INDEX

Satya Vasanth Reddy -CS13B1033
Arjun V Anand- CS13B1041

LLVM
Technical Report

1. LLVM DIRECTORY STRUCTURE

LLVM has in three parts.

1. All tools, libraries, and header files needed to use LLVM ,assembler, disassembler, bitcode analyzer and bitcode optimizer and also contains basic regression tests that can be used to test the LLVM tools and the Clang front end.

2.The second is the Clang front end. This compiles C, C++, Objective C, and Objective C++ code into LLVM bitcode.

3.It is optional .It is a suite of programs with a testing harness that can be used to further test LLVM's functionality and performance.

Directory Structure :

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. and 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 .

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 almost all of the source files of the LLVM system so that it becomes easy for different llvm tools to use the same source code

This has subdirectories containing source codes for

1.Implementing Instruction and Basic Block class etc of Intermediate Representation (/IR)

2.Parsing assembly language (/AsmParser)

3.Reading and Writing LLVM bitcode (/Bitcode)

4. LLVM to LLVM program transformations, such as Aggressive Dead Code Elimination, Sparse Conditional Constant Propagation, Inlining,etc.. for optimising (/Transformation)

5.Files describing the specs of various target machines (/Target)

6.Different program analyses, such as Dominator Information, Call Graphs, Induction Variables, Interval Identification, Natural Loop Identification, etc (/Analysis)

7.Helping the program to get the debugger to identify the source code location while program execution (/Debugger)

8.Executing LLVM bitcode directly at runtime in both interpreted and JIT compiled fashions (/ExecutionEngine)

9.Files located in llvm/include/ADT/ and llvm/include/Support/. (/Support)

llvm/tools

The tools directory contains the executables built out of the libraries above.They form the main part of the user interface.

llvm/runtime

This directory contains libraries which are compiled into LLVM bitcode and used when linking programs with the Clang front end.

llvm/test

This directory contains feature and regression tests and other basic sanity checks on the LLVM infrastructure

llvm/bugpoint

It 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.

llvm/utils

This contains utils that are required both for actual build process and also the tools used while working with the llvm code.

codegen-diff

This script generates difference between code generated by llc, lli

emacs

A text editor

getsrcs.sh

This script finds and outputs all non-generated source files

llvmgrep

This little tool performs an egrep -H -n on each source file in LLVM and passes to it a regular expression provided.

makellvm

The makellvm script compiles all files in the current directory and then compiles and links the tool that is the first argument.

TableGen/

The TableGen directory contains the tool used to generate register descriptions, instruction set descriptions, and even assemblers from common TableGen description files.

vim/

A text editor

2. CLANG DIRECTORY STRUCTURE

/include

It contains headers for the AST generation and several manipulations of AST nodes headers for formatting the program code

/lib

This diretory contains the source codes of the headers described in the /include directory of clang for ARC Migration, Code generation , Frontend ,Parsing , Semantic Analysis

/tools

This directory contains tools for code   generation ,Semantic Analysis , ASt building etc.. essential for the clang operations

/examples

This has examples for demonstrating the various operations of clang like parsing standard compiler command line arguments using the Driver library. ,Constructing a Clang compiler instance, using the appropriate argument ,Invoking the Clang compiler to lex, parse, syntax check, and then generate LLVM code, use the LLVM JIT functionality to execute the final module etc..

/test

Has various subdirectories for test different sections of the clang .

/docs

Contains the documentation of clang

3.STRUCTURE OF CLANG AST

Clang's AST representation is a rich form of AST representation. Parenthesis expressions and compile time constants are available in an unreduced form in the AST. As a result, it has close resemblance with the written C++ code. It had more than 100 thousand lines of code. I also noticed that its fully type resolved.

AST Context

All information about the AST for a translation unit is bundled up in the class ASTContext.
Some of its features are:
1. It keeps information around the AST by maintaining the identifier table and source manager.
   SourceManager class handles loading and caching of source files into memory.
   IdentifierTable class implements an efficient mapping from strings to IdentifierInfo nodes.
2. It is the entry point into the AST.

AST Nodes

Clang's AST nodes are modeled on a class hierarchy that does not have a common ancestor. Instead, there are some core classes from which many AST nodes derive. Some classes might derive from more than one core class.
Now, lets lock at the core classes:
1. Decl
2. Stmt
3. Type

Decl - This class represents one declaration (or definition).

This declaration can be of many types:
a. CXXRecordDecl - This class represents declaration of a C++ struct/union/class
b. VarDecl - An instance of this class is created for declaration or definition of a variable
c. UnresolvedUsingTypenameDecl - This class represents a dependent using declaration which was marked with typename. The type associated with an unresolved using typename decl is currently always a typename type.

Stmt - This represents one statement.

There are different kind of statements:
a. CompoundStmt - This represents a group of statements like { stmt stmt }
b. CXXTryStmt - This represents C++ try block, including all handlers
c. BinaryOperator - This expression node kind describes a builtin binary operation, such as "x + y" for integer values "x" and "y". The operands will already have been converted to appropriate types (e.g., by performing promotions or conversions).

Type - The base class of the type hierarchy.

A central concept with types is that each type always has a canonical type. A canonical type is the type with any typedef names stripped out of it or the types it references. Non-canonical types are useful for emitting diagnostics, without losing information about typedefs being used. Canonical types are useful for type comparisons (they allow by-pointer equality tests) and useful for reasoning about whether something has a particular form (e.g. is a function type), because they implicitly, recursively, strip all typedefs out of a type.

Some subclasses of Type are:
a. PointerType - Represents pointers
b. ParenType - Syntactic sugar for parentheses used when specifying types.
c. SubstTemplateTypeParm Type - Represents the result of substituting a type for a template type parameter.
Within an instantiated template, all template type parameters have been replaced with these. They are used solely to record that a type was originally written as a template type parameter; therefore they are never canonical.

Other than these basic classes there are other glue classes which are useful for adding more features to the base classes.

Examples of such classes are:

DeclContext - inherited by decls that contain other decls

TemplateArgument - accessors for the template argument

NestedNameSpecifier - Represents a C++ nested name specifier, such as "\::std::vector<int>::".

C++ nested name specifiers are the prefixes to qualified namespaces. For example, "foo::" in "foo::x" is a nested name specifier.

QualType - To store qualifiers.

One thing which we noticed with regard to this is that qualifiers like const are separated from the base type like int. This is useful because this reduces the number of node types to be defined. Otherwise it would have been a much difficult job to maintain a big pool of node types.

4.AST TRAVERSAL

RecursiveASTVisitor is the class that does preorder depth-first traversal on the entire Clang AST and visits each node. This class performs three main task:

1.Traverse the AST

2. At a particular node, walk up the class hierarchy, starting from the node's dynamic type, until the top-most class (e.g. Stmt, Decl, or Type)is reached.

3. given a (node, class) combination, where 'class' is some base class of the dynamic type of 'node', call a user-overridable function to actually visit the node.

These tasks are done by three groups of methods, respectively:

1. TraverseDecl(Decl *x) does task #1. It is the entry point for traversing an AST rooted at x. This method simply dispatches (i.e. forwards) to TraverseFoo(Foo *x) where Foo is the dynamic type of *x, which calls WalkUpFromFoo(x) and then recursively visits the child nodes of x. TraverseStmt(Stmt *x) and TraverseType(QualType x) work similarly.

2. WalkUpFromFoo(Foo *x) does task #2. It does not try to visit any child node of x. Instead, it first calls WalkUpFromBar(x) where Bar is the direct parent class of Foo (unless Foo has no parent), and then calls VisitFoo(x) (see the next list item).

3.VisitFoo(Foo *x) does task #3.

These three method groups are tiered (Traverse* > WalkUpFrom* > Visit*). A method (e.g. Traverse*) may call methods from the same tier (e.g. other Traverse*) or one tier lower (e.g. WalkUpFrom*). It may not call methods from a higher tier.

Note that since WalkUpFromFoo() calls WalkUpFromBar() (where Bar is Foo's super class) before calling VisitFoo(), the result is that the Visit*() methods for a given node are called in the top-down order (e.g. for a node of type NamespaceDecl, the order will be VisitDecl(), VisitNamedDecl(), and then VisitNamespaceDecl()).

This scheme guarantees that all Visit*() calls for the same AST node are grouped together. In other words, Visit*() methods for different nodes are never interleaved.

Stmts are traversed internally using a data queue to avoid a stack overflow with hugely nested ASTs.

By default, this visitor tries to visit every part of the explicit source code exactly once.

5. Findings about Name Mangling from Assembly code:

C++ supports function overloading, i.e., there can be more than one function with same name and differences in parameters. C++ compiler distinguishes between different functions when it generates object code by adding information about arguments. This technique of adding additional information to function names is called Name Mangling.

Lets see the assembly code generated for the simple piece of code below:

```
int  f (void) { return 1; }
int  f (int)  { return 0; }
void g (void) { int i = f(), j = f(0); }

int  __f_v (void) { return 1; }
int  __f_i (int)  { return 0; }
void __g_v (void) { int i = __f_v(), j = __f_i(0); }
```

But in C, function names were unchanged as C doesn't support function overloading.

When we link a C code in C++, we have to make sure that name of a symbol is not changed otherwise it will throw a compile error. The solution of problem is extern "C" in C++. When some code is put in extern "C" block, the C++ compiler ensures that the function names are un mangled – that the compiler emits a binary file with their names unchanged, as a C compiler would do.

6. ERROR messages and handling

Error handling in llvm through assert

Lets look at the below example . Error handling is done through either lib/Support/Errorhandling.cpp or LLVMContext Diagnostics or the assert functionality.

```
inline Value *getOperand(unsigned I) {
  assert(I < Operands.size() && "getOperand() out of range!");
  return Operands[I];
}
```

Here the LHS of && contains the expression that has to be evaluated and if the expression turns out to be false , the string that is RHS of && is shown as an error message.

Few more usages of assert:

assert(Ty->isPointerType() && "Can't allocate a non-pointer type!");
Here we are checking the type of Ty if its a pointer

assert((Opcode == Shl || Opcode == Shr) && "ShiftInst Opcode invalid!");

assert(idx < getNumSuccessors() && "Successor # out of range!");
Here we are assuring that the idx is <getNumSuccessors

assert(V1.getType() == V2.getType() && "Constant types must be identical!");

assert(isa<PHINode>(Succ->front()) && "Only works on PHId BBs!");

Other means of error handling

lib/Support/ErrorHandling.cpp to indicate fatal error conditions. Non-fatal (most of them) are handled through LLVMContext.

install_fatal_error_handler ()

Installs a new error handler which is later used for handling errors whenever a serious (non-recoverable) error is encountered by LLVM.

remove_fatal_error_handler():

Restores default error handling behaviour.

report_fatal_error()

Reports a serious error to any installed error handler. These are used for error conditions which are not in the compiler's control.

eg: (I/O errors, invalid user input, etc.)

If no error handler is installed the default is to print the error message to stderr, and call exit(1). If an error handler is installed the handler should log the message, it will no longer be sent to stderr. If the error handler returns, then exit(1) will be called.

llvm_unreachable_internal()

Marks that the current location is not supposed to be reachable.This is better to use instead of assert(0) to convey more information to compiler.

The macro NDEBUG controls how assert behaves. In !NDEBUG builds, the message and location info are printed to stderr. In NDEBUG builds, becomes an optimizer information that the current location is not supposed to be reachable. On compilers that don't support such hints, this prints a reduced message instead.

error handling via LLVMContext;

The Diagnostic Handlers are used to handle the diagnosis reports.There are different diagnostic handlers.

eg:InlineAsmDiagHandlerTy(),DiagnosticHandlerTy()

The Handlers are invoked when the backend wants to communicate to the user using setDiagnosticHandler(),

diagnose(DiagnosticInfo &DI)

Report's the message to the currently installed diagnostic handler.

The Diagnostic handlers can be invoked by using setDiagnosticHandler

The attribute Severity [DI.getSeverity()] is used while printing the errors by prefixing to the diagnostic message.

"error " for Severity =DS_error

"warning" for Severity=DS_warning

emitError - It emits an error message to the currently installed error handler with optional location information. Since this function returns, code should be prepared to drop the erroneous construct and "not crash".

7.LLVM IR

In IR ,instead of a fixed set of registers, IR uses an infinite set of temporaries of the form %0, %1, etc..

It follows static single assignment

The Code is organized as three-address instructions. Data processing instructions have two source operands and place the result in a distinct destination operand.

We observed that the ModuleID is the name of the file it is generated from.Suppose

if hello.ll is generated from hello.c using clang the module id is hello.c if its from hello.bc using llc , module id will be hello.bc

The target data layout construct contains information about endianness and type sizes for target triple that is described in target host. Some optimizations depend on knowing the specific data layout of the target to transform the code correctly.

```
    target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
    define i32 @main() #0 {

    }
```

This means that the function main returns an i32 datatype and takes no arguments .
Note that the #0 is not for the number of arguments.It maps to the function attributes mentioned later.

```
    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" }
```

This is how string constant is stored

@.str = internal constant [12 x i8] c"hello world\00"

Here internal constant stands for static const. The reference [12 x i8] string is later used to get the string "hello world".

The [] here indicates that its an array and here [12 x i8] means that its an array of 12 , 8 bits(charecters)

call i32 (i8*, ...)* @scanf(i8* getelementptr inbounds ([3 x i8]* @.str1, i64 0, i64 0), i32* %n) #2

This is a call to the function scanf .The call keyword is used. call i32 (i8*, ...) means that we are calling a funtion that returns a 32bit int and takes a variable number of i8 pointers (i.e.. charecter pointers)

There wont be load and store depending on the optimisations we use.
Clang -O0 does no optimizations, and the unnecessary loads and stores are not removed. If we compile with -O3 instead, the outcome is a much simpler code.
The following are the 2 IRs. for the sum function
Without optimization.

```
    define i32 @sum(i32 %a, i32 %b) #0 {
      %1 = alloca i32, align 4
      %2 = alloca i32, align 4
      store i32 %a, i32* %1, align 4
      store i32 %b, i32* %2, align 4
      %3 = load i32* %1, align 4
      %4 = load i32* %2, align 4
      %5 = add nsw i32 %3, %4
      ret i32 %5
    }
```

With opt.

```
    define i32 @sum(i32 %a, i32 %b) #0 {
      %1 = add nsw i32 %b, %a
      ret i32 %1
    }
```

8. Compiler Tool Chain and options.

The tool Chain:

The AST generated:

In each of the AST structures dumped, the expression block is treated as a node with branches to different statements in it. It starts a new node as soon as it sees a function declaration . This contains the name of the function, the keyword Functiondecl ,the reference of the function node and the position where the function starts anthe position where it ends and the return type.

FunctionDecl 0x103025c10 <nor.c:3:1, line:16:1> line:3:5 main 'int ()'

The entire block is put under the branch to the FunctionDecl called CompoundStatement

The line:column numbers of the code that a particular node is for tells us the scope of the code which the node corresponds. We observed that the line number is mentioned only if it changes from the parent node or the starting line of the current node.

We observed 2 implicit casts while using printf without return.One because printf returns a type but not taken care and the type of argument should be const char* but "Enter the integer " is char*.

Since strings are immutable it is stored in a char[18]

The same goes with scanf().

A <<NULL>> node is seen at the if statement which is unexplained.

If has got 3 branches one for the evaluation and the other two for the execution branches that are executed depending on what(t/f) the evaluation returns.

For binary operators ,

```
-BinaryOperator 0x103801ba8 <col:27, col:32> 'int' '%'
|   |   |     |-ImplicitCastExpr 0x103801b90 <col:27> 'int' <LValueToRValue>
|   |   |     | `-DeclRefExpr 0x103801b48 <col:27> 'int' lvalue Var 0x10302cbc0 'temp' 'int'
|   |   |     `-IntegerLiteral 0x103801b70 <col:32> 'int' 10
```

The 2 openrands the branches . If one of them is a variable ,then the reference location of the variable is also passed.

In case of multiple declarations

```
DeclStmt 0x10302cc38 <line:5:4, col:28>
|    | |-VarDecl 0x10302cac0 <col:4, col:8> col:8 used n 'int'
|    | |-VarDecl 0x10302cb30 <col:4, col:21> col:11 used reverse 'int' cinit
|    | | `-IntegerLiteral 0x10302cb88 <col:21> 'int' 0
|    | `-VarDecl 0x10302cbc0 <col:4, col:24> col:24 used temp 'int'
```

Here along with declaring int, we are also initialising it to 0 .The cinit gives that indication.

llvm compiler tools and commands

Frontend , Middle end and Optimizer options:

LLVM uses clang as its frontend.

In order to convert a given prog.c to a .bc bit code file

clang -O3 -llvm-emit hello.c -c -o hello.bc

But if we need to convert it into a user readable assembly code

clang -O3 -llvm-emit hello.c -S -o hello.ll

Here we can set the optimization level to 0 or 1 or 3 . In a typical .ll file which is not optimised i.e level 0 , we find the load and store commands which is not seen in the optimised assembly source code.

llvm-as :This command is used to convert the user readable .ll assembly code of llvm to binary code of llvm

llvm-as hello.ll -o hello.bc

llvm-dis :This command is used to convert the binary code of llvm to user readable .ll assembly code of llvm

llvm-dis hello.bc -o hello.ll

After the llvm files are generated by the clang front end, we can use opt command for specific optimisation.

opt:

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.

When -analyze is specified, opt performs various analyses of the input source. It will usually print the results on standard output.

While -analyze is not given, opt attempts to produce an optimized output file

I observed that the opt command output file should be a proper llvm filename (.ll/.bc) Else nothing is seen in the file

```
eg:opt -stats loop1.bc -o vasopt.bc
```

works but

```
opt -stats loop1.bc -o vasopt
```

doesnot work

If we wish to convert a given LLVM source files to assembly language of a particular architecture we can use llc command.The assembly language output can then be passed through a native assembler and linker to generate a native executable.

llc [options] [filename]

march=<arch> option is used to convert into assembly code of that architecture

mcpu=<cpu> converts to a particular chip type [The types of cpu can be found using help]

Programs in LLVM bitcode format can be directly executed using lli. It takes a program in LLVM bitcode format and executes it using a just-in-time compiler or an interpreter.

If a Just is Time compiler is available, it compiles the bitcode using the JIT else it interprets the bitcode.

-force-interepreter [true/false]

This option makes llvm use the interpreter even if a just-in-time compiler is available for this architecture.

There are also options for stats and measuring time for each pass , architecture specific options and floating point prescision options that disable/enable few optimisations.

This also has options for controling the code generation.

Options are available for changing the register allocation scheme , choose the default code style etc..

llvm-link

It takes many LLVM bitcode files and links them together into a single LLVM bitcode file.

There are options for enabling the output to terminal , specifying the filename and writing the output in IR not in bc

llvm-nm

This utility lists the names of symbols from the LLVM bitcode files, object files, or ar archives containing them,

eg:

```
00000000 T _main
         U _printf
         U _puts
         U _scanf
```

T means Global function

U means Named object but undefined in this file. printf ,scanf are having U because they are defined in stdio.h not here

llvm-diff

It compares the structure of two LLVM modules, primarily the differences in function definitions. Insignificant differences, such as changes in the ordering of globals or in the names of local values, are ignored.

function @Fibonacci exists only in left module

in function main:

in block %0 / %0:

```
      <   %1 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0,
i64 0), i32* %n) #3
          %puts = call i32 @puts(i8* getelementptr inbounds ([17 x i8], [17 x i8]* @str, i64 0, i64 0))
      >   %1 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str1, i64
0, i64 0), i32* %n) #2
          %2 = load i32, i32* %n, align 4, !tbaa !1
      >   %3 = icmp eq i32 %2, 0
      >   br i1 %3, label %._crit_edge, label %.lr.ph.preheader
      <   %3 = icmp slt i32 %2, 1
      <   br i1 %3, label %._crit_edge, label %.lr.ph.preheader
```

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.

See appendix for the analyser file:

9.REPORT ON KALEIDOSCOPE

Now, we first describe briefly the topics in the tutorial.The goal is to implement a simple language in llvm.

Firstly we start with defining the lexer and parser rules for the simple language which is not at all related to llvm .In the later stage we construct AST for the language.Now inorder to convert each AST node to a proper LLVM object we add a codegen() method to each AST node .

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 Builder object is a helper function that assists in building the Value Object for LLVM.

Now we need to add the Just In Time Optimisation support . For Kaleidoscope, we are currently generating functions on the fly, as the user types them in. We aren’t targeting the ultimate optimization experience .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.

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.

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.

Extending the Language:

Control flow extensions:

if/then/else:

Lexer:

We add 3 new tokens to the existing token list

```
if (IdentifierStr == "if")
  return tok_if;
if (IdentifierStr == "then")
  return tok_then;
if (IdentifierStr == "else")
  return tok_else;
```

Parser:

The ParseIfExpr extends the ExprAST. We eat the if and parse the condition and next we eat then.If there is no then then we return error else we move on and parse the expression to be executed (then) .Next we raise error if else is not seen .If else is seen we parse the else expression. The Cond,Then,Else are stored.

```
static std::unique_ptr<ExprAST> ParseIfExpr() {
  getNextToken();  // eat the if.

  // condition.
  auto Cond = ParseExpression();
  if (!Cond)
    return nullptr;

  if (CurTok != tok_then)
    return Error("expected then");
  getNextToken();  // eat the then

  auto Then = ParseExpression();
  if (!Then)
    return nullptr;

  if (CurTok != tok_else)
    return Error("expected else");

  getNextToken();

  auto Else = ParseExpression();
  if (!Else)
    return nullptr;

  return llvm::make_unique<IfExprAST>(std::move(Cond), std::move(Then),
                                      std::move(Else));
}
```

AST:

Using the parser , the AST stores the Cond,Then,Else . and creates a node for the if/then/else. The AST node as soon as its created calls the constructera nd the constructer here initialises the cond,Then,Else.

```
class IfExprAST : public ExprAST {
  std::unique_ptr<ExprAST> Cond, Then, Else;
public:
  IfExprAST(std::unique_ptr<ExprAST> Cond, std::unique_ptr<ExprAST> Then,
            std::unique_ptr<ExprAST> Else)
    : Cond(std::move(Cond)), Then(std::move(Then)), Else(std::move(Else)) {}
  virtual Value *codegen();
};
```

LLVM code generation:

We need to generate a code for if/then/else that looks like

```
%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 ]
```

So our codegen() should generate a similar to this one making use of the AST 's information.We first change the condition to a boolean value.After that we call the codegen of the then block.This can change the builder .We need to update the builder after calling the codegen() of then and the same with else too. To the PHI node we add the ThenV,ThenBB and ElseV,ElseBB and return this.

```
Value *IfExprAST::codegen() {
  Value *CondV = Cond->codegen();
  if (!CondV)
    return nullptr;


  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)
    return nullptr;

  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)
    return nullptr;

  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;
}
```

for loop

Lexer:

We add to lexer rules .

```
if (IdentifierStr == "for")
  return tok_for;
```

Parser Extension:

We first eat the for and then expect an identifier.If there is no identifier we flag an error else we eat the identifier and expect the '=' .After '=' is eaten up, we parse the expression and store it in Start.

Next we expect a cmma followed by the end expression which is parsed and stored in End .Next we look for an optimal step. i.e (similar to i++ ,- -i etc).

Now the body of the expression is parsed and is stored in Body .

```
static std::unique_ptr<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 '='.
  auto Start = ParseExpression();
  if (!Start)
```

```
    return nullptr;
if (CurTok != ',')
    return Error("expected ',' after for start value");
  getNextToken();

  auto End = ParseExpression();
  if (!End)
    return nullptr;

  // The step value is optional.
  std::unique_ptr<ExprAST> Step;
  if (CurTok == ',') {
    getNextToken();
    Step = ParseExpression();
    if (!Step)
      return nullptr;
  }

  if (CurTok != tok_in)
    return Error("expected 'in' after for");
  getNextToken();  // eat 'in'.

  auto Body = ParseExpression();
  if (!Body)
    return nullptr;

  return llvm::make_unique<ForExprAST>(IdName, std::move(Start),
                                       std::move(End), std::move(Step),
                                       std::move(Body));
}
```

AST Node:

Using the parser , the AST stores the Start ,End,Step,Body . and creates a node for the FOR. The AST node as soon as its created calls the constructer and the constructer here initialises Start , End , Step , Body.

```
class ForExprAST : public ExprAST {
  std::string VarName;
  std::unique_ptr<ExprAST> Start, End, Step, Body;

public:
  ForExprAST(const std::string &VarName, std::unique_ptr<ExprAST> Start,
             std::unique_ptr<ExprAST> End, std::unique_ptr<ExprAST> Step,
             std::unique_ptr<ExprAST> Body)
    : VarName(VarName), Start(std::move(Start)), End(std::move(End)),
      Step(std::move(Step)), Body(std::move(Body)) {}
  virtual Value *codegen();
};
```

Code Generation

This is how a for loop in LLVM IR looks like.

```
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
}
```

The codegen() would be:

```
// Output for-loop 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:
```

```
Value *ForExprAST::codegen() {
  Value *StartVal = Start->codegen();
  if (!StartVal)
    return nullptr;

  Function *TheFunction = Builder.GetInsertBlock()->getParent();
  BasicBlock *PreheaderBB = Builder.GetInsertBlock();
  BasicBlock *LoopBB =
      BasicBlock::Create(getGlobalContext(), "loop", TheFunction);

  Builder.CreateBr(LoopBB);

  Builder.SetInsertPoint(LoopBB);

  PHINode *Variable = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()),
                                        2, VarName.c_str());
  Variable->addIncoming(StartVal, PreheaderBB);

  Value *OldVal = NamedValues[VarName];
  NamedValues[VarName] = Variable;

  if (!Body->codegen())
    return nullptr;

  Value *StepVal = nullptr;
  if (Step) {
    StepVal = Step->codegen();
    if (!StepVal)
      return nullptr;
  } else {
    StepVal = ConstantFP::get(getGlobalContext(), APFloat(1.0));
  }

  Value *NextVar = Builder.CreateFAdd(Variable, StepVal, "nextvar");
  Value *EndCond = End->codegen();
  if (!EndCond)
    return nullptr;

  EndCond = Builder.CreateFCmpONE(
      EndCond, ConstantFP::get(getGlobalContext(), APFloat(0.0)), "loopcond");

  BasicBlock *LoopEndBB = Builder.GetInsertBlock();
  BasicBlock *AfterBB =
      BasicBlock::Create(getGlobalContext(), "afterloop", TheFunction);

  Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

  Builder.SetInsertPoint(AfterBB);

  Variable->addIncoming(NextVar, LoopEndBB);

  if (OldVal)
    NamedValues[VarName] = OldVal;
  else
    NamedValues.erase(VarName);

  return Constant::getNullValue(Type::getDoubleTy(getGlobalContext()));
}
```

Sources &Bibliography:

http://www.cis.upenn.edu/acg/papers/popl12_vellvm.pdf
http://llvm.org/docs/tutorial/LangImpl5.html#for-loop-expression
http://llvm.org/docs/GettingStarted.html#getting-started-with-llvm

APPENDIX

Analyser of Bitcode file

```
Summary of vasana.bc:
         Total size: 16160b/2020.00B/505W
        Stream type: LLVM IR
  # Toplevel Blocks: 1

Per-block Summary:
  Block ID #0 (BLOCKINFO_BLOCK):
      Num Instances: 1
         Total Size: 672b/84.00B/21W
    Percent of file: 4.1584%
      Num SubBlocks: 0
        Num Abbrevs: 0
        Num Records: 0

  Block ID #8 (MODULE_BLOCK):
      Num Instances: 1
         Total Size: 1837b/229.62B/57W
    Percent of file: 11.3676%
      Num SubBlocks: 9
        Num Abbrevs: 1
        Num Records: 11
    Percent Abbrevs: 0.0000%

   Record Histogram:
           Count    # Bits   %% Abv  Record Kind
               4      420         FUNCTION
               4      348         GLOBALVAR
               1      465         DATALAYOUT
               1      327         TRIPLE
               1       21        VERSION

  Block ID #9 (PARAMATTR_BLOCK):
      Num Instances: 1
         Total Size: 181b/22.62B/5W
    Percent of file: 1.1200%
      Num SubBlocks: 0
        Num Abbrevs: 0
        Num Records: 4
    Percent Abbrevs: 0.0000%

   Record Histogram:
           Count    # Bits   %% Abv  Record Kind
               4       96        ENTRY

  Block ID #10 (PARAMATTR_GROUP_BLOCK_ID):
      Num Instances: 1
         Total Size: 5077b/634.62B/158W
    Percent of file: 31.4171%
      Num SubBlocks: 0
        Num Abbrevs: 0
        Num Records: 4
    Percent Abbrevs: 0.0000%

   Record Histogram:
           Count    # Bits   %% Abv  Record Kind
               4     5010         UnknownCode3

  Block ID #11 (CONSTANTS_BLOCK):
      Num Instances: 2
         Total Size: 1492b/186.50B/46W
    Percent of file: 9.2327%
       Average Size: 746.00/93.25B/23W
  Tot/Avg SubBlocks: 0/0.000000e+00
    Tot/Avg Abbrevs: 4/2.000000e+00
    Tot/Avg Records: 23/1.150000e+01
    Percent Abbrevs: 73.9130%

   Record Histogram:
           Count    # Bits   %% Abv  Record Kind
               9       81 100.00  SETTYPE
               4      232         CE_INBOUNDS_GEP
               4      830 100.00  CSTRING
               4       48 100.00  INTEGER
```

```
            2       32        NULL

Block ID #12 (FUNCTION_BLOCK):
    Num Instances: 1
       Total Size: 1143b/142.88B/35W
  Percent of file: 7.0730%
    Num SubBlocks: 4
      Num Abbrevs: 0
      Num Records: 27
  Percent Abbrevs: 25.9259%

  Record Histogram:
          Count    # Bits   %% Abv  Record Kind
            7       190        INST_BR
            5       114 100.00  INST_BINOP
            4       310        INST_CALL
            4       178        INST_PHI
            3       120        INST_CMP2
            1        20 100.00  INST_LOAD
            1        46       INST_ALLOCA
            1        10 100.00  INST_RET
            1        22       DECLAREBLOCKS

Block ID #14 (VALUE_SYMTAB):
    Num Instances: 2
       Total Size: 1357b/169.62B/42W
  Percent of file: 8.3973%
     Average Size: 678.50/84.81B/21W
Tot/Avg SubBlocks: 0/0.000000e+00
  Tot/Avg Abbrevs: 0/0.000000e+00
  Tot/Avg Records: 19/9.500000e+00
  Percent Abbrevs: 100.0000%

  Record Histogram:
          Count    # Bits   %% Abv  Record Kind
           15       816 100.00  ENTRY
            4       390 100.00  BBENTRY

Block ID #15 (METADATA_BLOCK):
    Num Instances: 2
       Total Size: 3434b/429.25B/107W
  Percent of file: 21.2500%
     Average Size: 1717.00/214.62B/53W
Tot/Avg SubBlocks: 0/0.000000e+00
  Tot/Avg Abbrevs: 2/1.000000e+00
  Tot/Avg Records: 26/1.300000e+01
  Percent Abbrevs: 19.2308%

  Record Histogram:
          Count    # Bits   %% Abv  Record Kind
           14      2094        KIND
            5       141        NODE
            4       850 100.00  STRING
            1        21       NAMED_NODE
            1     89 100.00  NAME
            1     27         VALUE

Block ID #16 (METADATA_ATTACHMENT_BLOCK):
    Num Instances: 1
       Total Size: 116b/14.50B/3W
  Percent of file: 0.7178%
    Num SubBlocks: 0
      Num Abbrevs: 0
      Num Records: 1
  Percent Abbrevs: 0.0000%

  Record Histogram:
          Count    # Bits   %% Abv  Record Kind
            1     33         ATTACHMENT

Block ID #17 (TYPE_BLOCK_ID):
    Num Instances: 1
       Total Size: 661b/82.62B/20W
  Percent of file: 4.0903%
    Num SubBlocks: 0
      Num Abbrevs: 6
      Num Records: 24
  Percent Abbrevs: 66.6667%

  Record Histogram:
          Count    # Bits   %% Abv  Record Kind
            9     81 100.00  POINTER
            4     68 100.00  ARRAY
            4    100         INTEGER
            3     58 100.00  FUNCTION
            1     16         METADATA
            1     16         LABEL
            1     16         VOID
            1     22         NUMENTRY

Block ID #18 (USELIST_BLOCK_ID):
    Num Instances: 1
       Total Size: 148b/18.50B/4W
  Percent of file: 0.9158%
    Num SubBlocks: 0
      Num Abbrevs: 0
      Num Records: 1
  Percent Abbrevs: 0.0000%

  Record Histogram:
          Count    # Bits   %% Abv  Record Kind
            1     69         USELIST_CODE_DEFAULT
```