interface (the basicaa pass) just does a few simple checks that don't require significant analysis to compute (such as: two different globals can never alias each other, etc). Passes that use the AliasAnalysis interface (for example the gcse pass), do not care which implementation of alias analysis is actually provided, they just use the designated interface.

From the user's perspective, commands work just like normal. Issuing the command `opt -gcse ...` will cause the `basicaa` class to be instantiated and added to the pass sequence. Issuing the command `opt -somefancyaa -gcse ...` will cause the `gcse` pass to use the `somefancyaa` alias analysis (which doesn't actually exist, it's just a hypothetical example) instead.

Using `RegisterAnalysisGroup`

The `RegisterAnalysisGroup` template is used to register the analysis group itself, while the `INITIALIZE_AG_PASS` is used to add pass implementations to the analysis group. First, an analysis group should be registered, with a human readable name provided for it. Unlike registration of passes, there is no command line argument to be specified for the Analysis Group Interface itself, because it is "abstract":

```
static RegisterAnalysisGroup<AliasAnalysis> A("Alias Analysis");
```

Once the analysis is registered, passes can declare that they are valid implementations of the interface by using the following code:

```
namespace {
  // Declare that we implement the AliasAnalysis interface
  INITIALIZE_AG_PASS(FancyAA, AliasAnalysis , "somefancyaa",
      "A more complex alias analysis implementation",
      false,  // Is CFG Only?
      true,   // Is Analysis?
      false); // Is default Analysis Group implementation?
}
```

This just shows a class `FancyAA` that uses the `INITIALIZE_AG_PASS` macro both to register and to "join" the AliasAnalysis analysis group. Every implementation of an analysis group should join using this macro.

```
namespace {
  // Declare that we implement the AliasAnalysis interface
  INITIALIZE_AG_PASS(BasicAA, AliasAnalysis, "basicaa",
      "Basic Alias Analysis (default AA impl)",
      false, // Is CFG Only?
      true,  // Is Analysis?
      true); // Is default Analysis Group implementation?
}
```

Here we show how the default implementation is specified (using the final argument to the `INITIALIZE_AG_PASS` template). There must be exactly one default implementation available at all times for an Analysis Group to be used. Only default implementation can derive from `ImmutablePass`. Here we declare that the BasicAliasAnalysis pass is the default implementation for the interface.

4.20.4 Pass Statistics

The Statistic class is designed to be an easy way to expose various success metrics from passes. These statistics are printed at the end of a run, when the `-stats` command line option is enabled on the command line. See the *Statistics section* in the Programmer's Manual for details.

What PassManager does

The PassManager class takes a list of passes, ensures their *prerequisites* are set up correctly, and then schedules passes to run efficiently. All of the LLVM tools that run passes use the PassManager for execution of these passes.

The PassManager does two main things to try to reduce the execution time of a series of passes:

1. **Share analysis results.** The `PassManager` attempts to avoid recomputing analysis results as much as possible. This means keeping track of which analyses are available already, which analyses get invalidated, and which analyses are needed to be run for a pass. An important part of work is that the `PassManager` tracks the exact lifetime of all analysis results, allowing it to *free memory* allocated to holding analysis results as soon as they are no longer needed.

2. **Pipeline the execution of passes on the program.** The `PassManager` attempts to get better cache and memory usage behavior out of a series of passes by pipelining the passes together. This means that, given a series of consecutive *FunctionPass*, it will execute all of the *FunctionPass* on the first function, then all of the *FunctionPasses* on the second function, etc... until the entire program has been run through the passes.

   This improves the cache behavior of the compiler, because it is only touching the LLVM program representation for a single function at a time, instead of traversing the entire program. It reduces the memory consumption of compiler, because, for example, only one DominatorSet needs to be calculated at a time. This also makes it possible to implement some *interesting enhancements* in the future.

The effectiveness of the `PassManager` is influenced directly by how much information it has about the behaviors of the passes it is scheduling. For example, the "preserved" set is intentionally conservative in the face of an unimplemented *getAnalysisUsage* method. Not implementing when it should be implemented will have the effect of not allowing any analysis results to live across the execution of your pass.

The `PassManager` class exposes a `--debug-pass` command line options that is useful for debugging pass execution, seeing how things work, and diagnosing when you should be preserving more analyses than you currently are. (To get information about all of the variants of the `--debug-pass` option, just type "`opt -help-hidden`").

By using the –debug-pass=Structure option, for example, we can see how our *Hello World* pass interacts with other passes. Lets try it out with the gcse and licm passes:

```
$ opt -load ../../../Debug+Asserts/lib/Hello.so -gcse -licm --debug-pass=Structure < hello.bc > /dev/
Module Pass Manager
  Function Pass Manager
    Dominator Set Construction
    Immediate Dominators Construction
    Global Common Subexpression Elimination
--  Immediate Dominators Construction
--  Global Common Subexpression Elimination
    Natural Loop Construction
    Loop Invariant Code Motion
--  Natural Loop Construction
--  Loop Invariant Code Motion
    Module Verifier
--  Dominator Set Construction
--  Module Verifier
  Bitcode Writer
--Bitcode Writer
```

This output shows us when passes are constructed and when the analysis results are known to be dead (prefixed with "--"). Here we see that GCSE uses dominator and immediate dominator information to do its job. The LICM pass uses natural loop information, which uses dominator sets, but not immediate dominators. Because immediate dominators are no longer useful after the GCSE pass, it is immediately destroyed. The dominator sets are then reused to compute natural loop information, which is then used by the LICM pass.

After the LICM pass, the module verifier runs (which is automatically added by the **opt** tool), which uses the dominator set to check that the resultant LLVM code is well formed. After it finishes, the dominator set information is destroyed, after being computed once, and shared by three passes.

Lets see how this changes when we run the *Hello World* pass in between the two passes:

```
$ opt -load ../../../Debug+Asserts/lib/Hello.so -gcse -hello -licm --debug-pass=Structure < hello.bc
Module Pass Manager
  Function Pass Manager
    Dominator Set Construction
    Immediate Dominators Construction
    Global Common Subexpression Elimination
--  Dominator Set Construction
--  Immediate Dominators Construction
```

```
--  Global Common Subexpression Elimination
    Hello World Pass
--  Hello World Pass
    Dominator Set Construction
    Natural Loop Construction
    Loop Invariant Code Motion
--  Natural Loop Construction
--  Loop Invariant Code Motion
    Module Verifier
--  Dominator Set Construction
--  Module Verifier
  Bitcode Writer
--Bitcode Writer
Hello: __main
Hello: puts
Hello: main
```

Here we see that the *Hello World* pass has killed the Dominator Set pass, even though it doesn't modify the code at all! To fix this, we need to add the following *getAnalysisUsage* method to our pass:

```
// We don't modify the program, so we preserve all analyses
virtual void getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesAll();
}
```

Now when we run our pass, we get this output:

```
$ opt -load ../../../Debug+Asserts/lib/Hello.so -gcse -hello -licm --debug-pass=Structure < hello.bc
Pass Arguments:  -gcse -hello -licm
Module Pass Manager
  Function Pass Manager
    Dominator Set Construction
    Immediate Dominators Construction
    Global Common Subexpression Elimination
--  Immediate Dominators Construction
--  Global Common Subexpression Elimination
    Hello World Pass
--  Hello World Pass
    Natural Loop Construction
    Loop Invariant Code Motion
--  Loop Invariant Code Motion
--  Natural Loop Construction
    Module Verifier
--  Dominator Set Construction
--  Module Verifier
  Bitcode Writer
--Bitcode Writer
Hello: __main
Hello: puts
Hello: main
```

Which shows that we don't accidentally invalidate dominator information anymore, and therefore do not have to compute it twice.

The `releaseMemory` method

```
virtual void releaseMemory();
```

The `PassManager` automatically determines when to compute analysis results, and how long to keep them around for. Because the lifetime of the pass object itself is effectively the entire duration of the compilation process, we need some way to free analysis results when they are no longer useful. The `releaseMemory` virtual method is the way to do this.

If you are writing an analysis or any other pass that retains a significant amount of state (for use by another pass which "requires" your pass and uses the *getAnalysis* method) you should implement `releaseMemory` to, well, release the memory allocated to maintain this internal state. This method is called after the `run*` method for the class, before the next call of `run*` in your pass.

4.20.5 Registering dynamically loaded passes

Size matters when constructing production quality tools using LLVM, both for the purposes of distribution, and for regulating the resident code size when running on the target system. Therefore, it becomes desirable to selectively use some passes, while omitting others and maintain the flexibility to change configurations later on. You want to be able to do all this, and, provide feedback to the user. This is where pass registration comes into play.

The fundamental mechanisms for pass registration are the `MachinePassRegistry` class and subclasses of `MachinePassRegistryNode`.

An instance of `MachinePassRegistry` is used to maintain a list of `MachinePassRegistryNode` objects. This instance maintains the list and communicates additions and deletions to the command line interface.

An instance of `MachinePassRegistryNode` subclass is used to maintain information provided about a particular pass. This information includes the command line name, the command help string and the address of the function used to create an instance of the pass. A global static constructor of one of these instances *registers* with a corresponding `MachinePassRegistry`, the static destructor *unregisters*. Thus a pass that is statically linked in the tool will be registered at start up. A dynamically loaded pass will register on load and unregister at unload.

Using existing registries

There are predefined registries to track instruction scheduling (`RegisterScheduler`) and register allocation (`RegisterRegAlloc`) machine passes. Here we will describe how to *register* a register allocator machine pass.

Implement your register allocator machine pass. In your register allocator `.cpp` file add the following include:

```
#include "llvm/CodeGen/RegAllocRegistry.h"
```

Also in your register allocator `.cpp` file, define a creator function in the form:

```
FunctionPass *createMyRegisterAllocator() {
  return new MyRegisterAllocator();
}
```

Note that the signature of this function should match the type of `RegisterRegAlloc::FunctionPassCtor`. In the same file add the "installing" declaration, in the form:

```
static RegisterRegAlloc myRegAlloc("myregalloc",
                                   "my register allocator help string",
                                   createMyRegisterAllocator);
```

Note the two spaces prior to the help string produces a tidy result on the `-help` query.

```
$ llc -help
  ...
  -regalloc                - Register allocator to use (default=linearscan)
    =linearscan            -   linear scan register allocator
    =local                 -   local register allocator
    =simple                -   simple register allocator
    =myregalloc            -   my register allocator help string
  ...
```

And that's it. The user is now free to use `-regalloc=myregalloc` as an option. Registering instruction schedulers is similar except use the `RegisterScheduler` class. Note that the `RegisterScheduler::FunctionPassCtor` is significantly different from `RegisterRegAlloc::FunctionPassCtor`.

To force the load/linking of your register allocator into the **llc/lli** tools, add your creator function's global declaration to `Passes.h` and add a "pseudo" call line to `llvm/Codegen/LinkAllCodegenComponents.h`.

Creating new registries

The easiest way to get started is to clone one of the existing registries; we recommend `llvm/CodeGen/RegAllocRegistry.h`. The key things to modify are the class name and the `FunctionPassCtor` type.

Then you need to declare the registry. Example: if your pass registry is `RegisterMyPasses` then define:

```
MachinePassRegistry RegisterMyPasses::Registry;
```

And finally, declare the command line option for your passes. Example:

```
cl::opt<RegisterMyPasses::FunctionPassCtor, false,
        RegisterPassParser<RegisterMyPasses> >
MyPassOpt("mypass",
          cl::init(&createDefaultMyPass),
          cl::desc("my pass option help"));
```

Here the command option is "`mypass`", with `createDefaultMyPass` as the default creator.

Using GDB with dynamically loaded passes

Unfortunately, using GDB with dynamically loaded passes is not as easy as it should be. First of all, you can't set a breakpoint in a shared object that has not been loaded yet, and second of all there are problems with inlined functions in shared objects. Here are some suggestions to debugging your pass with GDB.

For sake of discussion, I'm going to assume that you are debugging a transformation invoked by **opt**, although nothing described here depends on that.

Setting a breakpoint in your pass

First thing you do is start gdb on the opt process:

```
$ gdb opt
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.6"...
(gdb)
```

Note that **opt** has a lot of debugging information in it, so it takes time to load. Be patient. Since we cannot set a breakpoint in our pass yet (the shared object isn't loaded until runtime), we must execute the process, and have it stop before it invokes our pass, but after it has loaded the shared object. The most foolproof way of doing this is to set a breakpoint in `PassManager::run` and then run the process with the arguments you want:

```
$ (gdb) break llvm::PassManager::run
Breakpoint 1 at 0x2413bc: file Pass.cpp, line 70.
(gdb) run test.bc -load $(LLVMTOP)/llvm/Debug+Asserts/lib/[libname].so -[passoption]
Starting program: opt test.bc -load $(LLVMTOP)/llvm/Debug+Asserts/lib/[libname].so -[passoption]
Breakpoint 1, PassManager::run (this=0xffbef174, M=@0x70b298) at Pass.cpp:70
70      bool PassManager::run(Module &M) { return PM->run(M); }
(gdb)
```

Once the **opt** stops in the `PassManager::run` method you are now free to set breakpoints in your pass so that you can trace through execution or do other standard debugging stuff.

Miscellaneous Problems

Once you have the basics down, there are a couple of problems that GDB has, some with solutions, some without.

- Inline functions have bogus stack information. In general, GDB does a pretty good job getting stack traces and stepping through inline functions. When a pass is dynamically loaded however, it somehow completely loses this capability. The only solution I know of is to de-inline a function (move it from the body of a class to a `.cpp` file).
- Restarting the program breaks breakpoints. After following the information above, you have succeeded in getting some breakpoints planted in your pass. Nex thing you know, you restart the program (i.e., you type "`run`" again), and you start getting errors about breakpoints being unsettable. The only way I have found to "fix" this problem is to delete the breakpoints that are already set in your pass, run the program, and re-set the breakpoints once execution stops in `PassManager::run`.

Hopefully these tips will help with common case debugging situations. If you'd like to contribute some tips of your own, just contact Chris.

Future extensions planned

Although the LLVM Pass Infrastructure is very capable as it stands, and does some nifty stuff, there are things we'd like to add in the future. Here is where we are going:

Multithreaded LLVM

Multiple CPU machines are becoming more common and compilation can never be fast enough: obviously we should allow for a multithreaded compiler. Because of the semantics defined for passes above (specifically they cannot maintain state across invocations of their `run*` methods), a nice clean way to implement a multithreaded compiler would be for the `PassManager` class to create multiple instances of each pass object, and allow the separate instances to be hacking on different parts of the program at the same time.

This implementation would prevent each of the passes from having to implement multithreaded constructs, requiring only the LLVM core to have locking in a few places (for global resources). Although this is a simple extension, we simply haven't had time (or multiprocessor machines, thus a reason) to implement this. Despite that, we have kept the LLVM passes SMP ready, and you should too.

4.21 How To Use Attributes

4.21.1 Introduction

Attributes in LLVM have changed in some fundamental ways. It was necessary to do this to support expanding the attributes to encompass more than a handful of attributes — e.g. command line options. The old way of handling attributes consisted of representing them as a bit mask of values. This bit mask was stored in a "list" structure that was reference counted. The advantage of this was that attributes could be manipulated with 'or's and 'and's. The disadvantage of this was that there was limited room for expansion, and virtually no support for attribute-value pairs other than alignment.

In the new scheme, an `Attribute` object represents a single attribute that's uniqued. You use the `Attribute::get` methods to create a new `Attribute` object. An attribute can be a single "enum" value (the enum being the `Attribute::AttrKind` enum), a string representing a target-dependent attribute, or an attribute-value pair. Some examples:

- Target-independent: `noinline`, `zext`
- Target-dependent: `"no-sse"`, `"thumb2"`
- Attribute-value pair: `"cpu" = "cortex-a8"`, `align = 4`

Note: for an attribute value pair, we expect a target-dependent attribute to have a string for the value.

4.21.2 `Attribute`

An `Attribute` object is designed to be passed around by value.

Because attributes are no longer represented as a bit mask, you will need to convert any code which does treat them as a bit mask to use the new query methods on the Attribute class.

4.21.3 `AttributeSet`

The `AttributeSet` class replaces the old `AttributeList` class. The `AttributeSet` stores a collection of Attribute objects for each kind of object that may have an attribute associated with it: the function as a whole, the return type, or the function's parameters. A function's attributes are at index `AttributeSet::FunctionIndex`; the return type's attributes are at index `AttributeSet::ReturnIndex`; and the function's parameters' attributes are at indices 1, ..., n (where 'n' is the number of parameters). Most methods on the `AttributeSet` class take an index parameter.

An `AttributeSet` is also a uniqued and immutable object. You create an `AttributeSet` through the `AttributeSet::get` methods. You can add and remove attributes, which result in the creation of a new `AttributeSet`.

An `AttributeSet` object is designed to be passed around by value.

Note: It is advised that you do *not* use the `AttributeSet` "introspection" methods (e.g. `Raw`, `getRawPointer`, etc.). These methods break encapsulation, and may be removed in a future release (i.e. LLVM 4.0).

4.21.4 AttrBuilder

Lastly, we have a "builder" class to help create the AttributeSet object without having to create several different intermediate uniqued AttributeSet objects. The AttrBuilder class allows you to add and remove attributes at will. The attributes won't be uniqued until you call the appropriate AttributeSet::get method.

An AttrBuilder object is *not* designed to be passed around by value. It should be passed by reference.

Note: It is advised that you do *not* use the AttrBuilder::addRawValue() method or the AttrBuilder(uint64_t Val) constructor. These are for backwards compatibility and may be removed in a future release (i.e. LLVM 4.0).

And that's basically it! A lot of functionality is hidden behind these classes, but the interfaces are pretty straight forward.

4.22 User Guide for NVPTX Back-end

4.22.1 Introduction

To support GPU programming, the NVPTX back-end supports a subset of LLVM IR along with a defined set of conventions used to represent GPU programming concepts. This document provides an overview of the general usage

of the back- end, including a description of the conventions used and the set of accepted LLVM IR.

Note: This document assumes a basic familiarity with CUDA and the PTX assembly language. Information about the CUDA Driver API and the PTX assembly language can be found in the CUDA documentation.

4.22.2 Conventions

Marking Functions as Kernels

In PTX, there are two types of functions: *device functions*, which are only callable by device code, and *kernel functions*, which are callable by host code. By default, the back-end will emit device functions. Metadata is used to declare a function as a kernel function. This metadata is attached to the `nvvm.annotations` named metadata object, and has the following format:

```
!0 = metadata !{<function-ref>, metadata !"kernel", i32 1}
```

The first parameter is a reference to the kernel function. The following example shows a kernel function calling a device function in LLVM IR. The function `@my_kernel` is callable from host code, but `@my_fmad` is not.

```
define float @my_fmad(float %x, float %y, float %z) {
  %mul = fmul float %x, %y
  %add = fadd float %mul, %z
  ret float %add
}

define void @my_kernel(float* %ptr) {
  %val = load float* %ptr
  %ret = call float @my_fmad(float %val, float %val, float %val)
  store float %ret, float* %ptr
  ret void
}

!nvvm.annotations = !{!1}
!1 = metadata !{void (float*)* @my_kernel, metadata !"kernel", i32 1}
```

When compiled, the PTX kernel functions are callable by host-side code.

Address Spaces

The NVPTX back-end uses the following address space mapping:

Address Space	Memory Space
0	Generic
1	Global
2	Internal Use
3	Shared
4	Constant
5	Local

Every global variable and pointer type is assigned to one of these address spaces, with 0 being the default address space. Intrinsics are provided which can be used to convert pointers between the generic and non-generic address spaces.

As an example, the following IR will define an array `@g` that resides in global device memory.

```
@g = internal addrspace(1) global [4 x i32] [ i32 0, i32 1, i32 2, i32 3 ]
```

LLVM IR functions can read and write to this array, and host-side code can copy data to it by name with the CUDA Driver API.

Note that since address space 0 is the generic space, it is illegal to have global variables in address space 0. Address space 0 is the default address space in LLVM, so the `addrspace(N)` annotation is *required* for global variables.

Triples

The NVPTX target uses the module triple to select between 32/64-bit code generation and the driver-compiler interface to use. The triple architecture can be one of `nvptx` (32-bit PTX) or `nvptx64` (64-bit PTX). The operating system should be one of `cuda` or `nvcl`, which determines the interface used by the generated code to communicate with the driver. Most users will want to use `cuda` as the operating system, which makes the generated PTX compatible with the CUDA Driver API.

Example: 32-bit PTX for CUDA Driver API: `nvptx-nvidia-cuda`

Example: 64-bit PTX for CUDA Driver API: `nvptx64-nvidia-cuda`

4.22.3 NVPTX Intrinsics

Address Space Conversion

'`llvm.nvvm.ptr.*.to.gen`' Intrinsics

Syntax: These are overloaded intrinsics. You can use these on any pointer types.

```
declare i8* @llvm.nvvm.ptr.global.to.gen.p0i8.p1i8(i8 addrspace(1)*)
declare i8* @llvm.nvvm.ptr.shared.to.gen.p0i8.p3i8(i8 addrspace(3)*)
declare i8* @llvm.nvvm.ptr.constant.to.gen.p0i8.p4i8(i8 addrspace(4)*)
declare i8* @llvm.nvvm.ptr.local.to.gen.p0i8.p5i8(i8 addrspace(5)*)
```

Overview: The '`llvm.nvvm.ptr.*.to.gen`' intrinsics convert a pointer in a non-generic address space to a generic address space pointer.

Semantics: These intrinsics modify the pointer value to be a valid generic address space pointer.

'`llvm.nvvm.ptr.gen.to.*`' Intrinsics

Syntax: These are overloaded intrinsics. You can use these on any pointer types.

```
declare i8* @llvm.nvvm.ptr.gen.to.global.p1i8.p0i8(i8 addrspace(1)*)
declare i8* @llvm.nvvm.ptr.gen.to.shared.p3i8.p0i8(i8 addrspace(3)*)
declare i8* @llvm.nvvm.ptr.gen.to.constant.p4i8.p0i8(i8 addrspace(4)*)
declare i8* @llvm.nvvm.ptr.gen.to.local.p5i8.p0i8(i8 addrspace(5)*)
```

Overview: The '`llvm.nvvm.ptr.gen.to.*`' intrinsics convert a pointer in the generic address space to a pointer in the target address space. Note that these intrinsics are only useful if the address space of the target address space of the pointer is known. It is not legal to use address space conversion intrinsics to convert a pointer from one non-generic address space to another non-generic address space.

Semantics: These intrinsics modify the pointer value to be a valid pointer in the target non-generic address space.

Reading PTX Special Registers

'`llvm.nvvm.read.ptx.sreg.*`'

Syntax:

```
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.tid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.tid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.warpsize()
```

Overview: The '`@llvm.nvvm.read.ptx.sreg.*`' intrinsics provide access to the PTX special registers, in particular the kernel launch bounds. These registers map in the following way to CUDA builtins:

CUDA Builtin	PTX Special Register Intrinsic
`threadId`	`@llvm.nvvm.read.ptx.sreg.tid.*`
`blockIdx`	`@llvm.nvvm.read.ptx.sreg.ctaid.*`
`blockDim`	`@llvm.nvvm.read.ptx.sreg.ntid.*`
`gridDim`	`@llvm.nvvm.read.ptx.sreg.nctaid.*`

Barriers

'`llvm.nvvm.barrier0`'

Syntax:

```
declare void @llvm.nvvm.barrier0()
```

Overview: The '`@llvm.nvvm.barrier0()`' intrinsic emits a PTX `bar.sync 0` instruction, equivalent to the `__syncthreads()` call in CUDA.

Other Intrinsics

For the full set of NVPTX intrinsics, please see the `include/llvm/IR/IntrinsicsNVVM.td` file in the LLVM source tree.

4.22.4 Linking with Libdevice

The CUDA Toolkit comes with an LLVM bitcode library called `libdevice` that implements many common mathematical functions. This library can be used as a high-performance math library for any compilers using the LLVM

NVPTX target. The library can be found under `nvvm/libdevice/` in the CUDA Toolkit and there is a separate version for each compute architecture.

For a list of all math functions implemented in libdevice, see libdevice Users Guide.

To accommodate various math-related compiler flags that can affect code generation of libdevice code, the library code depends on a special LLVM IR pass (`NVVMReflect`) to handle conditional compilation within LLVM IR. This pass looks for calls to the `@__nvvm_reflect` function and replaces them with constants based on the defined reflection parameters. Such conditional code often follows a pattern:

```
float my_function(float a) {
  if (__nvvm_reflect("FASTMATH"))
    return my_function_fast(a);
  else
    return my_function_precise(a);
}
```

The default value for all unspecified reflection parameters is zero.

The `NVVMReflect` pass should be executed early in the optimization pipeline, immediately after the link stage. The `internalize` pass is also recommended to remove unused math functions from the resulting PTX. For an input IR module `module.bc`, the following compilation flow is recommended:

1. Save list of external functions in `module.bc`
2. Link `module.bc` with `libdevice.compute_XX.YY.bc`
3. Internalize all functions not in list from (1)
4. Eliminate all unused internal functions
5. Run `NVVMReflect` pass
6. Run standard optimization pipeline

Note: `linkonce` and `linkonce_odr` linkage types are not suitable for the libdevice functions. It is possible to link two IR modules that have been linked against libdevice using different reflection variables.

Since the `NVVMReflect` pass replaces conditionals with constants, it will often leave behind dead code of the form:

```
entry:
  ..
  br i1 true, label %foo, label %bar
foo:
  ..
bar:
  ; Dead code
  ..
```

Therefore, it is recommended that `NVVMReflect` is executed early in the optimization pipeline before dead-code elimination.

Reflection Parameters

The libdevice library currently uses the following reflection parameters to control code generation:

Flag	Description
`__CUDA_FTZ=[0,1]`	Use optimized code paths that flush subnormals to zero

Invoking NVVMReflect

To ensure that all dead code caused by the reflection pass is eliminated, it is recommended that the reflection pass is executed early in the LLVM IR optimization pipeline. The pass takes an optional mapping of reflection parameter name to an integer value. This mapping can be specified as either a command-line option to `opt` or as an LLVM `StringMap<int>` object when programmatically creating a pass pipeline.

With `opt`:

```
# opt -nvvm-reflect -nvvm-reflect-list=<var>=<value>,<var>=<value> module.bc -o module.reflect.bc
```

With programmatic pass pipeline:

```
extern ModulePass *llvm::createNVVMReflectPass(const StringMap<int>& Mapping);

StringMap<int> ReflectParams;
ReflectParams["__CUDA_FTZ"] = 1;
Passes.add(createNVVMReflectPass(ReflectParams));
```

4.22.5 Executing PTX

The most common way to execute PTX assembly on a GPU device is to use the CUDA Driver API. This API is a low-level interface to the GPU driver and allows for JIT compilation of PTX code to native GPU machine code.

Initializing the Driver API:

```
CUdevice device;
CUcontext context;

// Initialize the driver API
cuInit(0);
// Get a handle to the first compute device
cuDeviceGet(&device, 0);
// Create a compute device context
cuCtxCreate(&context, 0, device);
```

JIT compiling a PTX string to a device binary:

```
CUmodule module;
CUfunction funcion;

// JIT compile a null-terminated PTX string
cuModuleLoadData(&module, (void*)PTXString);

// Get a handle to the "myfunction" kernel function
cuModuleGetFunction(&function, module, "myfunction");
```

For full examples of executing PTX assembly, please see the CUDA Samples distribution.

4.22.6 Common Issues

ptxas complains of undefined function: __nvvm_reflect

When linking with libdevice, the `NVVMReflect` pass must be used. See *Linking with Libdevice* for more information.

4.22.7 Tutorial: A Simple Compute Kernel

To start, let us take a look at a simple compute kernel written directly in LLVM IR. The kernel implements vector addition, where each thread computes one element of the output vector C from the input vectors A and B. To make this easier, we also assume that only a single CTA (thread block) will be launched, and that it will be one dimensional.

The Kernel

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v1
target triple = "nvptx64-nvidia-cuda"

; Intrinsic to read X component of thread ID
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x() readnone nounwind

define void @kernel(float addrspace(1)* %A,
                    float addrspace(1)* %B,
                    float addrspace(1)* %C) {
entry:
  ; What is my ID?
  %id = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x() readnone nounwind

  ; Compute pointers into A, B, and C
  %ptrA = getelementptr float addrspace(1)* %A, i32 %id
  %ptrB = getelementptr float addrspace(1)* %B, i32 %id
  %ptrC = getelementptr float addrspace(1)* %C, i32 %id

  ; Read A, B
  %valA = load float addrspace(1)* %ptrA, align 4
  %valB = load float addrspace(1)* %ptrB, align 4

  ; Compute C = A + B
  %valC = fadd float %valA, %valB

  ; Store back to C
  store float %valC, float addrspace(1)* %ptrC, align 4

  ret void
}

!nvvm.annotations = !{!0}
!0 = metadata !{void (float addrspace(1)*,
                      float addrspace(1)*,
                      float addrspace(1)*)* @kernel, metadata !"kernel", i32 1}
```

We can use the LLVM `llc` tool to directly run the NVPTX code generator:

```
# llc -mcpu=sm_20 kernel.ll -o kernel.ptx
```

Note: If you want to generate 32-bit code, change `p:64:64:64` to `p:32:32:32` in the module data layout string and use `nvptx-nvidia-cuda` as the target triple.

The output we get from `llc` (as of LLVM 3.4):

```
//
// Generated by LLVM NVPTX Back-End
//
```

```
.version 3.1
.target sm_20
.address_size 64

  // .globl kernel
                                        // @kernel
.visible .entry kernel(
  .param .u64 kernel_param_0,
  .param .u64 kernel_param_1,
  .param .u64 kernel_param_2
)
{
  .reg .f32   %f<4>;
  .reg .s32   %r<2>;
  .reg .s64   %rl<8>;

// BB#0:                                // %entry
  ld.param.u64    %rl1, [kernel_param_0];
  mov.u32         %r1, %tid.x;
  mul.wide.s32    %rl2, %r1, 4;
  add.s64         %rl3, %rl1, %rl2;
  ld.param.u64    %rl4, [kernel_param_1];
  add.s64         %rl5, %rl4, %rl2;
  ld.param.u64    %rl6, [kernel_param_2];
  add.s64         %rl7, %rl6, %rl2;
  ld.global.f32   %f1, [%rl3];
  ld.global.f32   %f2, [%rl5];
  add.f32         %f3, %f1, %f2;
  st.global.f32   [%rl7], %f3;
  ret;
}
```

Dissecting the Kernel

Now let us dissect the LLVM IR that makes up this kernel.

Data Layout

The data layout string determines the size in bits of common data types, their ABI alignment, and their storage size. For NVPTX, you should use one of the following:

32-bit PTX:

```
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v1
```

64-bit PTX:

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v1
```

Target Intrinsics

In this example, we use the `@llvm.nvvm.read.ptx.sreg.tid.x` intrinsic to read the X component of the current thread's ID, which corresponds to a read of register `%tid.x` in PTX. The NVPTX back-end supports a large

set of intrinsics. A short list is shown below; please see `include/llvm/IR/IntrinsicsNVVM.td` for the full list.

Intrinsic	CUDA Equivalent
`i32 @llvm.nvvm.read.ptx.sreg.tid.{x,y,z}`	threadIdx.{x,y,z}
`i32 @llvm.nvvm.read.ptx.sreg.ctaid.{x,y,z}`	blockIdx.{x,y,z}
`i32 @llvm.nvvm.read.ptx.sreg.ntid.{x,y,z}`	blockDim.{x,y,z}
`i32 @llvm.nvvm.read.ptx.sreg.nctaid.{x,y,z}`	gridDim.{x,y,z}
`void @llvm.cuda.syncthreads()`	__syncthreads()

Address Spaces

You may have noticed that all of the pointer types in the LLVM IR example had an explicit address space specifier. What is address space 1? NVIDIA GPU devices (generally) have four types of memory:

- Global: Large, off-chip memory
- Shared: Small, on-chip memory shared among all threads in a CTA
- Local: Per-thread, private memory
- Constant: Read-only memory shared across all threads

These different types of memory are represented in LLVM IR as address spaces. There is also a fifth address space used by the NVPTX code generator that corresponds to the "generic" address space. This address space can represent addresses in any other address space (with a few exceptions). This allows users to write IR functions that can load/store memory using the same instructions. Intrinsics are provided to convert pointers between the generic and non-generic address spaces.

See *Address Spaces* and *NVPTX Intrinsics* for more information.

Kernel Metadata

In PTX, a function can be either a *kernel* function (callable from the host program), or a *device* function (callable only from GPU code). You can think of *kernel* functions as entry-points in the GPU program. To mark an LLVM IR function as a *kernel* function, we make use of special LLVM metadata. The NVPTX back-end will look for a named metadata node called `nvvm.annotations`. This named metadata must contain a list of metadata that describe the IR. For our purposes, we need to declare a metadata node that assigns the "kernel" attribute to the LLVM IR function that should be emitted as a PTX *kernel* function. These metadata nodes take the form:

```
metadata !{<function ref>, metadata !"kernel", i32 1}
```

For the previous example, we have:

```
!nvvm.annotations = !{!0}
!0 = metadata !{void (float addrspace(1)*,
                      float addrspace(1)*,
                      float addrspace(1)*)* @kernel, metadata !"kernel", i32 1}
```

Here, we have a single metadata declaration in `nvvm.annotations`. This metadata annotates our `@kernel` function with the `kernel` attribute.

Running the Kernel

Generating PTX from LLVM IR is all well and good, but how do we execute it on a real GPU device? The CUDA Driver API provides a convenient mechanism for loading and JIT compiling PTX to a native GPU device, and launch-

ing a kernel. The API is similar to OpenCL. A simple example showing how to load and execute our vector addition code is shown below. Note that for brevity this code does not perform much error checking!

Note: You can also use the `ptxas` tool provided by the CUDA Toolkit to offline compile PTX to machine code (SASS) for a specific GPU architecture. Such binaries can be loaded by the CUDA Driver API in the same way as PTX. This can be useful for reducing startup time by precompiling the PTX kernels.

```
#include <iostream>
#include <fstream>
#include <cassert>
#include "cuda.h"


void checkCudaErrors(CUresult err) {
  assert(err == CUDA_SUCCESS);
}

/// main - Program entry point
int main(int argc, char **argv) {
  CUdevice    device;
  CUmodule    cudaModule;
  CUcontext   context;
  CUfunction  function;
  CUlinkState linker;
  int         devCount;

  // CUDA initialization
  checkCudaErrors(cuInit(0));
  checkCudaErrors(cuDeviceGetCount(&devCount));
  checkCudaErrors(cuDeviceGet(&device, 0));

  char name[128];
  checkCudaErrors(cuDeviceGetName(name, 128, device));
  std::cout << "Using CUDA Device [0]: " << name << "\n";

  int devMajor, devMinor;
  checkCudaErrors(cuDeviceComputeCapability(&devMajor, &devMinor, device));
  std::cout << "Device Compute Capability: "
            << devMajor << "." << devMinor << "\n";
  if (devMajor < 2) {
    std::cerr << "ERROR: Device 0 is not SM 2.0 or greater\n";
    return 1;
  }

  std::ifstream t("kernel.ptx");
  if (!t.is_open()) {
    std::cerr << "kernel.ptx not found\n";
    return 1;
  }
  std::string str((std::istreambuf_iterator<char>(t)),
                   std::istreambuf_iterator<char>());

  // Create driver context
  checkCudaErrors(cuCtxCreate(&context, 0, device));

  // Create module for object
  checkCudaErrors(cuModuleLoadDataEx(&cudaModule, str.c_str(), 0, 0, 0));
```

```
// Get kernel function
checkCudaErrors(cuModuleGetFunction(&function, cudaModule, "kernel"));

// Device data
CUdeviceptr devBufferA;
CUdeviceptr devBufferB;
CUdeviceptr devBufferC;

checkCudaErrors(cuMemAlloc(&devBufferA, sizeof(float)*16));
checkCudaErrors(cuMemAlloc(&devBufferB, sizeof(float)*16));
checkCudaErrors(cuMemAlloc(&devBufferC, sizeof(float)*16));

float* hostA = new float[16];
float* hostB = new float[16];
float* hostC = new float[16];

// Populate input
for (unsigned i = 0; i != 16; ++i) {
  hostA[i] = (float)i;
  hostB[i] = (float)(2*i);
  hostC[i] = 0.0f;
}

checkCudaErrors(cuMemcpyHtoD(devBufferA, &hostA[0], sizeof(float)*16));
checkCudaErrors(cuMemcpyHtoD(devBufferB, &hostB[0], sizeof(float)*16));


unsigned blockSizeX = 16;
unsigned blockSizeY = 1;
unsigned blockSizeZ = 1;
unsigned gridSizeX  = 1;
unsigned gridSizeY  = 1;
unsigned gridSizeZ  = 1;

// Kernel parameters
void *KernelParams[] = { &devBufferA, &devBufferB, &devBufferC };

std::cout << "Launching kernel\n";

// Kernel launch
checkCudaErrors(cuLaunchKernel(function, gridSizeX, gridSizeY, gridSizeZ,
                               blockSizeX, blockSizeY, blockSizeZ,
                               0, NULL, KernelParams, NULL));

// Retrieve device data
checkCudaErrors(cuMemcpyDtoH(&hostC[0], devBufferC, sizeof(float)*16));


std::cout << "Results:\n";
for (unsigned i = 0; i != 16; ++i) {
  std::cout << hostA[i] << " + " << hostB[i] << " = " << hostC[i] << "\n";
}


// Clean up after ourselves
delete [] hostA;
delete [] hostB;
delete [] hostC;
```

```
  // Clean-up
  checkCudaErrors(cuMemFree(devBufferA));
  checkCudaErrors(cuMemFree(devBufferB));
  checkCudaErrors(cuMemFree(devBufferC));
  checkCudaErrors(cuModuleUnload(cudaModule));
  checkCudaErrors(cuCtxDestroy(context));

  return 0;
}
```

You will need to link with the CUDA driver and specify the path to cuda.h.

```
# clang++ sample.cpp -o sample -O2 -g -I/usr/local/cuda-5.5/include -lcuda
```

We don't need to specify a path to `libcuda.so` since this is installed in a system location by the driver, not the CUDA toolkit.

If everything goes as planned, you should see the following output when running the compiled program:

```
Using CUDA Device [0]: GeForce GTX 680
Device Compute Capability: 3.0
Launching kernel
Results:
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
3 + 6 = 9
4 + 8 = 12
5 + 10 = 15
6 + 12 = 18
7 + 14 = 21
8 + 16 = 24
9 + 18 = 27
10 + 20 = 30
11 + 22 = 33
12 + 24 = 36
13 + 26 = 39
14 + 28 = 42
15 + 30 = 45
```

Note: You will likely see a different device identifier based on your hardware

---

4.22.8 Tutorial: Linking with Libdevice

In this tutorial, we show a simple example of linking LLVM IR with the libdevice library. We will use the same kernel as the previous tutorial, except that we will compute `C = pow(A, B)` instead of `C = A + B`. Libdevice provides an `__nv_powf` function that we will use.

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v1
target triple = "nvptx64-nvidia-cuda"

; Intrinsic to read X component of thread ID
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x() readnone nounwind
; libdevice function
declare float @__nv_powf(float, float)
```

```
define void @kernel(float addrspace(1)* %A,
                    float addrspace(1)* %B,
                    float addrspace(1)* %C) {
entry:
  ; What is my ID?
  %id = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x() readnone nounwind

  ; Compute pointers into A, B, and C
  %ptrA = getelementptr float addrspace(1)* %A, i32 %id
  %ptrB = getelementptr float addrspace(1)* %B, i32 %id
  %ptrC = getelementptr float addrspace(1)* %C, i32 %id

  ; Read A, B
  %valA = load float addrspace(1)* %ptrA, align 4
  %valB = load float addrspace(1)* %ptrB, align 4

  ; Compute C = pow(A, B)
  %valC = call float @__nv_powf(float %valA, float %valB)

  ; Store back to C
  store float %valC, float addrspace(1)* %ptrC, align 4

  ret void
}

!nvvm.annotations = !{!0}
!0 = metadata !{void (float addrspace(1)*,
                      float addrspace(1)*,
                      float addrspace(1)*)* @kernel, metadata !"kernel", i32 1}
```

To compile this kernel, we perform the following steps:

1. Link with libdevice
2. Internalize all but the public kernel function
3. Run `NVVMReflect` and set `__CUDA_FTZ` to 0
4. Optimize the linked module
5. Codegen the module

These steps can be performed by the LLVM `llvm-link`, `opt`, and `llc` tools. In a complete compiler, these steps can also be performed entirely programmatically by setting up an appropriate pass configuration (see *Linking with Libdevice*).

```
# llvm-link t2.bc libdevice.compute_20.10.bc -o t2.linked.bc
# opt -internalize -internalize-public-api-list=kernel -nvvm-reflect-list=__CUDA_FTZ=0 -nvvm-reflect
# llc -mcpu=sm_20 t2.opt.bc -o t2.ptx
```

Note: The `-nvvm-reflect-list=_CUDA_FTZ=0` is not strictly required, as any undefined variables will default to zero. It is shown here for evaluation purposes.

This gives us the following PTX (excerpt):

```
//
// Generated by LLVM NVPTX Back-End
//
```

```
.version 3.1
.target sm_20
.address_size 64

  // .globl kernel
                                        // @kernel
.visible .entry kernel(
  .param .u64 kernel_param_0,
  .param .u64 kernel_param_1,
  .param .u64 kernel_param_2
)
{
  .reg .pred  %p<30>;
  .reg .f32   %f<111>;
  .reg .s32   %r<21>;
  .reg .s64   %rl<8>;

// BB#0:                                // %entry
  ld.param.u64  %rl2, [kernel_param_0];
  mov.u32   %r3, %tid.x;
  ld.param.u64  %rl3, [kernel_param_1];
  mul.wide.s32  %rl4, %r3, 4;
  add.s64   %rl5, %rl2, %rl4;
  ld.param.u64  %rl6, [kernel_param_2];
  add.s64   %rl7, %rl3, %rl4;
  add.s64   %rl1, %rl6, %rl4;
  ld.global.f32   %f1, [%rl5];
  ld.global.f32   %f2, [%rl7];
  setp.eq.f32 %p1, %f1, 0f3F800000;
  setp.eq.f32 %p2, %f2, 0f00000000;
  or.pred   %p3, %p1, %p2;
  @%p3 bra  BB0_1;
  bra.uni   BB0_2;
BB0_1:
  mov.f32   %f110, 0f3F800000;
  st.global.f32   [%rl1], %f110;
  ret;
BB0_2:                                  // %__nv_isnanf.exit.i
  abs.f32   %f4, %f1;
  setp.gtu.f32  %p4, %f4, 0f7F800000;
  @%p4 bra  BB0_4;
// BB#3:                                // %__nv_isnanf.exit5.i
  abs.f32   %f5, %f2;
  setp.le.f32 %p5, %f5, 0f7F800000;
  @%p5 bra  BB0_5;
BB0_4:                                  // %.critedge1.i
  add.f32   %f110, %f1, %f2;
  st.global.f32   [%rl1], %f110;
  ret;
BB0_5:                                  // %__nv_isinff.exit.i

  ...

BB0_26:                                 // %__nv_truncf.exit.i.i.i.i.i
  mul.f32   %f90, %f107, 0f3FB8AA3B;
  cvt.rzi.f32.f32 %f91, %f90;
  mov.f32   %f92, 0fBF317200;
  fma.rn.f32  %f93, %f91, %f92, %f107;
```

```
  mov.f32   %f94, 0fB5BFBE8E;
  fma.rn.f32  %f95, %f91, %f94, %f93;
  mul.f32   %f89, %f95, 0f3FB8AA3B;
  // inline asm
  ex2.approx.ftz.f32 %f88,%f89;
  // inline asm
  add.f32   %f96, %f91, 0f00000000;
  ex2.approx.f32  %f97, %f96;
  mul.f32   %f98, %f88, %f97;
  setp.lt.f32 %p15, %f107, 0fC2D20000;
  selp.f32  %f99, 0f00000000, %f98, %p15;
  setp.gt.f32 %p16, %f107, 0f42D20000;
  selp.f32  %f110, 0f7F800000, %f99, %p16;
  setp.eq.f32 %p17, %f110, 0f7F800000;
  @%p17 bra   BB0_28;
// BB#27:
  fma.rn.f32  %f110, %f110, %f108, %f110;
BB0_28:                                 // %__internal_accurate_powf.exit.i
  setp.lt.f32 %p18, %f1, 0f00000000;
  setp.eq.f32 %p19, %f3, 0f3F800000;
  and.pred    %p20, %p18, %p19;
  @!%p20 bra  BB0_30;
  bra.uni   BB0_29;
BB0_29:
  mov.b32    %r9, %f110;
  xor.b32   %r10, %r9, -2147483648;
  mov.b32    %f110, %r10;
BB0_30:                                 // %__nv_powf.exit
  st.global.f32   [%rl1], %f110;
  ret;
}
```

4.23 Stack maps and patch points in LLVM

4.23.1 Definitions

In this document we refer to the “runtime” collectively as all components that serve as the LLVM client, including the LLVM IR generator, object code consumer, and code patcher.

A stack map records the location of `live values` at a particular instruction address. These `live values` do not refer to all the LLVM values live across the stack map. Instead, they are only the values that the runtime requires to

be live at this point. For example, they may be the values the runtime will need to resume program execution at that point independent of the compiled function containing the stack map.

LLVM emits stack map data into the object code within a designated *Stack Map Section*. This stack map data contains a record for each stack map. The record stores the stack map's instruction address and contains a entry for each mapped value. Each entry encodes a value's location as a register, stack offset, or constant.

A patch point is an instruction address at which space is reserved for patching a new instruction sequence at run time. Patch points look much like calls to LLVM. They take arguments that follow a calling convention and may return a value. They also imply stack map generation, which allows the runtime to locate the patchpoint and find the location of `live values` at that point.

4.23.2 Motivation

This functionality is currently experimental but is potentially useful in a variety of settings, the most obvious being a runtime (JIT) compiler. Example applications of the patchpoint intrinsics are implementing an inline call cache for polymorphic method dispatch or optimizing the retrieval of properties in dynamically typed languages such as JavaScript.

The intrinsics documented here are currently used by the JavaScript compiler within the open source WebKit project, see the FTL JIT, but they are designed to be used whenever stack maps or code patching are needed. Because the intrinsics have experimental status, compatibility across LLVM releases is not guaranteed.

The stack map functionality described in this document is separate from the functionality described in *Computing stack maps*. *GCFunctionMetadata* provides the location of pointers into a collected heap captured by the *GCRoot* intrinsic, which can also be considered a "stack map". Unlike the stack maps defined above, the *GCFunctionMetadata* stack map interface does not provide a way to associate live register values of arbitrary type with an instruction address, nor does it specify a format for the resulting stack map. The stack maps described here could potentially provide richer information to a garbage collecting runtime, but that usage will not be discussed in this document.

4.23.3 Intrinsics

The following two kinds of intrinsics can be used to implement stack maps and patch points: `llvm.experimental.stackmap` and `llvm.experimental.patchpoint`. Both kinds of intrinsics generate a stack map record, and they both allow some form of code patching. They can be used independently (i.e. `llvm.experimental.patchpoint` implicitly generates a stack map without the need for an additional call to `llvm.experimental.stackmap`). The choice of which to use depends on whether it is necessary to reserve space for code patching and whether any of the intrinsic arguments should be lowered according to calling conventions. `llvm.experimental.stackmap` does not reserve any space, nor does it expect any call arguments. If the runtime patches code at the stack map's address, it will destructively overwrite the program text. This is unlike `llvm.experimental.patchpoint`, which reserves space for in-place patching without overwriting surrounding code. The `llvm.experimental.patchpoint` intrinsic also lowers a specified number of arguments according to its calling convention. This allows patched code to make in-place function calls without marshaling.

Each instance of one of these intrinsics generates a stack map record in the *Stack Map Section*. The record includes an ID, allowing the runtime to uniquely identify the stack map, and the offset within the code from the beginning of the enclosing function.

'`llvm.experimental.stackmap`' Intrinsic

Syntax:

```
declare void
  @llvm.experimental.stackmap(i64 <id>, i32 <numShadowBytes>, ...)
```

Overview:

The '`llvm.experimental.stackmap`' intrinsic records the location of specified values in the stack map without generating any code.

Operands:

The first operand is an ID to be encoded within the stack map. The second operand is the number of shadow bytes following the intrinsic. The variable number of operands that follow are the `live values` for which locations will be recorded in the stack map.

To use this intrinsic as a bare-bones stack map, with no code patching support, the number of shadow bytes can be set to zero.

Semantics:

The stack map intrinsic generates no code in place, unless nops are needed to cover its shadow (see below). However, its offset from function entry is stored in the stack map. This is the relative instruction address immediately following the instructions that precede the stack map.

The stack map ID allows a runtime to locate the desired stack map record. LLVM passes this ID through directly to the stack map record without checking uniqueness.

LLVM guarantees a shadow of instructions following the stack map's instruction offset during which neither the end of the basic block nor another call to `llvm.experimental.stackmap` or `llvm.experimental.patchpoint` may occur. This allows the runtime to patch the code at this point in response to an event triggered from outside the code. The code for instructions following the stack map may be emitted in the stack map's shadow, and these instructions may be overwritten by destructive patching. Without shadow bytes, this destructive patching could overwrite program text or data outside the current function. We disallow overlapping stack map shadows so that the runtime does not need to consider this corner case.

For example, a stack map with 8 byte shadow:

```
call void @runtime()
call void (i64, i32, ...)* @llvm.experimental.stackmap(i64 77, i32 8,
                                                       i64* %ptr)
%val = load i64* %ptr
%add = add i64 %val, 3
ret i64 %add
```

May require one byte of nop-padding:

```
0x00 callq _runtime
0x05 nop                <--- stack map address
0x06 movq (%rdi), %rax
0x07 addq $3, %rax
```

```
0x0a popq %rdx
0x0b ret                <---- end of 8-byte shadow
```

Now, if the runtime needs to invalidate the compiled code, it may patch 8 bytes of code at the stack map's address at follows:

```
0x00 callq _runtime
0x05 movl  $0xffff, %rax <--- patched code at stack map address
0x0a callq *%rax         <---- end of 8-byte shadow
```

This way, after the normal call to the runtime returns, the code will execute a patched call to a special entry point that can rebuild a stack frame from the values located by the stack map.

'llvm.experimental.patchpoint.*' Intrinsic

Syntax:

```
declare void
  @llvm.experimental.patchpoint.void(i64 <id>, i32 <numBytes>,
                                     i8* <target>, i32 <numArgs>, ...)
declare i64
  @llvm.experimental.patchpoint.i64(i64 <id>, i32 <numBytes>,
                                    i8* <target>, i32 <numArgs>, ...)
```

Overview:

The 'llvm.experimental.patchpoint.*' intrinsics creates a function call to the specified <target> and records the location of specified values in the stack map.

Operands:

The first operand is an ID, the second operand is the number of bytes reserved for the patchable region, the third operand is the target address of a function (optionally null), and the fourth operand specifies how many of the following variable operands are considered function call arguments. The remaining variable number of operands are the live values for which locations will be recorded in the stack map.

Semantics:

The patch point intrinsic generates a stack map. It also emits a function call to the address specified by <target> if the address is not a constant null. The function call and its arguments are lowered according to the calling convention specified at the intrinsic's callsite. Variants of the intrinsic with non-void return type also return a value according to calling convention.

Requesting zero patch point arguments is valid. In this case, all variable operands are handled just like llvm.experimental.stackmap.*. The difference is that space will still be reserved for patching, a call will be emitted, and a return value is allowed.

The location of the arguments are not normally recorded in the stack map because they are already fixed by the calling convention. The remaining live values will have their location recorded, which could be a register, stack location, or constant. A special calling convention has been introduced for use with stack maps, anyregcc, which forces the arguments to be loaded into registers but allows those register to be dynamically allocated. These argument registers will have their register locations recorded in the stack map in addition to the remaining live values.

The patch point also emits nops to cover at least <numBytes> of instruction encoding space. Hence, the client must ensure that <numBytes> is enough to encode a call to the target address on the supported targets. If the call target is constant null, then there is no minimum requirement. A zero-byte null target patchpoint is valid.

The runtime may patch the code emitted for the patch point, including the call sequence and nops. However, the runtime may not assume anything about the code LLVM emits within the reserved space. Partial patching is not allowed. The runtime must patch all reserved bytes, padding with nops if necessary.

This example shows a patch point reserving 15 bytes, with one argument in $rdi, and a return value in $rax per native calling convention:

```
%target = inttoptr i64 -281474976710654 to i8*
%val = call i64 (i64, i32, ...)*
         @llvm.experimental.patchpoint.i64(i64 78, i32 15,
                                           i8* %target, i32 1, i64* %ptr)
%add = add i64 %val, 3
ret i64 %add
```

May generate:

```
0x00 movabsq $0xffff000000000002, %r11 <--- patch point address
0x0a callq   *%r11
0x0d nop
0x0e nop                                <--- end of reserved 15-bytes
0x0f addq    $0x3, %rax
0x10 movl    %rax, 8(%rsp)
```

Note that no stack map locations will be recorded. If the patched code sequence does not need arguments fixed to specific calling convention registers, then the anyregcc convention may be used:

```
%val = call anyregcc @llvm.experimental.patchpoint(i64 78, i32 15,
                                                   i8* %target, i32 1,
                                                   i64* %ptr)
```

The stack map now indicates the location of the %ptr argument and return value:

```
Stack Map: ID=78, Loc0=%r9 Loc1=%r8
```

The patch code sequence may now use the argument that happened to be allocated in %r8 and return a value allocated in %r9:

```
0x00 movslq 4(%r8) %r9              <--- patched code at patch point address
0x03 nop
...
0x0e nop                            <--- end of reserved 15-bytes
0x0f addq    $0x3, %r9
0x10 movl    %r9, 8(%rsp)
```

4.23.4 Stack Map Format

The existence of a stack map or patch point intrinsic within an LLVM Module forces code emission to create a *Stack Map Section*. The format of this section follows:

```
Header {
  uint8  : Stack Map Version (current version is 1)
  uint8  : Reserved (expected to be 0)
  uint16 : Reserved (expected to be 0)
}
uint32 : NumFunctions
```

```
uint32 : NumConstants
uint32 : NumRecords
StkSizeRecord[NumFunctions] {
  uint64 : Function Address
  uint64 : Stack Size
}
Constants[NumConstants] {
  uint64 : LargeConstant
}
StkMapRecord[NumRecords] {
  uint64 : PatchPoint ID
  uint32 : Instruction Offset
  uint16 : Reserved (record flags)
  uint16 : NumLocations
  Location[NumLocations] {
    uint8  : Register | Direct | Indirect | Constant | ConstantIndex
    uint8  : Reserved (location flags)
    uint16 : Dwarf RegNum
    int32  : Offset or SmallConstant
  }
  uint16 : Padding
  uint16 : NumLiveOuts
  LiveOuts[NumLiveOuts]
    uint16 : Dwarf RegNum
    uint8  : Reserved
    uint8  : Size in Bytes
  }
  uint32 : Padding (only if required to align to 8 byte)
}
```

The first byte of each location encodes a type that indicates how to interpret the `RegNum` and `Offset` fields as follows:

Encoding	Type	Value	Description
0x1	Register	Reg	Value in a register
0x2	Direct	Reg + Offset	Frame index value
0x3	Indirect	[Reg + Offset]	Spilled value
0x4	Constant	Offset	Small constant
0x5	ConstIndex	Constants[Offset]	Large constant

In the common case, a value is available in a register, and the `Offset` field will be zero. Values spilled to the stack are encoded as `Indirect` locations. The runtime must load those values from a stack address, typically in the form `[BP + Offset]`. If an `alloca` value is passed directly to a stack map intrinsic, then LLVM may fold the frame index into the stack map as an optimization to avoid allocating a register or stack slot. These frame indices will be encoded as `Direct` locations in the form `BP + Offset`. LLVM may also optimize constants by emitting them directly in the stack map, either in the `Offset` of a `Constant` location or in the constant pool, referred to by `ConstantIndex` locations.

At each callsite, a "liveout" register list is also recorded. These are the registers that are live across the stackmap and therefore must be saved by the runtime. This is an important optimization when the patchpoint intrinsic is used with a calling convention that by default preserves most registers as callee-save.

Each entry in the liveout register list contains a DWARF register number and size in bytes. The stackmap format deliberately omits specific subregister information. Instead the runtime must interpret this information conservatively. For example, if the stackmap reports one byte at `%rax`, then the value may be in either `%al` or `%ah`. It doesn't matter in practice, because the runtime will simply save `%rax`. However, if the stackmap reports 16 bytes at `%ymm0`, then the runtime can safely optimize by saving only `%xmm0`.

The stack map format is a contract between an LLVM SVN revision and the runtime. It is currently experimental

and may change in the short term, but minimizing the need to update the runtime is important. Consequently, the stack map design is motivated by simplicity and extensibility. Compactness of the representation is secondary because the runtime is expected to parse the data immediately after compiling a module and encode the information in its own format. Since the runtime controls the allocation of sections, it can reuse the same stack map space for multiple modules.

Stackmap support is currently only implemented for 64-bit platforms. However, a 32-bit implementation should be able to use the same format with an insignificant amount of wasted space.

Stack Map Section

A JIT compiler can easily access this section by providing its own memory manager via the LLVM C API `LLVMCreateSimpleMCJITMemoryManager()`. When creating the memory manager, the JIT provides a callback: `LLVMMemoryManagerAllocateDataSectionCallback()`. When LLVM creates this section, it invokes the callback and passes the section name. The JIT can record the in-memory address of the section at this time and later parse it to recover the stack map data.

On Darwin, the stack map section name is "__llvm_stackmaps". The segment name is "__LLVM_STACKMAPS".

4.23.5 Stack Map Usage

The stack map support described in this document can be used to precisely determine the location of values at a specific position in the code. LLVM does not maintain any mapping between those values and any higher-level entity. The runtime must be able to interpret the stack map record given only the ID, offset, and the order of the locations, which LLVM preserves.

Note that this is quite different from the goal of debug information, which is a best-effort attempt to track the location of named variables at every instruction.

An important motivation for this design is to allow a runtime to commandeer a stack frame when execution reaches an instruction address associated with a stack map. The runtime must be able to rebuild a stack frame and resume program execution using the information provided by the stack map. For example, execution may resume in an interpreter or a recompiled version of the same function.

This usage restricts LLVM optimization. Clearly, LLVM must not move stores across a stack map. However, loads must also be handled conservatively. If the load may trigger an exception, hoisting it above a stack map could be invalid. For example, the runtime may determine that a load is safe to execute without a type check given the current state of the type system. If the type system changes while some activation of the load's function exists on the stack, the load becomes unsafe. The runtime can prevent subsequent execution of that load by immediately patching any stack map location that lies between the current call site and the load (typically, the runtime would simply patch all stack map locations to invalidate the function). If the compiler had hoisted the load above the stack map, then the program could crash before the runtime could take back control.

To enforce these semantics, stackmap and patchpoint intrinsics are considered to potentially read and write all memory. This may limit optimization more than some clients desire. This limitation may be avoided by marking the call site as "readonly". In the future we may also allow meta-data to be added to the intrinsic call to express aliasing, thereby allowing optimizations to hoist certain loads above stack maps.

Direct Stack Map Entries

As shown in *Stack Map Section*, a Direct stack map location records the address of frame index. This address is itself the value that the runtime requested. This differs from Indirect locations, which refer to a stack locations from which the requested values must be loaded. Direct locations can communicate the address if an alloca, while Indirect locations handle register spills.

For example:

```
entry:
  %a = alloca i64...
  llvm.experimental.stackmap(i64 <ID>, i32 <shadowBytes>, i64* %a)
```

The runtime can determine this alloca's relative location on the stack immediately after compilation, or at any time thereafter. This differs from Register and Indirect locations, because the runtime can only read the values in those locations when execution reaches the instruction address of the stack map.

This functionality requires LLVM to treat entry-block allocas specially when they are directly consumed by an intrinsics. (This is the same requirement imposed by the llvm.gcroot intrinsic.) LLVM transformations must not substitute the alloca with any intervening value. This can be verified by the runtime simply by checking that the stack map's location is a Direct location type.

4.24 Design and Usage of the InAlloca Attribute

4.24.1 Introduction

The *inalloca* attribute is designed to allow taking the address of an aggregate argument that is being passed by value through memory. Primarily, this feature is required for compatibility with the Microsoft C++ ABI. Under that ABI, class instances that are passed by value are constructed directly into argument stack memory. Prior to the addition of inalloca, calls in LLVM were indivisible instructions. There was no way to perform intermediate work, such as object construction, between the first stack adjustment and the final control transfer. With inalloca, all arguments passed in memory are modelled as a single alloca, which can be stored to prior to the call. Unfortunately, this complicated feature comes with a large set of restrictions designed to bound the lifetime of the argument memory around the call.

For now, it is recommended that frontends and optimizers avoid producing this construct, primarily because it forces the use of a base pointer. This feature may grow in the future to allow general mid-level optimization, but for now, it should be regarded as less efficient than passing by value with a copy.

4.24.2 Intended Usage

The example below is the intended LLVM IR lowering for some C++ code that passes two default-constructed `Foo` objects to `g` in the 32-bit Microsoft C++ ABI.

```
// Foo is non-trivial.
struct Foo { int a, b; Foo(); ~Foo(); Foo(const Foo &); };
void g(Foo a, Foo b);
void f() {
  g(Foo(), Foo());
}

%struct.Foo = type { i32, i32 }
declare void @Foo_ctor(%struct.Foo* %this)
declare void @Foo_dtor(%struct.Foo* %this)
declare void @g(<{ %struct.Foo, %struct.Foo }>* inalloca %memargs)

define void @f() {
entry:
  %base = call i8* @llvm.stacksave()
  %memargs = alloca <{ %struct.Foo, %struct.Foo }>
  %b = getelementptr <{ %struct.Foo, %struct.Foo }>* %memargs, i32 1
  call void @Foo_ctor(%struct.Foo* %b)
```

```
  ; If a's ctor throws, we must destruct b.
  %a = getelementptr <{ %struct.Foo, %struct.Foo }>* %memargs, i32 0
  invoke void @Foo_ctor(%struct.Foo* %a)
      to label %invoke.cont unwind %invoke.unwind

invoke.cont:
  call void @g(<{ %struct.Foo, %struct.Foo }>* inalloca %memargs)
  call void @llvm.stackrestore(i8* %base)
  ...

invoke.unwind:
  call void @Foo_dtor(%struct.Foo* %b)
  call void @llvm.stackrestore(i8* %base)
  ...
}
```

To avoid stack leaks, the frontend saves the current stack pointer with a call to *llvm.stacksave*. Then, it allocates the argument stack space with alloca and calls the default constructor. The default constructor could throw an exception, so the frontend has to create a landing pad. The frontend has to destroy the already constructed argument `b` before restoring the stack pointer. If the constructor does not unwind, `g` is called. In the Microsoft C++ ABI, `g` will destroy its arguments, and then the stack is restored in `f`.

4.24.3 Design Considerations

Lifetime

The biggest design consideration for this feature is object lifetime. We cannot model the arguments as static allocas in the entry block, because all calls need to use the memory at the top of the stack to pass arguments. We cannot vend pointers to that memory at function entry because after code generation they will alias.

The rule against allocas between argument allocations and the call site avoids this problem, but it creates a cleanup problem. Cleanup and lifetime is handled explicitly with stack save and restore calls. In the future, we may want to introduce a new construct such as `freea` or `afree` to make it clear that this stack adjusting cleanup is less powerful than a full stack save and restore.

Nested Calls and Copy Elision

We also want to be able to support copy elision into these argument slots. This means we have to support multiple live argument allocations.

Consider the evaluation of:

```
// Foo is non-trivial.
struct Foo { int a; Foo(); Foo(const &Foo); ~Foo(); };
Foo bar(Foo b);
int main() {
  bar(bar(Foo()));
}
```

In this case, we want to be able to elide copies into `bar`'s argument slots. That means we need to have more than one set of argument frames active at the same time. First, we need to allocate the frame for the outer call so we can pass it in as the hidden struct return pointer to the middle call. Then we do the same for the middle call, allocating a frame and passing its address to `Foo`'s default constructor. By wrapping the evaluation of the inner `bar` with stack save and restore, we can have multiple overlapping active call frames.

Callee-cleanup Calling Conventions

Another wrinkle is the existence of callee-cleanup conventions. On Windows, all methods and many other functions adjust the stack to clear the memory used to pass their arguments. In some sense, this means that the allocas are automatically cleared by the call. However, LLVM instead models this as a write of undef to all of the inalloca values passed to the call instead of a stack adjustment. Frontends should still restore the stack pointer to avoid a stack leak.

Exceptions

There is also the possibility of an exception. If argument evaluation or copy construction throws an exception, the landing pad must do cleanup, which includes adjusting the stack pointer to avoid a stack leak. This means the cleanup of the stack memory cannot be tied to the call itself. There needs to be a separate IR-level instruction that can perform independent cleanup of arguments.

Efficiency

Eventually, it should be possible to generate efficient code for this construct. In particular, using inalloca should not require a base pointer. If the backend can prove that all points in the CFG only have one possible stack level, then it can address the stack directly from the stack pointer. While this is not yet implemented, the plan is that the inalloca attribute should not change much, but the frontend IR generation recommendations may change.

4.25 Using ARM NEON instructions in big endian mode

4.25.1 Introduction

Generating code for big endian ARM processors is for the most part straightforward. NEON loads and stores however have some interesting properties that make code generation decisions less obvious in big endian mode.

The aim of this document is to explain the problem with NEON loads and stores, and the solution that has been implemented in LLVM.

In this document the term "vector" refers to what the ARM ABI calls a "short vector", which is a sequence of items that can fit in a NEON register. This sequence can be 64 or 128 bits in length, and can constitute 8, 16, 32 or 64 bit items. This document refers to A64 instructions throughout, but is almost applicable to the A32/ARMv7 instruction sets also. The ABI format for passing vectors in A32 is sligtly different to A64. Apart from that, the same concepts apply.

Example: C-level intrinsics -> assembly

It may be helpful first to illustrate how C-level ARM NEON intrinsics are lowered to instructions.

This trivial C function takes a vector of four ints and sets the zero'th lane to the value "42":

```
#include <arm_neon.h>
int32x4_t f(int32x4_t p) {
    return vsetq_lane_s32(42, p, 0);
}
```

arm_neon.h intrinsics generate "generic" IR where possible (that is, normal IR instructions not `llvm.arm.neon.*` intrinsic calls). The above generates:

```
define <4 x i32> @f(<4 x i32> %p) {
  %vset_lane = insertelement <4 x i32> %p, i32 42, i32 0
  ret <4 x i32> %vset_lane
}
```

Which then becomes the following trivial assembly:

```
f:                                      // @f
        movz        w8, #0x2a
        ins         v0.s[0], w8
        ret
```

4.25.2 Problem

The main problem is how vectors are represented in memory and in registers.

First, a recap. The "endianness" of an item affects its representation in memory only. In a register, a number is just a sequence of bits - 64 bits in the case of AArch64 general purpose registers. Memory, however, is a sequence of addressable units of 8 bits in size. Any number greater than 8 bits must therefore be split up into 8-bit chunks, and endianness describes the order in which these chunks are laid out in memory.

A "little endian" layout has the least significant byte first (lowest in memory address). A "big endian" layout has the *most* significant byte first. This means that when loading an item from big endian memory, the lowest 8-bits in memory must go in the most significant 8-bits, and so forth.

4.25.3 `LDR` and `LD1`

A vector is a consecutive sequence of items that are operated on simultaneously. To load a 64-bit vector, 64 bits need to be read from memory. In little endian mode, we can do this by just performing a 64-bit load - `LDR q0, [foo]`. However if we try this in big endian mode, because of the byte swapping the lane indices end up being swapped! The zero'th item as laid out in memory becomes the n'th lane in the vector.

Because of this, the instruction `LD1` performs a vector load but performs byte swapping not on the entire 64 bits, but on the individual items within the vector. This means that the register content is the same as it would have been on a little endian system.

It may seem that `LD1` should suffice to peform vector loads on a big endian machine. However there are pros and cons to the two approaches that make it less than simple which register format to pick.

There are two options:

1. The content of a vector register is the same *as if* it had been loaded with an `LDR` instruction.
2. The content of a vector register is the same *as if* it had been loaded with an `LD1` instruction.

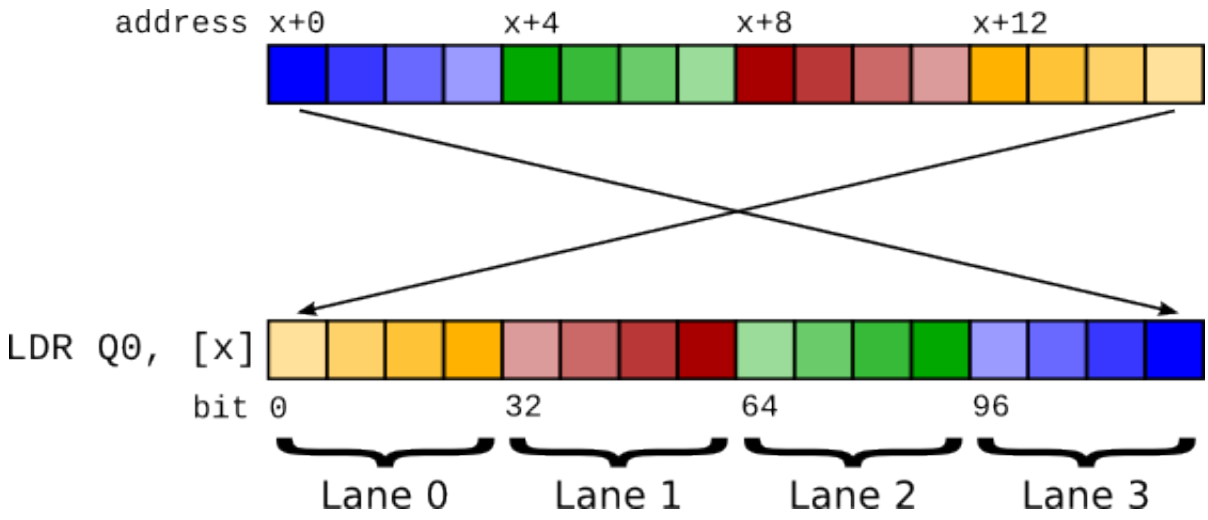


Figure 4.1: Big endian vector load using `LDR`.

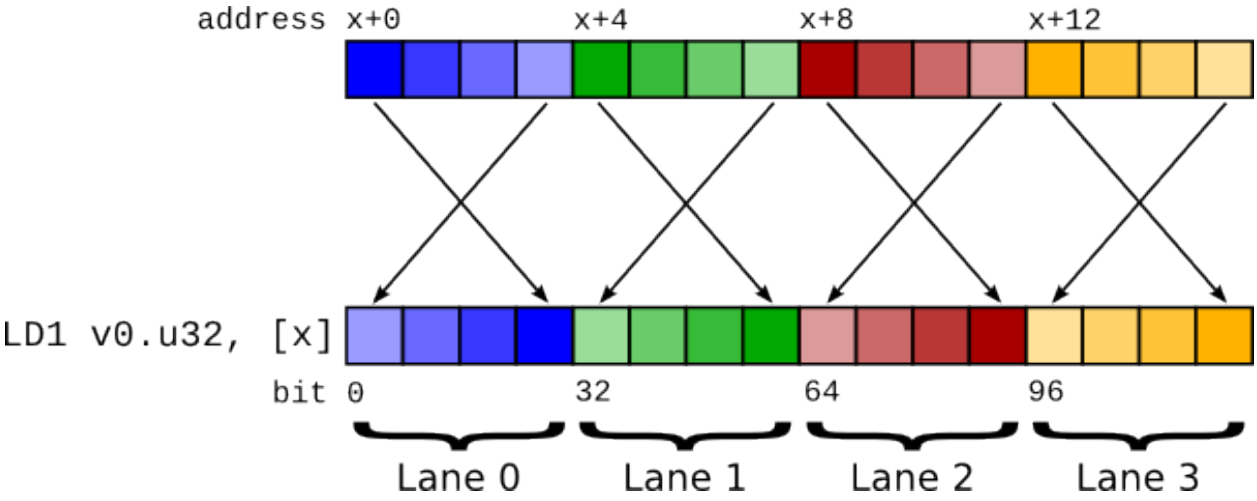


Figure 4.2: Big endian vector load using `LD1`. Note that the lanes retain the correct ordering.

Because `LD1 == LDR + REV` and similarly `LDR == LD1 + REV` (on a big endian system), we can simulate either type of load with the other type of load plus a `REV` instruction. So we're not deciding which instructions to use, but which format to use (which will then influence which instruction is best to use).

Note that throughout this section we only mention loads. Stores have exactly the same problems as their associated loads, so have been skipped for brevity.

4.25.4 Considerations

LLVM IR Lane ordering

LLVM IR has first class vector types. In LLVM IR, the zero'th element of a vector resides at the lowest memory address. The optimizer relies on this property in certain areas, for example when concatenating vectors together. The intention is for arrays and vectors to have identical memory layouts - `[4 x i8]` and `<4 x i8>` should be represented the same in memory. Without this property there would be many special cases that the optimizer would have to cleverly handle.

Use of `LDR` would break this lane ordering property. This doesn't preclude the use of `LDR`, but we would have to do one of two things:

1. Insert a `REV` instruction to reverse the lane order after every `LDR`.
2. Disable all optimizations that rely on lane layout, and for every access to an individual lane (`insertelement`/`extractelement`/`shufflevector`) reverse the lane index.

AAPCS

The ARM procedure call standard (AAPCS) defines the ABI for passing vectors between functions in registers. It states:

> When a short vector is transferred between registers and memory it is treated as an opaque object. That is a short vector is stored in memory as if it were stored with a single `STR` of the entire register; a short vector is loaded from memory using the corresponding `LDR` instruction. On a little-endian system this means that element 0 will always contain the lowest addressed element of a short vector; on a big-endian system element 0 will contain the highest-addressed element of a short vector.
>
> —Procedure Call Standard for the ARM 64-bit Architecture (AArch64), 4.1.2 Short Vectors

The use of `LDR` and `STR` as the ABI defines has at least one advantage over `LD1` and `ST1`. `LDR` and `STR` are oblivious to the size of the individual lanes of a vector. `LD1` and `ST1` are not - the lane size is encoded within them. This is important across an ABI boundary, because it would become necessary to know the lane width the callee expects. Consider the following code:

```
<callee.c>
void callee(uint32x2_t v) {
  ...
}

<caller.c>
extern void callee(uint32x2_t);
void caller() {
  callee(...);
}
```

If `callee` changed its signature to `uint16x4_t`, which is equivalent in register content, if we passed as `LD1` we'd break this code until `caller` was updated and recompiled.

There is an argument that if the signatures of the two functions are different then the behaviour should be undefined. But there may be functions that are agnostic to the lane layout of the vector, and treating the vector as an opaque value (just loading it and storing it) would be impossible without a common format across ABI boundaries.

So to preserve ABI compatibility, we need to use the `LDR` lane layout across function calls.

Alignment

In strict alignment mode, `LDR qX` requires its address to be 128-bit aligned, whereas `LD1` only requires it to be as aligned as the lane size. If we canonicalised on using `LDR`, we'd still need to use `LD1` in some places to avoid alignment faults (the result of the `LD1` would then need to be reversed with `REV`).

Most operating systems however do not run with alignment faults enabled, so this is often not an issue.

Summary

The following table summarises the instructions that are required to be emitted for each property mentioned above for each of the two solutions.

	`LDR` layout	**`LD1` layout**
Lane ordering	`LDR + REV`	`LD1`
AAPCS	`LDR`	`LD1 + REV`
Alignment for strict mode	`LDR` / `LD1 + REV`	`LD1`

Neither approach is perfect, and choosing one boils down to choosing the lesser of two evils. The issue with lane ordering, it was decided, would have to change target-agnostic compiler passes and would result in a strange IR in which lane indices were reversed. It was decided that this was worse than the changes that would have to be made to support `LD1`, so `LD1` was chosen as the canonical vector load instruction (and by inference, `ST1` for vector stores).

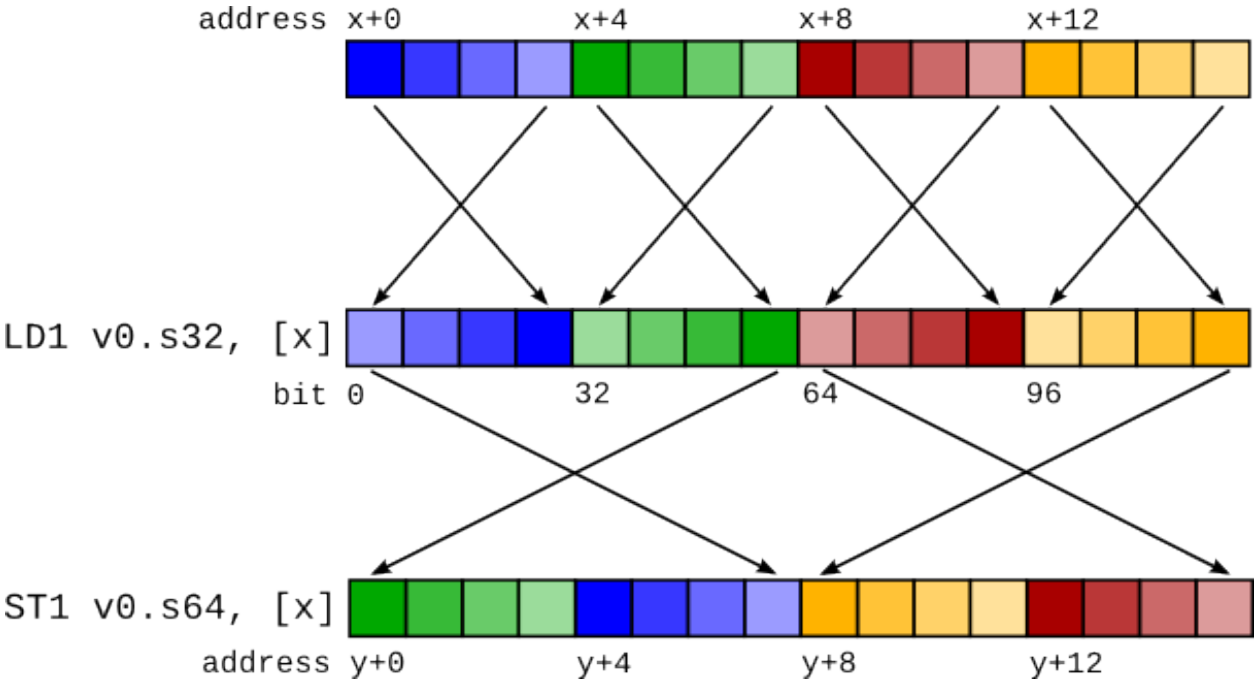
4.25.5 Implementation

There are 3 parts to the implementation:

1. Predicate `LDR` and `STR` instructions so that they are never allowed to be selected to generate vector loads and stores. The exception is one-lane vectors [1] - these by definition cannot have lane ordering problems so are fine to use `LDR`/`STR`.
2. Create code generation patterns for bitconverts that create `REV` instructions.
3. Make sure appropriate bitconverts are created so that vector values get passed over call boundaries as 1-element vectors (which is the same as if they were loaded with `LDR`).

[1] One lane vectors may seem useless as a concept but they serve to distinguish between values held in general purpose registers and values held in NEON/VFP registers. For example, an `i64` would live in an `x` register, but `<1 x i64>` would live in a `d` register.

Bitconverts



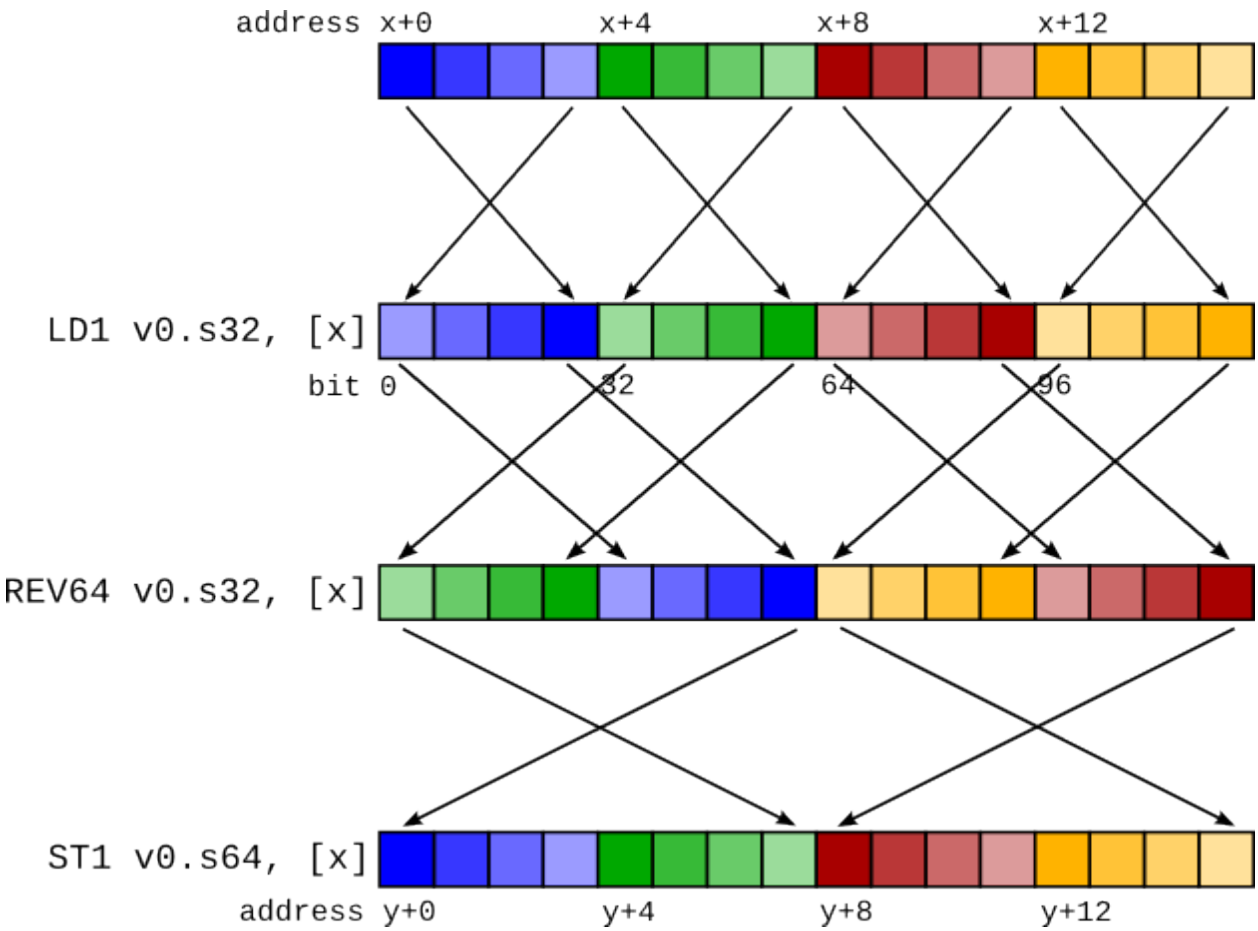
The main problem with the `LD1` solution is dealing with bitconverts (or bitcasts, or reinterpret casts). These are pseudo instructions that only change the compiler's interpretation of data, not the underlying data itself. A requirement is that if data is loaded and then saved again (called a "round trip"), the memory contents should be the same after the store as before the load. If a vector is loaded and is then bitconverted to a different vector type before storing, the round trip will currently be broken.

Take for example this code sequence:

```
%0 = load <4 x i32> %x
%1 = bitcast <4 x i32> %0 to <2 x i64>
     store <2 x i64> %1, <2 x i64>* %y
```

This would produce a code sequence such as that in the figure on the right. The mismatched `LD1` and `ST1` cause the stored data to differ from the loaded data.

When we see a bitcast from type `X` to type `Y`, what we need to do is to change the in-register representation of the data to be *as if* it had just been loaded by a `LD1` of type `Y`.



Conceptually this is simple - we can insert a `REV` undoing the `LD1` of type `X` (converting the in-register representation to the same as if it had been loaded by `LDR`) and then insert another `REV` to change the representation to be as if it had been loaded by an `LD1` of type `Y`.

For the previous example, this would be:

```
LD1   v0.4s, [x]

REV64 v0.4s, v0.4s                  // There is no REV128 instruction, so it must be synthesizedcd
EXT   v0.16b, v0.16b, v0.16b, #8    // with a REV64 then an EXT to swap the two 64-bit elements.

REV64 v0.2d, v0.2d
EXT   v0.16b, v0.16b, v0.16b, #8

ST1   v0.2d, [y]
```

It turns out that these `REV` pairs can, in almost all cases, be squashed together into a single `REV`. For the example above, a `REV128 4s` + `REV128 2d` is actually a `REV64 4s`, as shown in the figure on the right.

4.26 LLVM Code Coverage Mapping Format

4.26.1 Introduction

LLVM's code coverage mapping format is used to provide code coverage analysis using LLVM's and Clang's instrumenation based profiling (Clang's `-fprofile-instr-generate` option).

This document is aimed at those who use LLVM's code coverage mapping to provide code coverage analysis for their own programs, and for those who would like to know how it works under the hood. A prior knowledge of how Clang's profile guided optimization works is useful, but not required.

We start by showing how to use LLVM and Clang for code coverage analysis, then we briefly desribe LLVM's code coverage mapping format and the way that Clang and LLVM's code coverage tool work with this format. After the basics are down, more advanced features of the coverage mapping format are discussed - such as the data structures, LLVM IR representation and the binary encoding.

4.26.2 Quick Start

Here's a short story that describes how to generate code coverage overview for a sample source file called *test.c*.

- First, compile an instrumented version of your program using Clang's `-fprofile-instr-generate` option with the additional `-fcoverage-mapping` option:

```
clang -o test -fprofile-instr-generate -fcoverage-mapping test.c
```

- Then, run the instrumented binary. The runtime will produce a file called *default.profraw* containing the raw profile instrumentation data:

  ```
  ./test
  ```

- After that, merge the profile data using the *llvm-profdata* tool:

  ```
  llvm-profdata merge -o test.profdata default.profraw
  ```

- Finally, run LLVM's code coverage tool (*llvm-cov*) to produce the code coverage overview for the sample source file:

  ```
  llvm-cov show ./test -instr-profile=test.profdata test.c
  ```

4.26.3 High Level Overview

LLVM's code coverage mapping format is designed to be a self contained data format, that can be embedded into the LLVM IR and object files. It's described in this document as a **mapping** format because its goal is to store the data that is required for a code coverage tool to map between the specific source ranges in a file and the execution counts obtained after running the instrumented version of the program.

The mapping data is used in two places in the code coverage process:

1. When clang compiles a source file with `-fcoverage-mapping`, it generates the mapping information that describes the mapping between the source ranges and the profiling instrumentation counters. This information gets embedded into the LLVM IR and conveniently ends up in the final executable file when the program is linked.
2. It is also used by *llvm-cov* - the mapping information is extracted from an object file and is used to associate the execution counts (the values of the profile instrumentation counters), and the source ranges in a file. After that, the tool is able to generate various code coverage reports for the program.

The coverage mapping format aims to be a "universal format" that would be suitable for usage by any frontend, and not just by Clang. It also aims to provide the frontend the possibility of generating the minimal coverage mapping data in order to reduce the size of the IR and object files - for example, instead of emitting mapping information for each statement in a function, the frontend is allowed to group the statements with the same execution count into regions of code, and emit the mapping information only for those regions.

4.26.4 Advanced Concepts

The remainder of this guide is meant to give you insight into the way the coverage mapping format works.

The coverage mapping format operates on a per-function level as the profile instrumentation counters are associated with a specific function. For each function that requires code coverage, the frontend has to create coverage mapping data that can map between the source code ranges and the profile instrumentation counters for that function.

Mapping Region

The function's coverage mapping data contains an array of mapping regions. A mapping region stores the source code range that is covered by this region, the file id, the coverage mapping counter and the region's kind. There are several kinds of mapping regions:

- Code regions associate portions of source code and coverage mapping counters. They make up the majority of the mapping regions. They are used by the code coverage tool to compute the execution counts for lines, highlight the regions of code that were never executed, and to obtain the various code coverage statistics for a function. For example:

- Skipped regions are used to represent source ranges that were skipped by Clang's preprocessor. They don't associate with coverage mapping counters, as the frontend knows that they are never executed. They are used by the code coverage tool to mark the skipped lines inside a function as non-code lines that don't have execution counts. For example:
- Expansion regions are used to represent Clang's macro expansions. They have an additional property - *expanded file id*. This property can be used by the code coverage tool to find the mapping regions that are created as a result of this macro expansion, by checking if their file id matches the expanded file id. They don't associate with coverage mapping counters, as the code coverage tool can determine the execution count for this region by looking up the execution count of the first region with a corresponding file id. For example:

Source Range:

The source range record contains the starting and ending location of a certain mapping region. Both locations include the line and the column numbers.

File ID:

The file id an integer value that tells us in which source file or macro expansion is this region located. It enables Clang to produce mapping information for the code defined inside macros, like this example demonstrates:

Counter:

A coverage mapping counter can represents a reference to the profile instrumentation counter. The execution count for a region with such counter is determined by looking up the value of the corresponding profile instrumentation counter.

It can also represent a binary arithmetical expression that operates on coverage mapping counters or other expressions. The execution count for a region with an expression counter is determined by evaluating the expression's arguments and then adding them together or subtracting them from one another. In the example below, a subtraction expression is used to compute the execution count for the compound statement that follows the *else* keyword:

Finally, a coverage mapping counter can also represent an execution count of of zero. The zero counter is used to provide coverage mapping for unreachable statements and expressions, like in the example below:

The zero counters allow the code coverage tool to display proper line execution counts for the unreachable lines and highlight the unreachable code. Without them, the tool would think that those lines and regions were still executed, as it doesn't possess the frontend's knowledge.

4.26.5 LLVM IR Representation

The coverage mapping data is stored in the LLVM IR using a single global constant structure variable called *__llvm_coverage_mapping* with the *__llvm_covmap* section specifier.

For example, let's consider a C file and how it gets compiled to LLVM:

```
int foo() {
  return 42;
}
int bar() {
  return 13;
}
```

The coverage mapping variable generated by Clang is:

```
@__llvm_coverage_mapping = internal constant { i32, i32, i32, i32, [2 x { i8*, i32, i32 }], [40 x i8]
{ i32 2,  ; The number of function records
  i32 20, ; The length of the string that contains the encoded translation unit filenames
  i32 20, ; The length of the string that contains the encoded coverage mapping data
  i32 0,  ; Coverage mapping format version
  [2 x { i8*, i32, i32 }] [ ; Function records
   { i8*, i32, i32 } { i8* getelementptr inbounds ([3 x i8]* @__llvm_profile_name_foo, i32 0, i32 0),
     i32 3, ; Function's name length
     i32 9  ; Function's encoded coverage mapping data string length
   },
   { i8*, i32, i32 } { i8* getelementptr inbounds ([3 x i8]* @__llvm_profile_name_bar, i32 0, i32 0),
     i32 3, ; Function's name length
     i32 9  ; Function's encoded coverage mapping data string length
   }],
 [40 x i8] c"..." ; Encoded data (dissected later)
}, section "__llvm_covmap", align 8
```

Version:

The coverage mapping version number can have the following values:

- 0 — The first (current) version of the coverage mapping format.

Function record:

A function record is a structure of the following type:

```
{ i8*, i32, i32 }
```

It contains the pointer to the function's name, function's name length, and the length of the encoded mapping data for that function.

Encoded data:

The encoded data is stored in a single string that contains the encoded filenames used by this translation unit and the encoded coverage mapping data for each function in this translation unit.

The encoded data has the following structure:

```
[filenames, coverageMappingDataForFunctionRecord0, coverageMappingDataForFunctionRecord1,
..., padding]
```

If necessary, the encoded data is padded with zeroes so that the size of the data string is rounded up to the nearest multiple of 8 bytes.

Dissecting the sample:

Here's an overview of the encoded data that was stored in the IR for the coverage mapping sample that was shown earlier:

- The IR contains the following string constant that represents the encoded coverage mapping data for the sample translation unit:

  ```
  c"\01\12/Users/alex/test.c\01\00\00\01\01\01\0C\02\02\01\00\00\01\01\04\0C\02\02\00\00"
  ```

- The string contains values that are encoded in the LEB128 format, which is used throughout for storing integers. It also contains a string value.
- The length of the substring that contains the encoded translation unit filenames is the value of the second field in the *__llvm_coverage_mapping* structure, which is 20, thus the filenames are encoded in this string:

  ```
  c"\01\12/Users/alex/test.c"
  ```

  This string contains the following data:

  - Its first byte has a value of `0x01`. It stores the number of filenames contained in this string.
  - Its second byte stores the length of the first filename in this string.
  - The remaining 18 bytes are used to store the first filename.
- The length of the substring that contains the encoded coverage mapping data for the first function is the value of the third field in the first structure in an array of function records stored in the fifth field of the *__llvm_coverage_mapping* structure, which is the 9. Therefore, the coverage mapping for the first function record is encoded in this string:

  ```
  c"\01\00\00\01\01\01\0C\02\02"
  ```

  This string consists of the following bytes:

`0x01`	The number of file ids used by this function. There is only one file id used by the mapping data in this function.
`0x00`	An index into the filenames array which corresponds to the file "/Users/alex/test.c".
`0x00`	The number of counter expressions used by this function. This function doesn't use any expressions.
`0x01`	The number of mapping regions that are stored in an array for the function's file id #0.
`0x01`	The coverage mapping counter for the first region in this function. The value of 1 tells us that it's a coverage mapping counter that is a reference ot the profile instrumentation counter with an index of 0.
`0x01`	The starting line of the first mapping region in this function.
`0x0C`	The starting column of the first mapping region in this function.
`0x02`	The ending line of the first mapping region in this function.
`0x02`	The ending column of the first mapping region in this function.

- The length of the substring that contains the encoded coverage mapping data for the second function record is also 9. It's structured like the mapping data for the first function record.
- The two trailing bytes are zeroes and are used to pad the coverage mapping data to give it the 8 byte alignment.

4.26.6 Encoding

The per-function coverage mapping data is encoded as a stream of bytes, with a simple structure. The structure consists of the encoding types like variable-length unsigned integers, that are used to encode File ID Mapping, Counter Expressions and the Mapping Regions.

The format of the structure follows:

```
[file id mapping, counter expressions, mapping regions]
```

The translation unit filenames are encoded using the same encoding types as the per-function coverage mapping data, with the following structure:

```
[numFilenames : LEB128, filename0 : string, filename1 : string,
...]
```

Types

This section describes the basic types that are used by the encoding format and can appear after `:` in the `[foo : type]` description.

LEB128

LEB128 is an unsigned interger value that is encoded using DWARF's LEB128 encoding, optimizing for the case where values are small (1 byte for values less than 128).

Strings

```
[length :  LEB128, characters...]
```

String values are encoded with a LEB value for the length of the string and a sequence of bytes for its characters.

File ID Mapping

```
[numIndices :  LEB128, filenameIndex0 :  LEB128, filenameIndex1 :  LEB128,
...]
```

File id mapping in a function's coverage mapping stream contains the indices into the translation unit's filenames array.

Counter

```
[value :  LEB128]
```

A coverage mapping counter is stored in a single LEB value. It is composed of two things — the tag which is stored in the lowest 2 bits, and the counter data which is stored in the remaining bits.

Tag:

The counter's tag encodes the counter's kind and, if the counter is an expression, the expression's kind. The possible tag values are:

- 0 - The counter is zero.
- 1 - The counter is a reference to the profile instrumentation counter.
- 2 - The counter is a subtraction expression.
- 3 - The counter is an addition expression.

Data:

The counter's data is interpreted in the following manner:

- When the counter is a reference to the profile instrumentation counter, then the counter's data is the id of the profile counter.
- When the counter is an expression, then the counter's data is the index into the array of counter expressions.

Counter Expressions

```
[numExpressions : LEB128, expr0LHS : LEB128, expr0RHS : LEB128, expr1LHS :
LEB128, expr1RHS : LEB128, ...]
```

Counter expressions consist of two counters as they represent binary arithmetic operations. The expression's kind is determined from the tag of the counter that references this expression.

Mapping Regions

```
[numRegionArrays : LEB128, regionsForFile0, regionsForFile1, ...]
```

The mapping regions are stored in an array of sub-arrays where every region in a particular sub-array has the same file id.

The file id for a sub-array of regions is the index of that sub-array in the main array e.g. The first sub-array will have the file id of 0.

Sub-Array of Regions

```
[numRegions : LEB128, region0, region1, ...]
```

The mapping regions for a specific file id are stored in an array that is sorted in an ascending order by the region's starting location.

Mapping Region

```
[header, source range]
```

The mapping region record contains two sub-records — the header, which stores the counter and/or the region's kind, and the source range that contains the starting and ending location of this region.

Header

```
[counter]
```

or

```
[pseudo-counter]
```

The header encodes the region's counter and the region's kind.

The value of the counter's tag distinguishes between the counters and pseudo-counters — if the tag is zero, than this header contains a pseudo-counter, otherwise this header contains an ordinary counter.

Counter: A mapping region whose header has a counter with a non-zero tag is a code region.

Pseudo-Counter: `[value : LEB128]`

A pseudo-counter is stored in a single LEB value, just like the ordinary counter. It has the following interpretation:

- bits 0-1: tag, which is always 0.
- bit 2: expansionRegionTag. If this bit is set, then this mapping region is an expansion region.

- remaining bits: data. If this region is an expansion region, then the data contains the expanded file id of that region.

  Otherwise, the data contains the region's kind. The possible region kind values are:

  - 0 - This mapping region is a code region with a counter of zero.
  - 2 - This mapping region is a skipped region.

Source Range

```
[deltaLineStart : LEB128, columnStart : LEB128, numLines : LEB128,
columnEnd : LEB128]
```

The source range record contains the following fields:

- *deltaLineStart*: The difference between the starting line of the current mapping region and the starting line of the previous mapping region.

  If the current mapping region is the first region in the current sub-array, then it stores the starting line of that region.

- *columnStart*: The starting column of the mapping region.
- *numLines*: The difference between the ending line and the starting line of the current mapping region.
- *columnEnd*: The ending column of the mapping region.

Writing an LLVM Pass Information on how to write LLVM transformations and analyses.

Writing an LLVM Backend Information on how to write LLVM backends for machine targets.

The LLVM Target-Independent Code Generator The design and implementation of the LLVM code generator. Useful if you are working on retargetting LLVM to a new architecture, designing a new codegen pass, or enhancing existing components.

TableGen Describes the TableGen tool, which is used heavily by the LLVM code generator.

LLVM Alias Analysis Infrastructure Information on how to write a new alias analysis implementation or how to use existing analyses.

Accurate Garbage Collection with LLVM The interfaces source-language compilers should use for compiling GC'd programs.

Source Level Debugging with LLVM This document describes the design and philosophy behind the LLVM source-level debugger.

Auto-Vectorization in LLVM This document describes the current status of vectorization in LLVM.

Exception Handling in LLVM This document describes the design and implementation of exception handling in LLVM.

LLVM bugpoint tool: design and usage Automatic bug finder and test-case reducer description and usage information.

LLVM Bitcode File Format This describes the file format and encoding used for LLVM "bc" files.

System Library This document describes the LLVM System Library (`lib/System`) and how to keep LLVM source code portable

LLVM Link Time Optimization: Design and Implementation This document describes the interface between LLVM intermodular optimizer and the linker and its design

The LLVM gold plugin How to build your programs with link-time optimization on Linux.

Debugging JIT-ed Code With GDB How to debug JITed code with GDB.

MCJIT Design and Implementation Describes the inner workings of MCJIT execution engine.

LLVM Branch Weight Metadata Provides information about Branch Prediction Information.

LLVM Block Frequency Terminology Provides information about terminology used in the `BlockFrequencyInfo` analysis pass.

Segmented Stacks in LLVM This document describes segmented stacks and how they are used in LLVM.

LLVM's Optional Rich Disassembly Output This document describes the optional rich disassembly output syntax.

How To Use Attributes Answers some questions about the new Attributes infrastructure.

User Guide for NVPTX Back-end This document describes using the NVPTX back-end to compile GPU kernels.

Stack maps and patch points in LLVM LLVM support for mapping instruction addresses to the location of values and allowing code to be patched.

Using ARM NEON instructions in big endian mode LLVM's support for generating NEON instructions on big endian ARM targets is somewhat nonintuitive. This document explains the implementation and rationale.

LLVM Code Coverage Mapping Format This describes the format and encoding used for LLVM's code coverage mapping.

CHAPTER

FIVE

DEVELOPMENT PROCESS DOCUMENTATION

Information about LLVM's development process.

5.1 LLVM Developer Policy

5.1.1 Introduction

This document contains the LLVM Developer Policy which defines the project's policy towards developers and their contributions. The intent of this policy is to eliminate miscommunication, rework, and confusion that might arise from the distributed nature of LLVM's development. By stating the policy in clear terms, we hope each developer can know ahead of time what to expect when making LLVM contributions. This policy covers all llvm.org subprojects, including Clang, LLDB, libc++, etc.

This policy is also designed to accomplish the following objectives:

1. Attract both users and developers to the LLVM project.
2. Make life as simple and easy for contributors as possible.

3. Keep the top of Subversion trees as stable as possible.
4. Establish awareness of the project's *copyright, license, and patent policies* with contributors to the project.

This policy is aimed at frequent contributors to LLVM. People interested in contributing one-off patches can do so in an informal way by sending them to the llvm-commits mailing list and engaging another developer to see it through the process.

5.1.2 Developer Policies

This section contains policies that pertain to frequent LLVM developers. We always welcome one-off patches from people who do not routinely contribute to LLVM, but we expect more from frequent contributors to keep the system as efficient as possible for everyone. Frequent LLVM contributors are expected to meet the following requirements in order for LLVM to maintain a high standard of quality.

Stay Informed

Developers should stay informed by reading at least the "dev" mailing list for the projects you are interested in, such as llvmdev for LLVM, cfe-dev for Clang, or lldb-dev for LLDB. If you are doing anything more than just casual work on LLVM, it is suggested that you also subscribe to the "commits" mailing list for the subproject you're interested in, such as llvm-commits, cfe-commits, or lldb-commits. Reading the "commits" list and paying attention to changes being made by others is a good way to see what other people are interested in and watching the flow of the project as a whole.

We recommend that active developers register an email account with LLVM Bugzilla and preferably subscribe to the llvm-bugs email list to keep track of bugs and enhancements occurring in LLVM. We really appreciate people who are proactive at catching incoming bugs in their components and dealing with them promptly.

Please be aware that all public LLVM mailing lists are public and archived, and that notices of confidentiality or non-disclosure cannot be respected.

Making and Submitting a Patch

When making a patch for review, the goal is to make it as easy for the reviewer to read it as possible. As such, we recommend that you:

1. Make your patch against the Subversion trunk, not a branch, and not an old version of LLVM. This makes it easy to apply the patch. For information on how to check out SVN trunk, please see the Getting Started Guide.
2. Similarly, patches should be submitted soon after they are generated. Old patches may not apply correctly if the underlying code changes between the time the patch was created and the time it is applied.
3. Patches should be made with `svn diff`, or similar. If you use a different tool, make sure it uses the `diff -u` format and that it doesn't contain clutter which makes it hard to read.
4. If you are modifying generated files, such as the top-level `configure` script, please separate out those changes into a separate patch from the rest of your changes.

Once your patch is ready, submit it by emailing it to the appropriate project's commit mailing list (or commit it directly if applicable). Alternatively, some patches get sent to the project's development list or component of the LLVM bug tracker, but the commit list is the primary place for reviews and should generally be preferred.

When sending a patch to a mailing list, it is a good idea to send it as an *attachment* to the message, not embedded into the text of the message. This ensures that your mailer will not mangle the patch when it sends it (e.g. by making whitespace changes or by wrapping lines).

For Thunderbird users: Before submitting a patch, please open *Preferences* > *Advanced* > *General* > *Config Editor*, find the key `mail.content_disposition_type`, and set its value to `1`. Without this setting, Thunderbird sends your attachment using `Content-Disposition: inline` rather than `Content-Disposition: attachment`. Apple Mail gamely displays such a file inline, making it difficult to work with for reviewers using that program.

When submitting patches, please do not add confidentiality or non-disclosure notices to the patches themselves. These notices conflict with the LLVM License and may result in your contribution being excluded.

Code Reviews

LLVM has a code review policy. Code review is one way to increase the quality of software. We generally follow these policies:

1. All developers are required to have significant changes reviewed before they are committed to the repository.
2. Code reviews are conducted by email on the relevant project's commit mailing list, or alternatively on the project's development list or bug tracker.
3. Code can be reviewed either before it is committed or after. We expect major changes to be reviewed before being committed, but smaller changes (or changes where the developer owns the component) can be reviewed after commit.
4. The developer responsible for a code change is also responsible for making all necessary review-related changes.
5. Code review can be an iterative process, which continues until the patch is ready to be committed. Specifically, once a patch is sent out for review, it needs an explicit "looks good" before it is submitted. Do not assume silent approval, or request active objections to the patch with a deadline.

Sometimes code reviews will take longer than you would hope for, especially for larger features. Accepted ways to speed up review times for your patches are:

- Review other people's patches. If you help out, everybody will be more willing to do the same for you; goodwill is our currency.
- Ping the patch. If it is urgent, provide reasons why it is important to you to get this patch landed and ping it every couple of days. If it is not urgent, the common courtesy ping rate is one week. Remember that you're asking for valuable time from other professional developers.
- Ask for help on IRC. Developers on IRC will be able to either help you directly, or tell you who might be a good reviewer.
- Split your patch into multiple smaller patches that build on each other. The smaller your patch, the higher the probability that somebody will take a quick look at it.

Developers should participate in code reviews as both reviewers and reviewees. If someone is kind enough to review your code, you should return the favor for someone else. Note that anyone is welcome to review and give feedback on a patch, but only people with Subversion write access can approve it.

There is a web based code review tool that can optionally be used for code reviews. See *Code Reviews with Phabricator*.

Code Owners

The LLVM Project relies on two features of its process to maintain rapid development in addition to the high quality of its source base: the combination of code review plus post-commit review for trusted maintainers. Having both is a great way for the project to take advantage of the fact that most people do the right thing most of the time, and only commit patches without pre-commit review when they are confident they are right.

The trick to this is that the project has to guarantee that all patches that are committed are reviewed after they go in: you don't want everyone to assume someone else will review it, allowing the patch to go unreviewed. To solve this problem, we have a notion of an 'owner' for a piece of the code. The sole responsibility of a code owner is to ensure that a commit to their area of the code is appropriately reviewed, either by themself or by someone else. The list of current code owners can be found in the file CODE_OWNERS.TXT in the root of the LLVM source tree.

Note that code ownership is completely different than reviewers: anyone can review a piece of code, and we welcome code review from anyone who is interested. Code owners are the "last line of defense" to guarantee that all patches that are committed are actually reviewed.

Being a code owner is a somewhat unglamorous position, but it is incredibly important for the ongoing success of the project. Because people get busy, interests change, and unexpected things happen, code ownership is purely opt-in, and anyone can choose to resign their "title" at any time. For now, we do not have an official policy on how one gets elected to be a code owner.

Test Cases

Developers are required to create test cases for any bugs fixed and any new features added. Some tips for getting your testcase approved:

- All feature and regression test cases are added to the `llvm/test` directory. The appropriate sub-directory should be selected (see the *Testing Guide* for details).
- Test cases should be written in *LLVM assembly language*.
- Test cases, especially for regressions, should be reduced as much as possible, by *bugpoint* or manually. It is unacceptable to place an entire failing program into `llvm/test` as this creates a *time-to-test* burden on all developers. Please keep them short.

Note that llvm/test and clang/test are designed for regression and small feature tests only. More extensive test cases (e.g., entire applications, benchmarks, etc) should be added to the `llvm-test` test suite. The llvm-test suite is for coverage (correctness, performance, etc) testing, not feature or regression testing.

Quality

The minimum quality standards that any change must satisfy before being committed to the main development branch are:

1. Code must adhere to the LLVM Coding Standards.
2. Code must compile cleanly (no errors, no warnings) on at least one platform.
3. Bug fixes and new features should include a testcase so we know if the fix/feature ever regresses in the future.
4. Code must pass the `llvm/test` test suite.
5. The code must not cause regressions on a reasonable subset of llvm-test, where "reasonable" depends on the contributor's judgement and the scope of the change (more invasive changes require more testing). A reasonable subset might be something like "`llvm-test/MultiSource/Benchmarks`".

Additionally, the committer is responsible for addressing any problems found in the future that the change is responsible for. For example:

- The code should compile cleanly on all supported platforms.
- The changes should not cause any correctness regressions in the `llvm-test` suite and must not cause any major performance regressions.
- The change set should not cause performance or correctness regressions for the LLVM tools.

- The changes should not cause performance or correctness regressions in code compiled by LLVM on all applicable targets.
- You are expected to address any Bugzilla bugs that result from your change.

We prefer for this to be handled before submission but understand that it isn't possible to test all of this for every submission. Our build bots and nightly testing infrastructure normally finds these problems. A good rule of thumb is to check the nightly testers for regressions the day after your change. Build bots will directly email you if a group of commits that included yours caused a failure. You are expected to check the build bot messages to see if they are your fault and, if so, fix the breakage.

Commits that violate these quality standards (e.g. are very broken) may be reverted. This is necessary when the change blocks other developers from making progress. The developer is welcome to re-commit the change after the problem has been fixed.

Obtaining Commit Access

We grant commit access to contributors with a track record of submitting high quality patches. If you would like commit access, please send an email to Chris with the following information:

1. The user name you want to commit with, e.g. "hacker".
2. The full name and email address you want message to llvm-commits to come from, e.g. "J. Random Hacker <hacker@yoyodyne.com>".
3. A "password hash" of the password you want to use, e.g. "`2ACR96qjUqsyM`". Note that you don't ever tell us what your password is; you just give it to us in an encrypted form. To get this, run "`htpasswd`" (a utility that comes with apache) in crypt mode (often enabled with "`-d`"), or find a web page that will do it for you.

Once you've been granted commit access, you should be able to check out an LLVM tree with an SVN URL of "https://username@llvm.org/..." instead of the normal anonymous URL of "http://llvm.org/...". The first time you commit you'll have to type in your password. Note that you may get a warning from SVN about an untrusted key; you can ignore this. To verify that your commit access works, please do a test commit (e.g. change a comment or add a blank line). Your first commit to a repository may require the autogenerated email to be approved by a mailing list. This is normal and will be done when the mailing list owner has time.

If you have recently been granted commit access, these policies apply:

1. You are granted *commit-after-approval* to all parts of LLVM. To get approval, submit a patch to llvm-commits. When approved, you may commit it yourself.
2. You are allowed to commit patches without approval which you think are obvious. This is clearly a subjective decision — we simply expect you to use good judgement. Examples include: fixing build breakage, reverting obviously broken patches, documentation/comment changes, any other minor changes.
3. You are allowed to commit patches without approval to those portions of LLVM that you have contributed or maintain (i.e., have been assigned responsibility for), with the proviso that such commits must not break the build. This is a "trust but verify" policy, and commits of this nature are reviewed after they are committed.
4. Multiple violations of these policies or a single egregious violation may cause commit access to be revoked.

In any case, your changes are still subject to code review (either before or after they are committed, depending on the nature of the change). You are encouraged to review other peoples' patches as well, but you aren't required to do so.

Making a Major Change

When a developer begins a major new project with the aim of contributing it back to LLVM, they should inform the community with an email to the llvmdev email list, to the extent possible. The reason for this is to:

1. keep the community informed about future changes to LLVM,

2. avoid duplication of effort by preventing multiple parties working on the same thing and not knowing about it, and
3. ensure that any technical issues around the proposed work are discussed and resolved before any significant work is done.

The design of LLVM is carefully controlled to ensure that all the pieces fit together well and are as consistent as possible. If you plan to make a major change to the way LLVM works or want to add a major new extension, it is a good idea to get consensus with the development community before you start working on it.

Once the design of the new feature is finalized, the work itself should be done as a series of incremental changes, not as a long-term development branch.

Incremental Development

In the LLVM project, we do all significant changes as a series of incremental patches. We have a strong dislike for huge changes or long-term development branches. Long-term development branches have a number of drawbacks:

1. Branches must have mainline merged into them periodically. If the branch development and mainline development occur in the same pieces of code, resolving merge conflicts can take a lot of time.
2. Other people in the community tend to ignore work on branches.
3. Huge changes (produced when a branch is merged back onto mainline) are extremely difficult to code review.
4. Branches are not routinely tested by our nightly tester infrastructure.
5. Changes developed as monolithic large changes often don't work until the entire set of changes is done. Breaking it down into a set of smaller changes increases the odds that any of the work will be committed to the main repository.

To address these problems, LLVM uses an incremental development style and we require contributors to follow this practice when making a large/invasive change. Some tips:

- Large/invasive changes usually have a number of secondary changes that are required before the big change can be made (e.g. API cleanup, etc). These sorts of changes can often be done before the major change is done, independently of that work.
- The remaining inter-related work should be decomposed into unrelated sets of changes if possible. Once this is done, define the first increment and get consensus on what the end goal of the change is.
- Each change in the set can be stand alone (e.g. to fix a bug), or part of a planned series of changes that works towards the development goal.
- Each change should be kept as small as possible. This simplifies your work (into a logical progression), simplifies code review and reduces the chance that you will get negative feedback on the change. Small increments also facilitate the maintenance of a high quality code base.
- Often, an independent precursor to a big change is to add a new API and slowly migrate clients to use the new API. Each change to use the new API is often "obvious" and can be committed without review. Once the new API is in place and used, it is much easier to replace the underlying implementation of the API. This implementation change is logically separate from the API change.

If you are interested in making a large change, and this scares you, please make sure to first discuss the change/gather consensus then ask about the best way to go about making the change.

Attribution of Changes

When contributors submit a patch to an LLVM project, other developers with commit access may commit it for the author once appropriate (based on the progression of code review, etc.). When doing so, it is important to retain correct

attribution of contributions to their contributors. However, we do not want the source code to be littered with random attributions "this code written by J. Random Hacker" (this is noisy and distracting). In practice, the revision control system keeps a perfect history of who changed what, and the CREDITS.txt file describes higher-level contributions. If you commit a patch for someone else, please say "patch contributed by J. Random Hacker!" in the commit message. Overall, please do not add contributor names to the source code.

Also, don't commit patches authored by others unless they have submitted the patch to the project or you have been authorized to submit them on their behalf (you work together and your company authorized you to contribute the patches, etc.). The author should first submit them to the relevant project's commit list, development list, or LLVM bug tracker component. If someone sends you a patch privately, encourage them to submit it to the appropriate list first.

IR Backwards Compatibility

When the IR format has to be changed, keep in mind that we try to maintain some backwards compatibility. The rules are intended as a balance between convenience for llvm users and not imposing a big burden on llvm developers:

- The textual format is not backwards compatible. We don't change it too often, but there are no specific promises.
- The bitcode format produced by a X.Y release will be readable by all following X.Z releases and the (X+1).0 release.
- Newer releases can ignore features from older releases, but they cannot miscompile them. For example, if nsw is ever replaced with something else, dropping it would be a valid way to upgrade the IR.
- Debug metadata is special in that it is currently dropped during upgrades.
- Non-debug metadata is defined to be safe to drop, so a valid way to upgrade it is to drop it. That is not very user friendly and a bit more effort is expected, but no promises are made.

5.1.3 Copyright, License, and Patents

Note: This section deals with legal matters but does not provide legal advice. We are not lawyers — please seek legal counsel from an attorney.

This section addresses the issues of copyright, license and patents for the LLVM project. The copyright for the code is held by the individual contributors of the code and the terms of its license to LLVM users and developers is the University of Illinois/NCSA Open Source License (with portions dual licensed under the MIT License, see below). As contributor to the LLVM project, you agree to allow any contributions to the project to licensed under these terms.

Copyright

The LLVM project does not require copyright assignments, which means that the copyright for the code in the project is held by its respective contributors who have each agreed to release their contributed code under the terms of the LLVM License.

An implication of this is that the LLVM license is unlikely to ever change: changing it would require tracking down all the contributors to LLVM and getting them to agree that a license change is acceptable for their contribution. Since there are no plans to change the license, this is not a cause for concern.

As a contributor to the project, this means that you (or your company) retain ownership of the code you contribute, that it cannot be used in a way that contradicts the license (which is a liberal BSD-style license), and that the license for your contributions won't change without your approval in the future.

License

We intend to keep LLVM perpetually open source and to use a liberal open source license. **As a contributor to the project, you agree that any contributions be licensed under the terms of the corresponding subproject.** All of the code in LLVM is available under the University of Illinois/NCSA Open Source License, which boils down to this:

- You can freely distribute LLVM.
- You must retain the copyright notice if you redistribute LLVM.
- Binaries derived from LLVM must reproduce the copyright notice (e.g. in an included readme file).
- You can't use our names to promote your LLVM derived products.
- There's no warranty on LLVM at all.

We believe this fosters the widest adoption of LLVM because it **allows commercial products to be derived from LLVM** with few restrictions and without a requirement for making any derived works also open source (i.e. LLVM's license is not a "copyleft" license like the GPL). We suggest that you read the License if further clarification is needed.

In addition to the UIUC license, the runtime library components of LLVM (**compiler_rt, libc++, and libclc**) are also licensed under the MIT License, which does not contain the binary redistribution clause. As a user of these runtime libraries, it means that you can choose to use the code under either license (and thus don't need the binary redistribution clause), and as a contributor to the code that you agree that any contributions to these libraries be licensed under both licenses. We feel that this is important for runtime libraries, because they are implicitly linked into applications and therefore should not subject those applications to the binary redistribution clause. This also means that it is ok to move code from (e.g.) libc++ to the LLVM core without concern, but that code cannot be moved from the LLVM core to libc++ without the copyright owner's permission.

Note that the LLVM Project does distribute dragonegg, **which is GPL.** This means that anything "linked" into dragonegg must itself be compatible with the GPL, and must be releasable under the terms of the GPL. This implies that **any code linked into dragonegg and distributed to others may be subject to the viral aspects of the GPL** (for example, a proprietary code generator linked into dragonegg must be made available under the GPL). This is not a problem for code already distributed under a more liberal license (like the UIUC license), and GPL-containing subprojects are kept in separate SVN repositories whose LICENSE.txt files specifically indicate that they contain GPL code.

We have no plans to change the license of LLVM. If you have questions or comments about the license, please contact the LLVM Developer's Mailing List.

Patents

To the best of our knowledge, LLVM does not infringe on any patents (we have actually removed code from LLVM in the past that was found to infringe). Having code in LLVM that infringes on patents would violate an important goal of the project by making it hard or impossible to reuse the code for arbitrary purposes (including commercial use).

When contributing code, we expect contributors to notify us of any potential for patent-related trouble with their changes (including from third parties). If you or your employer own the rights to a patent and would like to contribute code to LLVM that relies on it, we require that the copyright owner sign an agreement that allows any other user of LLVM to freely use your patent. Please contact the oversight group for more details.

5.2 LLVM Makefile Guide

5.2.1 Introduction

This document provides *usage* information about the LLVM makefile system. While loosely patterned after the BSD makefile system, LLVM has taken a departure from BSD in order to implement additional features needed by LLVM. Although makefile systems, such as `automake`, were attempted at one point, it has become clear that the features needed by LLVM and the `Makefile` norm are too great to use a more limited tool. Consequently, LLVM requires simply GNU Make 3.79, a widely portable makefile processor. LLVM unabashedly makes heavy use of the features of GNU Make so the dependency on GNU Make is firm. If you're not familiar with `make`, it is recommended that you read the GNU Makefile Manual.

While this document is rightly part of the LLVM Programmer's Manual, it is treated separately here because of the volume of content and because it is often an early source of bewilderment for new developers.

5.2.2 General Concepts

The LLVM Makefile System is the component of LLVM that is responsible for building the software, testing it, generating distributions, checking those distributions, installing and uninstalling, etc. It consists of a several files throughout the source tree. These files and other general concepts are described in this section.

Projects

The LLVM Makefile System is quite generous. It not only builds its own software, but it can build yours too. Built into the system is knowledge of the `llvm/projects` directory. Any directory under `projects` that has both a `configure` script and a `Makefile` is assumed to be a project that uses the LLVM Makefile system. Building software that uses LLVM does not require the LLVM Makefile System nor even placement in the `llvm/projects` directory. However, doing so will allow your project to get up and running quickly by utilizing the built-in features that are used to compile LLVM. LLVM compiles itself using the same features of the makefile system as used for projects.

For further details, consult the Projects page.

Variable Values

To use the makefile system, you simply create a file named `Makefile` in your directory and declare values for certain variables. The variables and values that you select determine what the makefile system will do. These variables enable rules and processing in the makefile system that automatically Do The Right Thing (C).

Including Makefiles

Setting variables alone is not enough. You must include into your Makefile additional files that provide the rules of the LLVM Makefile system. The various files involved are described in the sections that follow.

`Makefile`

Each directory to participate in the build needs to have a file named `Makefile`. This is the file first read by `make`. It has three sections:

1. Settable Variables — Required that must be set first.
2. `include $(LEVEL)/Makefile.common` — include the LLVM Makefile system.
3. Override Variables — Override variables set by the LLVM Makefile system.

`Makefile.common`

Every project must have a `Makefile.common` file at its top source directory. This file serves three purposes:

1. It includes the project's configuration makefile to obtain values determined by the `configure` script. This is done by including the $(LEVEL)/Makefile.config file.
2. It specifies any other (static) values that are needed throughout the project. Only values that are used in all or a large proportion of the project's directories should be placed here.
3. It includes the standard rules for the LLVM Makefile system, $(LLVM_SRC_ROOT)/Makefile.rules. This file is the *guts* of the LLVM `Makefile` system.

Makefile.config

Every project must have a Makefile.config at the top of its *build* directory. This file is **generated** by the configure script from the pattern provided by the Makefile.config.in file located at the top of the project's *source* directory. The contents of this file depend largely on what configuration items the project uses, however most projects can get what they need by just relying on LLVM's configuration found in $(LLVM_OBJ_ROOT)/Makefile.config.

Makefile.rules

This file, located at $(LLVM_SRC_ROOT)/Makefile.rules is the heart of the LLVM Makefile System. It provides all the logic, dependencies, and rules for building the targets supported by the system. What it does largely depends on the values of make variables that have been set *before* Makefile.rules is included.

Comments

User Makefiles need not have comments in them unless the construction is unusual or it does not strictly follow the rules and patterns of the LLVM makefile system. Makefile comments are invoked with the pound (#) character. The # character and any text following it, to the end of the line, are ignored by make.

5.2.3 Tutorial

This section provides some examples of the different kinds of modules you can build with the LLVM makefile system. In general, each directory you provide will build a single object although that object may be composed of additionally compiled components.

Libraries

Only a few variable definitions are needed to build a regular library. Normally, the makefile system will build all the software into a single libname.o (pre-linked) object. This means the library is not searchable and that the distinction between compilation units has been dissolved. Optionally, you can ask for a shared library (.so) or archive library (.a) built. Archive libraries are the default. For example:

```
LIBRARYNAME = mylib
SHARED_LIBRARY = 1
BUILD_ARCHIVE = 1
```

says to build a library named mylib with both a shared library (mylib.so) and an archive library (mylib.a) version. The contents of all the libraries produced will be the same, they are just constructed differently. Note that you normally do not need to specify the sources involved. The LLVM Makefile system will infer the source files from the contents of the source directory.

The LOADABLE_MODULE=1 directive can be used in conjunction with SHARED_LIBRARY=1 to indicate that the resulting shared library should be openable with the dlopen function and searchable with the dlsym function (or your operating system's equivalents). While this isn't strictly necessary on Linux and a few other platforms, it is required on systems like HP-UX and Darwin. You should use LOADABLE_MODULE for any shared library that you intend to be loaded into an tool via the -load option. *Pass documentation* has an example of why you might want to do this.

Loadable Modules

In some situations, you need to create a loadable module. Loadable modules can be loaded into programs like `opt` or `llc` to specify additional passes to run or targets to support. Loadable modules are also useful for debugging a pass or providing a pass with another package if that pass can't be included in LLVM.

LLVM provides complete support for building such a module. All you need to do is use the `LOADABLE_MODULE` variable in your `Makefile`. For example, to build a loadable module named `MyMod` that uses the LLVM libraries `LLVMSupport.a` and `LLVMSystem.a`, you would specify:

```
LIBRARYNAME := MyMod
LOADABLE_MODULE := 1
LINK_COMPONENTS := support system
```

Use of the `LOADABLE_MODULE` facility implies several things:

1. **There will be no "`lib`" prefix on the module. This differentiates it from** a standard shared library of the same name.
2. The SHARED_LIBRARY variable is turned on.
3. The LINK_LIBS_IN_SHARED variable is turned on.

A loadable module is loaded by LLVM via the facilities of libtool's libltdl library which is part of `lib/System` implementation.

Tools

For building executable programs (tools), you must provide the name of the tool and the names of the libraries you wish to link with the tool. For example:

```
TOOLNAME = mytool
USEDLIBS = mylib
LINK_COMPONENTS = support system
```

says that we are to build a tool name `mytool` and that it requires three libraries: `mylib`, `LLVMSupport.a` and `LLVMSystem.a`.

Note that two different variables are used to indicate which libraries are linked: `USEDLIBS` and `LLVMLIBS`. This distinction is necessary to support projects. `LLVMLIBS` refers to the LLVM libraries found in the LLVM object directory. `USEDLIBS` refers to the libraries built by your project. In the case of building LLVM tools, `USEDLIBS` and `LLVMLIBS` can be used interchangeably since the "project" is LLVM itself and `USEDLIBS` refers to the same place as `LLVMLIBS`.

Also note that there are two different ways of specifying a library: with a `.a` suffix and without. Without the suffix, the entry refers to the re-linked (.o) file which will include *all* symbols of the library. This is useful, for example, to include all passes from a library of passes. If the `.a` suffix is used then the library is linked as a searchable library (with the `-l` option). In this case, only the symbols that are unresolved *at that point* will be resolved from the library, if they exist. Other (unreferenced) symbols will not be included when the `.a` syntax is used. Note that in order to use the `.a` suffix, the library in question must have been built with the `BUILD_ARCHIVE` option set.

JIT Tools

Many tools will want to use the JIT features of LLVM. To do this, you simply specify that you want an execution 'engine', and the makefiles will automatically link in the appropriate JIT for the host or an interpreter if none is available:

```
TOOLNAME = my_jit_tool
USEDLIBS = mylib
LINK_COMPONENTS = engine
```

Of course, any additional libraries may be listed as other components. To get a full understanding of how this changes the linker command, it is recommended that you:

```
% cd examples/Fibonacci
% make VERBOSE=1
```

5.2.4 Targets Supported

This section describes each of the targets that can be built using the LLVM Makefile system. Any target can be invoked from any directory but not all are applicable to a given directory (e.g. "check", "dist" and "install" will always operate as if invoked from the top level directory).

Target Name	Implied Targets	Target Description
`all`		Compile the software recursively. Default target.
`all-local`		Compile the software in the local directory only.
`check`		Change to the `test` directory in a project and run the test suite there.
`check-local`		Run a local test suite. Generally this is only defined in the `Makefile` of the project's `test` directory.
`clean`		Remove built objects recursively.
`clean-local`		Remove built objects from the local directory only.
`dist`	`all`	Prepare a source distribution tarball.
`dist-check`	`all`	Prepare a source distribution tarball and check that it builds.
`dist-clean`	`clean`	Clean source distribution tarball temporary files.
`install`	`all`	Copy built objects to installation directory.
`preconditions`	`all`	Check to make sure configuration and makefiles are up to date.
`printvars`	`all`	Prints variables defined by the makefile system (for debugging).
`tags`		Make C and C++ tags files for emacs and vi.
`uninstall`		Remove built objects from installation directory.

`all` (default)

When you invoke `make` with no arguments, you are implicitly instructing it to seek the `all` target (goal). This target is used for building the software recursively and will do different things in different directories. For example, in a `lib` directory, the `all` target will compile source files and generate libraries. But, in a `tools` directory, it will link libraries and generate executables.

`all-local`

This target is the same as all but it operates only on the current directory instead of recursively.

`check`

This target can be invoked from anywhere within a project's directories but always invokes the check-local target in the project's `test` directory, if it exists and has a `Makefile`. A warning is produced otherwise. If TESTSUITE is defined on the `make` command line, it will be passed down to the invocation of `make check-local` in the `test` directory. The intended usage for this is to assist in running specific suites of tests. If `TESTSUITE` is not set, the

implementation of `check-local` should run all normal tests. It is up to the project to define what different values for `TESTSUTE` will do. See the *Testing Guide* for further details.

check-local

This target should be implemented by the `Makefile` in the project's `test` directory. It is invoked by the `check` target elsewhere. Each project is free to define the actions of `check-local` as appropriate for that project. The LLVM project itself uses the *Lit* testing tool to run a suite of feature and regression tests. Other projects may choose to use **lit** or any other testing mechanism.

clean

This target cleans the build directory, recursively removing all things that the Makefile builds. The cleaning rules have been made guarded so they shouldn't go awry (via `rm -f $(UNSET_VARIABLE)/*` which will attempt to erase the entire directory structure).

clean-local

This target does the same thing as `clean` but only for the current (local) directory.

dist

This target builds a distribution tarball. It first builds the entire project using the `all` target and then tars up the necessary files and compresses it. The generated tarball is sufficient for a casual source distribution, but probably not for a release (see `dist-check`).

dist-check

This target does the same thing as the `dist` target but also checks the distribution tarball. The check is made by unpacking the tarball to a new directory, configuring it, building it, installing it, and then verifying that the installation results are correct (by comparing to the original build). This target can take a long time to run but should be done before a release goes out to make sure that the distributed tarball can actually be built into a working release.

dist-clean

This is a special form of the `clean` clean target. It performs a normal `clean` but also removes things pertaining to building the distribution.

install

This target finalizes shared objects and executables and copies all libraries, headers, executables and documentation to the directory given with the `--prefix` option to `configure`. When completed, the prefix directory will have everything needed to **use** LLVM.

The LLVM makefiles can generate complete **internal** documentation for all the classes by using `doxygen`. By default, this feature is **not** enabled because it takes a long time and generates a massive amount of data (>100MB). If you want this feature, you must configure LLVM with the –enable-doxygen switch and ensure that a modern version of doxygen (1.3.7 or later) is available in your `PATH`. You can download doxygen from here.

preconditions

This utility target checks to see if the `Makefile` in the object directory is older than the `Makefile` in the source directory and copies it if so. It also reruns the `configure` script if that needs to be done and rebuilds the `Makefile.config` file similarly. Users may overload this target to ensure that sanity checks are run *before* any building of targets as all the targets depend on `preconditions`.

printvars

This utility target just causes the LLVM makefiles to print out some of the makefile variables so that you can double check how things are set.

reconfigure

This utility target will force a reconfigure of LLVM or your project. It simply runs `$(PROJ_OBJ_ROOT)/config.status --recheck` to rerun the configuration tests and rebuild the configured files. This isn't generally useful as the makefiles will reconfigure themselves whenever its necessary.

spotless

Warning: Use with caution!

This utility target, only available when `$(PROJ_OBJ_ROOT)` is not the same as `$(PROJ_SRC_ROOT)`, will completely clean the `$(PROJ_OBJ_ROOT)` directory by removing its content entirely and reconfiguring the directory. This returns the `$(PROJ_OBJ_ROOT)` directory to a completely fresh state. All content in the directory except configured files and top-level makefiles will be lost.

tags

This target will generate a `TAGS` file in the top-level source directory. It is meant for use with emacs, XEmacs, or ViM. The TAGS file provides an index of symbol definitions so that the editor can jump you to the definition quickly.

uninstall

This target is the opposite of the `install` target. It removes the header, library and executable files from the installation directories. Note that the directories themselves are not removed because it is not guaranteed that LLVM is the only thing installing there (e.g. `--prefix=/usr`).

5.2.5 Variables

Variables are used to tell the LLVM Makefile System what to do and to obtain information from it. Variables are also used internally by the LLVM Makefile System. Variable names that contain only the upper case alphabetic letters and underscore are intended for use by the end user. All other variables are internal to the LLVM Makefile System and should not be relied upon nor modified. The sections below describe how to use the LLVM Makefile variables.

Control Variables

Variables listed in the table below should be set *before* the inclusion of $(LEVEL)/Makefile.common. These variables provide input to the LLVM make system that tell it what to do for the current directory.

BUILD_ARCHIVE If set to any value, causes an archive (.a) library to be built.

BUILT_SOURCES Specifies a set of source files that are generated from other source files. These sources will be built before any other target processing to ensure they are present.

CONFIG_FILES Specifies a set of configuration files to be installed.

DEBUG_SYMBOLS If set to any value, causes the build to include debugging symbols even in optimized objects, libraries and executables. This alters the flags specified to the compilers and linkers. Debugging isn't fun in an optimized build, but it is possible.

DIRS Specifies a set of directories, usually children of the current directory, that should also be made using the same goal. These directories will be built serially.

DISABLE_AUTO_DEPENDENCIES If set to any value, causes the makefiles to **not** automatically generate dependencies when running the compiler. Use of this feature is discouraged and it may be removed at a later date.

ENABLE_OPTIMIZED If set to 1, causes the build to generate optimized objects, libraries and executables. This alters the flags specified to the compilers and linkers. Generally debugging won't be a fun experience with an optimized build.

ENABLE_PROFILING If set to 1, causes the build to generate both optimized and profiled objects, libraries and executables. This alters the flags specified to the compilers and linkers to ensure that profile data can be collected from the tools built. Use the `gprof` tool to analyze the output from the profiled tools (`gmon.out`).

DISABLE_ASSERTIONS If set to 1, causes the build to disable assertions, even if building a debug or profile build. This will exclude all assertion check code from the build. LLVM will execute faster, but with little help when things go wrong.

EXPERIMENTAL_DIRS Specify a set of directories that should be built, but if they fail, it should not cause the build to fail. Note that this should only be used temporarily while code is being written.

EXPORTED_SYMBOL_FILE Specifies the name of a single file that contains a list of the symbols to be exported by the linker. One symbol per line.

EXPORTED_SYMBOL_LIST Specifies a set of symbols to be exported by the linker.

EXTRA_DIST Specifies additional files that should be distributed with LLVM. All source files, all built sources, all Makefiles, and most documentation files will be automatically distributed. Use this variable to distribute any files that are not automatically distributed.

KEEP_SYMBOLS If set to any value, specifies that when linking executables the makefiles should retain debug symbols in the executable. Normally, symbols are stripped from the executable.

LEVEL (required) Specify the level of nesting from the top level. This variable must be set in each makefile as it is used to find the top level and thus the other makefiles.

LIBRARYNAME Specify the name of the library to be built. (Required For Libraries)

LINK_COMPONENTS When specified for building a tool, the value of this variable will be passed to the `llvm-config` tool to generate a link line for the tool. Unlike `USEDLIBS` and `LLVMLIBS`, not all libraries need to be specified. The `llvm-config` tool will figure out the library dependencies and add any libraries that are needed. The `USEDLIBS` variable can still be used in conjunction with `LINK_COMPONENTS` so that additional project-specific libraries can be linked with the LLVM libraries specified by `LINK_COMPONENTS`.

LINK_LIBS_IN_SHARED By default, shared library linking will ignore any libraries specified with the LLVMLIBS or USEDLIBS. This prevents shared libs from including things that will be in the LLVM tool the shared library

will be loaded into. However, sometimes it is useful to link certain libraries into your shared library and this option enables that feature.

LLVMLIBS Specifies the set of libraries from the LLVM `$(ObjDir)` that will be linked into the tool or library.

LOADABLE_MODULE If set to any value, causes the shared library being built to also be a loadable module. Loadable modules can be opened with the dlopen() function and searched with dlsym (or the operating system's equivalent). Note that setting this variable without also setting `SHARED_LIBRARY` will have no effect.

NO_INSTALL Specifies that the build products of the directory should not be installed but should be built even if the `install` target is given. This is handy for directories that build libraries or tools that are only used as part of the build process, such as code generators (e.g. `tblgen`).

OPTIONAL_DIRS Specify a set of directories that may be built, if they exist, but it is not an error for them not to exist.

PARALLEL_DIRS Specify a set of directories to build recursively and in parallel if the `-j` option was used with `make`.

SHARED_LIBRARY If set to any value, causes a shared library (`.so`) to be built in addition to any other kinds of libraries. Note that this option will cause all source files to be built twice: once with options for position independent code and once without. Use it only where you really need a shared library.

SOURCES (optional) Specifies the list of source files in the current directory to be built. Source files of any type may be specified (programs, documentation, config files, etc.). If not specified, the makefile system will infer the set of source files from the files present in the current directory.

SUFFIXES Specifies a set of filename suffixes that occur in suffix match rules. Only set this if your local `Makefile` specifies additional suffix match rules.

TARGET Specifies the name of the LLVM code generation target that the current directory builds. Setting this variable enables additional rules to build `.inc` files from `.td` files.

TESTSUITE Specifies the directory of tests to run in `llvm/test`.

TOOLNAME Specifies the name of the tool that the current directory should build.

TOOL_VERBOSE Implies `VERBOSE` and also tells each tool invoked to be verbose. This is handy when you're trying to see the sub-tools invoked by each tool invoked by the makefile. For example, this will pass `-v` to the GCC compilers which causes it to print out the command lines it uses to invoke sub-tools (compiler, assembler, linker).

USEDLIBS Specifies the list of project libraries that will be linked into the tool or library.

VERBOSE Tells the Makefile system to produce detailed output of what it is doing instead of just summary comments. This will generate a LOT of output.

Override Variables

Override variables can be used to override the default values provided by the LLVM makefile system. These variables can be set in several ways:

- In the environment (e.g. setenv, export) — not recommended.
- On the `make` command line — recommended.
- On the `configure` command line.
- In the Makefile (only *after* the inclusion of $(LEVEL)/Makefile.common).

The override variables are given below:

AR (defaulted) Specifies the path to the `ar` tool.

PROJ_OBJ_DIR The directory into which the products of build rules will be placed. This might be the same as PROJ_SRC_DIR but typically is not.

PROJ_SRC_DIR The directory which contains the source files to be built.

BUILD_EXAMPLES If set to 1, build examples in `examples` and (if building Clang) `tools/clang/examples` directories.

BZIP2 (configured) The path to the `bzip2` tool.

CC (configured) The path to the 'C' compiler.

CFLAGS Additional flags to be passed to the 'C' compiler.

CPPFLAGS Additional flags passed to the C/C++ preprocessor.

CXX Specifies the path to the C++ compiler.

CXXFLAGS Additional flags to be passed to the C++ compiler.

DATE (configured) Specifies the path to the `date` program or any program that can generate the current date and time on its standard output.

DOT (configured) Specifies the path to the `dot` tool or `false` if there isn't one.

ECHO (configured) Specifies the path to the `echo` tool for printing output.

EXEEXT (configured) Provides the extension to be used on executables built by the makefiles. The value may be empty on platforms that do not use file extensions for executables (e.g. Unix).

INSTALL (configured) Specifies the path to the `install` tool.

LDFLAGS (configured) Allows users to specify additional flags to pass to the linker.

LIBS (configured) The list of libraries that should be linked with each tool.

LIBTOOL (configured) Specifies the path to the `libtool` tool. This tool is renamed `mklib` by the `configure` script.

LLVMAS (defaulted) Specifies the path to the `llvm-as` tool.

LLVMGCC (defaulted) Specifies the path to the LLVM version of the GCC 'C' Compiler.

LLVMGXX (defaulted) Specifies the path to the LLVM version of the GCC C++ Compiler.

LLVMLD (defaulted) Specifies the path to the LLVM bitcode linker tool

LLVM_OBJ_ROOT (configured) Specifies the top directory into which the output of the build is placed.

LLVM_SRC_ROOT (configured) Specifies the top directory in which the sources are found.

LLVM_TARBALL_NAME (configured) Specifies the name of the distribution tarball to create. This is configured from the name of the project and its version number.

MKDIR (defaulted) Specifies the path to the `mkdir` tool that creates directories.

ONLY_TOOLS If set, specifies the list of tools to build.

PLATFORMSTRIPOPTS The options to provide to the linker to specify that a stripped (no symbols) executable should be built.

RANLIB (defaulted) Specifies the path to the `ranlib` tool.

RM (defaulted) Specifies the path to the `rm` tool.

SED (defaulted) Specifies the path to the `sed` tool.

SHLIBEXT (configured) Provides the filename extension to use for shared libraries.

TBLGEN (defaulted) Specifies the path to the `tblgen` tool.

TAR (defaulted) Specifies the path to the `tar` tool.

ZIP (defaulted) Specifies the path to the `zip` tool.

Readable Variables

Variables listed in the table below can be used by the user's Makefile but should not be changed. Changing the value will generally cause the build to go wrong, so don't do it.

bindir The directory into which executables will ultimately be installed. This value is derived from the `--prefix` option given to `configure`.

BuildMode The name of the type of build being performed: Debug, Release, or Profile.

bytecode_libdir The directory into which bitcode libraries will ultimately be installed. This value is derived from the `--prefix` option given to `configure`.

ConfigureScriptFLAGS Additional flags given to the `configure` script when reconfiguring.

DistDir The *current* directory for which a distribution copy is being made.

Echo The LLVM Makefile System output command. This provides the `llvm[n]` prefix and starts with `@` so the command itself is not printed by `make`.

EchoCmd Same as Echo but without the leading `@`.

includedir The directory into which include files will ultimately be installed. This value is derived from the `--prefix` option given to `configure`.

libdir The directory into which native libraries will ultimately be installed. This value is derived from the `--prefix` option given to `configure`.

LibDir The configuration specific directory into which libraries are placed before installation.

MakefileConfig Full path of the `Makefile.config` file.

MakefileConfigIn Full path of the `Makefile.config.in` file.

ObjDir The configuration and directory specific directory where build objects (compilation results) are placed.

SubDirs The complete list of sub-directories of the current directory as specified by other variables.

Sources The complete list of source files.

sysconfdir The directory into which configuration files will ultimately be installed. This value is derived from the `--prefix` option given to `configure`.

ToolDir The configuration specific directory into which executables are placed before they are installed.

TopDistDir The top most directory into which the distribution files are copied.

Verb Use this as the first thing on your build script lines to enable or disable verbose mode. It expands to either an `@` (quiet mode) or nothing (verbose mode).

Internal Variables

Variables listed below are used by the LLVM Makefile System and considered internal. You should not use these variables under any circumstances.

```
Archive
AR.Flags
BaseNameSources
BCLinkLib
C.Flags
Compile.C
CompileCommonOpts
Compile.CXX
ConfigStatusScript
ConfigureScript
CPP.Flags
CPP.Flags
CXX.Flags
DependFiles
DestArchiveLib
DestBitcodeLib
DestModule
DestSharedLib
DestTool
DistAlways
DistCheckDir
DistCheckTop
DistFiles
DistName
DistOther
DistSources
DistSubDirs
DistTarBZ2
DistTarGZip
DistZip
ExtraLibs
FakeSources
INCFiles
InternalTargets
LD.Flags
LibName.A
LibName.BC
LibName.LA
LibName.O
LibTool.Flags
Link
LinkModule
LLVMLibDir
LLVMLibsOptions
LLVMLibsPaths
LLVMToolDir
LLVMUsedLibs
LocalTargets
Module
ObjectsLO
ObjectsO
ObjMakefiles
ParallelTargets
PreConditions
ProjLibsOptions
ProjLibsPaths
ProjUsedLibs
Ranlib
```

```
RecursiveTargets
SrcMakefiles
Strip
StripWarnMsg
TableGen
TDFiles
ToolBuildPath
TopLevelTargets
UserTargets
```

5.3 Creating an LLVM Project

5.3.1 Overview

The LLVM build system is designed to facilitate the building of third party projects that use LLVM header files, libraries, and tools. In order to use these facilities, a `Makefile` from a project must do the following things:

- Set `make` variables. There are several variables that a `Makefile` needs to set to use the LLVM build system:
  - `PROJECT_NAME` - The name by which your project is known.
  - `LLVM_SRC_ROOT` - The root of the LLVM source tree.
  - `LLVM_OBJ_ROOT` - The root of the LLVM object tree.
  - `PROJ_SRC_ROOT` - The root of the project's source tree.
  - `PROJ_OBJ_ROOT` - The root of the project's object tree.
  - `PROJ_INSTALL_ROOT` - The root installation directory.
  - `LEVEL` - The relative path from the current directory to the project's root `($PROJ_OBJ_ROOT)`.
- Include `Makefile.config` from `$(LLVM_OBJ_ROOT)`.
- Include `Makefile.rules` from `$(LLVM_SRC_ROOT)`.

There are two ways that you can set all of these variables:

- You can write your own `Makefiles` which hard-code these values.
- You can use the pre-made LLVM sample project. This sample project includes `Makefiles`, a configure script that can be used to configure the location of LLVM, and the ability to support multiple object directories from a single source directory.

If you want to devise your own build system, studying other projects and LLVM `Makefiles` will probably provide enough information on how to write your own `Makefiles`.

5.3.2 Source Tree Layout

In order to use the LLVM build system, you will want to organize your source code so that it can benefit from the build system's features. Mainly, you want your source tree layout to look similar to the LLVM source tree layout.

Underneath your top level directory, you should have the following directories:

lib

This subdirectory should contain all of your library source code. For each library that you build, you will have one directory in **lib** that will contain that library's source code.

Libraries can be object files, archives, or dynamic libraries. The **lib** directory is just a convenient place for libraries as it places them all in a directory from which they can be linked later.

include

This subdirectory should contain any header files that are global to your project. By global, we mean that they are used by more than one library or executable of your project.

By placing your header files in **include**, they will be found automatically by the LLVM build system. For example, if you have a file **include/jazz/note.h**, then your source files can include it simply with **#include "jazz/note.h"**.

tools

This subdirectory should contain all of your source code for executables. For each program that you build, you will have one directory in **tools** that will contain that program's source code.

test

This subdirectory should contain tests that verify that your code works correctly. Automated tests are especially useful.

Currently, the LLVM build system provides basic support for tests. The LLVM system provides the following:

- LLVM contains regression tests in `llvm/test`. These tests are run by the *Lit* testing tool. This test procedure uses `RUN` lines in the actual test case to determine how to run the test. See the *LLVM Testing Infrastructure Guide* for more details.
- LLVM contains an optional package called `llvm-test`, which provides benchmarks and programs that are known to compile with the Clang front end. You can use these programs to test your code, gather statistical information, and compare it to the current LLVM performance statistics.

  Currently, there is no way to hook your tests directly into the `llvm/test` testing harness. You will simply need to find a way to use the source provided within that directory on your own.

Typically, you will want to build your **lib** directory first followed by your **tools** directory.

5.3.3 Writing LLVM Style Makefiles

The LLVM build system provides a convenient way to build libraries and executables. Most of your project Makefiles will only need to define a few variables. Below is a list of the variables one can set and what they can do:

Required Variables

`LEVEL`

This variable is the relative path from this `Makefile` to the top directory of your project's source code. For example, if your source code is in `/tmp/src`, then the `Makefile` in `/tmp/src/jump/high` would set `LEVEL` to `"../.."`.

Variables for Building Subdirectories

`DIRS`

This is a space separated list of subdirectories that should be built. They will be built, one at a time, in the order specified.

`PARALLEL_DIRS`

This is a list of directories that can be built in parallel. These will be built after the directories in DIRS have been built.

`OPTIONAL_DIRS`

This is a list of directories that can be built if they exist, but will not cause an error if they do not exist. They are built serially in the order in which they are listed.

Variables for Building Libraries

`LIBRARYNAME`

This variable contains the base name of the library that will be built. For example, to build a library named `libsample.a`, `LIBRARYNAME` should be set to `sample`.

`BUILD_ARCHIVE`

By default, a library is a `.o` file that is linked directly into a program. To build an archive (also known as a static library), set the `BUILD_ARCHIVE` variable.

`SHARED_LIBRARY`

If `SHARED_LIBRARY` is defined in your Makefile, a shared (or dynamic) library will be built.

Variables for Building Programs

`TOOLNAME`

This variable contains the name of the program that will be built. For example, to build an executable named `sample`, `TOOLNAME` should be set to `sample`.

`USEDLIBS`

This variable holds a space separated list of libraries that should be linked into the program. These libraries must be libraries that come from your **lib** directory. The libraries must be specified without their `lib` prefix. For example, to link `libsample.a`, you would set `USEDLIBS` to `sample.a`.

Note that this works only for statically linked libraries.

`LLVMLIBS`

This variable holds a space separated list of libraries that should be linked into the program. These libraries must be LLVM libraries. The libraries must be specified without their `lib` prefix. For example, to link with a driver that performs an IR transformation you might set `LLVMLIBS` to this minimal set of libraries `LLVMSupport.a LLVMCore.a LLVMBitReader.a LLVMAsmParser.a LLVMAnalysis.a LLVMTransformUtils.a LLVMScalarOpts.a LLVMTarget.a`.

Note that this works only for statically linked libraries. LLVM is split into a large number of static libraries, and the list of libraries you require may be much longer than the list above. To see a full list of libraries use: `llvm-config --libs all`. Using `LINK_COMPONENTS` as described below, obviates the need to set `LLVMLIBS`.

`LINK_COMPONENTS`

This variable holds a space separated list of components that the LLVM `Makefiles` pass to the `llvm-config` tool to generate a link line for the program. For example, to link with all LLVM libraries use `LINK_COMPONENTS = all`.

`LIBS`

To link dynamic libraries, add `-l<library base name>` to the `LIBS` variable. The LLVM build system will look in the same places for dynamic libraries as it does for static libraries.

For example, to link `libsample.so`, you would have the following line in your `Makefile`:

```
LIBS += -lsample
```

Note that `LIBS` must occur in the Makefile after the inclusion of `Makefile.common`.

Miscellaneous Variables

`CFLAGS` & `CPPFLAGS`

This variable can be used to add options to the C and C++ compiler, respectively. It is typically used to add options that tell the compiler the location of additional directories to search for header files.

It is highly suggested that you append to `CFLAGS` and `CPPFLAGS` as opposed to overwriting them. The master `Makefiles` may already have useful options in them that you may not want to overwrite.

5.3.4 Placement of Object Code

The final location of built libraries and executables will depend upon whether you do a `Debug`, `Release`, or `Profile` build.

Libraries

All libraries (static and dynamic) will be stored in `PROJ_OBJ_ROOT/<type>/lib`, where *type* is `Debug`, `Release`, or `Profile` for a debug, optimized, or profiled build, respectively.

Executables

All executables will be stored in `PROJ_OBJ_ROOT/<type>/bin`, where *type* is `Debug`, `Release`, or `Profile` for a debug, optimized, or profiled build, respectively.

5.3.5 Further Help

If you have any questions or need any help creating an LLVM project, the LLVM team would be more than happy to help. You can always post your questions to the LLVM Developers Mailing List.

5.4 LLVMBuild Guide

5.4.1 Introduction

This document describes the `LLVMBuild` organization and files which we use to describe parts of the LLVM ecosystem. For description of specific LLVMBuild related tools, please see the command guide.

LLVM is designed to be a modular set of libraries which can be flexibly mixed together in order to build a variety of tools, like compilers, JITs, custom code generators, optimization passes, interpreters, and so on. Related projects in the LLVM system like Clang and LLDB also tend to follow this philosophy.

In order to support this usage style, LLVM has a fairly strict structure as to how the source code and various components are organized. The `LLVMBuild.txt` files are the explicit specification of that structure, and are used by the build systems and other tools in order to develop the LLVM project.

5.4.2 Project Organization

The source code for LLVM projects using the LLVMBuild system (LLVM, Clang, and LLDB) is organized into *components*, which define the separate pieces of functionality that make up the project. These projects may consist of many libraries, associated tools, build tools, or other utility tools (for example, testing tools).

For the most part, the project contents are organized around defining one main component per each subdirectory. Each such directory contains an `LLVMBuild.txt` which contains the component definitions.

The component descriptions for the project as a whole are automatically gathered by the LLVMBuild tools. The tools automatically traverse the source directory structure to find all of the component description files. NOTE: For performance/sanity reasons, we only traverse into subdirectories when the parent itself contains an `LLVMBuild.txt` description file.

5.4.3 Build Integration

The LLVMBuild files themselves are just a declarative way to describe the project structure. The actual building of the LLVM project is handled by another build system (currently we support both *Makefiles* and *CMake*).

The build system implementation will load the relevant contents of the LLVMBuild files and use that to drive the actual project build. Typically, the build system will only need to load this information at "configure" time, and use it to generative native information. Build systems will also handle automatically reconfiguring their information when the contents of the `LLVMBuild.txt` files change.

Developers generally are not expected to need to be aware of the details of how the LLVMBuild system is integrated into their build. Ideally, LLVM developers who are not working on the build system would only ever need to modify the contents of the `LLVMBuild.txt` description files (although we have not reached this goal yet).

For more information on the utility tool we provide to help interfacing with the build system, please see the *llvm-build* documentation.

5.4.4 Component Overview

As mentioned earlier, LLVM projects are organized into logical *components*. Every component is typically grouped into its own subdirectory. Generally, a component is organized around a coherent group of sources which have some kind of clear API separation from other parts of the code.

LLVM primarily uses the following types of components:

- *Libraries* - Library components define a distinct API which can be independently linked into LLVM client applications. Libraries typically have private and public header files, and may specify a link of required libraries that they build on top of.
- *Build Tools* - Build tools are applications which are designed to be run as part of the build process (typically to generate other source files). Currently, LLVM uses one main build tool called *TableGen* to generate a variety of source files.
- *Tools* - Command line applications which are built using the LLVM component libraries. Most LLVM tools are small and are primarily frontends to the library interfaces.

Components are described using `LLVMBuild.txt` files in the directories that define the component. See the LLVMBuild Format Reference section for information on the exact format of these files.

5.4.5 LLVMBuild Format Reference

LLVMBuild files are written in a simple variant of the INI or configuration file format (Wikipedia entry). The format defines a list of sections each of which may contain some number of properties. A simple example of the file format is below:

```
; Comments start with a semi-colon.

; Sections are declared using square brackets.
[component_0]

; Properties are declared using '=' and are contained in the previous section.
;
; We support simple string and boolean scalar values and list values, where
; items are separated by spaces. There is no support for quoting, and so
; property values may not contain spaces.
property_name = property_value
list_property_name = value_1 value_2 ... value_n
boolean_property_name = 1 (or 0)
```

LLVMBuild files are expected to define a strict set of sections and properties. A typical component description file for a library component would look like the following example:

```
[component_0]
type = Library
name = Linker
parent = Libraries
required_libraries = Archive BitReader Core Support TransformUtils
```

A full description of the exact sections and properties which are allowed follows.

Each file may define exactly one common component, named `common`. The common component may define the following properties:

- `subdirectories` **[optional]**

  If given, a list of the names of the subdirectories from the current subpath to search for additional LLVMBuild files.

Each file may define multiple components. Each component is described by a section who name starts with `component`. The remainder of the section name is ignored, but each section name must be unique. Typically components are just number in order for files with multiple components (`component_0`, `component_1`, and so on).

Warning: Section names not matching this format (or the `common` section) are currently unused and are disallowed.

Every component is defined by the properties in the section. The exact list of properties that are allowed depends on the component type. Components **may not** define any properties other than those expected by the component type.

Every component must define the following properties:

- `type` **[required]**

  The type of the component. Supported component types are detailed below. Most components will define additional properties which may be required or optional.

- `name` **[required]**

  The name of the component. Names are required to be unique across the entire project.

- `parent` **[required]**

  The name of the logical parent of the component. Components are organized into a logical tree to make it easier to navigate and organize groups of components. The parents have no semantics as far as the project build is concerned, however. Typically, the parent will be the main component of the parent directory.

  Components may reference the root pseudo component using `$ROOT` to indicate they should logically be grouped at the top-level.

Components may define the following properties:

- `dependencies` **[optional]**

  If specified, a list of names of components which *must* be built prior to this one. This should only be exactly those components which produce some tool or source code required for building the component.

  Note: `Group` and `LibraryGroup` components have no semantics for the actual build, and are not allowed to specify dependencies.

The following section lists the available component types, as well as the properties which are associated with that component.

- `type = Group`

  Group components exist purely to allow additional arbitrary structuring of the logical components tree. For example, one might define a `Libraries` group to hold all of the root library components.

  `Group` components have no additionally properties.

- `type = Library`

  Library components define an individual library which should be built from the source code in the component directory.

  Components with this type use the following properties:

  – `library_name` **[optional]**

    If given, the name to use for the actual library file on disk. If not given, the name is derived from the component name itself.

  - `required_libraries` **[optional]**

    If given, a list of the names of `Library` or `LibraryGroup` components which must also be linked in whenever this library is used. That is, the link time dependencies for this component. When tools are built, the build system will include the transitive closure of all `required_libraries` for the components the tool needs.

  - `add_to_library_groups` **[optional]**

    If given, a list of the names of `LibraryGroup` components which this component is also part of. This allows nesting groups of components. For example, the `X86` target might define a library group for all of the `X86` components. That library group might then be included in the `all-targets` library group.

  - `installed` **[optional] [boolean]**

    Whether this library is installed. Libraries that are not installed are only reported by `llvm-config` when it is run as part of a development directory.

- `type = LibraryGroup`

  `LibraryGroup` components are a mechanism to allow easy definition of useful sets of related components. In particular, we use them to easily specify things like "all targets", or "all assembly printers".

  Components with this type use the following properties:

  - `required_libraries` **[optional]**

    See the `Library` type for a description of this property.

  - `add_to_library_groups` **[optional]**

    See the `Library` type for a description of this property.

- `type = TargetGroup`

  `TargetGroup` components are an extension of `LibraryGroups`, specifically for defining LLVM targets (which are handled specially in a few places).

  The name of the component should always be the name of the target.

  Components with this type use the `LibraryGroup` properties in addition to:

  - `has_asmparser` **[optional] [boolean]**

    Whether this target defines an assembly parser.

  - `has_asmprinter` **[optional] [boolean]**

    Whether this target defines an assembly printer.

  - `has_disassembler` **[optional] [boolean]**

    Whether this target defines a disassembler.

  - `has_jit` **[optional] [boolean]**

    Whether this target supports JIT compilation.

- `type = Tool`

  `Tool` components define standalone command line tools which should be built from the source code in the component directory and linked.

  Components with this type use the following properties:

- `required_libraries` **[optional]**

  If given, a list of the names of `Library` or `LibraryGroup` components which this tool is required to be linked with.

  Note: The values should be the component names, which may not always match up with the actual library names on disk.

  Build systems are expected to properly include all of the libraries required by the linked components (i.e., the transitive closure of `required_libraries`).

  Build systems are also expected to understand that those library components must be built prior to linking – they do not also need to be listed under `dependencies`.

- `type = BuildTool`

  `BuildTool` components are like `Tool` components, except that the tool is supposed to be built for the platform where the build is running (instead of that platform being targeted). Build systems are expected to handle the fact that required libraries may need to be built for multiple platforms in order to be able to link this tool.

  `BuildTool` components currently use the exact same properties as `Tool` components, the type distinction is only used to differentiate what the tool is built for.

5.5 How To Release LLVM To The Public

5.5.1 Introduction

This document contains information about successfully releasing LLVM — including subprojects: e.g., `clang` and `dragonegg` — to the public. It is the Release Manager's responsibility to ensure that a high quality build of LLVM is released.

If you're looking for the document on how to test the release candidates and create the binary packages, please refer to the *How To Validate a New Release* instead.

5.5.2 Release Timeline

LLVM is released on a time based schedule — with major releases roughly every 6 months. In between major releases there may be dot releases. The release manager will determine if and when to make a dot release based on feedback from the community. Typically, dot releases should be made if there are large number of bug-fixes in the stable branch or a critical bug has been discovered that affects a large number of users.

Unless otherwise stated, dot releases will follow the same procedure as major releases.

The release process is roughly as follows:

- Set code freeze and branch creation date for 6 months after last code freeze date. Announce release schedule to the LLVM community and update the website.
- Create release branch and begin release process.

- Send out release candidate sources for first round of testing. Testing lasts 7-10 days. During the first round of testing, any regressions found should be fixed. Patches are merged from mainline into the release branch. Also, all features need to be completed during this time. Any features not completed at the end of the first round of testing will be removed or disabled for the release.
- Generate and send out the second release candidate sources. Only *critial* bugs found during this testing phase will be fixed. Any bugs introduced by merged patches will be fixed. If so a third round of testing is needed.
- The release notes are updated.
- Finally, release!

The release process will be accelerated for dot releases. If the first round of testing finds no critical bugs and no regressions since the last major release, then additional rounds of testing will not be required.

5.5.3 Release Process

Release Administrative Tasks

This section describes a few administrative tasks that need to be done for the release process to begin. Specifically, it involves:

- Creating the release branch,
- Setting version numbers, and
- Tagging release candidates for the release team to begin testing.

Create Release Branch

Branch the Subversion trunk using the following procedure:

1. Remind developers that the release branching is imminent and to refrain from committing patches that might break the build. E.g., new features, large patches for works in progress, an overhaul of the type system, an exciting new TableGen feature, etc.
2. Verify that the current Subversion trunk is in decent shape by examining nightly tester and buildbot results.
3. Create the release branch for `llvm`, `clang`, the `test-suite`, and `dragonegg` from the last known good revision. The branch's name is `release_XY`, where `X` is the major and `Y` the minor release numbers. The branches should be created using the following commands:

```
$ svn copy https://llvm.org/svn/llvm-project/llvm/trunk \
           https://llvm.org/svn/llvm-project/llvm/branches/release_XY

$ svn copy https://llvm.org/svn/llvm-project/cfe/trunk \
           https://llvm.org/svn/llvm-project/cfe/branches/release_XY

$ svn copy https://llvm.org/svn/llvm-project/dragonegg/trunk \
           https://llvm.org/svn/llvm-project/dragonegg/branches/release_XY

$ svn copy https://llvm.org/svn/llvm-project/test-suite/trunk \
           https://llvm.org/svn/llvm-project/test-suite/branches/release_XY
```

4. Advise developers that they may now check their patches into the Subversion tree again.
5. The Release Manager should switch to the release branch, because all changes to the release will now be done in the branch. The easiest way to do this is to grab a working copy using the following commands:

```
$ svn co https://llvm.org/svn/llvm-project/llvm/branches/release_XY llvm-X.Y

$ svn co https://llvm.org/svn/llvm-project/cfe/branches/release_XY clang-X.Y

$ svn co https://llvm.org/svn/llvm-project/dragonegg/branches/release_XY dragonegg-X.Y

$ svn co https://llvm.org/svn/llvm-project/test-suite/branches/release_XY test-suite-X.Y
```

Update LLVM Version

After creating the LLVM release branch, update the release branches' `autoconf` and `configure.ac` versions from '`X.Ysvn`' to '`X.Y`'. Update it on mainline as well to be the next version ('`X.Y+1svn`'). Regenerate the configure scripts for both `llvm` and the `test-suite`.

In addition, the version numbers of all the Bugzilla components must be updated for the next release.

Build the LLVM Release Candidates

Create release candidates for `llvm`, `clang`, `dragonegg`, and the LLVM `test-suite` by tagging the branch with the respective release candidate number. For instance, to create **Release Candidate 1** you would issue the following commands:

```
$ svn mkdir https://llvm.org/svn/llvm-project/llvm/tags/RELEASE_XYZ
$ svn copy https://llvm.org/svn/llvm-project/llvm/branches/release_XY \
           https://llvm.org/svn/llvm-project/llvm/tags/RELEASE_XYZ/rc1

$ svn mkdir https://llvm.org/svn/llvm-project/cfe/tags/RELEASE_XYZ
$ svn copy https://llvm.org/svn/llvm-project/cfe/branches/release_XY \
           https://llvm.org/svn/llvm-project/cfe/tags/RELEASE_XYZ/rc1
```

```
$ svn mkdir https://llvm.org/svn/llvm-project/dragonegg/tags/RELEASE_XYZ
$ svn copy https://llvm.org/svn/llvm-project/dragonegg/branches/release_XY \
           https://llvm.org/svn/llvm-project/dragonegg/tags/RELEASE_XYZ/rc1

$ svn mkdir https://llvm.org/svn/llvm-project/test-suite/tags/RELEASE_XYZ
$ svn copy https://llvm.org/svn/llvm-project/test-suite/branches/release_XY \
           https://llvm.org/svn/llvm-project/test-suite/tags/RELEASE_XYZ/rc1
```

Similarly, **Release Candidate 2** would be named `RC2` and so on. This keeps a permanent copy of the release candidate around for people to export and build as they wish. The final released sources will be tagged in the `RELEASE_XYZ` directory as `Final` (c.f. *Tag the LLVM Final Release*).

The Release Manager may supply pre-packaged source tarballs for users. This can be done with the following commands:

```
$ svn export https://llvm.org/svn/llvm-project/llvm/tags/RELEASE_XYZ/rc1 llvm-X.Yrc1
$ svn export https://llvm.org/svn/llvm-project/cfe/tags/RELEASE_XYZ/rc1 clang-X.Yrc1
$ svn export https://llvm.org/svn/llvm-project/dragonegg/tags/RELEASE_XYZ/rc1 dragonegg-X.Yrc1
$ svn export https://llvm.org/svn/llvm-project/test-suite/tags/RELEASE_XYZ/rc1 llvm-test-X.Yrc1

$ tar -cvf - llvm-X.Yrc1        | gzip > llvm-X.Yrc1.src.tar.gz
$ tar -cvf - clang-X.Yrc1       | gzip > clang-X.Yrc1.src.tar.gz
$ tar -cvf - dragonegg-X.Yrc1   | gzip > dragonegg-X.Yrc1.src.tar.gz
$ tar -cvf - llvm-test-X.Yrc1   | gzip > llvm-test-X.Yrc1.src.tar.gz
```

Building the Release

The builds of `llvm`, `clang`, and `dragonegg` *must* be free of errors and warnings in Debug, Release+Asserts, and Release builds. If all builds are clean, then the release passes Build Qualification.

The `make` options for building the different modes:

Mode	Options
Debug	`ENABLE_OPTIMIZED=0`
Release+Asserts	`ENABLE_OPTIMIZED=1`
Release	`ENABLE_OPTIMIZED=1 DISABLE_ASSERTIONS=1`

Build LLVM

Build `Debug`, `Release+Asserts`, and `Release` versions of `llvm` on all supported platforms. Directions to build `llvm` are *here*.

Build Clang Binary Distribution

Creating the `clang` binary distribution (Debug/Release+Asserts/Release) requires performing the following steps for each supported platform:

1. Build clang according to the directions here.
2. Build both a Debug and Release version of clang. The binary will be the Release build.
3. Package `clang` (details to follow).

Target Specific Build Details

The table below specifies which compilers are used for each Arch/OS combination when qualifying the build of `llvm`, `clang`, and `dragonegg`.

Architecture	OS	compiler
x86-32	Mac OS 10.5	gcc 4.0.1
x86-32	Linux	gcc 4.2.X, gcc 4.3.X
x86-32	FreeBSD	gcc 4.2.X
x86-32	mingw	gcc 3.4.5
x86-64	Mac OS 10.5	gcc 4.0.1
x86-64	Linux	gcc 4.2.X, gcc 4.3.X
x86-64	FreeBSD	gcc 4.2.X
ARMv7	Linux	gcc 4.6.X, gcc 4.7.X

Release Qualification Criteria

A release is qualified when it has no regressions from the previous release (or baseline). Regressions are related to correctness first and performance second. (We may tolerate some minor performance regressions if they are deemed necessary for the general quality of the compiler.)

Regressions are new failures in the set of tests that are used to qualify each product and only include things on the list. Every release will have some bugs in it. It is the reality of developing a complex piece of software. We need a very concrete and definitive release criteria that ensures we have monotonically improving quality on some metric. The metric we use is described below. This doesn't mean that we don't care about other criteria, but these are the criteria which we found to be most important and which must be satisfied before a release can go out.

Qualify LLVM

LLVM is qualified when it has a clean test run without a front-end. And it has no regressions when using either `clang` or `dragonegg` with the `test-suite` from the previous release.

Qualify Clang

`Clang` is qualified when front-end specific tests in the `llvm` regression test suite all pass, clang's own test suite passes cleanly, and there are no regressions in the `test-suite`.

Specific Target Qualification Details

Architecture	OS	clang baseline	tests
x86-32	Linux	last release	llvm regression tests, clang regression tests, test-suite (including spec)
x86-32	FreeBSD	last release	llvm regression tests, clang regression tests, test-suite
x86-32	mingw	none	QT
x86-64	Mac OS 10.X	last release	llvm regression tests, clang regression tests, test-suite (including spec)
x86-64	Linux	last release	llvm regression tests, clang regression tests, test-suite (including spec)
x86-64	FreeBSD	last release	llvm regression tests, clang regression tests, test-suite
ARMv7A	Linux	last release	llvm regression tests, clang regression tests, test-suite

Community Testing

Once all testing has been completed and appropriate bugs filed, the release candidate tarballs are put on the website and the LLVM community is notified. Ask that all LLVM developers test the release in 2 ways:

1. Download `llvm-X.Y`, `llvm-test-X.Y`, and the appropriate `clang` binary. Build LLVM. Run `make check` and the full LLVM test suite (`make TEST=nightly report`).
2. Download `llvm-X.Y`, `llvm-test-X.Y`, and the `clang` sources. Compile everything. Run `make check` and the full LLVM test suite (`make TEST=nightly report`).

Ask LLVM developers to submit the test suite report and `make check` results to the list. Verify that there are no regressions from the previous release. The results are not used to qualify a release, but to spot other potential problems. For unsupported targets, verify that `make check` is at least clean.

During the first round of testing, all regressions must be fixed before the second release candidate is tagged.

If this is the second round of testing, the testing is only to ensure that bug fixes previously merged in have not created new major problems. *This is not the time to solve additional and unrelated bugs!* If no patches are merged in, the release is determined to be ready and the release manager may move onto the next stage.

Release Patch Rules

Below are the rules regarding patching the release branch:

1. Patches applied to the release branch may only be applied by the release manager.
2. During the first round of testing, patches that fix regressions or that are small and relatively risk free (verified by the appropriate code owner) are applied to the branch. Code owners are asked to be very conservative in approving patches for the branch. We reserve the right to reject any patch that does not fix a regression as previously defined.
3. During the remaining rounds of testing, only patches that fix critical regressions may be applied.
4. For dot releases all patches must mantain both API and ABI compatibility with the previous major release. Only bugfixes will be accepted.

Release Final Tasks

The final stages of the release process involves tagging the "final" release branch, updating documentation that refers to the release, and updating the demo page.

Update Documentation

Review the documentation and ensure that it is up to date. The "Release Notes" must be updated to reflect new features, bug fixes, new known issues, and changes in the list of supported platforms. The "Getting Started Guide" should be updated to reflect the new release version number tag available from Subversion and changes in basic system requirements. Merge both changes from mainline into the release branch.

Tag the LLVM Final Release

Tag the final release sources using the following procedure:

```
$ svn copy https://llvm.org/svn/llvm-project/llvm/branches/release_XY \
           https://llvm.org/svn/llvm-project/llvm/tags/RELEASE_XYZ/Final

$ svn copy https://llvm.org/svn/llvm-project/cfe/branches/release_XY \
           https://llvm.org/svn/llvm-project/cfe/tags/RELEASE_XYZ/Final

$ svn copy https://llvm.org/svn/llvm-project/dragonegg/branches/release_XY \
           https://llvm.org/svn/llvm-project/dragonegg/tags/RELEASE_XYZ/Final

$ svn copy https://llvm.org/svn/llvm-project/test-suite/branches/release_XY \
           https://llvm.org/svn/llvm-project/test-suite/tags/RELEASE_XYZ/Final
```

Update the LLVM Demo Page

The LLVM demo page must be updated to use the new release. This consists of using the new `clang` binary and building LLVM.

Update the LLVM Website

The website must be updated before the release announcement is sent out. Here is what to do:

1. Check out the `www` module from Subversion.
2. Create a new subdirectory `X.Y` in the releases directory.
3. Commit the `llvm`, `test-suite`, `clang` source, `clang binaries`, `dragonegg` source, and `dragonegg` binaries in this new directory.
4. Copy and commit the `llvm/docs` and `LICENSE.txt` files into this new directory. The docs should be built with `BUILD_FOR_WEBSITE=1`.
5. Commit the `index.html` to the `release/X.Y` directory to redirect (use from previous release).
6. Update the `releases/download.html` file with the new release.
7. Update the `releases/index.html` with the new release and link to release documentation.
8. Finally, update the main page (`index.html` and sidebar) to point to the new release and release announcement. Make sure this all gets committed back into Subversion.

Announce the Release

Have Chris send out the release announcement when everything is finished.

5.6 Advice on Packaging LLVM

5.6.1 Overview

LLVM sets certain default configure options to make sure our developers don't break things for constrained platforms. These settings are not optimal for most desktop systems, and we hope that packagers (e.g., Redhat, Debian, MacPorts, etc.) will tweak them. This document lists settings we suggest you tweak.

LLVM's API changes with each release, so users are likely to want, for example, both LLVM-2.6 and LLVM-2.7 installed at the same time to support apps developed against each.

5.6.2 Compile Flags

LLVM runs much more quickly when it's optimized and assertions are removed. However, such a build is currently incompatible with users who build without defining `NDEBUG`, and the lack of assertions makes it hard to debug problems in user code. We recommend allowing users to install both optimized and debug versions of LLVM in parallel. The following configure flags are relevant:

`--disable-assertions` Builds LLVM with `NDEBUG` defined. Changes the LLVM ABI. Also available by setting `DISABLE_ASSERTIONS=0|1` in `make`'s environment. This defaults to enabled regardless of the optimization setting, but it slows things down.

`--enable-debug-symbols` Builds LLVM with `-g`. Also available by setting `DEBUG_SYMBOLS=0|1` in `make`'s environment. This defaults to disabled when optimizing, so you should turn it back on to let users debug their programs.

`--enable-optimized` (For svn checkouts) Builds LLVM with `-O2` and, by default, turns off debug symbols. Also available by setting `ENABLE_OPTIMIZED=0|1` in `make`'s environment. This defaults to enabled when not in a checkout.

5.6.3 C++ Features

RTTI LLVM disables RTTI by default. Add `REQUIRES_RTTI=1` to your environment while running `make` to re-enable it. This will allow users to build with RTTI enabled and still inherit from LLVM classes.

5.6.4 Shared Library

Configure with `--enable-shared` to build `libLLVM-<major>.<minor>.(so|dylib)` and link the tools against it. This saves lots of binary size at the cost of some startup time.

5.6.5 Dependencies

`--enable-libffi` Depend on libffi to allow the LLVM interpreter to call external functions.

`--with-oprofile`

Depend on libopagent (>=version 0.9.4) to let the LLVM JIT tell oprofile about function addresses and line numbers.

5.7 How To Validate a New Release

5.7.1 Introduction

This document contains information about testing the release candidates that will ultimately be the next LLVM release. For more information on how to manage the actual release, please refer to *How To Release LLVM To The Public*.

Overview of the Release Process

Once the release process starts, the Release Manager will ask for volunteers, and it'll be the role of each volunteer to:

- Test and benchmark the previous release
- Test and benchmark each release candidate, comparing to the previous release and candidates
- Identify, reduce and report every regression found during tests and benchmarks
- Make sure the critical bugs get fixed and merged to the next release candidate

Not all bugs or regressions are show-stoppers and it's a bit of a grey area what should be fixed before the next candidate and what can wait until the next release.

It'll depend on:

- The severity of the bug, how many people it affects and if it's a regression or a known bug. Known bugs are "unsupported features" and some bugs can be disabled if they have been implemented recently.
- The stage in the release. Less critical bugs should be considered to be fixed between RC1 and RC2, but not so much at the end of it.
- If it's a correctness or a performance regression. Performance regression tends to be taken more lightly than correctness.

5.7.2 Scripts

The scripts are in the `utils/release` directory.

test-release.sh

This script will check-out, configure and compile LLVM+Clang (+ most add-ons, like `compiler-rt`, `libcxx` and `clang-extra-tools`) in three stages, and will test the final stage. It'll have installed the final binaries on the Phase3/Releasei(+Asserts) directory, and that's the one you should use for the test-suite and other external tests.

To run the script on a specific release candidate run:

```
./test-release.sh \
     -release 3.3 \
     -rc 1 \
     -no-64bit \
```

```
    -test-asserts \
    -no-compare-files
```

Each system will require different options. For instance, x86_64 will obviously not need `-no-64bit` while 32-bit systems will, or the script will fail.

The important flags to get right are:

- On the pre-release, you should change `-rc 1` to `-final`. On RC2, change it to `-rc 2` and so on.
- On non-release testing, you can use `-final` in conjunction with `-no-checkout`, but you'll have to create the `final` directory by hand and link the correct source dir to `final/llvm.src`.
- For release candidates, you need `-test-asserts`, or it won't create a "Release+Asserts" directory, which is needed for release testing and benchmarking. This will take twice as long.
- On the final candidate you just need Release builds, and that's the binary directory you'll have to pack.

This script builds three phases of Clang+LLVM twice each (Release and Release+Asserts), so use screen or nohup to avoid headaches, since it'll take a long time.

Use the `--help` option to see all the options and chose it according to your needs.

findRegressions-nightly.py

TODO

5.7.3 Test Suite

Follow the LNT Quick Start Guide link on how to set-up the test-suite

The binary location you'll have to use for testing is inside the `rcN/Phase3/Release+Asserts/llvmCore-REL-RC.install`. Link that directory to an easier location and run the test-suite.

An example on the run command line, assuming you created a link from the correct install directory to `~/devel/llvm/install`:

```
./sandbox/bin/python sandbox/bin/lnt runtest \
    nt \
    -j4 \
    --sandbox sandbox \
    --test-suite ~/devel/llvm/test/test-suite \
    --cc ~/devel/llvm/install/bin/clang \
    --cxx ~/devel/llvm/install/bin/clang++
```

It should have no new regressions, compared to the previous release or release candidate. You don't need to fix all the bugs in the test-suite, since they're not necessarily meant to pass on all architectures all the time. This is due to the nature of the result checking, which relies on direct comparison, and most of the time, the failures are related to bad output checking, rather than bad code generation.

If the errors are in LLVM itself, please report every single regression found as blocker, and all the other bugs as important, but not necessarily blocking the release to proceed. They can be set as "known failures" and to be fix on a future date.

5.7.4 Pre-Release Process

When the release process is announced on the mailing list, you should prepare for the testing, by applying the same testing you'll do on the release candidates, on the previous release.

You should:

- Download the previous release sources from http://llvm.org/releases/download.html.
- Run the test-release.sh script on `final` mode (change `-rc 1` to `-final`).
- Once all three stages are done, it'll test the final stage.
- Using the `Phase3/Release+Asserts/llvmCore-MAJ.MIN-final.install` base, run the test-suite.

If the final phase's `make check-all` failed, it's a good idea to also test the intermediate stages by going on the obj directory and running `make check-all` to find if there's at least one stage that passes (helps when reducing the error for bug report purposes).

5.7.5 Release Process

When the Release Manager sends you the release candidate, download all sources, unzip on the same directory (there will be sym-links from the appropriate places to them), and run the release test as above.

You should:

- Download the current candidate sources from where the release manager points you (ex. http://llvm.org/pre-releases/3.3/rc1/).
- Repeat the steps above with `-rc 1`, `-rc 2` etc modes and run the test-suite the same way.
- Compare the results, report all errors on Bugzilla and publish the binary blob where the release manager can grab it.

Once the release manages announces that the latest candidate is the good one, you have to pack the `Release` (no Asserts) install directory on `Phase3` and that will be the official binary.

- Rename (or link) `clang+llvm-REL-ARCH-ENV` to the .install directory
- Tar that into the same name with `.tar.gz` extensioan from outside the directory
- Make it available for the release manager to download

5.7.6 Bug Reporting Process

If you found regressions or failures when comparing a release candidate with the previous release, follow the rules below:

- Critical bugs on compilation should be fixed as soon as possible, possibly before releasing the binary blobs.
- Check-all tests should be fixed before the next release candidate, but can wait until the test-suite run is finished.
- Bugs in the test suite or unimportant check-all tests can be fixed in between release candidates.
- New features or recent big changes, when close to the release, should have done in a way that it's easy to disable. If they misbehave, prefer disabling them than releasing an unstable (but untested) binary package.

5.8 Code Reviews with Phabricator

If you prefer to use a web user interface for code reviews, you can now submit your patches for Clang and LLVM at LLVM's Phabricator instance.

While Phabricator is a useful tool for some, the relevant -commits mailing list is the system of record for all LLVM code review. The mailing list should be added as a subscriber on all reviews, and Phabricator users should be prepared to respond to free-form comments in mail sent to the commits list.

5.8.1 Sign up

To get started with Phabricator, navigate to http://reviews.llvm.org and click the power icon in the top right. You can register with a GitHub account, a Google account, or you can create your own profile.

Make *sure* that the email address registered with Phabricator is subscribed to the relevant -commits mailing list. If your are not subscribed to the commit list, all mail sent by Phabricator on your behalf will be held for moderation.

Note that if you use your Subversion user name as Phabricator user name, Phabricator will automatically connect your submits to your Phabricator user in the Code Repository Browser.

5.8.2 Requesting a review via the command line

Phabricator has a tool called *Arcanist* to upload patches from the command line. To get you set up, follow the Arcanist Quick Start instructions.

You can learn more about how to use arc to interact with Phabricator in the Arcanist User Guide.

5.8.3 Requesting a review via the web interface

The tool to create and review patches in Phabricator is called *Differential*.

Note that you can upload patches created through various diff tools, including git and svn. To make reviews easier, please always include **as much context as possible** with your diff! Don't worry, Phabricator will automatically send a diff with a smaller context in the review email, but having the full file in the web interface will help the reviewer understand your code.

To get a full diff, use one of the following commands (or just use Arcanist to upload your patch):

- `git diff -U999999 other-branch`
- `svn diff --diff-cmd=diff -x -U999999`

To upload a new patch:

- Click *Differential*.
- Click *Create Diff*.
- Paste the text diff or upload the patch file. Note that TODO
- Leave the drop down on *Create a new Revision...* and click *Continue*.

- Enter a descriptive title and summary; add reviewers and mailing lists that you want to be included in the review. If your patch is for LLVM, cc llvm-commits; if your patch is for Clang, cc cfe-commits.
- Click *Save*.

To submit an updated patch:

- Click *Differential*.
- Click *Create Diff*.
- Paste the updated diff.
- Select the review you want to from the *Attach To* dropdown and click *Continue*.
- Click *Save*.

5.8.4 Reviewing code with Phabricator

Phabricator allows you to add inline comments as well as overall comments to a revision. To add an inline comment, select the lines of code you want to comment on by clicking and dragging the line numbers in the diff pane.

You can add overall comments or submit your comments at the bottom of the page.

Phabricator has many useful features, for example allowing you to select diffs between different versions of the patch as it was reviewed in the *Revision Update History*. Most features are self descriptive - explore, and if you have a question, drop by on #llvm in IRC to get help.

Note that as e-mail is the system of reference for code reviews, and some people prefer it over a web interface, we do not generate automated mail when a review changes state, for example by clicking "Accept Revision" in the web interface. Thus, please type LGTM into the comment box to accept a change from Phabricator.

5.8.5 Committing a change

Arcanist can manage the commit transparently. It will retrieve the description, reviewers, the `Differential Revision`, etc from the review and commit it to the repository.

```
arc patch D<Revision>
arc commit --revision D<Revision>
```

When committing an LLVM change that has been reviewed using Phabricator, the convention is for the commit message to end with the line:

```
Differential Revision: <URL>
```

where `<URL>` is the URL for the code review, starting with `http://reviews.llvm.org/`.

Note that Arcanist will add this automatically.

This allows people reading the version history to see the review for context. This also allows Phabricator to detect the commit, close the review, and add a link from the review to the commit.

5.8.6 Status

Please let us know whether you like it and what could be improved!

LLVM Developer Policy The LLVM project's policy towards developers and their contributions.

Creating an LLVM Project How-to guide and templates for new projects that *use* the LLVM infrastructure. The templates (directory organization, Makefiles, and test tree) allow the project code to be located outside (or inside) the `llvm/` tree, while using LLVM header files and libraries.

LLVMBuild Guide Describes the LLVMBuild organization and files used by LLVM to specify component descriptions.

LLVM Makefile Guide Describes how the LLVM makefiles work and how to use them.

How To Release LLVM To The Public This is a guide to preparing LLVM releases. Most developers can ignore it.

How To Validate a New Release This is a guide to validate a new release, during the release process. Most developers can ignore it.

Advice on Packaging LLVM Advice on packaging LLVM into a distribution.

Code Reviews with Phabricator Describes how to use the Phabricator code review tool hosted on http://reviews.llvm.org/ and its command line interface, Arcanist.

CHAPTER

SIX

COMMUNITY

LLVM has a thriving community of friendly and helpful developers. The two primary communication mechanisms in the LLVM community are mailing lists and IRC.

6.1 Mailing Lists

If you can't find what you need in these docs, try consulting the mailing lists.

Developer's List (llvmdev) This list is for people who want to be included in technical discussions of LLVM. People post to this list when they have questions about writing code for or using the LLVM tools. It is relatively low volume.

Commits Archive (llvm-commits) This list contains all commit messages that are made when LLVM developers commit code changes to the repository. It also serves as a forum for patch review (i.e. send patches here). It is useful for those who want to stay on the bleeding edge of LLVM development. This list is very high volume.

Bugs & Patches Archive (llvmbugs) This list gets emailed every time a bug is opened and closed. It is higher volume than the LLVMdev list.

Test Results Archive (llvm-testresults) A message is automatically sent to this list by every active nightly tester when it completes. As such, this list gets email several times each day, making it a high volume list.

LLVM Announcements List (llvm-announce) This is a low volume list that provides important announcements regarding LLVM. It gets email about once a month.

6.2 IRC

Users and developers of the LLVM project (including subprojects such as Clang) can be found in #llvm on irc.oftc.net.

This channel has several bots.

- Buildbot reporters
  - llvmbb - Bot for the main LLVM buildbot master. http://lab.llvm.org:8011/console
  - bb-chapuni - An individually run buildbot master. http://bb.pgr.jp/console
  - smooshlab - Apple's internal buildbot master.
- robot - Bugzilla linker. %bug <number>
- clang-bot - A geordi instance running near-trunk clang instead of gcc.

CHAPTER

SEVEN

RELEASE NOTE

This document was generated from LLVM's git repository, I manually edited the latex source code generated by sphinx, removed redundant content and updated some page format, so I tag my name here ☺

Weibo: http://weibo.com/u/2792942570

Homepage: https://chou.it