LLVM is currently being used by many companies and has a very active open source community. It provides a compiler infrastructure, which can be used to write a compiler for a language, generate machine code for a target architecture, and also with a lot of other utility tools that can be easily integrated into compiler projects.

This book details how you can use LLVM compiler infrastructure libraries effectively and will enable you to quickly design your own custom compiler with LLVM. We start with the basics, where you’ll get to know all about LLVM. We then cover how you can use LLVM library calls to produce intermediate representations (IR) of simple and complex high-level language paradigms. Moving on, we show you how to implement optimizations at different levels, write an optimization pass, generate code that is independent of a target, and then map the generated code to a backend. The book also walks you through IR-to-IR transformations, advanced IR block transformations, and target machines. By the end of this book, you’ll be able to easily utilize the LLVM libraries in your own projects.

Who this book is written for

This book is intended for those who already know some of the concepts concerning compilers and want to quickly become familiar with the LLVM’s infrastructure and the rich set of libraries that it provides.
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 'Playing with LLVM'
- A synopsis of the book’s content
- More information on LLVM Essentials
About the Authors

**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. Suyog was also involved in code performance improvements for the ARM and X86 architectures. He has been a part of the compiler team for the Tizen project. His interest in compiler development lies more in code optimization and vectorization.

Previously, he has authored a book on LLVM, titled *LLVM Cookbook* by Packt Publishing.

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 the Birla Institute of Technology, Dubai. He has earned a bachelor's degree in computer technology from the College of Engineering, Pune, India.

---

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

He has earned a bachelor's degree in Information Technology from Motilal Nehru National Institute of Technology, Allahabad, India. Currently, he lives in Bengaluru, India.
Preface

LLVM is one of the very hot topics in recent times. It is an open source project with an ever-increasing number of contributors. Every programmer comes across a compiler at some point or the other while programming. Simply speaking, a compiler converts a high-level language to machine-executable code. However, what goes on under the hood is a lot of complex algorithms at work. So, to get started with compiler, LLVM will be the simplest infrastructure to study. Written in object-oriented C++, modular in design, and with concepts that are very easy to map to theory, LLVM proves to be attractive for experienced compiler programmers and for novice students who are willing to learn.

As authors, we maintain that simple solutions frequently work better and are easier to grasp than complex solutions. Throughout the book we will look at various topics that will help you enhance your skills and drive you to learn more.

We also believe that this book will be helpful for people not directly involved in compiler development as knowledge of compiler development will help them write code optimally.

What this book covers

Chapter 1, Playing with LLVM, introduces you to the modular design of LLVM and LLVM Intermediate Representation. In this chapter, we also look into some of the tools that LLVM provides.

Chapter 2, Building LLVM IR, introduces you to some basic function calls provided by the LLVM infrastructure to build LLVM IR. This chapter demonstrates building of modules, functions, basic blocks, condition statements, and loops using LLVM APIs.

Chapter 3, Advanced LLVM IR, introduces you to some advanced IR paradigms. This chapter explains advanced IR to the readers and shows how LLVM function calls can be used to emit them in the IR.
Chapter 4, Basic IR Transformations, deals with basic transformation optimizations at the IR level using the LLVM optimizer tool opt and the LLVM Pass infrastructure. You will learn how to use the information of one pass in another and then look into Instruction Simplification and Instruction Combining Passes.

Chapter 5, Advanced IR Block Transformations, deals with optimizations at block level on IR. We will discuss various optimizations such as Loop Optimizations, Scalar Evolution, Vectorization, and so on, followed by the summary of this chapter.

Chapter 6, IR to Selection DAG phase, takes you on a journey through the abstract infrastructure of a target-independent code generator. We explore how LLVM IR is converted to Selection DAG and various phases thereafter. It also introduces you to instruction selection, scheduling, register allocation, and so on.

Chapter 7, Generating Code for Target Architecture, introduces the readers to the tablegen concept. It shows how target architecture specifications such as register sets, instruction sets, calling conventions, and so on can be represented using tablegen, and how the output of tablegen can be used to emit code for a given architecture. This chapter can be used by readers as a reference for bootstrapping a target machine code generator.
Playing with LLVM

The LLVM Compiler infrastructure project, started in 2000 in University of Illinois, was originally a research project to provide modern, SSA based compilation technique for arbitrary static and dynamic programming languages. Now it has grown to be an umbrella project with many sub projects within it, providing a set of reusable libraries having well defined interfaces.

LLVM is implemented in C++ and the main crux of it is the LLVM core libraries it provides. These libraries provide us with opt tool, the target independent optimizer, and code generation support for various target architectures. There are other tools which make use of core libraries, but our main focus in the book will be related to the three mentioned above. These are built around LLVM Intermediate Representation (LLVM IR), which can almost map all the high-level languages. So basically, to use LLVM’s optimizer and code generation technique for code written in a certain programming language, all we need to do is write a frontend for a language that takes the high level language and generates LLVM IR. There are already many frontends available for languages such as C, C++, Go, Python, and so on. We will cover the following topics in this chapter:

- Modular design and collection of libraries
- Getting familiar with LLVM IR
- LLVM Tools and using them at command line
Modular design and collection of libraries

The most important thing about LLVM is that it is designed as a collection of libraries. Let's understand these by taking the example of LLVM optimizer opt. There are many different optimization passes that the optimizer can run. Each of these passes is written as a C++ class derived from the Pass class of LLVM. Each of the written passes can be compiled into a .o file and subsequently they are archived into a .a library. This library will contain all the passes for opt tool. All the passes in this library are loosely coupled, that is they mention explicitly the dependencies on other passes.

When the optimizer is ran, the LLVM PassManager uses the explicitly mentioned dependency information and runs the passes in optimal way. The library based design allows the implementer to choose the order in which passes will execute and also choose which passes are to be executed based on the requirements. Only the passes that are required are linked to the final application, not the entire optimizer.

The following figure demonstrates 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.

The code generator also makes use of this modular design like the Optimizer, for splitting the code generation into individual passes, namely, instruction selection, register allocation, scheduling, code layout optimization, and assembly emission.
In each of the following phases mentioned there are some common things for almost every target, such as an algorithm for assigning physical registers available to virtual registers even though the set of registers for different targets vary. So, the compiler writer can modify each of the passes mentioned above and create custom target-specific passes. The use of the tablegen tool helps in achieving this using table description .td files for specific architectures. We will discuss how this happens later in the book.

Another capability that arises out of this is the ability to easily pinpoint a bug to a particular pass in the optimizer. A tool name Bugpoint makes use of this capability to automatically reduce the test case and pinpoint the pass that is causing the bug.

Getting familiar with LLVM IR

LLVM Intermediate Representation (IR) is the heart of the LLVM project. In general every compiler produces an intermediate representation on which it runs most of its optimizations. For a compiler targeting multiple-source languages and different architectures the important decision while selecting an IR is that it should neither be of very high-level, as in very closely attached to the source language, nor it should be very low-level, as in close to the target machine instructions. LLVM IR aims to be a universal IR of a kind, by being at a low enough level that high-level ideas may be cleanly mapped to it. Ideally the LLVM IR should have been target-independent, but it is not so because of the inherent target dependence in some of the programming languages itself. For example, when using standard C headers in a Linux system, the header files itself are target dependent, which may specify a particular type to an entity so that it matches the system calls of the particular target architecture.

Most of the LLVM tools revolve around this Intermediate Representation. The frontends of different languages generate this IR from the high-level source language. The optimizer tool of LLVM runs on this generated IR to optimize the code for better performance and the code generator makes use of this IR for target specific code generation. This IR has three equivalent forms:

- An in-memory compiler IR
- An on-disk bitcode representation
- A Human readable form (LLVM Assembly)
Now let's take an example to see how this LLVM IR looks like. We will take a small C code and convert it into LLVM IR using clang and try to understand the details of LLVM IR by mapping it back to the source language.

$ cat add.c
int globvar = 12;

int add(int a) {
    return globvar + a;
}

Use the clang frontend with the following options to convert it to LLVM IR:

$ clang -emit-llvm -c -S add.c
$ cat add.ll
@globvar = global i32 12, align 4
; Function Attrs: nounwind uwtable
define i32 @add(i32 %a) #0 {
    %1 = alloca i32, align 4
    store i32 %a, i32* %1, align 4
    %2 = load i32, i32* @globvar, align 4
    %3 = load i32, i32* %1, align 4
    %4 = add nsw i32 %2, %3
    ret i32 %4
}

attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-inf-s-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{10}

Now let's look at the IR generated and see what it is all about. You can see the very first line giving the ModuleID, that it defines the LLVM module for add.c file. An LLVM module is a top-level data structure that has the entire contents of the input LLVM file. It consists of functions, global variables, external function prototypes, and symbol table entries.
The following lines show the target data layout and target triple from which we can know that the target is x86_64 processor with Linux running on it. The datalayout string tells us what is the endianess of machine ('e' meaning little endian), and the name mangling (m : e denotes elf type). Each specification is separated by '–' and each following spec gives information about the type and size of that type. For example, i64:64 says 64 bit integer is of 64 bits.

Then we have a global variable globvar. In LLVM IR all globals start with '@' and all local variables start with '%'. There are two main reasons why the variables are prefixed with these symbols. The first one being, the compiler won't have to bother about a name clash with reserved words, the other being that the compiler can come up quickly with a temporary name without having to worry about a conflict with symbol table conflicts. This second property is useful for representing the IR in static single assignment (SSA) from where each variable is assigned only a single time and every use of a variable is preceded by its definition. So, while converting a normal program to SSA form, we create a new temporary name for every redefinition of a variable and limit the range of earlier definition till this redefinition.

LLVM views global variables as pointers, so an explicit dereference of the global variable using load instruction is required. Similarly, to store a value, an explicit store instruction is required.

Local variables have two categories:

- **Register allocated local variables**: These are the temporaries and allocated virtual registers. The virtual registers are allocated physical registers during the code generation phase which we will see in a later chapter of the book. They are created by using a new symbol for the variable like:

  ```
  %1 = some value
  ```

- **Stack allocated local variables**: These are created by allocating variables on the stack frame of a currently executing function, using the alloca instruction. The alloca instruction gives a pointer to the allocated type and explicit use of load and store instructions is required to access and store the value.

  ```
  %2 = alloca i32
  ```

Now let's see how the add function is represented in LLVM IR. `define i32 @add(i32 %a)` is very similar to how functions are declared in C. It specifies the function returns integer type i32 and takes an integer argument. Also, the function name is preceded by '@', meaning it has global visibility.
Within the function is actual processing for functionality. Some important things to note here are that LLVM uses a three-address instruction, that is a data processing instruction, which has two source operands and places the result in a separate destination operand (%4 = add i32 %2, %3). Also the code is in SSA form, that is each value in the IR has a single assignment which defines the value. This is useful for a number of optimizations.

The attributes string that follows in the generated IR specifies the function attributes which are very similar to C++ attributes. These attributes are for the function that has been defined. For each function defined there is a set of attributes defined in the LLVM IR.

The code that follows the attributes is for the `ident` directive that identifies the module and compiler version.

**LLVM tools and using them in the command line**

Until now, we have understood what LLVM IR (human readable form) is and how it can be used to represent a high-level language. Now, we will take a look at some of the tools that LLVM provides so that we can play around with this IR converting to other formats and back again to the original form. Let's take a look at these tools one by one along with examples.

- **llvm-as**: This is the LLVM assembler that takes LLVM IR in assembly form (human readable) and converts it to bitcode format. Use the preceding `add.ll` as an example to convert it into bitcode. To know more about the LLVM Bitcode file format refer to [http://llvm.org/docs/BitCodeFormat.html](http://llvm.org/docs/BitCodeFormat.html)

  ```
  $ llvm-as add.ll -o add.bc
  ```

  To view the content of this bitcode file, a tool such as `hexdump` can be used.

  ```
  $ hexdump -c add.bc
  ```

- **llvm-dis**: This is the LLVM disassembler. It takes a bitcode file as input and outputs the llvm assembly.

  ```
  $ llvm-dis add.bc -o add.ll
  ```

  If you check `add.ll` and compare it with the previous version, it will be the same as the previous one.
Chapter 1

- **llvm-link**: llvm-link links two or more llvm bitcode files and outputs one llvm bitcode file. To view a demo write a `main.c` file that calls the function in the `add.c` file.

  ```
  $ cat main.c
  #include<stdio.h>

  extern int add(int);

  int main() {
    int a = add(2);
    printf("%d\n",a);
    return 0;
  }
  ```

  Convert the C source code to LLVM bitcode format using the following command.

  ```
  $ clang -emit-llvm -c main.c
  ```

  Now link `main.bc` and `add.bc` to generate `output.bc`.

  ```
  $ llvm-link main.bc add.bc -o output.bc
  ```

- **lli**: lli directly executes programs in LLVM bitcode format using a just-in-time compiler or interpreter, if one is available for the current architecture. Lli is not like a virtual machine and cannot execute IR of different architecture and can only interpret for host architecture. Use the bitcode format file generated by llvm-link as input to lli. It will display the output on the standard output.

  ```
  $ lli output.bc
  14
  ```

- **llc**: llc is the static compiler. It compiles LLVM inputs (assembly form/bitcode form) into assembly language for a specified architecture. In the following example it takes the `output.bc` file generated by llvm-link and generates the assembly file `output.s`.

  ```
  $ llc output.bc -o output.s
  ```

  Let's look at the content of the `output.s` assembly, specifically the two functions of the generated code, which is very similar to what a native assembler would have generated.

  ```
  Function main:
  .type main,@function
  main:               # @main
    .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
    subq  $16, %rsp
    movl  $0, -4(%rbp)
    movl  $2, %edi
    callq  add
    movl  %eax, %ecx
    movl  %ecx, -8(%rbp)
    movl  $.L.str, %edi
    xorl  %eax, %eax
    movl  %ecx, %esi
    callq  printf
    xorl  %eax, %eax
    addq  $16, %rsp
    popq  %rbp
    retq
.Lfunc_end0:

Function: add
add: # @add
    .cfi_startproc
# BB#0:
    pushq  %rbp
.Ltmp3:
    .cfi_def_cfa_offset 16
.Ltmp4:
    .cfi_offset %rbp, -16
    movq  %rsp, %rbp
.Ltmp5:
    .cfi_def_cfa_register %rbp
    movl  %edi, -4(%rbp)
    addl  globvar(%rip), %edi
    movl  %edi, %eax
    popq  %rbp
    retq
.Lfunc_end1:
• **opt:** This is modular LLVM analyzer and optimizer. It takes the input file and runs the optimization or analysis specified on the command line. Whether it runs the analyzer or optimizer depends on the command-line option.

```
opt [options] [input file name]
```

When the `--analyze` option is provided it performs various analysis on the input. There is a set of analysis options already provided that can be specified through command line or else one can write down their own analysis pass and provide the library to that analysis pass. Some of the useful analysis passes that can be specified using the following command line arguments are:

- **basicaa:** basic alias analysis
- **da:** dependence analysis
- **instcount:** count the various instruction types.
- **loops:** information about loops
- **scalar evolution:** analysis of scalar evolution

When the `--analyze` option is not passed, the opt tool does the actual optimization work and tries to optimize the code depending upon the command-line options passed. Similarly to the preceding case, you can use some of the optimization passes already present or write your own pass for optimization. Some of the useful optimization passes that can be specified using the following command-line arguments are:

- **constprop:** simple constant propagation.
- **dce:** dead code elimination pass
- **globalopt:** pass for global variable optimization
- **inline:** pass for function inlining
- **instcombine:** for combining redundant instructions
- **licm:** loop invariant code motion
- **tailcallelim:** Tail Call elimination

Before going ahead we must note that all the tools mentioned in this chapter are meant for compiler writers. An end user can directly use clang for compilation of C code without converting the C code into intermediate representation.
Summary

In this chapter, we looked into the modular design of LLVM: How it is used in the opt tool of LLVM, and how it is applicable across LLVM core libraries. Then we took a look into LLVM intermediate representation, and how various entities (variables, functions etc.) of a language are mapped to LLVM IR. In the last section, we discussed about some of the important LLVM tools, and how they can be used to transform the LLVM IR from one form to another.

In the next chapter, we will see how we can write a frontend for a language that can output LLVM IR using the LLVM machinery.
Where to buy this book

You can buy LLVM Essentials 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.