LLVM Cookbook

LLVM is a compiler framework with libraries that provides a modern source-and target-independent optimizer, along with a code generator.

This book not only explains the effective use of the compiler infrastructure that LLVM provides, but also helps you implement it in one of your projects. You start with a simple task to get you up-and-running with LLVM, followed by learning the process of writing a frontend for a language, which includes writing a lexer, a parser, and generating IR code. You will then see how to implement optimizations at different levels, generate target-independent code, and then map this generated code to a backend. Finally, you will look into the functionalities that the LLVM infrastructure provides, such as exception handling, LLVM Utility Passes, using sanitizers, the garbage collector, and how we can use these in our projects.

What this book will do for you...

- Introduction to LLVM modular design and LLVM tools
- Write a frontend for a language
- Add JIT support and use frontends for different languages
- Learn about the LLVM Pass infrastructure and the LLVM Pass Manager
- Create analyses and transform optimization passes
- Build a LLVM TOY backend from scratch
- Optimize the code at SelectionDAG level and allocate registers to variables

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving problems efficiently
- Clear explanations of what you did
- Apply solutions to other situations

Over 80 engaging recipes that will help you build a compiler frontend, optimizer, and code generator using LLVM

In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 1 'LLVM Design and Use'
- A synopsis of the book’s content
- More information on LLVM Cookbook

About the Authors

**Mayur Pandey** is a professional software engineer and an open source enthusiast. He focuses on compiler development and compiler tools. He is an active contributor to the LLVM open source community. He has been part of the compiler team for the Tizen project, and has hands-on experience with other proprietary compilers.

Mayur earned a bachelor's degree in information technology from Motilal Nehru National Institute of Technology Allahabad, India. Currently, he lives in Bengaluru, India.

**Suyog Sarda** is a professional software engineer and an open source enthusiast. He focuses on compiler development and compiler tools. He is an active contributor to the LLVM open source community. He has been part of the compiler team for the Tizen project. Suyog was also involved in code performance improvements for the ARM and the x86 architecture. He has hands-on experience in other proprietary compilers. His interest in compiler development lies more in code optimization and vectorization.

Apart from compilers, Suyog is also interested in Linux kernel development. He has published a technical paper titled Secure Co-resident Virtualization in Multicore Systems by VM Pinning and Page Coloring at the IEEE Proceedings of the 2012 International Conference on Cloud Computing, Technologies, Applications, and Management at Birla Institute of Technology, Dubai. He earned a bachelor's degree in computer technology from College of Engineering, Pune, India. Currently, he lives in Bengaluru, India.
LLVM Cookbook

A programmer might have come across compilers at some or the other point when programming. Simply speaking, a compiler converts a human-readable, high-level language into machine-executable code. But have you ever wondered what goes on under the hood? A compiler does lot of processing before emitting optimized machine code. Lots of complex algorithms are involved in writing a good compiler.

This book travels through all the phases of compilation: frontend processing, code optimization, code emission, and so on. And to make this journey easy, LLVM is the simplest compiler infrastructure to study. It's a modular, layered compiler infrastructure where every phase is dished out as a separate recipe. Written in object-oriented C++, LLVM gives programmers a simple interface and lots of APIs to write their own compiler.

As authors, we maintain that simple solutions frequently work better than complex solutions; throughout this book, we'll look at a variety of recipes that will help develop your skills, make you consider all the compiling options, and understand that there is more to simply compiling code than meets the eye.

We also believe that programmers who are not involved in compiler development will benefit from this book, as knowledge of compiler implementation will help them code optimally next time they write code.

We hope you will find the recipes in this book delicious, and after tasting all the recipes, you will be able to prepare your own dish of compilers. Feeling hungry? Let's jump into the recipes!

What This Book Covers

Chapter 1, LLVM Design and Use, introduces the modular world of LLVM infrastructure, where you learn how to download and install LLVM and Clang. In this chapter, we play with some examples to get accustomed to the workings of LLVM. We also see some examples of various frontends.

Chapter 2, Steps in Writing a Frontend, explains the steps to write a frontend for a language. We will write a bare-metal toy compiler frontend for a basic toy language. We will also see how a frontend language can be converted into the LLVM intermediate representation (IR).

Chapter 3, Extending the Frontend and Adding JIT Support, explores the more advanced features of the toy language and the addition of JIT support to the frontend. We implement some powerful features of a language that are found in most modern programming languages.
Chapter 4, *Preparing Optimizations*, takes a look at the pass infrastructure of the LLVM IR. We explore various optimization levels, and the optimization techniques kicking at each level. We also see a step-by-step approach to writing our own LLVM pass.

Chapter 5, *Implementing Optimizations*, demonstrates how we can implement various common optimization passes on LLVM IR. We also explore some vectorization techniques that are not yet present in the LLVM open source code.

Chapter 6, *Target-independent Code Generator*, takes us on a journey through the abstract infrastructure of a target-independent code generator. We explore how LLVM IR is converted to Selection DAGs, which are further processed to emit target machine code.

Chapter 7, *Optimizing the Machine Code*, examines how Selection DAGs are optimized and how target registers are allocated to variables. This chapter also describes various optimization techniques on Selection DAGs as well as various register allocation techniques.

Chapter 8, *Writing an LLVM Backend*, takes us on a journey of describing a target architecture. This chapter covers how to describe registers, instruction sets, calling conventions, encoding, subtarget features, and so on.

Chapter 9, *Using LLVM for Various Useful Projects*, explores various other projects where LLVM IR infrastructure can be used. Remember that LLVM is not just a compiler; it is a compiler infrastructure. This chapter explores various projects that can be applied to a code snippet to get useful information from it.
Introduction

In this recipe, you get to know about LLVM, its design, and how we can make multiple uses out of the various tools it provides. You will also look into how you can transform a simple C code to the LLVM intermediate representation and how you can transform it into various forms. You will also learn how the code is organized within the LLVM source tree and how can you use it to write a compiler on your own later.
Understanding modular design

LLVM is designed as a set of libraries unlike other compilers such as GNU Compiler Collection (GCC). In this recipe, LLVM optimizer will be used to understand this design. As LLVM optimizer's design is library-based, it allows you to order the passes to be run in a specified order. Also, this design allows you to choose which optimization passes you can run—that is, there might be a few optimizations that might not be useful to the type of system you are designing, and only a few optimizations will be specific to the system. When looking at traditional compiler optimizers, they are built as a tightly interconnected mass of code, that is difficult to break down into small parts that you can understand and use easily. In LLVM, you need not know about how the whole system works to know about a specific optimizer. You can just pick one optimizer and use it without having to worry about other components attached to it.

Before we go ahead and look into this recipe, we must also know a little about LLVM assembly language. The LLVM code is represented in three forms: in memory compiler Intermediate Representation (IR), on disk bytecode representation, and as human readable assembly. LLVM is a Static Single Assignment (SSA)-based representation that provides type safety, low level operations, flexibility, and the capability to represent all the high-level languages cleanly. This representation is used throughout all the phases of LLVM compilation strategy. The LLVM representation aims to be a universal IR by being at a low enough level that high-level ideas may be cleanly mapped to it. Also, LLVM assembly language is well formed. If you have any doubts about understanding the LLVM assembly mentioned in this recipe, refer to the link provided in the See also section at the end of this recipe.

Getting ready

We must have installed the LLVM toolchain on our host machine. Specifically, we need the opt tool.

How to do it...

We will run two different optimizations on the same code, one-by-one, and see how it modifies the code according to the optimization we choose.

1. First of all, let us write a code we can input for these optimizations. Here we will write it into a file named testfile.ll:

   $ cat testfile.ll
   define i32 @test1(i32 %A) {
     %B = add i32 %A, 0
     ret i32 %B
   }
define internal i32 @test(i32 %X, i32 %dead) {
    ret i32 %X
}

define i32 @caller() {
    %A = call i32 @test(i32 123, i32 456)
    ret i32 %A
}

2. Now, run the opt tool for one of the optimizations—that is, for combining the instruction:
   $ opt -S -instcombine testfile.ll -o output1.ll

3. View the output to see how instcombine has worked:
   $ cat output1.ll
   ; ModuleID = 'testfile.ll'

   define i32 @test1(i32 %A) {
       ret i32 %A
   }

   define internal i32 @test(i32 %X, i32 %dead) {
       ret i32 %X
   }

   define i32 @caller() {
       %A = call i32 @test(i32 123, i32 456)
       ret i32 %A
   }

4. Run the opt command for dead argument elimination optimization:
   $ opt -S -deadargelim testfile.ll -o output2.ll
5. View the output, to see how deadargelim has worked:

    $ cat output2.ll
    ; ModuleID = testfile.ll'

    define i32 @test1(i32 %A) {
        %B = add i32 %A, 0
        ret i32 %B
    }

    define internal i32 @test(i32 %X) {
        ret i32 %X
    }

    define i32 @caller() {
        %A = call i32 @test(i32 123)
        ret i32 %A
    }

How it works...

In the preceding example, we can see that, for the first command, the instcombine pass is run, which combines the instructions and hence optimizes %B = add i32 %A, 0; ret i32 %B to ret i32 %A without affecting the code.

In the second case, when the deadargelim pass is run, we can see that there is no modification in the first function, but the part of code that was not modified last time gets modified with the function arguments that are not used getting eliminated.

LLVM optimizer is the tool that provided the user with all the different passes in LLVM. These passes are all written in a similar style. For each of these passes, there is a compiled object file. Object files of different passes are archived into a library. The passes within the library are not strongly connected, and it is the LLVM PassManager that has the information about dependencies among the passes, which it resolves when a pass is executed. The following image shows how each pass can be linked to a specific object file within a specific library. In the following figure, the PassA references LLVMPasses.a for PassA.o, whereas the custom pass refers to a different library MyPasses.a for the MyPass.o object file.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
There's more...

Similar to the optimizer, the LLVM code generator also makes use of its modular design, splitting the code generation problem into individual passes: instruction selection, register allocation, scheduling, code layout optimization, and assembly emission. Also, there are many built-in passes that are run by default. It is up to the user to choose which passes to run.

See also

- In the upcoming chapters, we will see how to write our own custom pass, where we can choose which of the optimization passes we want to run and in which order. Also, for a more detailed understanding, refer to [http://www.aosabook.org/en/llvm.html](http://www.aosabook.org/en/llvm.html).
- To understand more about LLVM assembly language, refer to [http://llvm.org/docs/LangRef.html](http://llvm.org/docs/LangRef.html).

Cross-compiling Clang/LLVM

By cross-compiling we mean building a binary on one platform (for example, x86) that will be run on another platform (for example, ARM). The machine on which we build the binary is called the host, and the machine on which the generated binary will run is called the target. The compiler that builds code for the same platform on which it is running (the host and target platforms are the same) is called a native assembler, whereas the compiler that builds code for a target platform different from the host platform is called a cross-compiler.
In this recipe, cross-compilation of LLVM for a platform different than the host platform will be shown, so that you can use the built binaries for the required target platform. Here, cross-compiling will be shown using an example where cross-compilation from host platform x86_64 for target platform ARM will be done. The binaries thus generated can be used on a platform with ARM architecture.

**Getting ready**

The following packages need to be installed on your system (host platform):

- cmake
- ninja-build (from backports in Ubuntu)
- gcc-4.x-arm-linux-gnueabihf
- gcc-4.x-multilib-arm-linux-gnueabihf
- binutils-arm-linux-gnueabihf
- libgcc1-armhf-cross
- libstdc++-6-armhf-cross
- libstdc++-6-4.x-dev-armhf-cross
- install llvm on your host platform

**How to do it...**

To compile for the ARM target from the host architecture, that is X86_64 here, you need to perform the following steps:

1. Add the following cmake flags to the normal cmake build for LLVM:

   ```
   -DCMAKE_CROSSCOMPILING=True
   -DCMAKE_INSTALL_PREFIX= path-where-you-want-the-toolchain(optional)
   -DLLVM_TABLEGEN=<path-to-host-installed-llvm-toolchain-bin>/llvm-tblgen
   -DCLANG_TABLEGEN=< path-to-host-installed-llvm-toolchain-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.x.x/arm-linux-gnueabihf/ -I/usr/arm-linux-gnueabihf/include/ -mfloat-abi=hard -ccc-gcc-name arm-linux-gnueabihf-gcc'
   ```
2. If using your platform compiler, run:

   ```
   $ cmake -G Ninja <llvm-source-dir> <options above>
   ```

   If 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 work fine.

3. To build LLVM, simply type:

   ```
   $ ninja
   ```

4. After the LLVM/Clang has built successfully, install it with the following command:

   ```
   $ ninja install
   ```

   This will create a sysroot on the install-dir location if you have specified the DCMAKE_INSTALL_PREFIX options.

**How it works...**

The cmake package builds the toolchain for the required platform by making the use of option flags passed to cmake, and the tblgen tools are used to translate the target description files into C++ code. Thus, by using it, the information about targets is obtained, for example—what instructions are available on the target, the number of registers, and so on.

If Clang is used as the cross-compiler, there is a problem in the LLVM ARM backend that produces absolute relocations on position-independent code (PIC), so as a workaround, disable PIC at the moment.

The ARM libraries will not be available on the host system. So, either download a copy of them or build them on your system.

**Converting a C source code to LLVM assembly**

Here we will convert a C code to intermediate representation in LLVM using the C frontend Clang.
Getting ready

Clang must be installed in the PATH.

How to do it...

1. Let's create a C code in the multiply.c file, which will look something like the following:
   
   ```
   $ cat multiply.c
   int mult() {
   int a = 5;
   int b = 3;
   int c = a * b;
   return c;
   }
   ```

2. Use the following command to generate LLVM IR from the C code:
   
   ```
   $ clang -emit-llvm -S multiply.c -o multiply.ll
   ```

3. Have a look at the generated IR:
   
   ```
   $ cat multiply.ll
   ; ModuleID = 'multiply.c'
   target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
   target triple = "x86_64-unknown-linux-gnu"

   ; Function Attrs: nounwind uwtable
define i32 @mult() #0 {
   %a = alloca i32, align 4
   %b = alloca i32, align 4
   %c = alloca i32, align 4
   store i32 5, i32* %a, align 4
   store i32 3, i32* %b, align 4
   %1 = load i32* %a, align 4
   %2 = load i32* %b, align 4
   %3 = mul nsw i32 %1, %2
   store i32 %3, i32* %c, align 4
   %4 = load i32* %c, align 4
   ret i32 %4
   }
   ```
We can also use the `cc1` for generating IR:

```
$ clang -cc1 -emit-llvm testfile.c -o testfile.ll
```

**How it works...**

The process of C code getting converted to IR starts with the process of lexing, wherein the C code is broken into a token stream, with each token representing an Identifier, Literal, Operator, and so on. This stream of tokens is fed to the parser, which builds up an abstract syntax tree with the help of **Context free grammar (CFG)** for the language. Semantic analysis is done afterwards to check whether the code is semantically correct, and then we generate code to IR.

Here we use the Clang frontend to generate the IR file from C code.

**See also**

- In the next chapter, we will see how the lexer and parser work and how code generation is done. To understand the basics of LLVM IR, you can refer to [http://llvm.org/docs/LangRef.html](http://llvm.org/docs/LangRef.html).

**Converting IR to LLVM bitcode**

In this recipe, you will learn to generate LLVM bit code from IR. The LLVM bit code file format (also known as bytecode) is actually two things: a bitstream container format and an encoding of LLVM IR into the container format.

**Getting Ready**

The `llvm-as` tool must be installed in the PATH.

**How to do it...**

Do the following steps:

1. First create an IR code that will be used as input to `llvm-as`:
   ```
   $ cat test.ll
   define i32 @mult(i32 %a, i32 %b) #0 {
     %1 = mul nsw i32 %a, %b
     ret i32 %1
   }
   ```

2. Now use `llvm-as` to generate the bit code:
   ```
   $ llvm-as test.ll
   ```
2. To convert LLVM IR in `test.ll` to bitcode format, you need to use the following command:

   `llvm-as test.ll -o test.bc`

3. The output is generated in the `test.bc` file, which is in bit stream format; so, when we want to have a look at output in text format, we get it as shown in the following screenshot:

Since this is a bitcode file, the best way to view its content would be by using the `hexdump` tool. The following screenshot shows the output of `hexdump`:
How it works...

The `llvm-as` is the LLVM assembler. It converts the LLVM assembly file that is the LLVM IR into LLVM bitcode. In the preceding command, it takes the `test.ll` file as the input and outputs, and `test.bc` as the bitcode file.

There's more...

To encode LLVM IR into bitcode, the concept of blocks and records is used. Blocks represent regions of bitstream, for example—a function body, symbol table, and so on. Each block has an ID specific to its content (for example, function bodies in LLVM IR are represented by ID 12). Records consist of a record code and an integer value, and they describe the entities within the file such as instructions, global variable descriptors, type descriptions, and so on.

Bitcode files for LLVM IR might be wrapped in a simple wrapper structure. This structure contains a simple header that indicates the offset and size of the embedded BC file.

See also

- To get a detailed understanding of the LLVM the bitstream file format, refer to http://llvm.org/docs/BitCodeFormat.html#abstract

Converting LLVM bitcode to target machine assembly

In this recipe, you will learn how to convert the LLVM bitcode file to target specific assembly code.

Getting ready

The LLVM static compiler `llc` should be in installed from the LLVM toolchain.

How to do it...

Do the following steps:

1. The bitcode file created in the previous recipe, `test.bc`, can be used as input to `llc` here. Using the following command, we can convert LLVM bitcode to assembly code:

   ```bash
   $ llc test.bc -o test.s
   ```
2. The output is generated in the `test.s` file, which is the assembly code. To have a look at that, use the following command lines:

```bash
$ cat test.s
.text
.file "test.bc"
.globl mult
.align 16, 0x90
.type mult,@function
mult:
    # @mult
    .cfi_startproc
    # BB#0:
    Pushq %rbp
    .Ltmp0:
    .cfi_def_cfa_offset 16
    .Ltmp1:
    .cfi_offset %rbp, -16
    movq %rsp, %rbp
    .Ltmp2:
    .cfi_def_cfa_register %rbp
    imull %esi, %edi
    movl %edi, %eax
    popq %rbp
    retq
    .Ltmp3:
    .size mult, .Ltmp3-mult
    .cfi_endproc
```

3. You can also use Clang to dump assembly code from the bitcode file format. By passing the `-S` option to Clang, we get `test.s` in assembly format when the `test.bc` file is in bitstream file format:

```bash
$ clang -S test.bc -o test.s -fomit-frame-pointer # using the clang front end
```

The `test.s` file output is the same as that of the preceding example. We use the additional option `fomit-frame-pointer`, as Clang by default does not eliminate the frame pointer whereas `llc` eliminates it by default.
How it works...

The `llc` command compiles LLVM input into assembly language for a specified architecture. If we do not mention any architecture as in the preceding command, the assembly will be generated for the host machine where the `llc` command is being used. To generate executable from this assembly file, you can use assembler and linker.

There's more...

By specifying `-march=architecture` flag in the preceding command, you can specify the target architecture for which the assembly needs to be generated. Using the `-mcpu=cpu` flag setting, you can specify a CPU within the architecture to generate code. Also by specifying `-regalloc=basic/greedy/fast/pbqp`, you can specify the type of register allocation to be used.

Converting LLVM bitcode back to LLVM assembly

In this recipe, you will convert LLVM bitcode back to LLVM IR. Well, this is actually possible using the LLVM disassembler tool called `llvm-dis`.

Getting ready

To do this, you need the `llvm-dis` tool installed.

How to do it...

To see how the bitcode file is getting converted to IR, use the `test.bc` file generated in the recipe Converting IR to LLVM Bitcode. The `test.bc` file is provided as the input to the `llvm-dis` tool. Now proceed with the following steps:

1. Using the following command shows how to convert a bitcode file to an the one we had created in the IR file:
   ```bash
   $ llvm-dis test.bc -o test.ll
   ```

2. Have a look at the generated LLVM IR by the following:
   ```bash
   $ cat test.ll
   ; ModuleID = 'test.bc'
   define i32 @mult(i32 %a, i32 %b) #0 {
     %1 = mul nsw i32 %a, %b
   }
   ```
ret i32 %l
}

The output test.ll file is the same as the one we created in the recipe Converting IR to LLVM Bitcode.

How it works...

The llvm-dis command is the LLVM disassembler. It takes an LLVM bitcode file and converts it into LLVM assembly language.

Here, the input file is test.bc, which is transformed to test.ll by llvm-dis.

If the filename is omitted, llvm-dis reads its input from standard input.

Transforming LLVM IR

In this recipe, we will see how we can transform the IR from one form to another using the opt tool. We will see different optimizations being applied to IR code.

Getting ready

You need to have the opt tool installed.

How to do it...

The opt tool runs the transformation pass as in the following command:

$opt -passname input.ll -o output.ll

1. Let's take an actual example now. We create the LLVM IR equivalent to the C code used in the recipe Converting a C source code to LLVM assembly:

$ cat multiply.c
int mult() {
    int a =5;
    int b = 3;
    int c = a * b;
    return c;
}
2. Converting and outputting it, we get the unoptimized output:

   $ clang -emit-llvm -S multiply.c -o multiply.ll
   $ cat multiply.ll

   ; ModuleID = 'multiply.c'
   target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
   target triple = "x86_64-unknown-linux-gnu"

   ; Function Attrs: nounwind uwtable
   define i32 @mult() #0 {
     %a = alloca i32, align 4
     %b = alloca i32, align 4
     %c = alloca i32, align 4
     store i32 5, i32* %a, align 4
     store i32 3, i32* %b, align 4
     %1 = load i32* %a, align 4
     %2 = load i32* %b, align 4
     %3 = mul nsw i32 %1, %2
     store i32 %3, i32* %c, align 4
     %4 = load i32* %c, align 4
     ret i32 %4
   }

3. Now use the opt tool to transform it to a form where memory is promoted to register:

   $ opt -mem2reg -S multiply.ll -o multiply1.ll
   $ cat multiply1.ll

   ; ModuleID = 'multiply.ll'
   target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
   target triple = "x86_64-unknown-linux-gnu"

   ; Function Attrs: nounwind uwtable
   define i32 @mult(i32 %a, i32 %b) #0 {
     %1 = mul nsw i32 %a, %b
     ret i32 %1
   }
How it works...

The opt, LLVM optimizer, and analyzer tools take the input.ll file as the input and run the pass passname on it. The output after running the pass is obtained in the output.ll file that contains the IR code after the transformation. There can be more than one pass passed to the opt tool.

There's more...

When the -analyze option is passed to opt, it performs various analyses of the input source and prints results usually on the standard output or standard error. Also, the output can be redirected to a file when it is meant to be fed to another program.

When the -analyze option is not passed to opt, it runs the transformation passes meant to optimize the input file.

Some of the important transformations are listed as follows, which can be passed as a flag to the opt tool:

- adce: Aggressive Dead Code Elimination
- bb-vectorize: Basic-Block Vectorization
- constprop: Simple constant propagation
- dce: Dead Code Elimination
- deadargelim: Dead Argument Elimination
- globaldce: Dead Global Elimination
- globalopt: Global Variable Optimizer
- gvn: Global Value Numbering
- inline: Function Integration/Inlining
- instcombine: Combine redundant instructions
- licm: Loop Invariant Code Motion
- loop: unswitch: Unswitch loops
- loweratomic: Lower atomic intrinsics to non-atomic form
- lowerinvoke: Lower invokes to calls, for unwindless code generators
- lowerswitch: Lower SwitchInsts to branches
- mem2reg: Promote Memory to Register
- memcpyopt: MemCpy Optimization
- simplifycfg: Simplify the CFG
- sink: Code sinking
- tailcallelim: Tail Call Elimination
Run at least some of the preceding passes to get an understanding of how they work. To get to the appropriate source code on which these passes might be applicable, go to the `llvm/test/Transforms` directory. For each of the above mentioned passes, you can see the test codes. Apply the relevant pass and see how the test code is getting modified.

To see the mapping of how C code is converted to IR, after converting the C code to IR, as discussed in an earlier recipe Converting a C source code to LLVM assembly, run the `mem2reg` pass. It will then help you understand how a C instruction is getting mapped into IR instructions.

## Linking LLVM bitcode

In this section, you will link previously generated `.bc` files to get one single bitcode file containing all the needed references.

### Getting ready

To link the `.bc` files, you need the `llvm-link` tool.

### How to do it...

Do the following steps:

1. To show the working of `llvm-link`, first write two codes in different files, where one makes a reference to the other:

   ```
   $ cat test1.c
   int func(int a) {
     a = a*2;
     return a;
   }
   $ cat test2.c
   #include<stdio.h>
   extern int func(int a);
   int main() {
     int num = 5;
     num = func(num);
     printf("number is %d\n", num);
     return num;
   }
   ```
2. Using the following formats to convert this C code to bitstream file format, first convert to .ll files, then from .ll files to .bc files:

   $ clang -emit-llvm -S test1.c -o test1.ll
   $ clang -emit-llvm -S test2.c -o test2.ll
   $ llvm-as test1.ll -o test1.bc
   $ llvm-as test2.ll -o test2.bc

   We get test1.bc and test2.bc with test2.bc making a reference to func syntax in the test1.bc file.

3. Invoke the llvm-link command in the following way to link the two LLVM bitcode files:

   $ llvm-link test1.bc test2.bc –o output.bc

We provide multiple bitcode files to the llvm-link tool, which links them together to generate a single bitcode file. Here, output.bc is the generated output file. We will execute this bitcode file in the next recipe Executing LLVM bitcode.

**How it works...**

The llvm-link works using the basic functionality of a linker—that is, if a function or variable referenced in one file is defined in the other file, it is the job of linker to resolve all the references made in a file and defined in the other file. But note that this is not the traditional linker that links various object files to generate a binary. The llvm-link tool links bitcode files only.

In the preceding scenario, it is linking test1.bc and test2.bc files to generate the output.bc file, which has references resolved.

After linking the bitcode files, we can generate the output as an IR file by giving –S option to the llvm-link tool.

### Executing LLVM bitcode

In this recipe, you will execute the LLVM bitcode that was generated in previous recipes.

### Getting ready

To execute the LLVM bitcode, you need the lli tool.
How to do it...

We saw in the previous recipe how to create a single bitstream file after linking the two .bc files with one referencing the other to define func. By invoking the lli command in the following way, we can execute the output.bc file generated. It will display the output on the standard output:

```bash
$ lli output.bc
    number is 10
```

The output.bc file is the input to lli, which will execute the bitcode file and display the output, if any, on the standard output. Here the output is generated as number is 10, which is a result of the execution of the output.bc file formed by linking test1.c and test2.c in the previous recipe. The main function in the test2.c file calls the function func in the test1.c file with integer 5 as the argument to the function. The func function doubles the input argument and returns the result to main the function that outputs it on the standard output.

How it works...

The lli tool command executes the program present in LLVM bitcode format. It takes the input in LLVM bitcode format and executes it using a just-in-time compiler, if there is one available for the architecture, or an interpreter.

If lli is making use of a just-in-time compiler, then it effectively takes all the code generator options as that of llc.

See also

- The Adding JIT support for a language recipe in Chapter 3, Extending the Frontend and Adding JIT support.

Using the C frontend Clang

In this recipe, you will get to know how the Clang frontend can be used for different purposes.

Getting ready

You will need Clang tool.
Clang can be used as the high-level compiler driver. Let us show it using an example:

1. Create a hello world C code, test.c:
   
   ```
   $ cat test.c
   #include<stdio.h>
   int main() {
   printf("hello world\n");
   return 0; }
   ```

2. Use Clang as a compiler driver to generate the executable a.out file, which on execution gives the output as expected:
   
   ```
   $ clang test.c
   $ ./a.out
   hello world
   ```
   Here the test.c file containing C code is created. Using Clang we compile it and produce an executable that on execution gives the desired result.

3. Clang can be used in preprocessor only mode by providing the -E flag. In the following example, create a C code having a #define directive defining the value of MAX and use this MAX as the size of the array you are going to create:
   
   ```
   $ cat test.c
   #define MAX 100
   void func() {
   int a[MAX];
   }
   ```

4. Run the preprocessor using the following command, which gives the output on standard output:
   
   ```
   $ clang test.c -E
   # 1 "test.c"
   # 1 "<built-in>" 1
   # 1 "<built-in>" 3
   # 308 "<built-in>" 3
   # 1 "<command line>" 1
   # 1 "<built-in>" 2
   # 1 "test.c" 2
   ```
void func() {
    int a[100];
}

In the `test.c` file, which will be used in all the subsequent sections of this recipe, MAX is defined to be 100, which on preprocessing is substituted to MAX in a [MAX], which becomes a[100].

5. You can print the AST for the `test.c` file from the preceding example using the following command, which displays the output on standard output:

```bash
$ clang -cc1 test.c -ast-dump
TranslationUnitDecl 0x3f72c50 <<invalid sloc>> <invalid sloc>
  | -TypedefDecl 0x3f73148 <<invalid sloc>> <invalid sloc> implicit
  |          _int128_t '__int128'
  | -TypedefDecl 0x3f731a8 <<invalid sloc>> <invalid sloc> implicit
  |          _uint128_t 'unsigned __int128'
  | -TypedefDecl 0x3f73518 <<invalid sloc>> <invalid sloc> implicit
  |          __builtin_va_list '__va_list_tag [1]'
  | -FunctionDecl 0x3f735b8 <test.c:3:1, line:5:1> line:3:6 func
    |   'void ()'
    |   `-CompoundStmt 0x3f73790 <col:13, line:5:1>
    |      `-DeclStmt 0x3f73778 <line:4:1, col:11>
    |        `-VarDecl 0x3f73718 <col:1, col:10> col:5 a 'int [100]'
```

Here, the `–cc1` option ensures that only the compiler front-end should be run, not the driver, and it prints the AST corresponding to the `test.c` file code.

6. You can generate the LLVM assembly for the `test.c` file in previous examples, using the following command:

```bash
$ clang test.c -S -emit-llvm -o -
; ModuleID = '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: nounwind uwtable
|define void @func() #0 {
|    | %a = alloca [100 x i32], align 16
|    | ret void
|
```

The `–S` and `–emit-llvm` flag ensure the LLVM assembly is generated for the `test.c` code.
7. To get machine code use for the same `test.c` testcode, pass the `-S` flag to Clang. It generates the output on standard output because of the option `-o -`:

```
$ clang -S test.c -o -
.text
.file "test.c"
.globl func
.align 16, 0x90
.type func,@function

func: # @func
    .cfi_startproc

# BB#0:
    pushq %rbp
.Ltmp0:
    .cfi_def_cfa_offset 16
.Ltmp1:
    .cfi_offset %rbp, -16
    movq %rsp, %rbp
.Ltmp2:
    .cfi_def_cfa_register %rbp
    popq %rbp
    retq
.Ltmp3:
    .size func, .Ltmp3-func
    .cfi_endproc
```

When the `-S` flag is used alone, machine code is generated by the code generation process of the compiler. Here, on running the command, machine code is output on the standard output as we use `-o -` options.

**How it works...**

Clang works as a preprocessor, compiler driver, frontend, and code generator in the preceding examples, thus giving the desired output as per the input flag given to it.

**See also**

- This was a basic introduction to how Clang can be used. There are also many other flags that can be passed to Clang, which makes it perform different operation. To see the list, use Clang `--help`. 
Using the GO frontend

The llgo compiler is the LLVM-based frontend for Go written in Go language only. Using this frontend, we can generate the LLVM assembly code from a program written in Go.

Getting ready

You need to download the llgo binaries or build llgo from the source code and add the binaries in the PATH file location as configured.

How to do it...

Do the following steps:

1. Create a Go source file, for example, that will be used for generating the LLVM assembly using llgo. Create test.go:

   ```
   $ cat test.go
   package main
   import "fmt"
   func main() {
      fmt.Println("Test Message")
   }
   ```

2. Now, use llgo to get the LLVM assembly:

   ```
   $llgo -dump test.go
   ; ModuleID = 'main'
   target datalayout = "e-p:64:64:64..."
   target triple = "x86_64-unknown-linux"
   %0 = type { i8*, i8* }
   ....
   ```

How it works...

The llgo compiler is the frontend for the Go language; it takes the test.go program as its input and emits the LLVM IR.

See also

- For information about how to get and install llgo, refer to https://github.com/go-llvm/llgo
Using DragonEgg

Dragonegg is a gcc plugin that allows gcc to make use of the LLVM optimizer and code generator instead of gcc's own optimizer and code generator.

Getting ready

You need to have gcc 4.5 or above, with the target machine being x86-32/x86-64 and an ARM processor. Also, you need to download the dragonegg source code and build the dragonegg.so file.

How to do It...

Do the following steps:

1. Create a simple hello world program:

   $ cat testprog.c
   #include<stdio.h>
   int main() {
     printf("hello world");
   }

2. Compile this program with your gcc; here we use gcc-4.5:

   $ gcc testprog.c -S -O1 -o -
   .file  " testprog.c"
   .section  .rodata.str1.l,"aMS",@progbits,1
   .LC0:
   .string  "Hello world!"
   .text
   .globl main
   .type  main, @function
   main:
   subq  $8, %rsp
   movl  $.LC0, %edi
   call  puts
   movl  $0, %eax
   addq  $8, %rsp
   ret
   .size  main, -.main
3. Using the \texttt{-fplugin=path/dragonegg.so} flag in the command line of \texttt{gcc} makes \texttt{gcc} use LLVM's optimizer and LLVM codegen:

```
$ gcc testprog.c -S -O1 -o - -fplugin=./dragonegg.so
    .file  "testprog.c"
# Start of file scope inline assembly
    .ident  "GCC: (GNU) 4.5.0 20090928 (experimental) LLVM:
            82450:82981"
# End of file scope inline assembly

.text
.align 16
.globl main
.type main,@function
main:
  subq $8, %rsp
  movl $.L.str, %edi
  call puts
  xorl %eax, %eax
  addq $8, %rsp
  ret
.size main, .-main
.type .L.str,@object
.section .rodata.str1.1,"aMS",@progbits,1
.L.str:
.asciz "Hello world!"
.size .L.str,13

.section .note.GNU-stack,"",@progbits
```

\textbf{See also}

- To know about how to get the source code and installation procedure, refer to \url{http://dragonegg.llvm.org/}
Where to buy this book

You can buy LLVM Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.