Chapter 1

Compilers: Principles, Techniques, and Tools Ed. 1 (1986)

1. Compilers

Compiler : Translate source languages into an equivalent program in another (target language)

Compilers are generally made up of two main parts, analysis and synthesis.

- Analysis breaks up the source into constituents and creates an intermediate representation (IR)
- Synthesis constructs the target program from IR

During analysis, the program is parsed into a syntax tree, where each node is an operation, and the children of the said node are the arguments.

Whilst most compilers translate source code into machine code, there are others that may have more specialized use cases:

- Text formatters take a stream of characters as input to typeset. An example is LaTeX.
- Silicon compilers have variables that represent not positions in memory, but logical signals (0 or 1).
- Query interpreters translate a predicate containing relational and boolean operations into commands to search databases.

In addition to compilers, many executables require other programs to run, as the source code is often split into multiple smaller files.

Preprocessing : The task of collecting the source program in one location. Preprocessing also expands shorthands, such as macros.

There may be more steps; for example, with C/C++, it compiles into an assembler, which compiles into machine code, which then passes through a linker into the code that actually runs.

2. Analysis of the Source Program

Analysis is split up into 3 main stages:

- Linear Analysis: Aka lexing, the characters are grouped into tokens, which are sequences of characters having meaning.
- Hierarchical/Syntactic Analysis: Aka Abstract Syntax Tree (AST) Generation, tokens are grouped into collections with meaning.

- Semantic Analysis: Checks are performed to ensure that components fit together in a meaningful way. Done by traversing the AST.

2.1. Lexical Analysis

Consider the following:

```
position := initial + rate * 60
```

would be grouped into:

1. the identifier `position`.
2. the assignment symbol `:=`.
3. the identifier `initial`.
4. the operator `+`.
5. the identifier `rate`.
6. the operator `*`.
7. the integer literal `60`.

Newlines, comments, and blanks are also resolved during lexical analysis.

2.2. Syntax Analysis

Hierarchical analysis is also known as syntax analysis or parsing. This involves grouping tokens in a manner defined by a **grammar**, and organizing them into a syntax tree. Syntactic/grammatical errors are also resolved in this step.
Consider the expression `initial + rate * 60`, we must define an order of precedence to perform the `rate * 60` first, even if `initial + rate` is lexed first. We create a recursivly defined grammar of expressions. An example may be:

1. Any $identifier$ is an expression.
2. Any *number* is an expression.
3. If $expression_1$ and $expression_2$ are expressions, then so are

$$\begin{array}{l} expression_1 \ + \ expression_2 \\ expression_1 \ * \ expression_2 \\ (\quad expression_1 \quad) \end{array}$$

By rule (1), `initial` and `rate` are expressions. By rule (2), `60` is an expression, and rule (3) tells us that `rate * 60` is also an expression. Finally, we deduce that `initial + rate * 60` is also an expression, thus the statement has valid syntax. These rules can also be represented as:

1. If $identifier_1$ is an identifier, and $expression_2$ is an expression, then

$$identifier_1 \;:=\; expression_2$$

is a statement.

2. If $expression_1$ is an expression and $statement_2$ is a statement, then

   while ($expression_1$) **do** $statement_2$
   if ($expression_1$) **then** $statement_2$

   are statements.

The formalization of these grammars in a form known as **context free grammars** will be discussed later.

The difference between lexical and syntactic analysis is somewhat arbitrary, and is chosen by considering the simplification of the overall task of analysis.

It is important to note that there are certain expressions that don't require recursion to define. For example, to recognize identifiers, we can simply just scan the input stream, waiting until the next character is not alphanumeric. These are then added to a lookup table called a **symbol table**, and then removed from the input so that the lexing of the next token can begin.

However, this style of linear scanning is not powerful enough for all expressions/statements, as we cannot evaluate expressions in parenthesis or conditionals without some kind of hierarchical structure on the input. Thus, as described earlier, the tokens are added to a tree, where nodes represent operations, and children to that node are operands.

2.3. Semantic Analysis

Semantic analysis involves checking the source program, now formatted into a syntax tree, for semantic errors and gathers relevant information for code generation.

One of the key steps in semantic analysis is **type checking**. Here, the compiler checks that each operation has operarands of valid types for their corresponding operator. For example, if we encounter

```
array[5.5]
```

we must report an error, as we cannot index an array with non integer valued numbers. However, some languages may also permit some operand type coercion, such as arithmetic between an integer and a float. In this case, the compiler would need to convert the integer into a float.

Example 1.1. Inside a machine, the bit representation for a float and an integer are different, even if they happen to have the same value. Consider `initial + rate * 60`. Type checking shows that the `*` is applied to a float, `rate`, and an integer, `60`. The general approach to this is to convert the integer into a real, by introducing an extra node **inttoreal**.

3. Phases of a Compiler

Conceptually, a compiler can be thought as operating in phases, each of which transforms the source code from one representation to another. For example, many compilers may have the following pipeline:
source program →lexical analyzer →syntax analyzer →semantic analyzer →IR generator →code optimizer →code generator →target program.

3.1. Symbol-Table Management

Symbol table : A lookup table that contains keys as identifiers, and values as relevant attributes.

Symbol-table management a function done through all stages of compiling. It is responsible for recording all identifiers used throughout the source program, as well as relevant attributes such as memory allocation, type, scope, and, in the case of function declarations, number and types of arguments.

When an identifier in the source program is detected during lexing, it is added to the symbol table. However, the attributes cannot be inferred as easily. Consider the following, written in Pascal syntax:

```
var position, initial, rate : real ;
```

The type `real` is not known when the identifiers `position, initial`, and `rate` are seen by the lexical analyzer.

3.2. Error Detection and Reporting

A compiler that instantly stops after encountering an error is not as useful as it seems. It helps to continue for various reasons, including catching multiple errors. Therefore there must be a process such that the compiler can proceed even after encountering an error. This error handling is mostly done by the syntax and semantic analysis phases.

3.3. Analysis Phases

As translation progresses, the internal representation of the source code strips away a layer of abstraction. Consider

```
position := initial + rate * 60
```

The lexical analyzer reads the characters and groups them into tokens. Certain tokens will also contain a lexical value attribute. For example, when the lexer sees `rate`, it doesn't only create a token, for example **id**, but also enters the lexeme `rate` into the symbol table, if it is not already there. For this section, we will use $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$ for `position, literal,` and `rate` respectively, to emphasize that the internal representation of identifiers is completely different to that of the source code. Thus, our expression now becomes:

$$\mathbf{id}_1 \; := \; \mathbf{id}_2 \; + \; \mathbf{id}_3 \; * \; 60$$

This is not fully accurate, as there should also be unique tokens for each operation. However for ease of reading, this will do. Syntactic and Semantic analysis are then done to check for errors and to generate the AST, which is then translated into the following sample intermediate representation:

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 * temp2
id1 := temp3
```

where the temp variables are stored in the symbol table. Next, code optimization is done, to remove redundant operations. This is usually implemented using third party tools, such as LLVM.

```
temp1:= id3 * 60.0
id1: = id2 + temp1
```

Finally, the target language code is created.

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Each step is described in-depth below.

3.4. Intermediate Code Generation

After syntax and semantic analysis, some compilers create an explicit intermediate representation. We can think of this as a source code for an abstract machine (such as a virtual machine). This IR should satisfy two properties: it should be easy to read, and easy to translate into the target program.

3.5. Code Optimization

The code optimization step attempts to improve the IR. For example, many AST algorithms will have a seperate operation for converting `60` into a real, and then multiplying with `id3`, even if there is a more optimal way. This isn't necessarily a bad thing, as optimizations like these are resolved during code optimization. That is, the compiler can deduce that the conversion can be done at compile time, and the `inttoreal` operation is eliminated. The difficulty in this comes in the variation of the different kinds of optimizations possible. There are dedicated tools, such as LLVM described above, that implement heuristic optimization algorithms due to the NP nature of this problem.

3.6. Code Generation

The final step of compilation is the generation of target code, usually machine or assembly code. In the code shown above, it is written in the form of many assembly languages, The first keyword tells us the operation, and the first and second operands of each instruction specify a source and destination, respectively. This code moves the contents of address `id3` into register 2, and then multiplies it by `60.0`. The `#` signifies that `60` is to be treated as a constant. The third instruction moves `id2` into register 1 and line four adds it with the contents of register 2. Finally, the contents of register 1 are assigned to `id1`, thus the code accurately implements the source code.

4. Cousins of the Compiler

Many times there is a need for further processing for the input and output for compilers. The following sections discuss the context in which compilers are implemented in.

4.1. Preprocessors

Preprocessors provide input to compilers. Their functionalities can vary, however many times they include:

1. *Macro processing*: A preprocessor will resolve any macros defined by the user. For example, `#define PI 3.14159`
2. *File inclusion*: A preprocessor will include any header files into the program. For example, a C preprocessor will replace the statement `#include <stdio.h>` with the contents in `<stdio.h>`.

4.2. Assemblers

Some compilers output assembly code, which needs to be passed to an assembler to be translated into machine code.
Assembly Code: A mnemonic version of machine code, where opcodes are assigned names. An example can include:

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

The program above moves the contents of address `a` into `R1`, then adds the constant 2, and then moves that to the address of `b`. Thus, this is equivalent to `b := a + 2`.

4.3. Two-Pass Assembly

Machine Code: Code comprised of only bits (0s and 1s), in a form that the CPU can directly execute.
A *pass* refers to reading an input file once. Thus, a two-pass assembly, which is the simplest form of an assembler, will read over the input file twice. In the first pass, all identifiers are found, and stored in a symbol table. Note that this symbol table is separate to that of the compiler. For the example above, assuming that `a` and `b` are 4 bytes each, the first pass could store `a` at address `0`, and `b` at address `4`.
In the second pass, the assembler will translate each operation into a predefined sequence of bits, following an *opcode*. This opcode is specific to the type of hardware used. When it encounters an identifier, it replaces it with the address stored. A hypothetical machine code for the program above may look like:

```
0001 01 00 00000000 *
0011 01 10 00000010
0010 01 00 00000100 *
```

Each instruction is split up into four sections. The first four bits encode the operation to be performed. In this case, `0001` represents load, `0010` represents store, and `0011` represents add (Note that load denotes moving from address to register, and store denotes moving from register to address). The next two bits represent the target register. The two bits after that represent a kind of "tag". `00` denotes that the next eight bits refer to a specific memory address. The tag `10` denotes immediate; the last eight bits represent an operand (in this case, the number 2). Note that this is why it is important to distinguish constant values, as it can free up register space in not having to load the value into a register.
We also notice that the first and third instructions have a `*`. This represents a *relocation bit*, indicating that the addresses are relative. Then, we denote a starting address, say L, and add L to every address with `*`. For example, if $L =$ `00001111`, then the instructions would become:

```
0001 01 00 00001111
0011 01 10 00000010
0010 01 00 00010011
```

Notice that there are no `*`, thus this is referred to as *absolute* machine code. As well, since the second instruction didnt have a `*`, it's address remained unchanged.

4.4. Loaders and Link-Editors

Usually, a program called a *loader* performs loading and link-editing. Loading consists of taking relocatable machine code, altering the relocatable operations, and replacing their addresses. The link-editor allows a project consisting of multiple files to be compiled into a single program. If the files are to be used together in a meaningful fashion, there may be some *external references.* These are lines of code in one file that refer to a location in another file. The relocatable machine code file must be able to store in the symbol table the location and or instruction label that is referred to externally. Since we dont know the context in which these external references are in during compiling, we must include the entire symbol table as part of the relocatable machine code. From the example above, if another file referenced the variable `b`, then that reference would have to be replaced by $4 + L$, where L is the relocation offset.

5. The Grouping of Phases

In many production compilers, there are different phases that are grouped together.

5.1. Front and Back Ends

These phases are normally separated into at least a *front end back end.* The front end usually comprises of lexing, parsing, and creating the IR, and the back end is responsible for all components that are dependent on the target machine. As well, there may be a *middle end*, which is responsible for performing optimizations on the IR.

5.2. Passes

As mentioned before, a pass simply refers to reading through an input file once. Several phases of compilation are usually implemented within a single pass. For example, lexical analysis, syntax analysis, semantic analysis, and code generation may be grouped into one pass.

5.3. Reducing the Number of Passes

While more passes can lead to more optimized code and thus optimized run time, more passes can also greatly increase the compile time. Finding a balance between the two is not a trivial matter. For some phases, grouping them into one pass can lead to issues. For example, it is generally fine to group lexical and syntactic analysis into one pass. However, code generation in the same pass can be extremely difficult, and is generally not feasable to do until after the IR is generated. Many programs allow you to use a variable or function before implementing a concrete definition, which is not possible to do within one pass.
In some cases, we are able to use a technique known as *backpatching*. This will be discussed more indepth in later sections, however the idea is to replace forward references with a placeholder address. Once you get the definition of the object, you can assign it to it's corresponding address.