Anatomy of a Compiler

Handout written by Maggie Johnson and Julie Zelenski, with edits by Keith.
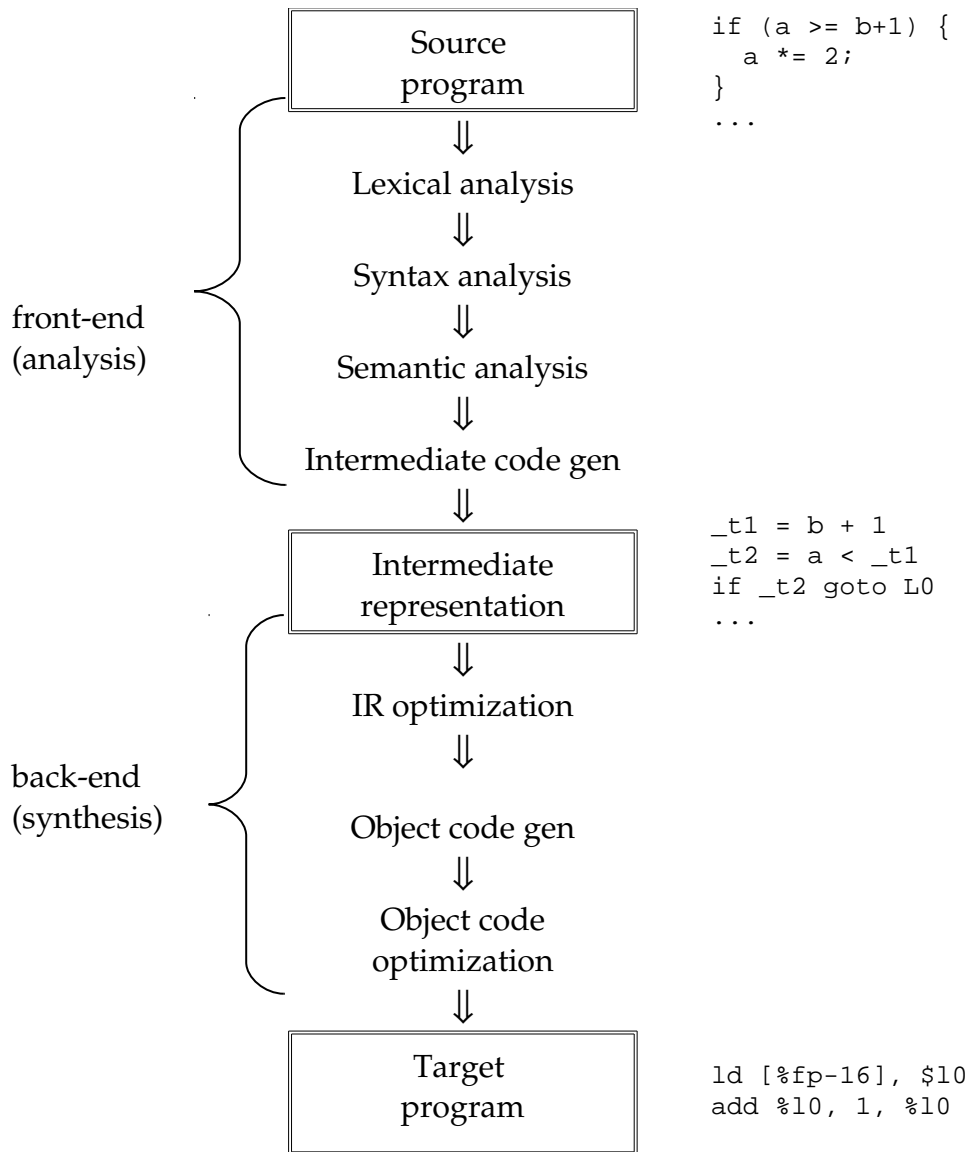
What is a compiler?

A *compiler* is a program that takes as input a program written in one language (the *source* language) and translates it into a functionally equivalent program in another language (the *target* language). The source language is usually a high-level language like C++, Java, Objective C, or C#, and the target language is usually a low-level language like assembly or machine code. As it translates, a compiler also reports errors and warnings to help the programmer make corrections to the source, so the translation can be completed. Theoretically, the source and target can be any language, but the most common use of a compiler is translating an ASCII source program written in a language such as C++ into a machine-specific result like x86 assembly that can execute on that designated hardware.

Although we will focus on writing a compiler for a programming language, the techniques you learn can be valuable and useful for a wide variety of parsing and translating tasks: translating **javadoc** comments to HTML, generating a table from the results of an SQL query, collating responses from e-mail surveys, implementing a server that responds to a network protocol like **http** or **imap**, or "screen scraping" information from an on-line source. Your printer uses parsing to render PostScript files. Hardware engineers use a full-blown compiler to translate from a hardware description language to the schematic of a circuit. Your spam filter most likely scans and parses email content. The list goes on and on.

How does a compiler work?

From the diagram on the next page, you can see there are two main stages in the compiling process: *analysis* and *synthesis*. The analysis stage breaks up the source program into pieces, and creates a generic (language-independent) intermediate representation of the program. Then, the synthesis stage constructs the desired target program from the intermediate representation. Typically, a compiler's analysis stage is called its *front end* and the synthesis stage its *back end*. Each of the stages is broken down into a set of "phases" that handle different parts of the tasks. (Why do you think typical compilers separate the compilation process into front and back-end phases?)

Diagram of the compilation process

```
            ┌──────────────────────┐      if (a >= b+1) {
            │       Source         │        a *= 2;
            │      program         │      }
            └──────────────────────┘      ...
                      ⇓
                Lexical analysis
                      ⇓
                Syntax analysis
  front-end           ⇓
  (analysis)     Semantic analysis
                      ⇓
            Intermediate code gen
                      ⇓
            ┌──────────────────────┐      _t1 = b + 1
            │     Intermediate     │      _t2 = a < _t1
            │    representation    │      if _t2 goto L0
            └──────────────────────┘      ...
                      ⇓
                IR optimization
                      ⇓
  back-end       Object code gen
  (synthesis)         ⇓
                 Object code
                 optimization
                      ⇓
            ┌──────────────────────┐      ld [%fp-16], $l0
            │       Target         │      add %l0, 1, %l0
            │      program         │
            └──────────────────────┘
```

Front-End Analysis Stage

There are four phases in the analysis stage of compiling:
  1) *Lexical Analysis* or *Scanning*: The stream of characters making up a source program
     is read from left to right and grouped into *tokens*, which are sequences of characters
     that have a collective meaning. Examples of tokens are *identifiers* (user-defined
     names), reserved words, integers, doubles or floats, delimiters, operators, and
     special symbols.

Example of lexical analysis:

```
int a;
a = a + 2;
```
A lexical analyzer scanning the code fragment above might return:

int	T_INT	(reserved word)
a	T_IDENTIFIER	(variable name)
;	T_SPECIAL	(special symbol with value of **";"**)
a	T_IDENTIFIER	(variable name)
=	T_OP	(operator with value of **"="**)
a	T_IDENTIFIER	(variable name)
+	T_OP	(operator with value of **"+"**)
2	T_INTCONSTANT	(integer constant with value of **2**)
;	T_SPECIAL	(special symbol with value of **";"**)

2) *Syntax Analysis* or *Parsing*: The tokens found during scanning are grouped together using a *context-free grammar*. A grammar is a set of rules that define valid structures in the programming language. Each token is associated with a specific rule, and grouped together accordingly. This process is called parsing. The output of this phase is called a *parse tree* or a *derivation*, i.e., a record of which grammar rules were used to create the source program.

Example of syntax analysis:

Part of a grammar for simple arithmetic expressions in C might look like this:

```
Expression ->      Expression + Expression |
                   Expression - Expression |
                   ...
                   Variable                |
                   Constant                |
                   ...

Variable ->        T_IDENTIFIER

Constant ->        T_INTCONSTANT            |
                   T_DOUBLECONSTANT
```

The symbol on the left side of the "->" in each rule can be replaced by the symbols on the right. To parse **a + 2**, we would apply the following rules:

```
Expression -> Expression + Expression
           -> Variable + Expression
           -> T_IDENTIFIER + Expression
           -> T_IDENTIFIER + Constant
           -> T_IDENTIFIER + T_INTCONSTANT
```

When we reach a point in the parse where we have only tokens, we have finished. By knowing which rules are used to parse, we can determine the structures present in the source program.

3) *Semantic Analysis*: The parse tree or derivation is next checked for semantic errors, i.e., a statement that is syntactically correct (associates with a grammar rule correctly), but disobeys the semantic rules of the source language. Semantic analysis is the phase where we detect such things as use of an undeclared variable, a function called with improper arguments, access violations, and incompatible operands and type mismatches, e.g., an array variable added to a function name.

Example of semantic analysis:

```
int arr[2], c;
c = arr * 10;
```

Most semantic analysis pertains to the checking of types. Although the C fragment above will scan into valid tokens and successfully match the rules for a valid expression, it isn't semantically valid. In the semantic analysis phase, the compiler checks the types and reports that you cannot use an array variable in a multiplication expression and that the type of the right-hand-side of the assignment is not compatible with the left.

4) *Intermediate Code Generation*: This is where the intermediate representation of the source program is created. We want this representation to be easy to generate, and easy to translate into the target program. The representation can have a variety of forms, but a common one is called *three-address code* (TAC), which is a lot like a generic assembly language that doesn't commit to a particular architecture. Three-address code is a sequence of simple instructions, each of which can have at most three operands.

Example of intermediate code generation:

```
a = b * c + b * d              _t1 = b * c
                               _t2 = b * d
                               _t3 = _t1 + _t2
                               a = _t3
```

The single C statement on the left is translated into a sequence of four instructions in three-address code on the right. Note the use of temp variables that are created by the compiler as needed to keep the number of operands down to three.

Of course, it's a little more complicated than this, because we have to translate branching and looping instructions, as well as function calls. Here is some TAC for a branching translation:

```
if (a <= b)                          _t1 = a > b
    a = a - c;                       if _t1 goto L0
c = b * c;                           _t2 = a - c
                                     a = _t2
                                 L0: _t3 = b * c
                                     c = _t3
```

The synthesis stage (back-end)

There can be up to three phases in the synthesis stage of compiling:
1) *Intermediate Code Optimization*: The optimizer accepts input in the intermediate representation (e.g., TAC) and outputs a streamlined version still in the intermediate representation. In this phase, the compiler attempts to produce the smallest, fastest and most efficient running result by applying various techniques such as

- suppressing code generation of unreachable code segments,
- ridding of unused variables,
- eliminating multiplication by 1 and addition by 0,
- loop optimization (e.g., remove statements that are not modified in the loop),
- common sub-expression elimination,
.....

The optimization phase can really slow down a compiler, so most compilers allow this feature to be suppressed or turned off by default. The compiler may even have fine-grain controls that allow the developer to make tradeoffs between the time spent compiling versus optimization quality.

Example of code optimization:

```
_t1 = b * c                  _t1 = b * c
_t2 = _t1 + 0                _t2 = _t1 + _t1
_t3 = b * c                  a = _t2
_t4 = _t2 + _t3
a = _t4
```

In the example shown above, the optimizer was able to eliminate an addition to the zero and a re-evaluation of the same expression, allowing the original five TAC statements to be re-written in just three statements and use two fewer temporary variables.

2) *Object Code Generation*: This is where the target program is generated. The output of this phase is usually machine code or assembly code. Memory locations are selected for each variable. Instructions are chosen for each operation. The three-address code is translated into a sequence of assembly or machine language instructions that perform the same tasks.

Example of code generation:

```
_t1 = b * c          ld [%fp-16], %l1    # load
_t2 = _t1 + _t1      ld [%fp-20], %l2    # load
a = _t2              smul %l1, %l2, %l3  # mult
                     add %l3, %l3, %l0   # add
                     st %l0, [%fp-24]    # store
```

In the example above, the code generator translated the TAC input into Sparc assembly output.

3) *Object Code Optimization*: There may also be another optimization pass that follows code generation, this time transforming the object code into tighter, more efficient object code. This is where we consider features of the hardware itself to make efficient usage of the processor(s) and registers. The compiler can take advantage of machine-specific idioms (specialized instructions, pipelining, branch prediction, and other peephole optimizations) in reorganizing and streamlining the object code itself. As with IR optimization, this phase of the compiler is usually configurable or can be skipped entirely.

The symbol table

There are a few activities that interact with various phases across both stages. One is *symbol table management*; a symbol table contains information about all the identifiers in the program along with important attributes such as type and scope. Identifiers can be found in the lexical analysis phase and added to the symbol table. During the two phases that follow (syntax and semantic analysis), the compiler updates the identifier entry in the table to include information about its type and scope. When generating intermediate code, the type of the variable is used to determine which instructions to emit. During optimization, the "live range" of each variable may be placed in the table to aid in register allocation. The memory location determined in the code generation phase might also be kept in the symbol table.

Error-handling

Another activity that occurs across several phases is *error handling*. Most error handling occurs in the first three phases of the analysis stage. The scanner keeps an eye out for stray tokens, the syntax analysis phase reports invalid combinations of tokens, and the semantic analysis phase reports type errors and the like. Sometimes these are fatal

errors that stop the entire process, while others are less serious and can be circumvented so the compiler can continue.

One-pass versus multi-pass

In looking at this phased approach to the compiling process, one might think that each phase generates output that is then passed on to the next phase. For example, the scanner reads through the entire source program and generates a list of tokens. This list is the input to the parser that reads through the entire list of tokens and generates a parse tree or derivation. If a compiler works in this manner, we call it a *multi-pass* compiler. The "pass" refers to how many times the compiler must read through the source program. In reality, most compilers are one-pass up to the code optimization phase. Thus, scanning, parsing, semantic analysis and intermediate code generation are all done simultaneously as the compiler reads through the source program once. Once we get to code optimization, several passes are usually required, which is why this phase slows the compiler down so much.

Historical perspective

In the early days of programming languages, compilers were considered very difficult programs to write. The first FORTRAN compiler (1957) took 18 person-years to implement. Since then, lots of techniques have been developed that simplify the task considerably, many of which you will learn about in the coming weeks.

To understand how compilers developed, we have to go all the way back to the 1940's with some of the earliest computers. A common computer at this time had perhaps 32 bytes of memory (that's *bytes*, not gigabytes or even megabytes). It also might have one *register* (a register is high-speed memory accessed directly by the processor) and 7 *opcodes* (an opcode is a low-level instruction for the processor). Each opcode was numbered from 0 to 7 and represented a specific task for the processor to perform. This type of language representation is called machine language. For example:

instruction	meaning
011	store contents of register to some memory location
100	subtract value stored in memory from register value
111	stop

The earliest "coders" of these computers (which was the term for programmers) used the binary codes to write their programs. With such a small set of instructions, this was not too difficult, but even these coders looked for ways to speed things up. They made up shorthand versions so they would not need to remember the binary codes:

instruction	shorthand	
`011`	`r,M`	(store register value r to location M)
`100`	`r-M`	(subtract value in M from register value r)
`111`	`stop`	

This is an example of an early assembly language. Assembly language is a transliteration of machine language. Many early assembly languages also provided the capability for working with symbolic memory locations as opposed to physical memory locations. To run a program, the shorthand symbols and symbolic addresses had to be translated back to the binary codes and physical addresses, first by hand, and later the coders created a program to do the translation. Such programs were called *assemblers*. Assembly language is much easier to deal with than machine language, but it is just as verbose and hardware-oriented.

As time went on, the computers got bigger (UNIVAC I, one of the early vacuum tube machines had a "huge" 1000 word memory) and the coders got more efficient. One timesaving trick they started to do was to copy programs from each other. There were some problems though (according to Admiral Grace Murray Hopper):

> "There were two problems with this technique: one was that the subroutines all started at line 0 and went on sequentially from there. When you copied them into another program you had to add all those addresses as you copied them. And programmers are lousy adders! The second thing that inhibited this was that programmers are also lousy copyists. It was amazing how often a 4 would turn into a delta (which was our space symbol) or into an A; and even B's turned into 13's." [WEXELBLAT]

Out of all this came what is considered the first compiler created by Grace Hopper and her associates at Remington Rand: A-0. It's not really a compiler in the sense that we know it; all it did was automate the subroutine copying and allow for parameter passing to the subroutines. But A-0 quickly grew into something more like a compiler. The motivation for this growth was the realization on the part of the coders that they had to get faster. The earliest computers (as described above) could do three additions per second while the UNIVAC I could do 3000. Needless to say, the coders had not accelerated in the same fashion. They wanted to write correct programs faster.
A-0 became A-2 when a three-address machine code module was placed on top of it. This meant the coders could program in TAC, which was very natural for them, for two reasons. There's the natural mathematical aspect: something plus something equals something. In addition, the machines had 12 bits to a word: the first three defined the operation and the other 9 were the three addresses.

These developments were important precursors to the development of "real" compilers, and higher-level programming languages. As we get into the 1950's, we find two tracks of development: a scientific/mathematical track, and a business track. The researchers in both tracks wanted to develop languages more in line with the way people thought about algorithms, as opposed to the way the program had to be coded to get it to run. On the scientific/mathematical track, we find researchers interested in finding a way to input algebraic equations, as they were originally written, and have the machine calculate them. A-3, a follow-on to A-2, was an early mathematical language. Another early one was the Laning and Zierler system at MIT (1953). This language had conditional branches, loops, mathematical functions in a library (including a function to solve differential equations), and print statements. It was an *interpreted* language, meaning each line is translated as it is encountered, and its actions are carried out immediately. Remember that a compiler creates a complete representation of the source program in the target language prior to execution.

By 1954, IBM was all over these new ideas. They created the Formula Translation System (FORTRAN), a set of programs that enabled an IBM 704 to accept a concise formulation of a problem in a precise mathematical notation, and to produce automatically a high-speed 704 program for the solution of the problem. The initial report on this system had several pages on its advantages including "the virtual elimination of coding and debugging, a reduction in elapsed time, and a doubling in machine output." [SAMMET].

The first FORTRAN compiler emerged in 1957 after 18 person-years of effort. It embodied all the language features one would expect, and added much more. Interestingly, IBM had a difficult time getting programmers to use this compiler. Customers did not buy into it right away because they felt it could not possibly turn out *object code* (the output of a compiler) as good as their best programmers. At this time, programming languages existed prior to compilers for those languages. So, "human compilers" would take the programs written in a programming language and translate them to assembly or machine language for a particular machine. Customers felt that human compilers were much better at optimizing code than a machine compiler could ever be. The short-term speed advantage that the machine compiler offered (i.e., it compiled a lot faster than a human) was not as important as the long-term speed advantage of an efficiently optimized executable.

On the business track, we find COBOL (Common Business-Oriented Language). As its name suggests, it was oriented toward business computing, as opposed to FORTRAN, with its emphasis on mathematical computing. Grace Hopper played a key role in the development of COBOL:

"Mathematical programs should be written in mathematical notation; data processing programs should be written in English statements." [WEXELBLAT]

It had a more "English-like" format, and had intensive built-in data storage and reporting capabilities.

Quite a bit of work had to be done on both FORTRAN and COBOL, and their compilers, before programmers and their employers were convinced of their merit. By 1962, things had taken off. There were 43 different FORTRAN compilers for all the different machines of the day. COBOL, despite some very weak compilers in its early days, survived because it was the first programming language to be mandated by the Department of Defense. By the early 60's, it was at the forefront of the mechanization of accounting in most large businesses around the world.

As we explore how to build a compiler, we will continue to trace the history of programming languages, and look at how they were designed and implemented. It will be useful to understand how things were done then, in order to appreciate what we do today.

Programming language design

An important consideration in building a compiler is the design of the source language that it must translate. This design is often based on the motivation for the language: FORTRAN looks a lot like mathematical formulas; COBOL looks a lot like a to-do list for an office clerk. In later languages, we find more generic designs in general-purpose programming languages. The features of these modern programming languages might include a block-structure, variable declaration sections, procedure and function capabilities with various forms of parameter passing, information hiding/data encapsulation, recursion, etc. Today, there is a standard set of principles one follows in designing a general-purpose programming language. Here is a brief list adapted from some important textbooks on programming languages:

- A language should provide a conceptual framework for thinking about algorithms and a means of expressing those algorithms.
- The syntax of a language should be well-defined, and should allow for programs that are easy to design, easy to write, easy to verify, and easy to understand and modify later on.
- A language should be as simple as possible. There should be a minimal number of constructs with simple rules for their combination and those rules should be regular, without exceptions.

- A language should provide data structures, data types and operations to be defined and maintained as self-contained abstractions. Specifically, the language should permit modules designed so that the user has all the information needed to use the module correctly, and nothing more; and the implementer has all the information needed to implement the module correctly, and nothing more.

- The static structure of a program should correspond in a simple way to the dynamic structure of the corresponding computations. In other words, it should be possible to visualize the behavior of a program from its written form.

- The costs of compiling and executing programs written in the language should be carefully managed.

- No program that violates the definition of the language should escape detection.

- The language should (hopefully!) not incorporate features or facilities that tie the language to a particular machine.

We will want to keep these features in mind as we explore programming languages, and think about how to design a programming language ourselves.

Programming paradigms

Programming languages come in various flavors based on the fundamental mechanism for describing a computation. This underlying computational model of a language is called its *paradigm*. Many programming paradigms exist, the four most common of which are

1. *Imperative*: This paradigm has been the most popular over the past 40 years. The languages in this paradigm are statement-oriented, with the most important statement being the assignment. The basic idea is we have machine states that are characterized by the current values of the registers, memory and external storage. As the statements of an imperative language execute, we move from state to state. The assignment statement allows us to change states directly. Control structures are used to route us from assignment to assignment so that statements (and machine states) occur in the correct order. The goal is a specific machine state when the program completes its execution. Languages of this paradigm include FORTRAN, COBOL, ALGOL, PL/I, C, Pascal, and Ada.

2. *Functional*: Another way of viewing computation is to think about the function that a program performs as opposed to state changes as a program executes. Thus, instead of looking at the sequence of states the machine must pass through, we look at the function that must be applied to the initial state to get the desired result. We develop programs in functional languages by writing functions from previously developed

functions, in order to build more complex functions until a final function is reached which computes the desired result. LISP, ML, and Haskell are all excellent examples of the functional paradigm.

3. *Rule-Based* or *Declarative*: Languages in this paradigm check for certain conditions; when the conditions are met, actions take place. Prolog is an important language of this paradigm where the conditions are a set of predicate logic expressions. We will see later that Bison, a parser generation tool, also works along these same lines.

4. *Object-Oriented*: This paradigm is an extension of the imperative paradigm, where the data objects are viewed in a different way. Abstract data types are a primary feature of the languages of this paradigm, but they are different from the ADTs one builds in a language like C. In the object-oriented paradigm, we build *objects*, which consist not only of data structures, but also operations that manipulate those data, structures. We can define complex objects from simpler objects by allowing objects to inherit properties of other objects. Then, the objects that we have created interact with one another in very carefully defined ways. This is a different way of working with data than in the imperative paradigm. Languages of this paradigm include Smalltalk, Eiffel, C++, and Java.

In recent years, the lines of distinction between these paradigms have become blurred. In most imperative languages, we tend to write small functions and procedures that can be considered similar to the functions we might write in a functional language. In addition, the concept of ADTs as implemented in an imperative language is very close in nature to that of objects in an object-oriented language (without the inheritance). Object-oriented features have been added to traditionally imperative languages (C++) and functional ones (CLOS for LISP).

It will be important as we study programming languages and their compilers to know the paradigm of a given language. Many implementation decisions are automatically implied by the paradigm. For example, functional languages are almost always interpreted and not compiled.

Bibliography

Aho, A., Sethi, R., Ullman, J. Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

Appleby, D. Programming Languages: Paradigm and Practice. New York, NY: McGraw-Hill, 1991.

Bennett, J.P. Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

MacLennan, B. <u>Principles of Programming Languages</u>. New York, NY: Holt, Rinehart, Winston, 1987.

Sammet, J. <u>Programming Languages: History and Fundamentals</u>. Englewood-Cliffs, NJ: Prentice-Hall, 1969.

Wexelblat, R.L. <u>History of Programming Languages</u>. London: Academic Press, 1981.