Chapter

11

Compilers and Language Translation

Chapter Introduction

r studying this chapter, you will be able to:

List the phases of a typical compiler and describe the purpose of each phase
Demonstrate how to break up a string of text into tokens
Understand grammar rules written in BNF and use them to parse statements
Explain how semantic analysis uses semantic records to determine meaning
Show what a code generator does
Explain the historical importance of code optimization, and why it is less central today

Give an example of local code optimization and an example of global code optimization

11.1 Introduction

Although the high-level languages you learned about in the previous two chapters vary greatly in structure and behavior, they all share one feature: No computer in the world can understand them. There are no "Java computers" or "C++ processors" that can directly execute programs written in the high-level languages of Chapters 9 and 10. In Chapter 6, you learned that assembly language must first be translated into machine language prior to execution. High-level languages must also be translated into machine language prior to execution—in this case by a special piece of system software called a ***compiler***. Compilers for languages like those discussed in Chapters 9 and 10 are very complex programs. They contain tens of thousands of lines of code and may require dozens or hundreds of person-years to complete. Unlike the assemblers of Chapter 6, these translators are not easy to design or implement.

There is a simple explanation for the vast difference in complexity between assemblers and compilers. Assembly language and machine language are related *one to one*; that is, one assembly language instruction produces exactly one machine language instruction. Therefore, translation is really a replacement process in which the assembler looks up a symbolic value in a table (either the op code table or the symbol table) and replaces it by its numeric equivalent:

This is equivalent to translating English into Spanish by looking up each individual English word in an English/Spanish dictionary and replacing it with exactly one Spanish word:

This is a simple way to do translation, and this approach does work for assemblers. Unfortunately, it does not work for most English sentences. Often, a single English word must be translated into a multiword Spanish phrase or vice versa. This same problem exists in the translation of high-level programming languages like Java, C++, or Python.

The relationship between a high-level language and machine language is not one to one but *one to many*. That is, one high-level language statement, such as an assignment or conditional statement, usually produces *many* machine language or assembly language instructions. For example,

Java	***Assembly Language***
; →	LOAD B
	ADD C
	SUBTRACT D
	STORE A

To determine which machine language instructions must be generated, a compiler cannot simply look up a name in a table. Instead, it must do a thorough linguistic analysis of the structure (syntax) and meaning (semantics) of each high-level language statement before deciding what to do. This is far more difficult than table lookup. In fact, building a compiler for a modern high-level programming language can be one of the most difficult system software projects.

When performing a translation, a compiler has two distinct goals. The first is *correctness*. The machine language code produced by the compiler must do exactly what the high-level language statement describes, and nothing else. For example, here is a typical Java assignment statement:

Assume that a compiler translates this statement into the following assembly language code:

```
--Compute the term (B + C)
LOAD B        --Register R holds the value of B
ADD C         --Now it holds the result (B + C)
STORE B       --Let's store the result temporarily in B (see comments below)
--Next compute the term (D + E)
LOAD D        --Register R holds the value of D
ADD E         --Now it holds the result (D + E)
STORE D       --Let's store the result temporarily in D (see comments below)
--Finally, subtract the two terms and store the result in A
LOAD B        --This loads (B + C)
SUBTRACT D    --This is (B + C) − (D + E)
STORE A       --Put the result in A. We are done translating the statement
```

This translation is *wrong*. Although the code does evaluate the expression (B + C) − (D + E) and store the result into A, it does two things it should not do. The translated program destroys the original contents of the variables B and D when it does the first two STORE operations. This is not what the Java assignment operator is supposed to do, and this compiler has produced an incorrect machine language translation of the original high-level language statement.

In addition to correctness, a compiler has a second goal. The code it produces should be reasonably *efficient and concise*. Even though memory costs have come down and processors are much faster, programmers will not accept gross inefficiencies in either execution speed or size of the compiled program. They might not care whether a compiler eliminates every wasted nanosecond or every unnecessary memory cell, but they do want it to produce reasonably fast and efficient machine language code. For example, to compute the sum , an inexperienced programmer might write something like the following:

This loop includes the time-consuming multiplication operation (2.0 * $x[i]$). By the rules of arithmetic, this operation can be moved outside the loop and done just once. A "smart" compiler should recognize this and translate the previous fragment of code as though it had been written:

By restructuring the loop, a smart compiler eliminates 49,999 unnecessary multiplications.

As you can see, we have our work cut out for us in this chapter. We want to describe how to construct a compiler that can read and interpret high-level language statements, understand what they are trying to do, correctly translate their intentions into machine language without any errors or unexpected side effects, and do all of this efficiently and concisely. Now you can appreciate why building a compiler is such a major undertaking.

The remainder of this chapter gives an overview of the steps involved in building a compiler for a procedural, Java-like, or C++-like language. No single chapter could investigate the subtleties and complexities of this huge subject. We can, however, give you an appreciation for some of the issues and concepts involved in designing and implementing this important piece of system software.

Main content

Chapter Contents

11.2 The Compilation Process

The general structure of a compiler is shown in Figure 11.1. Because there is a good deal of variability in the design and organization of a compiler, this diagram should be viewed as a generalized model rather than an exact description of how all compilers are structured.

Figure 11.1 General structure of a compiler

The four phases of compilation listed in Figure 11.1 are the following:

- *Phase I: Lexical analysis*—The compiler examines the individual characters in the source program and groups them into syntactical units, called tokens, that will be analyzed in succeeding stages. This operation is analogous to grouping letters into words prior to analyzing natural language text.
- *Phase II: Parsing*—The sequence of tokens formed by the scanner is checked to see whether it is syntactically correct according to the rules of the programming language. This phase is roughly equivalent to checking whether individual words in a natural language text are connected together in a way that forms grammatically correct sentences.
- *Phase III: Semantic analysis and code generation*—If the high-level language statement is structurally correct, then the compiler analyzes its meaning and generates the proper sequence of machine language instructions to carry out the intended actions.
- *Phase IV: Code optimization*—The compiler takes the generated code and sees whether it can be made more efficient, either by making it run faster, having it occupy less memory, or possibly both.

When these four phases are complete, we have a correct and efficient machine language translation of the original high-level language *source program*. In the final step, this machine language code, called the object program, is written to an *object file*. We have reached the stage labeled "Machine language program" from Chapter 6, Figure 6.4, and the resulting object program can be handled in exactly the fashion shown there. That is, it can be loaded into memory and executed by the processor to produce the desired results.

The overall sequence of operations performed on a high-level language program is summarized in Figure 11.2. The following sections take a closer look at each of the four phases of the compilation process.

Figure 11.2 Overall execution sequence of a high-level language program

Main content

Chapter Contents

11.2.1 Phase I: Lexical Analysis

The program that performs **lexical analysis** is called a **lexical analyzer**, or more commonly a **scanner**. Its job is to group input characters into units called **tokens**—syntactical units that are treated as single, indivisible entities for the purposes of translation. For example, take a look at the following assignment statement:

You probably see an assignment statement containing some symbols (*a*, *b*, *delta*), a number (319), and some operators (=, +, −, ;). However, your eyes and your brain have already done a great deal of processing to recognize and mentally create these objects, just as they do a great deal of processing to create words, sentences, and paragraphs from the individual characters on this section. In the assignment statement shown previously, high-level linguistic objects such as symbols and numbers do not yet exist. Initially, there are only the following 21 characters:

tab, a, blank, =, blank, b, blank, +, blank, 3, 1, 9, blank, –, blank, d, e, l, t, a, ;

It is the task of the scanner to discard nonessential characters, such as blanks and tabs, and then group the remaining characters into high-level syntactical units such as symbols, numbers, and operators. A scanner would construct from the preceding example the following eight tokens:

a
=
b
+
319
–

delta
;

Once this task has been completed, our compiler no longer has to deal with individual characters. Instead, it can work at the level of symbols (*a*, *b*, *delta*), numbers (319), and operators (=, +, −, ;).

In addition to building tokens, a scanner must classify tokens by their type—that is, is the token a symbol, is it a number, is it an assignment operator? Whereas a modern high-level language like C++, Java, or Python may have 50 or more different token types, our simple examples in this chapter are limited to the 11 classifications listed in Figure 11.3.

Figure 11.3

Typical token classifications

Token Type	Classification Number
symbol	1
number	2
=	3
+	4
-	5
;	6
==	7
if	8
else	9
(	10
)	11

The scanner assigns the classification number 1 to all legal symbols, such as *a*, *b*, and *delta*. Similarly, all unsigned numbers, regardless of value, are assigned classification number 2. The reason all symbols and all numbers can be grouped into single classifications is that the grammatical correctness of a statement depends only on whether a legal symbol or a legal number appears in a given location. It does not depend on exactly which symbol or which number is actually used. For example, given the following model of an assignment statement:

"symbol" = "symbol" + "number";

it is possible to determine that a given assignment statement is syntactically correct, regardless of which specific "symbol" and "number" are actually used (as long as they are all legal in the programming language being used).

Using the token types and classification values shown in Figure 11.3, it is now possible to describe exactly what a scanner must do:

The input to a scanner is a high-level language statement from the source program. Its output is a list of all the tokens contained in that statement, as well as the classification number of each token found.

Here are some examples (using the classification values shown in Figure 11.3):

Input:
```
a = b + 319
- delta;
```

Output:

Token	**Classification**
a	1
=	3
b	1
+	4
319	2
-	5
delta	1
;	6

Input:
```
if (a == b) xx =
13; else xx = 2;
```

Output:

Token	**Classification**
if	8
(	10
a	1
==	7
b	1
)	11
xx	1
=	3
13	2

;	6
else	9
xx	1
=	3
2	2
;	6

Regardless of which programming language is being analyzed, every scanner performs virtually the same set of operations:

- (1)

  It discards blanks and other nonessential characters and looks for the beginning of a token;

- (2)

  when it finds the beginning, it puts characters together until

- (3)

  it detects the end of the token, at which point it classifies the token and begins looking for the next one.

  This algorithm works properly regardless of what the tokens look like.

  We can see this process more clearly by looking at an algorithm for grouping natural language characters into words:

This Is English.

Este es Espanol.

Kore wa Nihongo desu.

Even though these three sentences are written in very different languages—English, Spanish, and Japanese—the algorithm for constructing words is identical:

- (1)

  Discard blanks until you find a nonblank character;

- (2)

  group characters together until

- (3)

  you encounter either a blank or the character ".".

  You have now built a word. Go back to Step 1 and repeat the entire sequence to locate the next word. This is essentially the same algorithm that is used to build a lexical scanner for high-level programming languages.

11.2.2 Phase II: Parsing

During the parsing phase, a compiler determines whether the tokens recognized by the scanner during Phase I fit together in a grammatically meaningful way. That is, it determines whether they are a syntactically legal statement of the programming language. This step is analogous to diagramming a sentence. For example, to prove that the sequence of words

The man bit the dog.

is a correctly formed sentence, we must show that the individual words can be grouped together structurally to form a proper English language sentence:

Practice Problems

Using the token types and classification numbers given in Figure 11.3, determine the output of a scanner given the following input statements:

```
x = x + 1;
```

Answer

```
if (a + b42 == 0) a = zz - 12;
```

Answer

Do you think a scanner would classify the following sequence of symbols as a single token or as multiple tokens? Give a reason for your choice.

abc–def

Answer

This would most likely be classified as three tokens: the name abc, the minus sign '-', and the name def. The reason is that a minus sign is rarely allowed to be part of a variable name.

abc_def

Answer

This would likely be classified as a single token with the name abc_def. In many programming languages, the underscore character is allowed to be part of a name to allow the different parts of the variable to stand out, for example, the names inches_per_yard or cost_per_square_meter.

abc def (there is exactly one space between the c and the d)

Answer

This would definitely be classified as two separate tokens, abc and def, because a space is always used to end one token and to identify the start of the next one.

The following character sequence is illegal in virtually all programming languages:

What would you expect a scanner to do if it encounters such a sequence?

Answer

A scanner is not involved in the determination of the legality or illegality of a programming language statement; that is done in the next two steps. A scanner is only concerned with finding and classifying tokens. So it would simply return the following six tokens:

Explain what will happen if the user writes the following assignment statement:

Answer

The symbol “else”, which was chosen as a variable name, will not be assigned symbol category 1 as we would expect. Instead, it will be assigned to category 9, the reserved word “else”. In the next phase of translation, when the compiler analyzes the syntactic structure of this statement, it will not be able to understand what the user meant and will produce an error message. This is why in most programming languages you are not allowed to use reserved words (such as while, do, if, else) as names of variables.

If we are unable to diagram the sentence, then it is not correctly formed. For example, when we try to analyze the sequence, “The man bit the”, here is what happens:

At this point in the analysis, we are stuck because there is no object for the verb “bit.” We cannot diagram the sentence and must conclude that it is not properly formed.

The same thing happens with statements in a programming language, which are roughly analogous to sentences in a natural language. If a compiler is able to “diagram” a statement such as , it concludes that the statement is structurally correct:

The structure shown above is called a **parse tree**. It starts from the individual tokens in the statement, *a*, =, *b*, +, and *c*, and shows how these tokens can be grouped into predefined grammatical categories such as <symbol>, <assignment operator>, and <expression> until the desired goal is reached—in this case, <assignment statement>. (We will explain shortly why we are writing the names of these grammatical categories inside the angle brackets “<” and “>”.) The successful construction of a parse tree is proof that this statement is correctly formed according to the rules of the language. If a parser cannot produce such a parse tree, then the statement is not correctly formed.

In the field of compiler design, the process of diagramming a high-level language statement is called **parsing**, and it is done by a program called a **parser**. The output of a parser is either a completed parse tree or an error message if such a tree cannot be constructed.

Grammars, Languages, and BNF. How does a parser know how to construct the parse tree? What tells it how the pieces of a language fit together? For example, in

the statement shown previously, you might wonder how the parser knows that the format of an assignment statement in our language is

<symbol> = <expression>

The answer is that it does not know; we must tell it. The parser must be given a formal description of the *syntax*—the grammatical structure—of the language that it is going to analyze. The most widely used notation for representing the syntax of a programming language is called **BNF**, an acronym for Backus-Naur Form, named after its designers John Backus and Peter Naur.

In BNF, the syntax of a language is specified as a set of **rules**, also called **productions**. The entire collection of rules is called a **grammar**. Each individual BNF rule looks like this:

left-hand side ::= "definition"

The *left-hand side* of a BNF rule is the name of a single grammatical category, such as <symbol>, <expression>, or <assignment statement>. The BNF operator ::= means "is defined as," and "definition," which is also called the *right-hand side*, specifies the grammatical structure of the symbol appearing on the left-hand side of the rule. The definition may contain any number of objects. For example, here is a BNF rule that defines how an <assignment statement> is formed:

<assignment statement> ::= <symbol> = <expression>

This rule says that the syntactical construct called <assignment statement> is defined as a <symbol> followed by the token = followed by the syntactical construct called <expression>. To have a structurally correct assignment statement, these three objects must all be present in exactly that order.

A BNF rule that gives one possible definition for the English language construct called <sentence> follows.

<sentence> ::= <subject> <verb> <object> .

This BNF rule says that a <sentence> is defined as a <subject> followed by a <verb> followed by an <object> and ending with a period. It is this rule that allowed us to correctly parse the sentence "The man bit the dog." since "The man" is a <subject>, "bit" is a <verb>, "the dog" is an <object>, and there is a period at the end.

Finally, the simple BNF rule

<addition operator> ::= +

says that the grammatical construct <addition operator> is defined as the single character +.

If a parser is analyzing a statement in a language and it sees exactly the same sequence of objects that appears on the right-hand side of a BNF rule, it is allowed to replace them with the one grammatical object on the left-hand side of that rule. For example, given our BNF rule for <assignment statement>:

<assignment statement> ::= <symbol> = <expression>

if a parser encounters the three objects <symbol>, =, and <expression> next to each other in the input, it can replace them with the object appearing on the left-hand side of the rule—in this case, <assignment statement>. In a sense, the parser is constructing one branch of the parse tree, which looks like this:

We say that the three objects, <symbol>, =, and <expression>, *produce* the grammatical category called <assignment statement>, and that is why a BNF rule is also called a production.

BNF rules use two different types of objects, called **terminals** and nonterminals, on the right-hand side of a production. Terminals are the actual tokens of the language recognized and returned by a scanner. The terminals of our language are the 11 tokens listed in Figure 11.3:

<symbol>	==
<number>	if
=	else
+	(
-	)
;	

The important characteristic of terminals is that they are not defined any further by other rules of the grammar. That is, there is no rule in the grammar that explains the "meaning" of such objects as <symbol>, =, +, and *if*. They are simply elements of the language, much like the words *man*, *bit*, and *dog* in our earlier example.

The second type of object used in a BNF rule is a **nonterminal**. A nonterminal is not an actual element of the language but an intermediate grammatical category used to help explain and organize the language. For example, in the analysis of the English sentence "The man bit the dog.", we used grammatical categories called article, noun, verb, noun phrase, subject, and object. These categories help us understand the structure of the sentence and show that it is correctly formed, but they are not actual words of the sentence being studied.

In every grammar, there is one special nonterminal called the **goal symbol**. This is the nonterminal object that the parser is trying to produce as it builds the parse tree. When the parser has produced the goal symbol using all the elements of the sentence or statement, it has proved the syntactical correctness of the sentence or statement being analyzed. In our English language example, the goal symbol is the nonterminal object <sentence>. In our assignment statement example, it is, naturally, <assignment statement>. When this nonterminal goal symbol has been produced, the parser has finished building the tree, and the statement has been successfully parsed. The collection of all statements that can be successfully parsed is called the **language defined by a grammar**.

All nonterminals are written inside angle brackets; examples include <expression> and <assignment statement>. Some terminals are also written in angle brackets when they

do not represent actual characters of the language but rather groups of characters constructed by the scanner, such as <symbol> or <number>. However, it is easy to tell the difference between the two. A terminal such as <symbol> is not defined by any other rule of the language. That is, there is no rule anywhere in the grammar that looks like this:

<symbol> ::= “definition of a symbol”

Terminal symbols are like the words and punctuation marks of a language, and a parser does not have to know anything more about their syntactical structure to analyze a sentence.

However, nonterminals are constructed by the parser from more elementary syntactical units. Therefore, nonterminals such as <expression> and <assignment statement> must be further defined by one or more rules that specify exactly how this nonterminal is constructed. For example, there must exist at least one rule in our grammar that has the nonterminal <expression> as the left-hand side. This rule tells the parser how to form expressions from other terminals and nonterminals:

<expression> ::= “definition of expression”

Similarly, there must be at least one rule that specifies the structure of an assignment statement:

<assignment statementglt; ::= “definition of assignment statement”

We can summarize the difference between terminals and nonterminals by saying that terminals never appear on the left-hand side of a BNF rule, whereas nonterminals must appear on the left-hand side of one or more rules.

The three symbols <, >, and ::= used as part of BNF rules are termed **metasymbols**. This means that they are symbols of one language (BNF) that are being used to describe the characteristics of another language. In addition to these three, there are two other metasymbols used in BNF definitions. The vertical bar, |, means OR, and it is used to separate two alternative definitions of a nonterminal. This could be done without the vertical bar by just writing two separate rules:

<nonterminal> ::= “definition 1”

<nonterminal> ::= “definition 2”

However, it is sometimes more convenient to use the | character and write a single rule:

<nonterminal> ::= “definition 1” | “definition 2”

For example, the rule

<arithmetic operator> ::= + | − | * | /

says that an arithmetic operator is defined as either a +, or a −, or an *, or a /. Without the | operator, we would need to write four separate rules, which would make the grammar much larger. Here is a rule that defines the nonterminal <digit>:

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

We will see many more examples of the use of the OR operator.

The final metasymbol used in BNF definitions is the Greek character lambda, Λ, which represents the **null string**—nothing at all. It is possible that a nonterminal can be "empty," and the symbol Λ is used to indicate this. For example, the nonterminal <signed integer> can be defined as an optional sign preceding an integer value, such as +7 or −5 or 8. To define the idea of an optional sign in BNF, we could say:

<signed integer> ::= <sign> <number>

<sign> ::= + | − | Λ

which says that <sign> may be either a + or a −, or it may be omitted entirely.

Practice Problems

Write a single BNF rule that defines the nonterminal <Boolean operator>. (Assume that the three possible Boolean operators are AND, OR, and NOT.)

Answer

Create a BNF grammar that describes all one- or two-character identifiers that begin with the letter *i* or *j*. The second character, if present, can be any letter or digit. What is the goal symbol of your grammar?

Answer

Write a BNF grammar that describes Boolean expressions of the form *(var op var)*

where var can be one of the symbols *x, y,* and *z*, and *op* can be one of the three relational operators ==, >, and <. The parentheses are part of the expression.

Answer

Using the grammar created in Practice Problem 3, show the parse tree for the expression (*x* > *y*).

Answer

Using the grammar created in Practice Problem 3, show what happens when you try to parse the illegal expression (*x* ==).

Answer

You eventually reach the point in the parse where you have the following sequence:

which does not match the right-hand side of any rule, and the parse fails.

Modify your grammar from Practice Problem 3 so that the enclosing parentheses are optional. That is, Boolean expressions can be written as either (var op var) or var op var.

Answer

The first rule of Practice Problem 3 could be changed to

Do the symbols "Λ" (Greek lambda) and " " (blank) have the same meaning and can they be used interchangeably? Explain why or why not.

Answer

The symbol Λ and the blank are NOT interchangeable. Blank is a character, just like A or 5 or *. However, the symbol Λ means the absence of any character, including a blank space.

Parsing Concepts and Techniques. Given this brief introduction to grammars, languages, and BNF, we can now explain how a parser works. A parser receives as input the BNF description of a high-level language and a sequence of tokens recognized by the scanner. The fundamental rule of parsing follows.

If, by repeated applications of the rules of the grammar, a parser can convert the complete sequence of input tokens into the goal symbol, then that sequence of tokens is a syntactically valid statement of the language. If it cannot convert the input tokens into the goal symbol, then this is not a syntactically valid statement of the language.

To illustrate this idea, here is a three-rule grammar:

Number	***Rule***
1	<sentence> ::= <noun> <verb>.
2	<noun> ::= bees | dogs
3	<verb> ::= buzz | bite

The grammar contains five terminals: bees, dogs, buzz, bite, and the character "." (a period). It also contains three nonterminals: <sentence>, <noun>, and <verb>. The goal symbol is <sentence> because it is the one nonterminal that does not appear on the right-hand side of any other rule. Given this three-rule grammar, we can provide a sequence of tokens such as *dogs*, *bite*, and "." and have the parser attempt to transform these tokens into the goal symbol <sentence> using the three BNF rules given above:

In this case, the parse was successful. (The numbers in the diagram indicate which rule is being applied.) Thus, "dogs bite." is a syntactically valid sentence of the language defined by this three-rule grammar. However, the following sequence of tokens:

leads to a dead end. We have not yet produced the goal symbol <sentence>, but there is no rule in the grammar that can be applied to the sequence <noun> <noun> ".". That is, no sequence of terminals and nonterminals in the parse tree constructed so far matches the right-hand side of any rule. This means that "bees dogs." is not a valid sentence of this language.

Grammars for "real" high-level languages like C++, Python, or Java are very large, containing many hundreds of productions; therefore, it is not feasible to use these grammars as examples in our discussions. Even a grammar describing individual statements can be quite complex. For example, the BNF description of a Java assignment statement, complete with variables, constants, operators, parentheses, and function calls, can easily require 20 or 30 rules. Therefore, the following examples all use highly simplified "toy" languages to keep the level of detail manageable and enable us to focus on important concepts.

Our first example is a grammar for a highly simplified assignment statement in which the only operator is +, numbers are not permitted, and the only allowable variable names are *x*, *y*, and *z*. A first attempt at designing a grammar for this simplified assignment statement is shown in Figure 11.4.

Figure 11.4

First attempt at a grammar for a simplified assignment statement

Number	Rule
1	<assignment statement> ::= <variable> = <expression>
2	<expression> ::= <variable> | <variable> + <variable>
3	<variable> ::= *x* | *y* | *z*

If the input statement is *x* = *y* + *z*, then the parser can determine that this statement is correctly formed because it can construct a parse tree (Figure 11.5). The parse tree of Figure 11.5 is the output of the parser, and it is the information that is passed on to the next stage in the compilation process.

Figure 11.5 Parse tree produced by the parser

Building a parse tree like the one in Figure 11.5 is not as easy as it may appear. Often two or more rules of a grammar may be applied to the current input string, and the parser is not sure which one to choose. For example, assume that a grammar contains the following two rules:

Number	*Rule*
1	<t1> ::= A B
2	<t2> ::= B C

and that the statement being parsed contains the three-character string . . . A B C . . . . We could apply either Rule 1:

or Rule 2:

One of these choices might be correct, whereas the other might lead down a grammatical dead end, and the parser has no idea which is which.

You are probably not aware that a similar situation occurs in the example shown earlier in Figure 11.5. Assume that the parser reaches this position in building the parse tree for the statement :

In Figure 11.5, the parser next groups the three objects <variable>, +, and <variable> into an <expression> using Rule 2. However, at this point the parser has other options. For example, it could choose to parse the nonterminal <variable> generated from the symbol *y* to <expression> using Rule 2 and then parse the sequence <variable> = <expression> to <assignment statement> using Rule 1. This produces the following parse tree:

Unfortunately, this is the wrong choice. Although the parser does generate the goal symbol <assignment statement>, it does not use all of the tokens contained in the statement. An extra plus sign and <variable> are not used. (It accidentally parsed the assignment statement instead of .) The parser has gone down the wrong path and reached a point where it is unable to continue. It must now go back to the point where it made the incorrect choice and try something else. For example, it might choose to parse the nonterminal <variable> generated from *z* to <expression> using Rule 2. Unfortunately, this is also a dead end; it produces the sequence <variable> + <expression>, which does not match the right-hand side of any rule.
The process of parsing is a complex sequence of applying rules, building grammatical constructs, seeing whether things are moving toward the correct answer (the goal symbol), and, if not, "undoing" the rule just applied and trying another. It is much like finding one's way through a maze. You try one path and if it works, fine. If not, you back up to where you made your last choice and try another, hoping that this time it will lead in the right direction.

This sounds like a haphazard and disorganized way to analyze statements, and in fact, it is. However, "real" parsing algorithms don't rely on a random selection of rules, as our previous discussion may have implied. Instead, they try to be a little more clever in their choices by looking ahead to see whether the rule they plan to apply will or will not help them to reach the goal. For example, assume we have the following input sequence:

A B C

and this grammar:

<goal> ::= <term> C

<term> ::= A B | B C

We have two choices on how to parse the input string. We can either group the two characters A B to form a <term>, or we can group B C instead. A random choice causes us to be wrong about half the time, but if a parser is clever and looks ahead, it can do a lot better. It is easy to see that grouping B C to produce the nonterminal <term> leads to trouble, because there is norule telling us what to do with the sequence A <term>. We quickly come to a dead end:

However, by choosing to group the tokens A B into <term> instead of B C, the parser quickly produces a correct parse tree:

There are many well-known **look-ahead parsing algorithms** that use the ideas just described. These algorithms "look down the road" a few tokens to see what would happen if a certain choice is made. This helps keep the parser moving in the right direction, and it significantly reduces the number of false starts and dead ends. These algorithms can do very efficient parsing, even for large languages with hundreds of rules.

There is another important issue in the design of grammars. Let's assume we attempt to parse the following assignment statement:

using the grammar in Figure 11.4. No matter how hard we try to build a parse tree, it is just not possible:

All other attempts lead to a similar result.

The problem is that the grammar in Figure 11.4 does not correctly describe the desired language. We wanted a language that allowed expressions containing an *arbitrary number* of plus signs. However, the grammar of Figure 11.4 describes a language in which expressions may contain at most a single addition operator. More complicated expressions such as $x + y + z$ cannot be parsed, and they are erroneously excluded from our language.

One of the biggest problems in building a compiler for a programming language is designing a grammar that:

- Includes every valid statement that we want to include in the language, and
- Excludes every invalid statement that we do not want to include in the language.

In this case, a statement that should be a part of our language was excluded. If this statement were contained in a program, the parser would not recognize it and the user would receive an error message for a statement that is not really in error. The grammar in Figure 11.4 is wrong in the sense that it does not define the language that we want. Let's redo the grammar of Figure 11.4 so that it describes an assignment statement that allows expressions containing an arbitrary number of occurrences of the plus sign. That is, our language will include such statements as

This second attempt at a grammar is shown in Figure 11.6.

Figure 11.6

Second attempt at a grammar for assignment statements

Number	Rule
1	<assignment statement> ::= <variable> = <expression>
2	<expression> ::= <variable> | <expression> + <expression>
3	<variable> ::= *x* | *y* | *z*

The grammar in Figure 11.6 does recognize and accept expressions with more than one plus sign. For example, here is a parse tree for the statement :

Note that Rule 2 of Figure 11.6 uses the nonterminal <expression> on both the left-hand and the right-hand side of the same rule. In essence, the rule defines the nonterminal symbol <expression> in terms of itself. This is called a **recursive definition**, and its use is very common in BNF. It is recursion that allows us to describe an expression not just

with one or two or three or . . . plus signs but with an *arbitrary and unbounded* number, as shown here.

We have solved one problem: that of making sure our grammar defines a language that includes expressions with multiple addition operators. Unfortunately, though, while one problem has disappeared, another one has arisen, and the grammar of Figure 11.6 is still not quite correct. To demonstrate this new problem, let's take the same statement that we have been analyzing:

and construct a second parse tree using the grammar of Figure 11.6. Both trees are shown in Figure 11.7.

Figure 11.7 Two parse trees for the statement

Using this assignment statement and the grammar in Figure 11.6, it is possible to construct *two* distinct parse trees. This might not seem to be a problem because the construction of a parse tree has been used only to demonstrate that a statement is correctly formed. Building two parse trees implies that the parser has demonstrated correctness in two different ways, perhaps twice as good.

However, a parse tree not only serves to demonstrate that a statement is correct, it also assigns it a specific *meaning*, or *interpretation*. The next phase of compilation uses this parse tree to understand what a statement means, and it generates code on the basis of that meaning. The existence of two different parse trees implies two different interpretations of the same statement, which is disastrous. A grammar that allows the construction of two or more distinct parse trees for the same statement is said to be **ambiguous**.

This problem can occur in natural languages as well as programming languages. Consider the following ambiguous sentence:

I saw the man in the store with the dogs.

This sentence has two distinct meanings depending on how we choose to parse it:

These two interpretations say very different things, so the sentence leaves us confused about what the speaker meant. In the areas of languages and grammars, ambiguity is decidedly not a desirable property.

The two parse trees shown in Figure 11.7 correspond to the following two interpretations of the assignment statement .

`x = (x + y) + z` (Do the operation *x + y* first.)

`x = x + (y + z)` (Do the operation *y + z* first.)

Because addition is associative—that is, —in this case the ambiguity does not cause a serious problem. However, if the statement were changed slightly to

then these two different interpretations lead to completely different results:

`x = (x - y) - z`	which evaluates to `x - y - z`
`x = x - (y - z)`	which evaluates to `x - y + z`

We now have a situation in which a statement could mean one thing using compiler C on machine M and something totally different using compiler C9 on machine M9, depending on which parse tree it happens to construct. This contradicts the spirit of machine independence, which is a basic characteristic of all high-level languages.

To solve the problem, the assignment statement grammar must be rewritten a third time so that it is no longer ambiguous. This new grammar is shown in Figure 11.8. To see that the grammar of Figure 11.8 is not ambiguous, try parsing the statement in the two ways shown in Figure 11.7. You will see that one of these two parse trees cannot be built (see Problem 2 in the next set of Practice Problems).

Figure 11.8

Third attempt at a grammar for assignment statements

Number	Rule
1	<assignment statement> ::= <variable> = <expression>
2	<expression> ::= <variable> | <expression> + <variable>
3	<variable> ::= *x* | *y* | *z*

Figure 11.9 shows the BNF grammar for a simplified version of an *if-else* statement that allows only a single assignment statement in the two separate clauses and allows the else clause to be omitted. The <Boolean expression> can include at most a single use of the relational operators ==, <, and >. The nonterminal <assignment statement> is defined in the same way as in Figure 11.8. Figure 11.10 then shows the parse tree for the statement

using the grammar of Figure 11.9.

Figure 11.9 Grammar for a simplified version of an *if-else* statement

Figure 11.10 Parse tree for the statement if (*x* = = *y*) *x* = *z*; else *x* = *y*;

Even though this *if-else* statement has been greatly simplified, its grammar still requires seven rules. The parse trees for actual conditional statements can become quite large and "bushy."

Practice Problems

Using the grammar of Figure 11.8, show the parse tree for the assignment statement

Answer

Using the grammar of Figure 11.8, show the parse tree for the assignment statement

Answer

Using the grammar of Figure 11.9, show the parse tree for the statement if

Answer

What language is described by the following pair of rules?

<string> ::= <character> | <character> <string>

<character> ::= a | b

Answer

The language consists of all strings of length 1 or more containing an arbitrary sequence of *a*s and *b*s.

Write a BNF grammar that describes strings containing any number of repetitions of the character pair AB. That is, all of the following strings are part of the language:

AB ABAB ABABAB ABABABABAB ...

Answer

Is the following grammar ambiguous?

<goal> ::= <left> <op> <right>

<left> ::= a | b

<op> ::= *

<right> ::= c | d

Answer

No, this grammar is not ambiguous. There are only four sentences that belong to the language defined by this grammar:

, , , and
and all of these sentences have only a single possible parse tree.

Main content

Chapter Contents

11.2.3 Phase III: Semantics and Code Generation

Let's analyze the following three-rule grammar:

<sentence> ::= <noun> <verb>.

<noun> ::= dogs | bees

<verb> ::= bite | bark

The language defined by this grammar contains exactly four sentences:

dogs bite.

dogs bark.

bees bite.

bees bark.

For each of these four sentences, we can construct a parse tree showing that it is (structurally, at least) a valid sentence of the language:

There is one problem, though. Although the sentence "bees bark." is structurally valid, it makes no sense whatsoever! During parsing, a compiler deals only with the *syntax* of a statement—that is, its grammatical structure. At that point, the only "correctness" that a compiler can determine is grammatical correctness with respect to the syntactical rules of the language. In this limited sense the sentence "bees bark." is perfectly correct. Another example of this limitation is the sentence, "The man bit the dog." This sentence is structurally correct, but its meaning is somewhat unusual!

The next phase of translation, during which a compiler examines the semantics of a programming language statement, deals with this issue. It analyzes the meaning of the tokens and tries to understand the *actions* they perform. If the statement is meaningless, as "bees bark." is, then it is semantically rejected, even though it is syntactically correct. If the statement is meaningful, then the compiler translates it into machine language.

It is easy to give examples of English-language sentences that are syntactically correct but semantically meaningless:

The orange artichoke flew through the pink eight-legged elephant.

But what are semantically meaningless statements in high-level programming languages?

One possibility is the following assignment statement:

This is obviously correct syntactically, but what if the variables *sum*, *a*, and *b* are declared as follows:

What does it mean to add a character to a real number? What would possibly be the result of adding the letter 'Q' to 3.1416? In most cases, this operation has no meaning, and perhaps it should be rejected as semantically invalid.

To check for this semantic error, a compiler must look at the parse tree to see whether there is a branch that looks something like this:

If there is such a branch, then the compiler must examine the data types of the two expressions to see whether they "make sense." That is, it must determine whether addition is defined for the data types of the two expressions.

The compiler does this by examining the **semantic records** associated with each nonterminal symbol in the grammar, such as <expression> and <variable>. A semantic record is a data structure that stores information about a nonterminal, such as the actual name of the object that it represents and its data type. For example, the nonterminal <variable> might have been constructed from a character variable named CH. This relationship is represented by a link between the nonterminal <variable> and a semantic record containing the name CH and its data type, char. Pictorially, we can represent this link as:

The initial semantic records in our parse tree are built by the compiler when it sees the declarations of new objects; that is, when it sees declarations such as `char CH;`. Additional semantic records are constructed as the parse tree grows and new nonterminals are produced. Thus, a more realistic picture of the parse tree for the expression *a* + *b* (assuming both are declared as integers) would look like this:

This parse tree says that we are adding two <expression>s that are integer variables named *a* and *b*. The result is an <expression> stored in the integer variable *temp*, a name picked by the compiler. Because addition is well defined for integers, this operation makes perfectly good sense, and the compiler can generate machine language instructions to carry out this addition. If, however, the parse tree and its associated semantic records looked like this:

the compiler determines that this is not a meaningful operation because addition is not defined between a real number and a character. The compiler rejects this parse tree for semantic rather than syntactical reasons.

Thus, the first part of code generation involves a pass over the parse tree to determine whether all branches of the tree are semantically valid. If so, then the compiler can generate machine language instructions. If not, there is a semantic error, and generation of the machine language is suppressed because we do not want the processor to execute meaningless code. This step is called **semantic analysis**.

Following semantic analysis, the compiler makes a second pass over the parse tree, not to determine correctness (it has already done that), but to produce the translated

machine language code. Each branch of the parse tree represents an action, a transformation of one or more grammatical objects into other grammatical objects. The compiler must determine how that transformation can be accomplished in machine language. This step is called **code generation**.

Let's work through the complete semantic analysis and code generation process using the parse tree for the assignment statement , where *x*, *y*, and *z* are all integers. The example uses the instruction set shown in Chapter 6, Figure 6.5.

Typically, code generation begins at the productions in the tree that are nearest to the original input tokens. The compiler takes each production and, one branch at a time, translates that production into machine language operations or data generation pseudo-ops. For example, the following branch in the parse tree:

can be implemented by allocating space for the variable *y* using the .DATA pseudo-op

```
Y:   .DATA   0
```

In addition to generating this pseudo-op, the compiler must build the initial semantic record associated with the nonterminal <variable>. This semantic record contains, at a minimum, the name of this <variable>, which is *y*, and its data type, which is integer. (The data type information comes from the *int* declaration, which is not shown.) Here is what is produced after analyzing and translating the first branch of the parse tree:

Identical operations are done for the branches of the parse tree that produce the nonterminal <variable> from the symbols *x* and *z*, leading to the following situation:

The production that transforms the nonterminal <variable> generated from *y* into the nonterminal <expression>

does not generate any machine language code. This branch of the tree is really just the renaming of a nonterminal to avoid the ambiguity problem discussed earlier. This demonstrates an important point: Although most branches of a parse tree produce code, some do not. Although no code is produced, the compiler must still create a semantic record for the new nonterminal <expression>. It is identical to the one built for the nonterminal <variable>.

The branch of the parse tree that implements addition:

can be translated into machine language using the assembly language instruction set presented in Section 6.3.1. The compiler loads the value of <expression> into a register, adds the value of <variable>, and stores the resulting <expression> into a temporary

memory location. This can be accomplished using the LOAD, ADD, and STORE operations in our instruction set. The names used in the address field of the instructions are determined by looking in the semantic records associated with the nonterminals <expression> and <variable>. The code generated by this branch of the parse tree is

```
LOAD    Y
ADD     Z
STORE   TEMP
```

TEMP is the name of a memory cell picked by the compiler to hold the result (Y + Z). Whenever the compiler creates one of these temporary variables, it must also remember to allocate memory space for this new variable using the DATA pseudo-op

```
TEMP:   .DATA 0     0
```

In addition, the compiler records the name (TEMP) and the data type (*integer*) of the result in the semantic record associated with this new nonterminal called <expression>. Here is what is produced by this branch of the parse tree:

The final branch of the parse tree builds the nonterminal called <assignment statement>:

This production is translated into machine language by loading the value of the <expression> on the right-hand side of the assignment operator, using a LOAD instruction, and storing it, via a STORE operation, into the <variable> on the left-hand side of the assignment operator. Again, the names used in the address fields of the machine language instructions are obtained from the semantic records associated with <variable> and <expression>. The machine language code generated by this branch of the parse tree is

```
LOAD    TEMP
STORE   X
```

The compiler must also build the semantic record associated with the newly created nonterminal <assignment statement>. The name (*x*) and the data type (*integer*) of the variable on the left-hand side of the assignment operator are copied into that semantic record because the value stored in that variable is considered the value of the entire assignment statement.

Our compiler has now analyzed every branch in the parse tree, and it has produced the following translation. (We have separated the pseudo-ops and executable instructions for clarity.)

```
            LOAD        Y
            ADD         Z
            STORE       TEMP
            LOAD        TEMP
            STORE       X
            .
            .
            .
X:          .DATA       0
Y:          .DATA       0
Z:          .DATA       0
TEMP:       .DATA       0
```

This is an exact translation of the assignment statement .

Figure 11.11 shows the code generation process for the slightly more complex assignment statement . The branches of the parse tree are labeled and referenced by comments in the code. (The parse tree was constructed using the grammar shown in Figure 11.8.)

Figure 11.11Code generation for the assignment statement

The code of Figure 11.11 could represent the end of the compilation process because generating a correct machine language translation was our original goal. (*Note*: We have shown the output of the compiler in assembly language, not binary machine language. This assembly language result could easily be converted to machine language using the assembler techniques described in Chapter 6.) However, we are not quite finished. In the beginning of the chapter, we said that a compiler really has *two* goals: correctness and efficiency. The first goal has been achieved, but not necessarily the second. We have produced correct code, but not necessarily good code. Therefore, the next and final operation is *optimization*, where the compiler polishes and fine-tunes the translation so that it runs a little faster or occupies a little less memory.

Practice Problem

Go through the code generation process for the simple assignment statement where *x* and *y* are integers. The parse tree for this statement is

For each branch in the tree, show what semantic records are created and what code is generated.

Answer

The parse tree for this expression is

During the construction of this parse tree, you will build four semantic records: two for <variable>, one for <expression>, and one for <assignment statement>.

The code generated is

11.2.4 Phase IV: Code Optimization

As you learned in Chapter 10, the first high-level language and compiler was Fortran, which appeared in 1957. (It was created by John Backus, the *B* of BNF.) At that time, everyone programmed in assembly language because nothing else was available. Given all the shortcomings of assembly language, you might think that programmers would have flocked to Fortran and thanked their lucky stars that it was available. After all, it is certainly a lot easier to understand the statement than the rather cryptic sequence LOAD B, ADD C, STORE A.

In fact, programmers did not accept this new language very quickly. The reason had nothing to do with the power and expressiveness of Fortran. Everyone admitted that it was far superior to assembly language in terms of clarity and ease of use. The problem had to do with efficiency—the ability to write highly optimized programs that contained no wasted microseconds or unnecessary memory cells.

In 1957 (early second-generation computing), computers were still enormously expensive; they typically cost millions of dollars. Therefore, programmers cared more about avoiding wasted computing resources than simplifying their job. The productivity of programmers earning $2 per hour was unimportant compared with optimizing the use of a multimillion-dollar computer system. In 1957, the guiding principle was "Programmers are cheap, hardware is expensive!"

When programmers used assembly language, they were working on the actual machine, not the virtual machine created by the system software (and described in Chapters 6 and 7). They were free to choose the instructions that ran most quickly or used the least amount of memory. For example, if the INCREMENT, LOAD, and STORE instructions execute in 1 μsec, whereas an ADD takes 2 μsec, then translating the arithmetic operation as

```
INCREMENT       X -- x + 1      This takes 1 μsec
INCREMENT       X -- x + 2      This takes 1 μsec
INCREMENT       X -- x + 3      and this takes 1 μsec
```

requires 3 μsec to execute. This code runs 25% faster than if it had been translated as

```
        LOAD        X           -- This takes 1 μsec
        ADD         THREE       -- x + 3 This takes 2 μsec
        STORE       X           -- and this takes 1 μsec
        .
        .
        .
THREE:  .DATA       3
```

which takes 4 μsec to execute and requires an additional memory cell to hold the integer constant 3. When programmers wrote in assembly language, they were free to choose the first of these sequences rather than the second, knowing that it is faster and more compact. However, in a high-level language like Fortran, a programmer can only write and hope that the compiler is "smart enough" to select the faster of the two implementations.

Because efficiency was so important to programmers of the 1950s and 1960s, these early first- and second-generation compilers spent a great deal of time doing **code**

optimization. In fact, Backus himself did not regard language design as a difficult problem, but merely a prelude to the real problem: designing a compiler that could produce efficient programs. These compiler pioneers were quite successful in solving many of the problems of optimization, and early Fortran compilers produced object programs that ran nearly as fast as highly optimized assembly language code produced by top-notch programmers. After seeing these startling results, programmers of the 1950s and 1960s were eventually won over. They could gain the benefits of high-level languages—a powerful virtual environment—without any loss of efficiency. The code optimization techniques developed by Backus and others were one of the most important reasons for the rapid acceptance of high-level programming languages during the early years of computer science.

However, conditions have changed dramatically since 1957. Because of impressive reductions in hardware costs, code optimization no longer plays the central role it did 50 or 60 years ago. Programmers rarely worry about saving a few memory cells when even an inexpensive tablet has 64 GB of memory. Similarly, as processor speeds increase to 1–10 Gflops (billions of floating-point instructions per second), removing a few instructions becomes much less important. For example, eliminating the execution of 1,000 unnecessary instructions saves only 0.000001 second on a 1 Gflop machine. Therefore, compilers are no longer judged solely on whether they produce highly optimized code.

Whereas hardware costs are dropping, programmer costs are rising dramatically. A powerful high-speed graphics workstation can be purchased for as little as $1,000, but the programmers developing software for that system may earn 75 to 150 times that in annual salary. The operational phrase of the 21st century is the exact opposite of what was true in the 1950s: "Hardware is cheap, people are expensive!" The primary goal in compiler design today is to provide a wide array of compiler tools that simplify the programmer's task and increase his or her productivity. This includes such tools as **visual development environments** that use graphics and video to let the programmer see what is happening, sophisticated **online debuggers** to help programmers locate and correct errors, and reusable code libraries, which contain a large collection of prewritten and fully debugged program units. When a compiler is embedded within a collection of supporting software development routines such as debuggers, editors, toolkits, and libraries, it is called an integrated development environment (IDE). It is these types of programmer productivity optimizations, rather than computer speed and memory optimizations, that have taken center stage in language and compiler design. Often, these sophisticated IDEs are provided transparently to software developers via the cloud computing techniques described in Section 7.5.

However, this does not mean that code optimization is no longer of any importance or that programmers will tolerate any level of code inefficiency. A little bit of effort by a compiler can often pay large dividends in reduced memory space and lower running time. Thus, optimization algorithms are still included, at least to some level, as a component of modern compilers.

There are two types of compiler optimizations: local optimization and global optimization. The former is relatively easy and is part of virtually all compilers. The latter is more difficult and is often omitted from all but the most sophisticated and expensive production-level *optimizing compilers*.

In **local optimization**, the compiler looks at a very small block of instructions, typically from one to five. It tries to determine how it can improve the efficiency of this local code block without regard for what instructions come before or after. It is as though the compiler has placed a tiny "window" over the code, and it optimizes only the instructions inside this optimization window:

Here is a list of some possible local optimizations:

1. **Constant evaluation**—Arithmetic expressions are fully evaluated at compile time if possible, rather than at execution time.

2. **Strength reduction**—Slow arithmetic operations are replaced with faster ones. For example, on most computers increment is faster than addition, and addition is faster than multiplication, which in turn is faster than division. Whenever possible, the compiler replaces an operation with one that is equivalent but executes more quickly.

3. **Eliminating unnecessary operations**—Instructions that are correct, but not necessary, are discarded. For example, because of the nondestructive read principle, when a value is stored from a register into memory, its value is still in the register, and it does not need to be reloaded. However, because the code generation phase translates each statement individually, there may be some unnecessary LOAD and STORE operations:

The code in Figure 11.11 contains two opportunities for local optimizations:

- There are unnecessary LOAD and STORE operations. For example, the first four instructions in Figure 11.11 read

```
LOAD    X
ADD     Y
STORE   TEMP
LOAD    TEMP
```

- The STORE and LOAD operations on Lines 3 and 4 are both unnecessary because the sum (X + Y) is still in register R.

- The code uses two memory cells called TEMP and TEMP2 to hold temporary values. Neither of these variables is needed.
  Locally optimized code for the assignment statement is shown in Figure 11.12. It uses only 7 instructions and data generation pseudo-ops rather than the 13 of Figure 11.11, a savings of about 45%.

Figure 11.12 Optimized code for the assignment statement

The second type of optimization is **global optimization**, and it can be much more difficult. In global optimization, the compiler looks at large segments of the program, not just small pieces, to determine how to improve performance. The compiler examines large blocks of code such as *while* loops, *if* statements, and procedures to determine how to speed up execution. This is a much harder problem, both for a compiler and for a human programmer, but it can produce enormous savings in time and space. For example, earlier in the chapter we showed a loop that looked like this:

"Now I Understand," Said the Machine

Chapter 6 showed that translating assembly language into machine language is relatively easy. This chapter demonstrated that translating high-level programming languages into machine language is more difficult, but it still can be done. What about the next step—the translation of natural languages such as English, Spanish, or Chinese into machine language?

Getting computers to understand and use natural language is a far more difficult problem than translating programming languages like Java and C++. In fact, for many years **natural language understanding** was viewed as the single most difficult research problem in computer science. Demonstrated success was always "just over the next hill," and for many years true natural language understanding remained an unattainable goal. Many in computer science were pessimistic about the possibility of ever giving a computer true language understanding capabilities. However, that pessimism was shown to be unfounded when, in February 2011, a computer program called Watson, developed by artificial intelligence and natural language researchers at IBM, defeated two human players in a game of *Jeopardy!* The computer was presented with the same questions as the human contestants, Ken Jennings and Brad Rutter, the two most successful players in the game's history. Watson had to parse and understand English sentences that might include puns, similes, metaphors, and obscure pop culture references, then search its vast database to find the correct answer, and "buzz in" before either of the two human contestants. Watson won the game and the first place prize of $1 million. For the last half dozen years IBM researchers have continued to refine and improve the natural language and data access algorithms used on that TV show. Today, Watson and its sophisticated language capabilities are being used in over 45 different countries and 20 different industries, doing things like assisting physicians with medical diagnostics, improving online customer support by quickly and efficiently responding to user questions, translating news feeds in real time, and even chatting with management of the Toronto Raptors to help them select the best players in the NBA draft.

By moving the multiplication operation outside the loop, it is possible to eliminate 49,999 unnecessary and time-consuming operations. A good optimizing compiler would analyze the entire loop and restructure it as follows:

Such restructuring requires the ability to look at more than a few instructions at a time. The compiler cannot look at only a small "optimization window" but must be able to examine and analyze large segments of code. It requires a compiler that can see the "big picture," not just a small scene. Seeing this big picture is difficult, and many compilers are unable to do the type of global optimizations just discussed.

We close this section with one extremely important fact about code optimization: It *cannot* make an inefficient algorithm efficient. As we learned in Chapter 3, the efficiency of an algorithm is an inherent characteristic of its structure; It is not

something programmed in by a programmer or optimized in by a compiler. A sequential search program, written by a team of world-class programmers and optimized by the best compiler available, will not run as fast as a nonoptimized binary search program written by firstyear computer science students. Code optimization should not be seen as a way to create fast, efficient programs. That goal is achieved when we decide which algorithm to use. Optimization is more like the "frosting on the cake," whereby we take a good algorithm and make it just a tiny bit better.

Main content

Chapter Contents

11.3 Conclusion

This chapter has touched on some of the many issues involved in compiler design. Topics such as syntax, grammars, parsing, semantics, and optimization are rich and complex, each worthy of an entire book rather than one brief chapter. In addition, there are many topics not mentioned here that play an important role in compiler design:

- Integrated development environments (IDEs) and support tools
- Compilers for alternative languages, such as functional, object-oriented, or parallel languages
- Language standardization
- Top-down versus bottom-up parsing algorithms
- Error detection and recovery

The key point is that, unlike the assemblers of Chapter 6, a compiler is hard to build, and compilers for languages like C++, Python, and Java are large, complicated pieces of software. John Backus reported that the construction of the first Fortran compiler in 1957 required about 18 person-years of effort to design, code, and test. Even though we know much more today about how to build compilers, and numerous support tools are available to assist in this effort, it still requires a large team of programmers working months or years to build a correct and efficient compiler for a modern high-level programming language.

This chapter and the previous two chapters looked at the implementation phase of software development. They focused on the languages used to write programs and the methods used to translate programs into instructions that can be executed by the hardware. However, there are limits to computing. Chapter 12 will show that, no matter how powerful your hardware capabilities and no matter how sophisticated and expressive your programming language, there are some problems that simply cannot be solved algorithmically.