Introduction to Compilers and Language Design

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at this website:
`http://compilerbook.org`

Draft version: September 14, 2018

Chapter 2 – A Quick Tour

2.1 The Compiler Toolchain

A compiler is one component in a **toolchain** of programs used to create executables from source code. Typically, when you invoke a single command to compile a program, a whole sequence of programs are invoked in the background. Figure 2.1 shows the programs typically used in a Unix system for compiling C source code to assembly code.

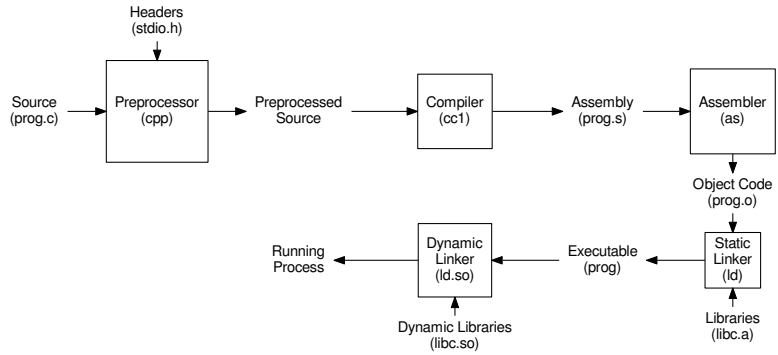

Figure 2.1: A Typical Compiler Toolchain

- The **preprocessor** prepares the source code for the compiler proper. In the C and C++ languages, this means consuming all directives that start with the `#` symbol. For example, an `#include` directive causes the pre-processor to open the named file and insert its contents into the source code. A `#define` directive causes the preprocessor to substitute a value wherever a macro name is encountered. (Not all languages rely on a preprocessor.)

- The **compiler** proper consumes the clean output of the preprocessor. It scans and parses the source code, performs typechecking and

other semantic routines, optimizes the code, and then produces assembly language as the output. This part of the toolchain is the main focus of this book.

- The **assembler** consumes the assembly code and produces **object code**. Object code is "almost executable" in that it contains raw machine language instructions in the form needed by the CPU. However, object code does not know the final memory addresses in which it will be loaded, and so it contains gaps that must be filled in by the linker.

- The **linker** consumes one or more object files and library files and combines them into a complete, executable program. It selects the final memory locations where each piece of code and data will be loaded, and then "links" them together by writing in the missing address information. For example, an object file that calls the `printf` function does not initially know the address of the function. An empty (zero) address will be left where the address must be used. Once the linker selects the memory location of `printf`, it must go back and write in the address at every place where `printf` is called.

In Unix-like operating systems, the preprocessor, compiler, assembler, and linker are historically named `cpp`, `cc1`, `as`, and `ld` respectively. The user-visible program `cc` simply invokes each element of the toolchain in order to produce the final executable.

2.2 Stages Within a Compiler

In this book, our focus will be primarily on the compiler proper, which is the most interesting component in the toolchain. The compiler itself can be divided into several stages:

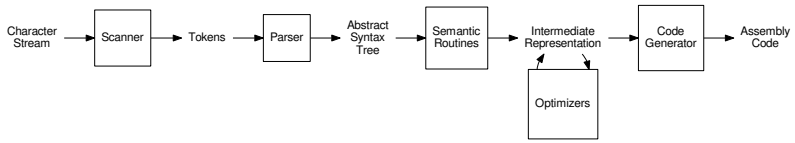

Figure 2.2: The Stages of a Unix Compiler

- The **scanner** consumes the plain text of a program, and groups together individual characters to form complete **tokens**. This is much like grouping characters into words in a natural language.

- The **parser** consumes tokens and groups them together into complete statements and expressions, much like words are grouped into sentences in a natural language. The parser is guided by a **grammar** which states the formal rules of composition in a given language. The output of the parser is an **abstract syntax tree (AST)** that captures the grammatical structures of the program. The AST also remembers where in the source file each construct appeared, so it is able to generate targeted error messages, if needed.
- The **semantic routines** traverse the AST and derive additional meaning (semantics) about the program from the rules of the language and the relationship between elements of the program. For example, we might determine that `x + 10` is a `float` expression by observing the type of `x` from an earlier declaration, then applying the language rule that addition between `int` and `float` values yields a float. After the semantic routines, the AST is often converted into an **intermediate representation (IR)** which is a simplified form of assembly code suitable for detailed analysis. There are many forms of IR which we will discuss in Chapter 8.
- One or more **optimizers** can be applied to the intermediate representation, in order to make the program smaller, faster, or more efficient. Typically, each optimizer reads the program in IR format, and then emits the same IR format, so that each optimizer can be applied independently, in arbitrary order.
- Finally, a **code generator** consumes the optimized IR and transforms it into a concrete assembly language program. Typically, a code generator must perform **register allocation** to effectively manage the limited number of hardware registers, and **instruction selection** and **sequencing** to order assembly instructions in the most efficient form.

2.3 Example Compilation

Suppose we wish to compile this fragment of code into assembly:

```
height = (width+56) * factor(foo)
```

The first stage of the compiler (the scanner) will read in the text of the source code character by character, identify the boundaries between symbols, and emit a series of **tokens**. Each token is a small data structure that describes the nature and contents of each symbol:

id:height	**=**	**(**	**id:width**	**+**	**int:56**	**)**	*****	**id:factor**	**(**	**id:foo**	**)**	**;**

At this stage, the purpose of each token is not yet clear. For example, `factor` and `foo` are simply known to be identifiers, even though one is

the name of a function, and the other is the name of a variable. Likewise, we do not yet know the type of `width`, so the `+` could potentially represent integer addition, floating point addition, string concatenation, or something else entirely.

The next step is to determine whether this sequence of tokens forms a valid program. The parser does this by looking for patterns that match the **grammar** of a language. Suppose that our compiler understands a language with the following grammar:

Grammar G_1

1. expr → expr + expr
2. expr → expr * expr
3. expr → expr = expr
4. expr → id (expr)
5. expr → (expr)
6. expr → id
7. expr → int

Each line of the grammar is called a rule, and explains how various parts of the language are constructed. Rules 1-3 indicate that an expression can be formed by joining two expressions with operators. Rule 4 describes a function call. Rule 5 describes the use of parentheses. Finally, rules 5 and 6 indicate that identifiers and integers are atomic expressions. [1]

The parser looks for sequences of tokens that can be replaced by the left side of a rule in our grammar. Each time a rule is applied, the parser creates a node in a tree, and connects the sub-expressions into the **abstract syntax tree (AST)**. The AST shows the structural relationships between each symbol: addition is performed on `width` and `56`, while a function call is applied to `factor` and `foo`.

With this data structure in place, we are now prepared to analyze the meaning of the program. The **semantic routines** traverse the AST and derive additional meaning by relating parts of the program to each other, and to the definition of the programming language. An important component of this process is **typechecking**, in which the type of each expression is determined, and checked for consistency with the rest of the program. To keep things simple here, we will assume that all of our variables are plain integers.

To generate linear intermediate code, we perform a post-order traversal of the AST and generate an IR instruction for each node in the tree. A typical IR looks like an abstract assembly language, with load/store instructions, arithmetic operations, and an infinite number of registers. For example, this is a possible IR representation of our example program:

[1] The careful reader will note that this example grammar has ambiguities. We will discuss that in some detail in Chapter 4.

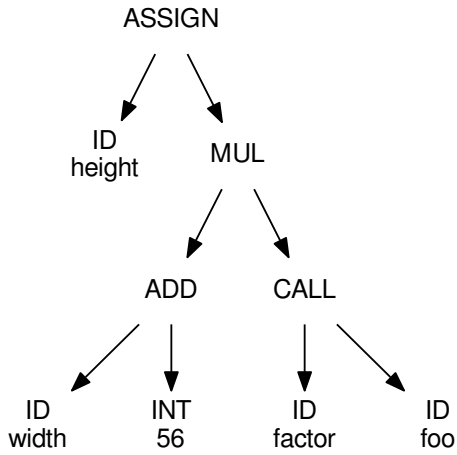

Figure 2.3: Example AST

```
LOAD $56    -> r1
LOAD width  -> r2
IADD r1, r2 -> r3
PUSH foo
CALL factor -> r4
IMUL r3, r4 -> r5
STOR r6     -> height
```

Figure 2.4: Example Intermediate Representation

The intermediate representation is where most forms of optimization occur. Dead code is removed, common operations are combined, and code is generally simplified to consume fewer resources and run more quickly.

Finally, the intermediate code must be converted to the desired assembly code. Figure 2.5 shows X86 assembly code that is one possible translation of the IR given above. Note that the assembly instructions do not necessarily correspond one-to-one with IR instructions.

A well-engineered compiler is highly modular, so that common code elements can be shared and combined as needed. To support multiple languages, a compiler can provide distinct scanners and parsers, each emitting the same intermediate representation. Different optimization techniques can be implemented as independent modules (each reading and

```
MOVL    width, %eax     # load width into eax
ADDL    $56, %eax       # add 56 to eax
MOVL    %eax, -8(%rbp)  # save sum in temporary
MOVL    foo, %edi       # load foo into arg 0 register
CALL    factor          # invoke function
MOVL    -8(%rbp), %edx  # recover sum from temporary
IMULL   %eax, %edx      # multiply them together
MOVL    %edx, height    # store result into height
```

Figure 2.5: Example Assembly Code

writing the same IR) so that they can be enabled and disable independently. A retargetable compiler contains multiple code generators, so that the same IR can be emitted for a variety of microprocessors.

2.4 Exercises

1. Determine how to invoke the preprocessor, compiler, assembler, and linker manually in your local computing environment. Compile a small complete program that computes a simple expression, and examine the output at each stage. Are you able to follow the flow of the program in each form?

2. Determine how to change the optimization level for your local compiler. Find a non-trivial source program and compile it at multiple levels of optimization. How does the compile time, program size, and run time vary with optimization levels?

3. Search the internet for the formal grammars for three languages that you are familiar with, such as C++, Ruby, and Rust. Compare them side by side. Which language is inherently more complex? Do they share any common structures?