By Dr. Muhammad Nur Yanhaona

Lesson 1: Architecture of a Compiler and its Behavior

Introduction

Most of us write computer programs using some high-level language such as C, Java, Python that allows expressing our computation in terms of human-understandable, easy to use, but formal constructs such as user defined data structures, composite expressions, lengthy and nested if/else-if/else branching, for/while loops, and regular and recursive functions. However, we all know that the machine – that is the computer – that executes our program only supports simple arithmetic logic operations, data load and store with memory, and I/O device read/writes at the unit level. Therefore, the high-level program programmers write must be converted (that is, translated) into a machine-readable program (that is, executable) that the computer can execute. The program that does this translation is the compiler. Notice that the compiler itself is a program. Since other programs require a compiler to get translated into machine executables, the first compiler itself must be written directly as a machine-readable program. That is why compiler is often called the mother of all programs.

The process of translating a high-level program into a machine-readable program is called compilation, which is a complex endeavor. As we breakdown a real-life problem into component pieces then deal with the pieces one after another to keep things manageable, compilation of high-level programs is also broken down into steps (or stages) where each step addresses a particular aspect of the translation process. Interestingly, but not surprisingly, we can associate the stages of the compiler with the task of understanding natural language speeches in our daily communication.

Stage-full Architecture of a Compiler

Consider the following hypothetical speech made by a student interested in the Compiler Design course:

> I am thinking about taking CSE 420 (Compiler Design) course this semester. I am also taking 3 other CSE courses at the same time. Do you think I am putting too much pressure on myself? My CGPA is 3.5. Do you think I am studious enough to do well in four CSE courses in a single semester? If you think I should drop CSE 420, what course you would suggest me to take in its place?

Now the above speech must be made to some other person, assume another student. To understand the speech, the listener must understand its sentences. Understanding a sentence requires understanding its type (question/statement) and its parts (aka fragments) such as 'If you think I should drop CSE 420.' Understanding a sentence part requires understanding the words and punctuation marks. Finally, all these understanding is possible because you memorized the alphabet, digits, and punctuation marks of English language. Once the listener understands the speech then he/she can take action according to it, which in this case is providing some guidance to the speaker.

As we are so accustomed to natural language, we often do not realize our brain find meanings in our conversation and speeches from the smallest elements to the largest. That it first recognizes the letters, digits, and punctuation marks; then it recognizes the words and deduces their meaning, then it identifies sentence fragments and sentences using word sequences and the knowledge of language grammar; then it attempts to find meaning of those sentences from existing knowledge and earlier sentences; afterword, it combines the meaning of the sentences to find the meaning of the whole speech; finally, the action follows.

Compiler stages are eerily similar to this process and virtually the same regardless of what programming language we are using. The alphabet of the programming language is the English language alphabet, all digits, and few punctuation marks (called delimiters in programming term) combined. The first stage of the compiler determines how the words are made out of the letters of the alphabet. However, in compiler terminology, we call the words **tokens**. The second stage grammar checks sequence of tokens to determine how sentences and sentence fragments are made, which are respectively called **statements** and **expressions** in compiler terminology. The next stage of the compiler determines the meaning of these statements and expressions. Then it takes step to generate the machine-executable, which is the action. Now hold on! Compiler did not try to understand the meaning of the entire program as a human listener does in real-life conversation. Actually, no compiler can determine the meaning of the whole program given to it. The meaning of the program is only understood by the programmer who wrote it. Compiler generates the machine executable based on the understanding of the building block expression and sentences.

As the first three stages of the compiler only analyzed the source program to gather information, they are called in-combine the analysis phase of a compiler, or the compiler frontend. The stages are given names that indicate their functions. Token generation is lexical analyses, grammar checking is syntax analysis, and meaning determination is semantic analysis. Typically, once the compiler is done finding meanings of statements and expressions, it converts the original source program into an intermediate form where information about the meanings is tagged with those expressions and statements as attributes to simply machine executable generation later. This process actually happens concurrently with semantic analysis. However, it is standard practice to mention another stage, called intermediate code generation, as part of the compiler frontend.

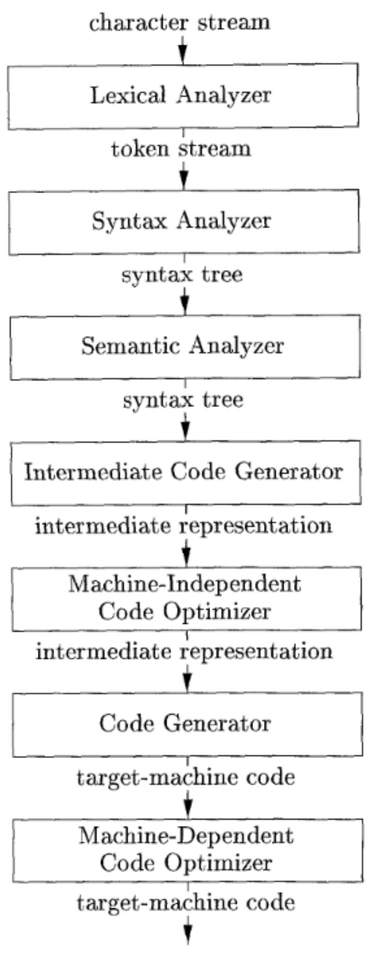
The latter stages of the compiler form its backend. Only after validating a program is syntactically correct and meaningful – but not necessarily accurate – the compiler goes on to generating the executable. Note that, different machines support different set of instructions, as you learned in your Computer Architecture course. The set of instructions the machine support forms its instruction set architecture, aka the machine language. Therefore, the action part, that is machine executable generation, is target machine's architecture dependent. However, the analysis phase is independent of the machine architecture. That is why it is common practice to write a single compiler frontend when you develop a new programming language and write different backends for different architectures.

Since machine-level instructions that standard CPUs support are very fine-grained compared to the statements and expressions in our program, typically a sequence of several machine level instructions is needed to translate each statement/expression into a machine language equivalent. That is why the process of machine code generation after completing program analysis is often called program synthesis and the component stages of the compiler backend forms the synthesis phase of the compiler.

So, what stages constitute the synthesis phase? Strictly speaking, only a single stage is needed to generate the code after meaning of the program is found at the end of the analysis phase. That stage diligently goes through the statements and expressions of the source program and generates the machine code sequence for each of them in the order it found those statements and expressions in the program. However, there is more to it.

Consider a conversation between two important state persons coming from USA and Russia. Assume they only know how to talk in their own languages. Then we must have a language interpreter person who will listen in English and translate speeches to Russian and vice versa. The compiler is like that language interpreter. However, just as in natural language interpretation, the interpreter does not translate each sentence as it is from one language to another, rather finds the best way to convey the exact meaning with lesser number of sentences; a compiler backend stage also tries to eliminate unnecessary expressions and statements from the program synthesis process while keeping the program's meaning exactly the same. Notice that this stage is optional. After this optimization, the machine code generation stage generates the machine instruction sequence for the program. Things should have ended there, however, advanced compilers also try to optimize the generated code using very detailed information of the machine architecture through another and final optimization stage. What could be optimized even after code generation? Well, for example, you can covert a multiplication by a two's power into a shift operation or you can reorder some machine instruction sequence to reduce pipeline stalls in the CPU.

So overall, we get the following stage-full architecture of any standard compiler regardless of what high-level programming language it deals with:



This picture has been copied from the reference textbook written by Aho, Lam, Sethi and Ullman.

Errors in Compilation

Note that even if the compiler can generate an executable behaviorally exact to your source program, it is not guaranteed that your program is correct. When you run the executable, the program may crush with some memory fault, file read-write errors, etc. So, what do you understand by compilation errors? Compilation errors are those errors that are identified during the analysis phase (the synthesis phase never reports an error). If the compiler cannot identify what token you wanted in a situation, or finds grammatical mistakes, or cannot determine any meaning out of an expression/statement only then it throws errors. Such errors make it impossible for it to generate a machine code equivalent.