Anatomy of a Compiler

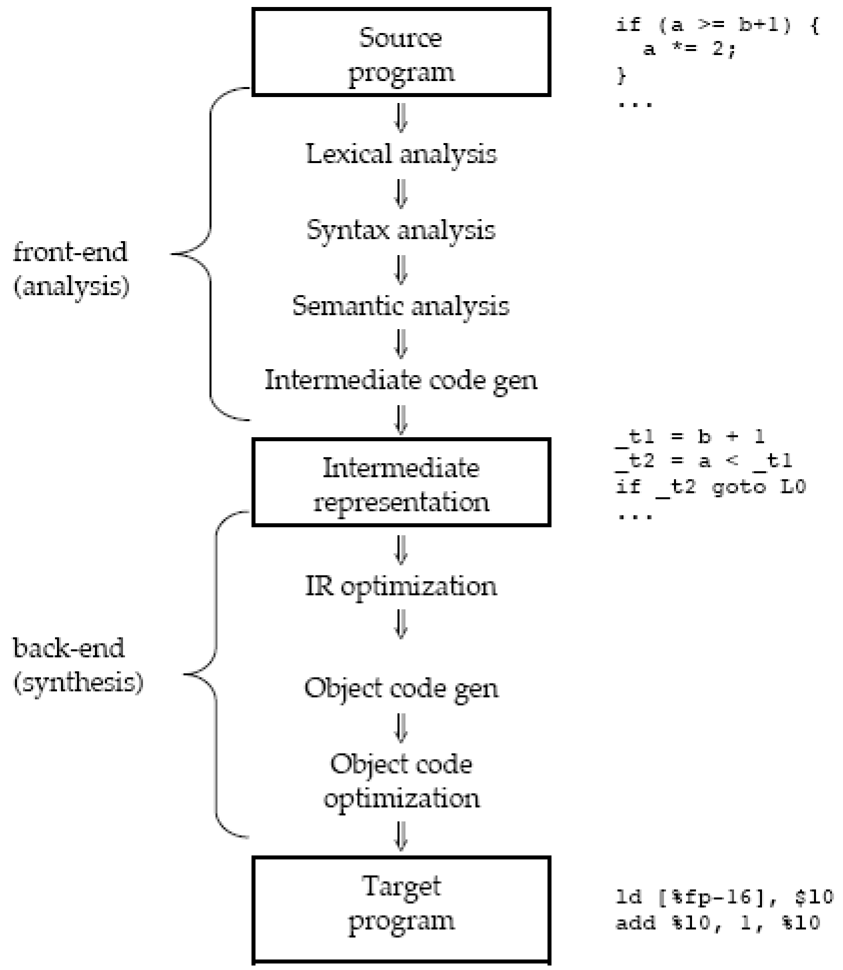
What is a compiler?

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 Sparc assembly that can execute on that designated hardware.

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.

Diagram of the compilation process



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 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. Three-address code is a sequence of simple instructions, each of which can have at most three operands.

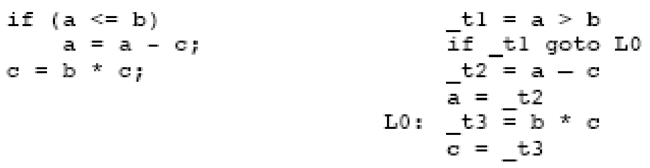
Example of intermediate code generation:

```
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.

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 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.

Reference:

Compiler: http://www.stanford.edu/class/cs143/