Introduction to Clang/LLVM

Christian Sharpsten

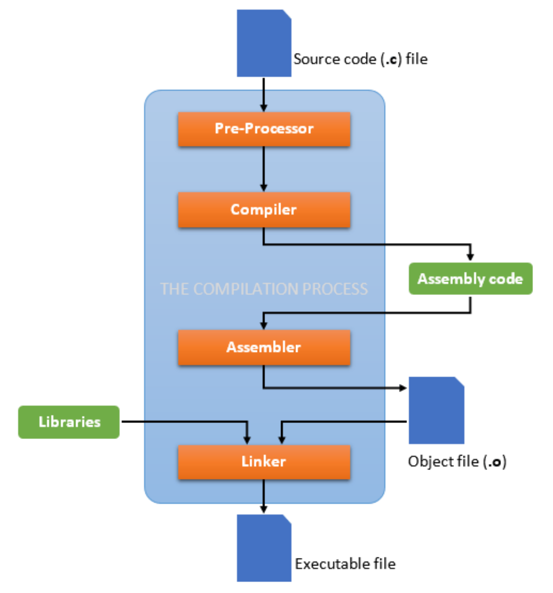
01 NOV 2019

Outline

- Background
  - General Compilation Process
  - About LLVM
- LLVM Compilation Process
- Writing an Optimization Pass
  - Building LLVM
  - A Simple Pass
  - Control-Flow Obfuscation

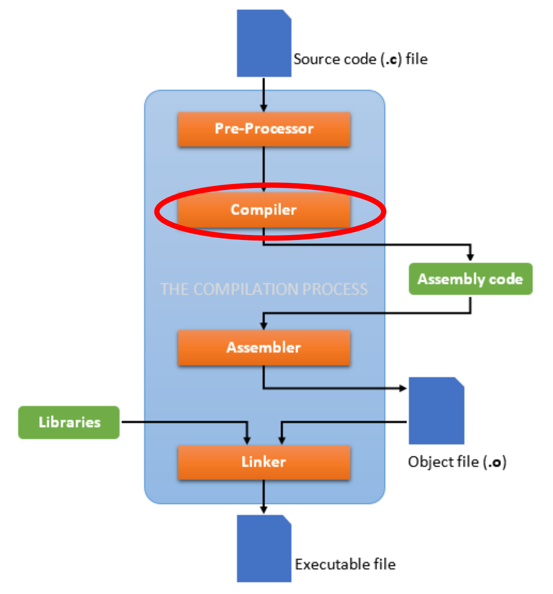
Background

General Compilation Process



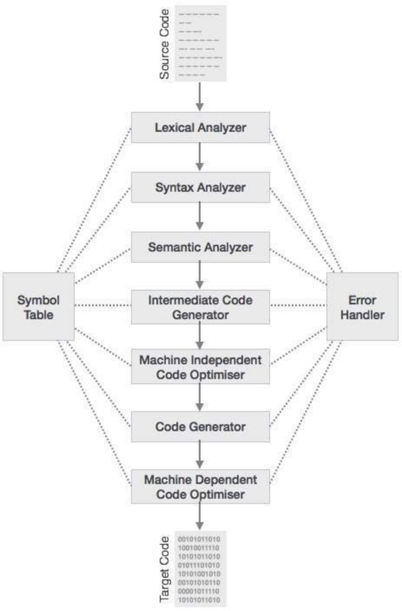
[4]

General Compilation Process



[4]

General Compilation Process



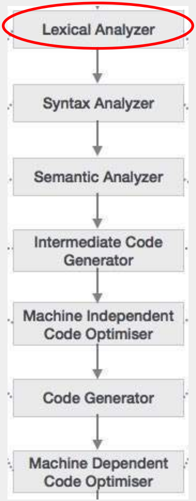
[1]

General Compilation Process (Lexing)

Convert a stream of characters into a stream of tokens. A lexer recognizes a regular language.

```
int main() {
    return 0;
}
```

```
TOK_KEYWORD            "int"      1
TOK_IDENTIFIER         "main"     1
TOK_L_PAREN            "("        1
TOK_R_PAREN            ")"        1
TOK_L_BRACE            "{"        1
TOK_KEYWORD            "return"   2
TOK_NUMERIC_CONSTANT   "0"        2
TOK_SEMI               ";"        2
TOK_R_BRACE            "}"        3
```



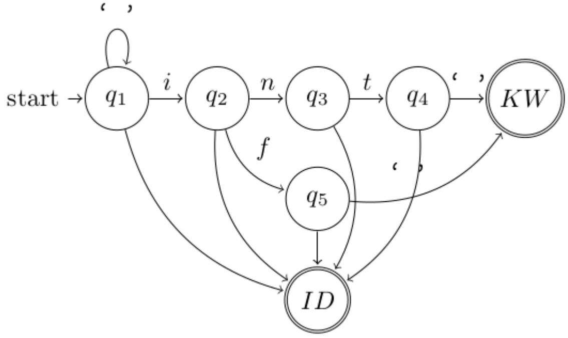
`llvm-project/clang/include/clang/Basic/TokenKinds.def`

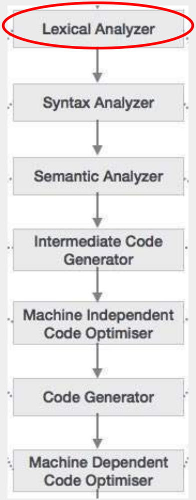
General Compilation Process (Lexing)

Since the lexer recognizes a regular language, we can model the lexer as a Deterministic Finite Automata (DFA). This allows to lex in one pass.

Partial DFA recognizing the language:
“ *(int|if|[a-z]+) ”

Alphabet = “[a-z]”

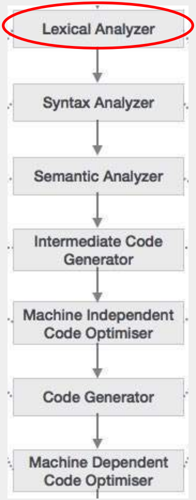




General Compilation Process (Lexing)

In fact, there are tools that do this, such as Flex.

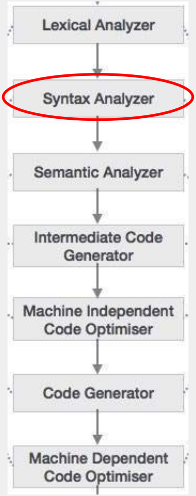
```
DIGIT     [0-9]
ID        [_a-zA-Z][_a-zA-Z0-9]*

%%
{DIGIT}+   {
           printf("Number: %d\n", atoi(yytext));
           }
{ID}       {
           printf("Identifier: %s\n", yytext);
           }
int|print  {
           printf("Keyword: %s\n", yytext);
           }
%%
```



- Parsing is the process of building a Parse Tree from the token stream.
- The grammar of the language is normally defined by a series of production rules.

$$
\begin{aligned}
&EXPR \rightarrow EXPR + TERM \\
&EXPR \rightarrow EXPR - TERM \\
&EXPR \rightarrow TERM \\
&TERM \rightarrow 0 \\
&TERM \rightarrow 1 \\
&TERM \rightarrow \ldots \\
&TERM \rightarrow 9
\end{aligned}
$$

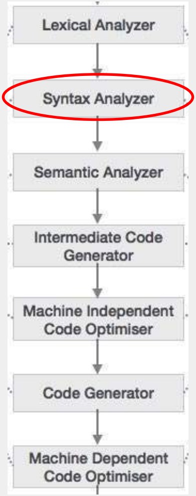
If at any point no production is available, we have encountered a parse error.



General Compilation Process (Parsing)

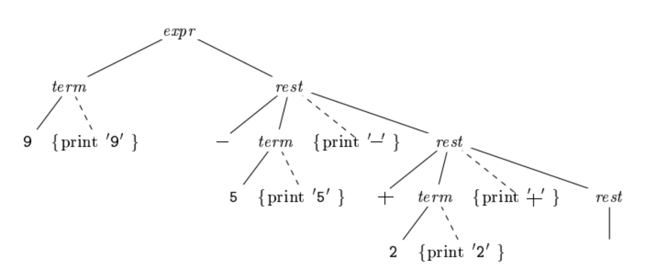
After removing left-recursion, we can use this equivalent grammar instead for top-down parsing:

```
EXPR  —> TERM  REST
REST  —> + TERM  REST
       | — TERM  REST
       | e
TERM  —> 0
       | 1
         ...
       | 9
```

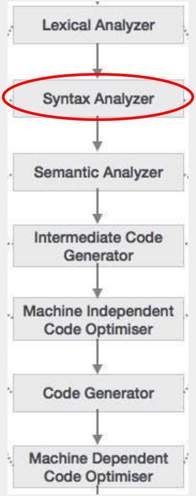


Let's build the parse tree for a simple statement.

9 — 5 + 2



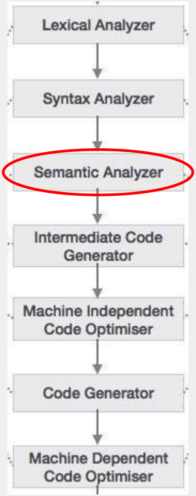
[8]



You can use bison to generate a parser for you.

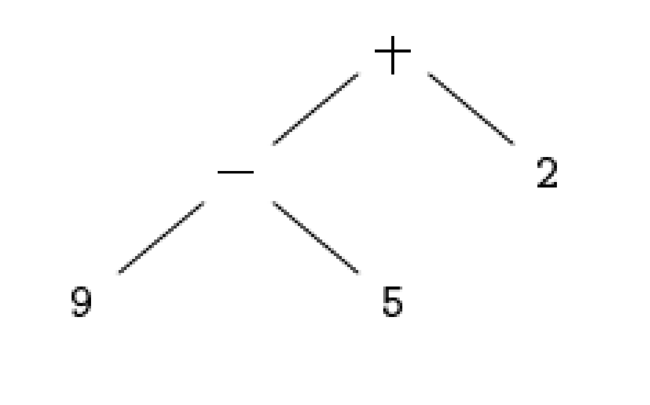
General Compilation Process (Parsing)

- We can then process the parse tree to remove redundant information and perform semantic analysis, like type checking.
- This is often where type coercions are introduced.
- This phase produces the Abstract Syntax Tree (AST).

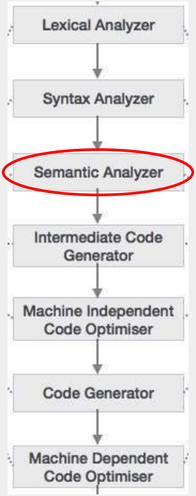


Continuing the earlier example...

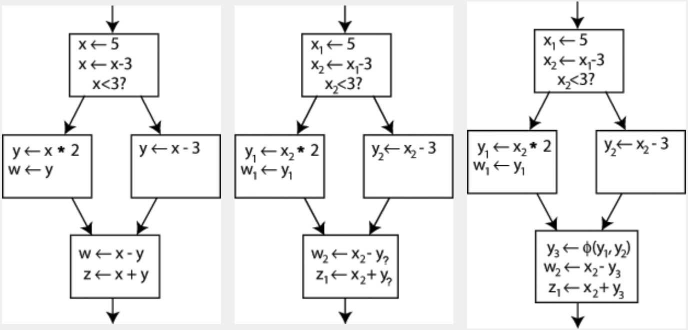
9 — 5 + 2



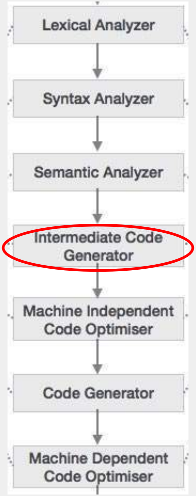
[8]



Many compilers generate IR that is in Static Single Assignment (SSA) form - Each variable is assigned exactly once, and every variable is defined before it is used.



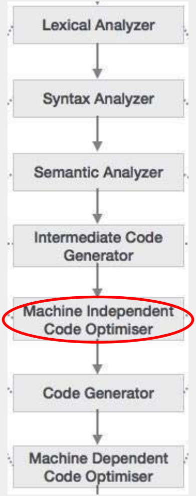
[7]



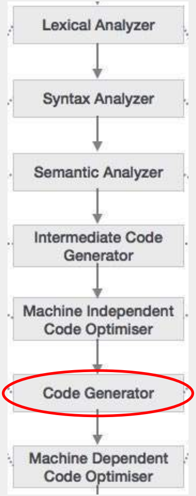
General Compilation Process (Optimization)

The compiler will run a number of optimization passes. A few of the common ones are listed here.

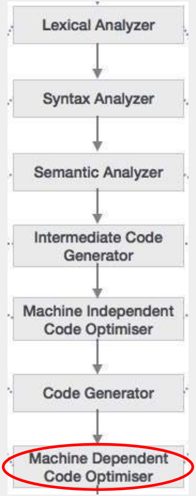
- Strength Reduction
- Constant Propagation
- Dead Code Elimination
- Loop Invariant Code Motion (Hoisting and Sinking)
- Scalar Replacement of Aggregates & mem2reg



During this phase, the compiler will generate assembly code for the given target.



Depending on how the backend code generator is implemented, the compiler may apply target-specific optimizations.



What is LLVM?

LLVM is a "collection of modular and reusable compiler and toolchain technologies." [5]

LLVM is composed of multiple sub-projects including:

1. **LLVM Core** - A set of libraries implementing an optimizer and code generators for common CPUs
2. **Clang** - A front-end compiler
3. **LLDB** - A native debugger
4. **libc++** - A C++14 compliant STL
5. **compiler-rt** - Compiler run-time libraries (intrinsics, ASAN, TSAN, MSAN, etc.)
6. **klee** - A symbolic executor
7. **LLD** - A drop-in replacement for system linkers such as `ld`

Why LLVM?

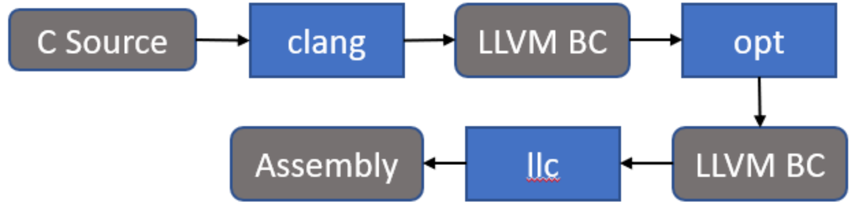
Why is LLVM interesting?

- Modular
- Easy to hack on
- It has a JIT Engine
- Can cross-compile for multiple architectures with one build

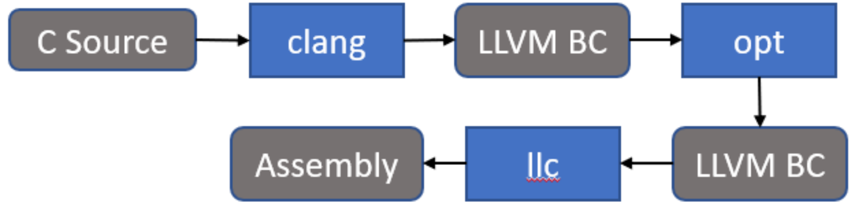
LLVM has been used in a number of open source tools:

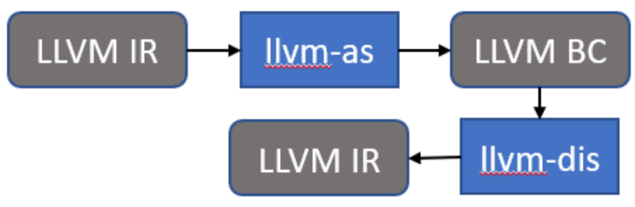
- Keystone
- Capstone
- McSema

LLVM Core includes a number of tools:



LLVM Core includes a number of tools:





LLVM Compilation Process

Clang: Pre-processing, Lexing, and Parsing

```
int main(void) {
    int a = 5 + 2;
    return a;
}

$ clang -Xclang -ast-dump test.c
TranslationUnitDecl <<invalid sloc>> <invalid sloc>
|-...
|-...
`-FunctionDecl <test.c:1:1, line:4:1> line:1:5 main 'int (void)'
  `-CompoundStmt <col:16, line:4:1>
    |-DeclStmt <line:2:5, col:18>
    | `-VarDecl <col:5, col:17> col:9 used a 'int' cinit
    |   `-BinaryOperator <col:13, col:17> 'int' '+'
    |     |-IntegerLiteral <col:13> 'int' 5
    |     `-IntegerLiteral <col:17> 'int' 2
    `-ReturnStmt <line:3:5, col:12>
      `-ImplicitCastExpr <col:12> 'int' <LValueToRValue>
        `-DeclRefExpr <col:12> 'int' lvalue Var 'a' 'int'
```

Use the `-emit-llvm` option to enable bitcode generation.

```
$ clang -c -emit-llvm test.c -o test.bc
$ llvm-dis < test.bc
; ModuleID = '<stdin>'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
%retval = alloca i32, align 4
%a = alloca i32, align 4
store i32 0, i32* %retval, align 4
store i32 7, i32* %a, align 4
%0 = load i32, i32* %a, align 4
ret i32 %0
}
...
```

The data layout string describes how data is to be laid out in memory. Elements are separated by the minus sign.

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
```

Spec	Description	Value
e	Endianness	little-endian
m:e	IR Name Mangling Type	ELF mangling
i64:64	Alignment for 64-bit integers (bits)	64
f80:128	Alignment for 80-bit floats (bits)	128
n8:16:32:64	Set of native integer widths for the CPU (bits)	8, 16, 32, 64
S128	Stack Alignment (bits)	128

The `opt` tool runs target-independent optimizations on LLVM bitcode.

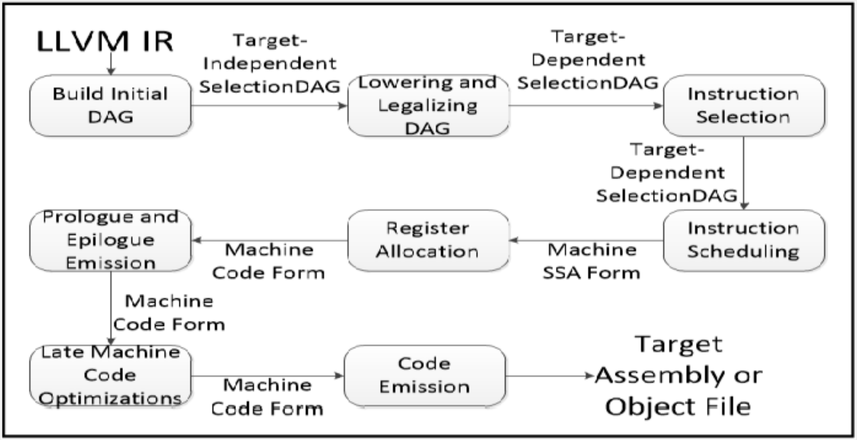
There are different types of passes, each registered with a pass manager. A pass can be an analysis or a modification pass.

- ModulePass - Inter-procedural optimizations
- FunctionPass - Intra-procedural optimizations
- BasicBlockPass - Useful for local and “peephole” optimizations
- CallGraphSCCPass, LoopPass, RegionPass - Specialized pass types

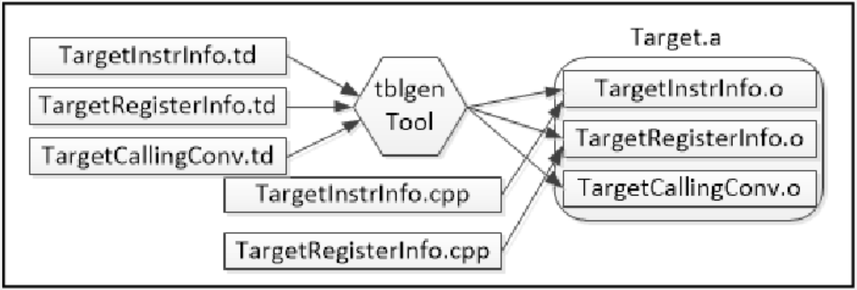
The PassManager is responsible for scheduling passes in an order that makes sense (analysis dependencies, SROA before DCE, etc.)

You can view the structure of passes by using the –debug-pass=Structure option.

```
$ opt -O1 --debug-pass=Structure test.bc > test_opt.bc
Pass Arguments:  -verify -simplifycfg -domtree -sroa ...
  FunctionPass Manager
    Module Verifier
    Simplify the CFG
    Dominator Tree Construction
    SROA
Pass Arguments:  -simplifycfg -verify -write-bitcode ...
  ModulePass Manager
    Dead Argument Elimination
    FunctionPass Manager
      Dominator Tree Construction
      Simplify the CFG
...
...
```


LLVM IR
Build Initial DAG
Target-Independent SelectionDAG
Lowering and Legalizing DAG
Target-Dependent SelectionDAG
Instruction Selection
Target-Dependent SelectionDAG
Instruction Scheduling
Machine SSA Form
Register Allocation
Machine Code Form
Prologue and Epilogue Emission
Machine Code Form
Late Machine Code Optimizations
Machine Code Form
Code Emission
Target Assembly or Object File

LLVM uses a tool called `tblgen` to translate target description (.td) files into C++ code that implements part of the target code generator.



Writing an Optimization Pass

Building LLVM

- LLVM switched from svn to a single git monorepo as of 21 OCT 2019 (exciting!)
- LLVM uses CMake. You can control the build in a number of ways:
  - Generator (Ninja, Unix Makefiles, VS, Xcode)
  - Build type (Debug, Release, RelWithDebInfo, MinSizeRel)
  - Enabled sub-projects (test suite, libcxx, lldb, lld, etc.)
  - Backend targets (X86, Mips, PowerPC, etc.)
- Depending on which features you enable, LLVM can take a long time to compile.

Building LLVM

For this exercise, we only need to build Clang and the X86 backend.

```
$ git clone https://github.com/llvm/llvm-project.git
```

For this exercise, we only need to build Clang and the X86 backend.

```
$ git clone https://github.com/llvm/llvm-project.git
$ cd llvm-project && git checkout llvmorg-9.0.0
```

For this exercise, we only need to build Clang and the X86 backend.

```
$ git clone https://github.com/llvm/llvm-project.git
$ cd llvm-project && git checkout llvmorg-9.0.0
$ mkdir build && cd build
```

Building LLVM

For this exercise, we only need to build Clang and the X86 backend.

```
$ git clone https://github.com/llvm/llvm-project.git
$ cd llvm-project && git checkout llvmorg-9.0.0
$ mkdir build && cd build
$ cmake -G Ninja                          \
    -DLLVM_ENABLE_PROJECTS='clang'    \
    -DCMAKE_BUILD_TYPE=Debug           \
    -DLLVM_TARGETS_TO_BUILD=X86
    ../llvm
```

Building LLVM

For this exercise, we only need to build Clang and the X86 backend.

```
$ git clone https://github.com/llvm/llvm-project.git
$ cd llvm-project && git checkout llvmorg-9.0.0
$ mkdir build && cd build
$ cmake -G Ninja                          \
    -DLLVM_ENABLE_PROJECTS='clang'    \
    -DCMAKE_BUILD_TYPE=Debug          \
    -DLLVM_TARGETS_TO_BUILD=X86
    ../llvm
$ time ninja
$ sudo ninja install
```

LLVM can take a while to compile...

Generator	Build Type	Sub-Projects	Targets	Time (m)	Size (GB)
Ninja	**Debug**	**Clang**	**X86**	**120.15**	**44.0**
Ninja	Release	Clang	X86	75.03	1.7
Ninja	Debug	Clang	All	205.65	59.5
Ninja	Release	Clang	All	106.33	2.5
Make	Release	Clang	X86	433.30	1.8
Make (-j8)	Release	Clang	X86	77.13	1.7

Table: LLVM Compile Time Benchmarks (Ubuntu 18.04 VM, 6 cores, 16GB RAM)

Keep in mind that debug artifacts can be quite large as well.

High Memory Usage

Watch out for out-of-memory errors when linking. Restart ninja/make with less threads if a link process is killed.

```
[2361/2742] Linking CXX shared module
    lib/CheckerOptionHandlingAnalyzerPlugin.so
FAILED: lib/CheckerOptionHandlingAnalyzerPlugin.so
: && /usr/bin/c++ -fPIC -fPIC ...
...
collect2: fatal error: ld terminated with signal 9 [Killed]
compilation terminated.
ninja: build stopped: subcommand failed.
```

High Memory Usage

Watch out for out-of-memory errors when linking. Restart ninja/make with less threads if a link process is killed.

```
[2359/2742] Linking CXX executable bin/clang-diff
FAILED: bin/clang-diff
: && /usr/bin/c++  -fPIC -fvisibility-inlines-hidden ...
...
/usr/bin/ld: BFD (GNU Binutils for Ubuntu) 2.30 internal error,
    aborting at ../../bfd/merge.c:908 in
    _bfd_merged_section_offset

/usr/bin/ld: Please report this bug.

collect2: error: ld returned 1 exit status
ninja: build stopped: subcommand failed.
```

https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=874674

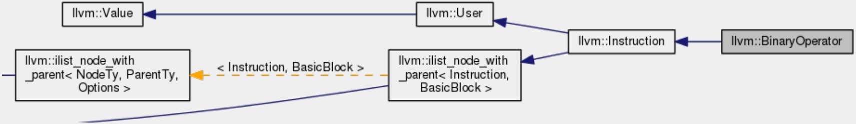
LLVM IR

LLVM comes with IR and tblgen syntax highlighting for vim, emacs, and vscode, among other editors.

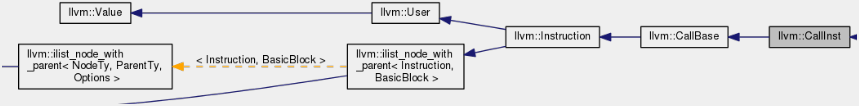
```
$ cd ~/.vim
$ ln -s ~/llvm-project/llvm/utils/vim/
        {ftdetect,ftplugin,indent,syntax}
```

Demo

LLVM BinaryOperator class:



LLVM CallInst class:



The llvm::Value, llvm::User, and llvm::Use classes implement use-def chains.

llvm::User provides an op_iterator that returns llvm::Use * for operands.
llvm::Value provides a use_iterator that returns uses of this value.

A *def-use* chain is the list of all Users of a particular Value.

```
Function *F = ...;
for (User *U : F->users()) {
    if (Instruction *Inst = dyn_cast<Instruction>(U)) {
        errs() << "F is used in instruction:\n";
        errs() << *Inst << "\n";
    }
```

A *use-def* chain is the list of all Values used by a User.

```
Instruction *pi = ...;

for (Use &U : pi->operands()) {
    Value *v = U.get();
    // ...
}
```

- Many classes provide iterators for common collections (BBs in a function, functions in a module)
- LLVM makes extensive use of a custom form of RTTI, similar to C++ `dynamic_cast<>`. It provides a number of operators such as “isa<>”, “cast<>”, and “dyn_cast<>”.

```
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());
}
```

- The Builder API allows you to generate new code during your pass
- Some quirks - In Windows calls that are inside a __try/__except block are emitted as ‘Invoke’ instructions

The LLVM documentation recommends building your pass a shared object, to be loaded by clang or opt.

```
$ clang -c -emit-llvm chal.c -o chal.bc
$ opt -load /usr/local/lib/LLVMExamplePass.so --example chal.bc > chal_c
Function: decrypt
External: rand
Function: main
External: llvm.memset.p0i8.i64
External: srand
External: printf
External: __isoc99_scanf
External: strlen
External: memcmp
External: puts
```

Writing a Pass (ExamplePass)

The LLVM documentation recommends building your pass a shared object, to be loaded by clang or opt.

```
$ clang -Xclang -load -Xclang /usr/local/lib/LLVMExamplePass.so chal.c
Function: decrypt
External: rand
Function: main
External: llvm.memset.p0i8.i64
External: srand
External: printf
External: __isoc99_scanf
External: strlen
External: memcmp
External: puts
```

There are tons of interesting things you can do with a pass.

- Obfuscation - bogus arguments, constant obfuscation, control flow obfuscation, string encryption, etc.
- Source and target independent taint tracing to detect vulnerabilities
- Measure statistics about code you compile

ExtractBB Pass

What if we could obfuscate a program by making control-flow interprocedural?

ExtractBB Pass

What if we could obfuscate a program by making control-flow interprocedural?

1. Conduct a liveness analysis to determine the set of live variables at the entry of each basic block.

ExtractBB Pass

What if we could obfuscate a program by making control-flow interprocedural?

1. Conduct a liveness analysis to determine the set of live variables at the entry of each basic block.
2. Extract each basic block into a new function.

ExtractBB Pass

What if we could obfuscate a program by making control-flow interprocedural?

1. Conduct a liveness analysis to determine the set of live variables at the entry of each basic block.
2. Extract each basic block into a new function.
3. Convert branches to calls.

ExtractBB Pass

What if we could obfuscate a program by making control-flow interprocedural?

1. Conduct a liveness analysis to determine the set of live variables at the entry of each basic block.
2. Extract each basic block into a new function.
3. Convert branches to calls.
4. Fixup operand uses with their new argument Values.

ExtractBB Pass

What if we could obfuscate a program by making control-flow interprocedural?

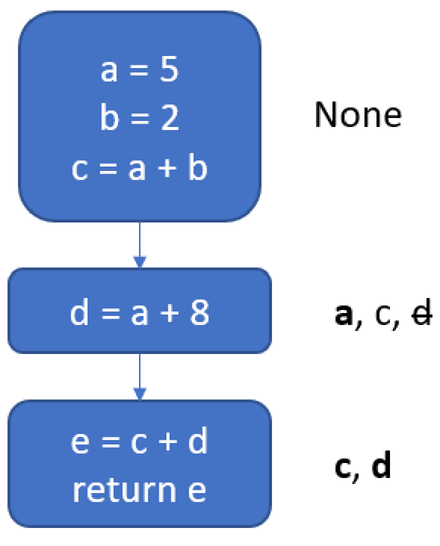
1. Conduct a liveness analysis to determine the set of live variables at the entry of each basic block.
2. Extract each basic block into a new function.
3. Convert branches to calls.
4. Fixup operand uses with their new argument Values.
5. Remove PhiNodes.

ExtractBB Pass (1 - Liveness Analysis)

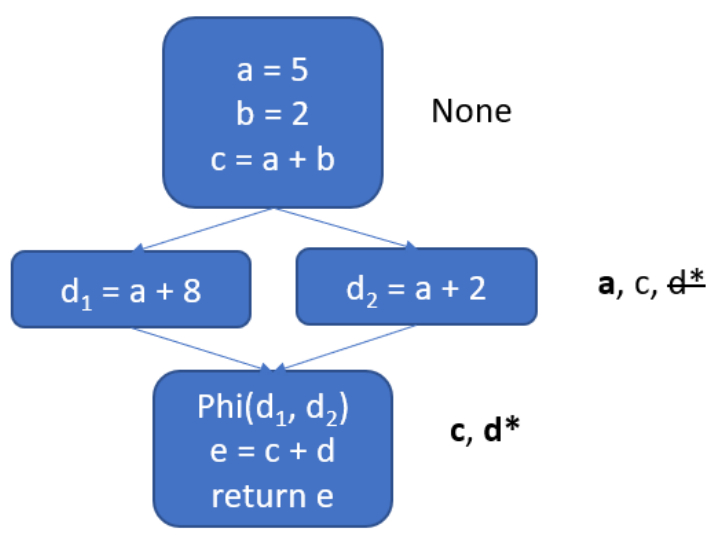
The arguments for each new function need to be the set of variables that are still **live** at that point in the function. We can do liveness analysis with multiple postorder traversals of the CFG.

We know LLVM IR is already in SSA. So we can just follow use-def chain for each inst to get these values.
`https://github.com/shareef12/ExtractBB/blob/8cc4ddf2502353450e88f435b252e39d4bc31c8d/ExtractBB/Extract.cpp#L99`

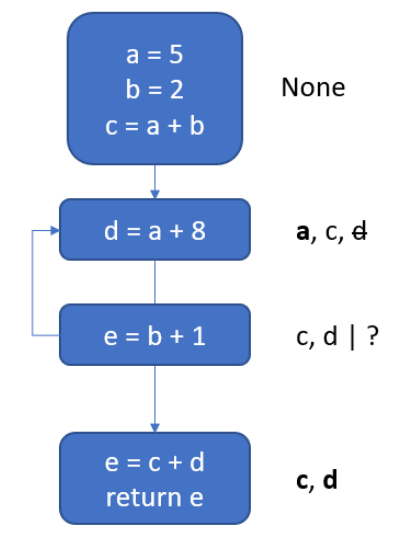
ExtractBB Pass (1 - Liveness Analysis)



ExtractBB Pass (1 - Liveness Analysis)



ExtractBB Pass (1 - Liveness Analysis)



ExtractBB Pass (2 - Extract Basic Blocks)

https://github.com/shareef12/ExtractBB/blob/master/ExtractBB/Extract.cpp#L257

https://github.com/shareef12/ExtractBB/blob/8cc4ddf2502353450e88f435b252e39d4bc31c8d/ExtractBB/Extract.cpp#L331

ExtractBB Pass (4 - Fixup Argument Uses)

https://github.com/shareef12/ExtractBB/blob/8cc4ddf2502353450e88f435b252e39d4bc31c8d/ExtractBB/Extract.cpp#L421

ExtractBB Pass (5 - Remove PHINodes)

https://github.com/shareef12/ExtractBB/blob/8cc4ddf2502353450e88f435b252e39d4bc31c8d/ExtractBB/Extract.cpp#L450

QUESTIONS?

References I

Compiler design – phases of compiler.
https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm.
Accessed: 2019-10-31.

Getting started with the LLVM system.
https://llvm.org/docs/GettingStarted.html.
Accessed: 2019-10-28.

Introduction to the clang AST.
https://clang.llvm.org/docs/IntroductionToTheClangAST.html.
Accessed: 2019-10-31.

Let's learn about the C program compilation process.
https://medium.com/@victormdnguyen/lets-learn-about-the-c-program-compilation-process-e4
Accessed: 2019-10-31.

References II

The llvm compiler infrastructure.
https://llvm.org/.
Accessed: 2019-10-30.

Llvm language reference manual.
https://llvm.org/docs/LangRef.html.
Accessed: 2019-10-31.

Static single assignment form.
https://en.wikipedia.org/wiki/Static_single_assignment_form.
Accessed: 2019-10-31.

A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman.
Compilers: Principles, Techniques, & Tools, Second Edition.
Pearson Education, Inc., 2007.

References III

Jin-Chuan See, Wai-Kong Lee, Kai-Ming Mok, and Hock-Guan Goh.
Development of llvm compilation toolchain for iot processor targeting wireless measurement applications.
2017 IEEE 4th International Conference on Smart Instrumentation, Measurement and Application (ICSIMA), pages 1–4, 2017.