Chapter 2 - Issues of Language Design

This chapter presents a number of concepts that are necessary for a complete understanding of the remainder of the text, including language specification and a description of several languages that can be used to describe the syntax of other languages. These are called metalanguages and will prove useful throughout the book for expressing language syntax. Approaches that are used to formally describe the semantics of a programming language are briefly outlined. The structure of a programming language is frequently affected by the process used to translate programs from that language into an executable form. An overview of that translation process is presented.

A number of characteristics of a programming language that enhance its effectiveness are outlined, standards against which languages and their various features can be judged are described, and criteria useful for choosing the appropriate language for a given application are listed.

2.1 Characteristics of Language Design

The specification of a natural language, such as English, evolves over time through use by those who communicate in that language. Problems arise, however, when the population that communicates in that language is widely separated geographically and culturally. In this case, dialects develop that inhibit the ability of some groups to communicate with others, even though they share the same language. For this reason, formal specifications of natural languages are created in the form of dictionaries and guides to grammar and style, providing a common, formalized description of the language.

Usually, even people who speak different dialects of the same language are able to communicate without a common specification, because natural languages have a great deal of built-in redundancy, and people are able to derive meaning from statements intelligently even if they do not understand everything. It is extremely important, however, for programming languages to be formally specified, because they lack the redundancy of natural languages, and one of the communicating parties (the computer) requires strict conformance of expressions to the rules of the language in order to understand their meaning. There are three classes of people who need formal programming language specifications:

1. Language designers -- those who design a language need some means to express the language they create.

2. Language implementors -- those who write compilers and interpreters for languages need to have a formal specification to unambiguously describe the language they are implementing. Compilers and interpreters are actually language specifications that describe a language to the computer. It is important for different translation programs to share a common language syntax definition so that their implementations can be consistent.

3. Language users -- those who program in a language must have specifications that formally describe legal structures and the meaning of those structures so they can form correct programs in that language.

It is convenient to divide the formal specification of programming languages into two
parts: syntax and semantics. This division is useful because each of these two has its own set of
tools used in expressing its part of the specification.

Syntax refers to the description of those sets of strings that represent valid structures in a
language. In fact, a syntactic specification of a language divides the set of all strings of characters
into two mutually exclusive groups: those that are in the language and those that are not. Tools
used to specify syntax formally must, therefore, consist of rules that can be applied to a string of
characters in order to determine in which of the two groups it lies. This set of rules is the syntactic
specification of a language. The tool we introduce and use in this book for syntactic specification
is EBNF.

The semantics of a programming language associate a meaning with each syntactically
valid construct. More specifically, the semantics of a language describe the actions that will occur
when the program associated with any valid construct in that language is executed by a computer.
On a functional level, the semantic specification gives a way of determining, for any valid pro-
gram and any valid input to that program, what the resulting output of that program will be.

To understand the difference between syntactic and semantic specifications, let us suppose
we have such specifications for some hypothetical language Z. The syntactic specification of the
language will describe a process whereby any string of characters can be tested as to whether it
forms a valid program in Z. The semantic specification of Z will define a notation by which any
string that passes the syntactic test can have its actions specified. As you can see from this illustra-
tion, the application of the syntactic specification precedes that of the semantic specification.

It is common to refer to a program as being syntactically correct if the code for the pro-
gram satisfies all of the syntactic rules of the language. In other words, a syntactically correct pro-
gram is any valid : Give an example of an English sentence that is syntactically correct, but not
semantically correct.program in a language. A program is said to be semantically correct if it is
syntactically correct and if the semantic interpretation of the program, namely what it does,
matches the specifications of the program that define what it is supposed to do.

In the following two sections, we examine some tools for constructing syntactic and
semantic specifications. You will see there that syntactic specification is a much more exact pro-
cess than semantic specification. Whereas the tool introduced for syntactic specification is univer-
sally accepted because it can so thoroughly and accurately specify syntax, there is no tool that is
similarly successful in semantic specification. Three approaches to semantic specification are pre-
SENTed in Section 2.3.

Discuss: Give an example of an English sentence that is syntactically correct but not
semantically correct.

Discuss: Give an example of an English sentence that is semantically correct but not
syntactically correct.

Discuss: How do you resolve questions about the syntax of a programming language
that you have used?

2.2 Language Specification

2.2.1 Syntax Specification
Recall from Section 2.1 that the description of a language is commonly broken into two parts: syntax and semantics. The syntax of a language is the set of rules that determines which constructs are correctly formed programs and which are not. The semantics of a language is the description of the way a syntactically correct program is interpreted, or carried out. For example, the syntax of Java tells us that

\[ a = b; \]

forms a correct assignment statement, whereas the interpretation of the statement as "replace the value of \( a \) with the current value of \( b \)" is a representation of Java's semantics.

In our discussions in this text, we will use a specific formal tool, known as EBNF, to describe language syntax. It's precursor, BNF, is described in Section 2.2.1.1 and EBNF is defined in detail in Section 2.2.1.2. Language syntax specification is important for two reasons. First, it is useful in describing a language, and we will use it throughout the text to describe both real and theoretical programming language constructs. But language specifications are also useful in the verification of the validity of programs, because they give us a set of rules against which an allegedly legal program can be tested. Such syntactic verification is typically carried out by language translation programs such as compilers.

### 2.2.1.1 Grammars

The syntax of a language is described by a **grammar**, which is a set of rules that defines all of the valid constructs that can be accepted in the language. The basic elements of a grammar are as follows:

1. **Set of terminal symbols**—These symbols are the atomic (non-divisible) symbols that can be combined to form valid constructs in the language. **Terminal symbols** most commonly are a set of characters, although some languages may consider certain character strings to be symbols as well.

2. **Set of nonterminal symbols**—These symbols are not included in the program text of the language itself but are symbols used to represent intermediate definitions within the language as defined by productions. These **nonterminal symbols** represent syntactic classes or categories.

3. **Set of productions**—A **production** is a definition of a nonterminal symbol. It is of the form

\[ x ::= y \]

where \( x \) is a nonterminal symbol and \( y \) is a sequence of symbols each of which can be either terminal or nonterminal.

4. **Start symbol**—One symbol from the set of nonterminal symbols is specified as the **start symbol**. This is also sometimes called the distinguished symbol or the goal symbol.

Two rules must be obeyed for these four components to form a grammar:

1. Every nonterminal symbol must appear to the left of the ::= of at least one production.
2. The start symbol must not appear to the right of the ::= of any production.

To illustrate this concept of a grammar, let us construct a simple grammar for describing a calculator language. This grammar is given in Figure 2.1. Note that this grammar has 16 terminal
symbols and 8 nonterminals. Also note that every nonterminal except for the start symbol has multiple productions defining it. Although such multiple productions are not required, they are common and represent alternate definitions. For example, \texttt{value} is defined alternatively as a \texttt{number} or a \texttt{number} preceded by a \texttt{sign}. 
Figure 2.1 Grammar for Calculator Language

Terminal Symbols: 0 1 2 3 4 5 6 7 8 9 + - * / = .

Nonterminal Symbols: calculation
                  expression
                  value
                  number
                  unsigned
                  digit
                  sign
                  operator

Start Symbol: calculation

Productions:
1. calculation ::= expression =
2. expression ::= value
3. expression ::= value operator expression
4. value ::= number
5. value ::= sign number
6. number ::= unsigned
7. number ::= unsigned . unsigned
8. unsigned ::= digit
9. unsigned ::= digit unsigned
10. digit ::= 0
11. digit ::= 1
12. digit ::= 2
13. digit ::= 3
14. digit ::= 4
15. digit ::= 5
16. digit ::= 6
17. digit ::= 7
18. digit ::= 8
19. digit ::= 9
20. sign ::= +
21. sign ::= -
22. operator ::= +
23. operator ::= -
24. operator ::= *
25. operator ::= /

You can also see that recursive productions are permitted, that is, productions where the nonterminal being defined is also found in its own definition on the right-hand side of the production. The nonterminal unsigned, for example, is defined recursively as either a digit or a
digit followed by another unsigned. Another way of stating this is an unsigned consists of one or more digits.

There are two ways a grammar such as that found in Figure 2.1 can be used. The first is to generate valid programs in the language. If we begin with the start symbol and at each step substitute some definition for a nonterminal, proceeding until all remaining symbols are terminal symbols, we have generated a valid program. Examine the generation sequence found in Figure 2.2. At each step, the left-most nonterminal was replaced by a definition from one of the productions. Because multiple productions might apply to any nonterminal, there are possibly several choices that can be made at each nonterminal. The choice of a production to apply is arbitrary in these cases. The choice of the left-most nonterminal for expansion is also arbitrary. An infinite number of valid calculations can be generated in this way.

Figure 2.2 Generation of a calculation using the grammar in Figure 2.1

<table>
<thead>
<tr>
<th>Current String</th>
<th>Production Applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculation</td>
<td>1</td>
</tr>
<tr>
<td>expression =</td>
<td>3</td>
</tr>
<tr>
<td>value operator expression =</td>
<td>4</td>
</tr>
<tr>
<td>number operator expression =</td>
<td>6</td>
</tr>
<tr>
<td>unsigned operator expression =</td>
<td>9</td>
</tr>
<tr>
<td>digit unsigned operator expression =</td>
<td>12</td>
</tr>
<tr>
<td>2 unsigned operator expression =</td>
<td>8</td>
</tr>
<tr>
<td>2 digit operator expression =</td>
<td>15</td>
</tr>
<tr>
<td>25 operator expression =</td>
<td>24</td>
</tr>
<tr>
<td>25* expression =</td>
<td>2</td>
</tr>
<tr>
<td>25* value =</td>
<td>4</td>
</tr>
<tr>
<td>25* number =</td>
<td>7</td>
</tr>
<tr>
<td>25* unsigned . unsigned =</td>
<td>8</td>
</tr>
<tr>
<td>25* digit . unsigned =</td>
<td>11</td>
</tr>
<tr>
<td>25*1. unsigned =</td>
<td>8</td>
</tr>
<tr>
<td>25*1. digit =</td>
<td>15</td>
</tr>
<tr>
<td>25*1.5=</td>
<td></td>
</tr>
</tbody>
</table>

The second way a grammar can be used is in the reduction of a valid program back to the start symbol through the reverse application of productions. By “reverse application,” we mean replacing a sequence of terminal and nonterminal symbols that match the right-hand side of a production by the single nonterminal symbol found on the left-hand side of the same production. This verifies that a string of terminal symbols is indeed a program in the language defined by the grammar. This derivation is possible only if a prudent choice is made of the sequence of production reversals to be applied, as in Figure 2.3. For example, an initial choice of sign for + would have led to an early dead end in our reduction. Furthermore, if this process is attempted on a string that is not a valid program, the start symbol can never be reached. Figure 2.4 shows this through an example, where every step is uniquely determined and the final string matches the start symbol.

Figure 2.3 Verification that 6+3/12= is a calculation
6 + 3 / 1 2 =

digit  operator  digit  operator  digit  digit

unsigned  unsigned  unsigned

number  number  unsigned

value  value  number

expression  expression  value

expression

calculation
Figure 2.4 Reduction of invalid string

Reinforce: Use the calculator grammar in Figure 2.1 to generate the following strings from the start symbol. Your result should be in the same form as Figure 2.2.

(a) -6+-3=
(b) 72.5--5=
(c) 16.05+2=

Reinforce: Do the verification of the above three strings for the calculator grammar in Figure 2.1. Your result should be in the same form as Figure 2.3.

Reinforce: Show that the verification process deadends in the case of the following strings that are not in the language generated by the calculator grammar.

(a) ==
(b) 6.2++7=
(c) 15=7*2+1

Reinforce: Determine whether each of the following is a valid calculation in the calculator language of Figure 2.5. If so, show its derivation tree. If not, tell why not.

(a) 4+2=
(b) 6=
(c) 21-14/7=
(d) 6-(2+5)=
(e) +4+2=
2.2.1.2 BNF and EBNF

In this section, we will describe a language for expressing grammars. Because this is a language for describing languages, we call it a metalanguage. The metalanguage we describe is BNF, which stands for Backus-Naur Form. This language was created to express the syntax of ALGOL 60 and has become the standard metalanguage. Throughout this text, we will describe syntax by using a form of BNF that has three additional features. The extended language will be called EBNF for Extended BNF.

The metalanguage described in the previous section is what we will define as BNF with one notational addition. In BNF, nonterminals will have their names enclosed in angle brackets (<>) to allow a string of characters in a nonterminal name to be distinguished from the corresponding string of terminal characters. For example, the nonterminal symbol <id> can be distinguished from id, a string of two terminal symbols. See Figure 2.5 for a BNF definition of the calculator grammar. This is a straightforward translation of Figure 2.1.

Figure 2.5 Calculator Grammar in BNF

```
<calculation> ::= <expression> =
<expression> ::= <value>
<expression> ::= <value> <operator> <expression>
[value]    ::= <number>
[value]    ::= <sign> <number>
[number]   ::= <unsigned>
[number]   ::= <unsigned> . <unsigned>
<unsigned> ::= <digit>+
<unsigned> ::= <digit> <unsigned>+
<digit>    ::= 0
<digit>    ::= 1
<digit>    ::= 2
<digit>    ::= 3
<digit>    ::= 4
<digit>    ::= 5
<digit>    ::= 6
<digit>    ::= 7
<digit>    ::= 8
<digit>    ::= 9
<sign>     ::= +
<sign>     ::= -
<operator> ::= +
<operator> ::= -
<operator> ::= *
```
<\text{operator}> ::= / \\

We are now ready to add three features to BNF to transform it into our EBNF. These features do not add any capabilities to the language but rather improve the compactness of the expression of a grammar. These features are in such common use that many people include them in the definition of BNF.

The first new feature, \textit{alternation}, is the use of the $\texttt{\texttt{or}}$ symbol to express alternate definitions for the same nonterminal within a single production. This symbol is a vertical bar ($|$). For example, the set of productions

\begin{align*}
<\text{odd}> ::= & 1 \\
<\text{odd}> ::= & 3 \\
<\text{odd}> ::= & 5 \\
<\text{odd}> ::= & 7 \\
<\text{odd}> ::= & 9 \\
\end{align*}

can be compressed using this notation to read

\begin{align*}
<\text{odd}> ::= & 1 \mid 3 \mid 5 \mid 7 \mid 9 \\
\end{align*}

We also add \textit{optionality} by the use of brackets ([ ]) to specify an optional item and \textit{repetition} by the use of braces (\{ \}) to specify a repeated item. The brackets indicate zero or one occurrence of the enclosed specification, while the braces indicate zero or more repetitions of the enclosed specification.

For example, the production

\begin{align*}
<\text{goal}> ::= & [a] b \{c\} \\
\end{align*}

specifies the following strings as valid goals:

\begin{itemize}
  \item \texttt{b}
  \item \texttt{ab}
  \item \texttt{bc}
  \item \texttt{abc}
  \item \texttt{bcc}
  \item \texttt{abcc}
  \item \texttt{bccc}
  \item \texttt{abccc}
  \item \ldots
\end{itemize}

This indicates there are zero or one \texttt{a}'s followed by one \texttt{b} followed by zero or more \texttt{c}'s.

Although these brackets and braces are simply defined, their interpretation can be quite complicated when they are nested and/or include the $\texttt{\texttt{or}}$ metasymbol. Consider, for example, the production

\begin{align*}
<\text{goal}> ::= & \{a\} b \{d \mid e\} \\
\end{align*}
Although this expresses a production compactly, we see that some clarity has been lost in the process as the definition is rather complex.

One final extension we will find helpful is the use of parentheses to group parts of the specification for priority in evaluation. Parentheses are necessary to eliminate ambiguity in the application of alternation and concatenation. For example, parentheses can specify whether the production

\[
<x> ::= a \ b \mid c
\]

means

\[
<x> ::= (a \ b) \mid c \text{ or } <x> ::= a (b \mid c)
\]

The first of these interpretations would recognize the string \(c\), while the second would not. The second would recognize \(ac\), while the first would not.

When used with care, these extensions can improve not only the compactness but also the clarity of grammar definitions. Figure 2.6 shows how these extensions can be used in defining our calculator grammar. As you can see, this new notation frequently eliminates the need for recursion and alternatives. For example, without the brackets, \(<value>\) might be defined by

\[
<value> ::= <unsigned> \mid <sign><unsigned> \\
\mid <unsigned>.<unsigned> \mid <sign><unsigned>.<unsigned>
\]

Figure 2.6 Calculator Grammar in EBNF

\[
<calculation> ::= <expression> = \\
<expression> ::= <value> [<operator> <expression>]
\]

\[
<value> ::= [<sign>] <unsigned> [. <unsigned>]
\]

\[
<unsigned> ::= <digit> {<digit>}
\]

\[
<digit> ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

\[
<sign> ::= + \mid -
\]

\[
<operator> ::= + \mid - \mid * \mid /
\]

One purpose of EBNF is to enhance the clarity of expression for grammars. The one difficulty that arises in its use is when a symbol used in EBNF (a metasymbol) is also a terminal symbol in a grammar. For example, if \(|\) were a terminal symbol in the grammar being defined, then the production

\[
<x> ::= a \mid b
\]

could be interpreted in either of two ways. First, the nonterminal \(x\) might have two alternative definitions, \(a\) or \(b\), if \(|\) is interpreted as a metasymbol. If, however, \(|\) is interpreted as a terminal symbol of the grammar, a single definition consisting of three terminal symbols is specified.

We avoid this confusion by using the following convention: when a metasymbol is also a
terminal symbol of the grammar being defined, the symbol will be underlined if it is to represent the terminal symbol and will represent the metasymbol otherwise. Using this convention, the preceding production would represent two alternative single symbol definitions of $x$. If we wished to express the three-symbol definition, it would be written

$$<x> ::= a \mid b$$

Research: Who were Backus and Naur and what role did they play in the development of BNF?

Reinforce: The text states that the EBNF extensions do not add any functionality to BNF. For example,

$$<n> ::= a \mid b$$

is equivalent to

$$<n> ::= a$$
$$<n> ::= b$$

Show equivalent BNF forms for

$$<n> ::= [a]$$
and

$$<n> ::= \{b\}$$

Discuss: What are the advantages and disadvantages of EBNF in comparison to BNF?

Reinforce: Given the EBNF production

$$<goal> ::= ([a]b)c(d|e)$$

express this as an equivalent set of BNF productions.

Reinforce: For the EBNF production above, determine which of the following strings form valid goal nonterminals.

(a) abc
(b) ababcd
(c) cde
(d) aac
(e) aabce
(f) bcde
de
(g) cede

Reinforce: Given the EBNF production

$$<goal> ::= ([ab]c)[d]e$$

determine which of the following form valid goal nonterminals.

(a) abcede
(b) ce
(c) abe
(d) ababce
(e) ccee
(f) dddde
(g) cddd
(h) aabbce
(i) e
(j) cccccce

Reinforce: Modify the calculator grammar to add the option of expressing numbers using the exponential format such as
- 6.2e5
- -16.23e21
- 15E-3

Reinforce: Give a BNF specification for the language that consists of an even number of a's followed by an odd number of b's.

Laboratory: Write an interpreter for the calculator language described in this chapter.

2. Write a program that translates a grammar written in EBNF into a BNF grammar.

3. Write a program that accepts as input an expression in the calculator language and generates the parse tree for that expression.

2.2.1.3 Syntax Diagrams

Wirth uses a somewhat different approach to expressing grammars in his definition of Pascal. This tool is called a syntax diagram. It expresses productions as two dimensional directed graphs whose nodes are symbols. The possible paths through the graph represent the possible sequences of symbols that define the nonterminal of the production.

Terminal symbols are represented by ovals and nonterminals, by rectangular nodes. Syntax diagrams have the advantage of using two dimensions to enhance understandability. Their disadvantage is the difficulty in generating the diagrams using a linear input device, such as a keyboard.

Figure 2.7 shows the syntax diagram for our calculator grammar. Here the definition of each nonterminal from Figure 2.6 is represented by a graph. The definition of expression shows how optional components are represented. The definition of digit gives an example of representing alternative definitions. An example of the syntax diagram representation of repetition is shown in the definition of unsigned.
Figure 2.7 Syntax diagram for calculator grammar
Research: Find a description of Pascal using syntax diagrams. Study the structure of Pascal if statements.

Discuss: What are the relative advantages and disadvantages of syntax diagrams as compared to EBNF syntax specification.

Reinforce: Construct syntax diagrams for the grammars given below:
(a) \( \text{<goal>} ::= [\text{<a>}][\text{xy}]\text{<b>} \)
\( \text{<a>} ::= z[z] \)
\( \text{<b>} ::= x[y]z \)

(b) \( \text{<goal>} ::= <\text{expression}> \)
\( <\text{expression}> ::= <\text{term}> \{+ <\text{expression}>\} \)
\( <\text{term}> ::= <\text{factor}> \{* <\text{term}>\} \)
\( <\text{factor}> ::= a \mid b \mid c \mid <\text{expression}> \)

(c) \( \text{<goal>} ::= <\text{a}> \)
\( <\text{goal}> ::= x <\text{b}> <\text{a}> \)
\( <\text{a}> ::= y \)
\( <\text{a}> ::= x <\text{a}> \)
\( <\text{b}> ::= <\text{a}> \)
\( <\text{b}> ::= <\text{a}> <\text{b}> \)
\( <\text{b}> ::= y <\text{b}> \)

(d) \( \text{<goal>} ::= <\text{a}> \)
\( <\text{a}> ::= <\text{b}> \)
\( <\text{a}> ::= x <\text{a}> \)
\( <\text{b}> ::= <\text{c}> \)
\( <\text{b}> ::= y <\text{c}> \)
\( <\text{c}> ::= z \)
\( <\text{c}> ::= z <\text{c}> \)

2.2.1.4 Problems with Specifications

Grammars that can be described by BNF and, equivalently, by EBNF are known as context-free grammars. This terminology indicates that the valid definitions of a nonterminal symbol are independent of the context in which the symbol is found. Most programming languages cannot be completely specified by a context-free grammar because they contain some rules that are context sensitive, i.e., their nonterminal definition depends on context. For example, the common requirement that a variable must be declared before it is used cannot be expressed in a context-free grammar, because the validity of a variable depends upon whether its declaration is in the context of its use or not. Although formal tools do exist to express context-sensitive grammars, these are much more complex than the context-free tools we have already seen. These context-sensitive tools are therefore not commonly used in language specification. The common approach
is to specify formally the context-free portion of a language's syntax using BNF or a similar tool and specify informally the context-sensitive portion in English text.

Another problem in formal specification is that of ambiguity. Ambiguity occurs when there is more than one possible derivation tree associated with a given valid string in a language. To illustrate this, consider the grammar in Figure 2.6 modified by replacing the production for `<expression>` by

\[
<expression> ::= <value> \mid ( <expression> <operator> <expression> )
\]

The resulting grammar is an ambiguous grammar, as Figures 2.8a and 2.8b demonstrate. Here, two different valid derivation trees are given for the string

\[4 + 2 \times 3 =\]

If the grammar is being used only to determine whether the string is valid or not, this ambiguity causes no problem, because either derivation is sufficient for this purpose. Often, however, a grammar is used to interpret the meaning of the string as well. Notice that in this case, the two possible derivations have different meanings in the sense that they imply that the two operators are applied in the opposite order when the statement is executed. The application of the tree in Figure 2.8a then represents a calculation whose result is 18, whereas the second tree represents a calculation whose result is 10.

It is useful to eliminate ambiguity by modifying the grammar whenever possible. There are no general techniques for doing this. Rather, each situation must be carefully analyzed to find and eliminate the ambiguities. Some languages have no possible representation through an unambiguous grammar, so attempts to find such representations are fruitless. These languages are called ambiguous languages.

The grammar of Figure 2.6 results in the derivation of a tree equivalent to the tree of Figure 2.8b. In general, this grammar specifies the application of operators from right to left. This is reversed from the left-to-right application we commonly expect. Careful examination shows we could generate left-to-right operator application if the production for `<expression>` is replaced by

\[
<expression> ::= [ <expression> <operator> ] <value>
\]

The difference between this production and the one given in Figure 2.6 is the placement of the recursive symbol with respect to the operator. When the recursive symbol is to the left of the operator, the operator will be applied left to right, and when it is to the right, the operator will be applied right to left.

We frequently wish to override the left-to-right application of operators in arithmetic expressions in order to give certain operators precedence over others. For example, it is common for multiplication and division to be given precedence over addition and subtraction. In this way, the expression

\[6 + 3 \times 4\]

is evaluated by applying the multiplication first, even though it lies to the right of the addition.
Grammars that implement precedence can be constructed by inserting additional nonterminal symbols for each level of precedence. The lower the level of precedence for an operator, the nearer to the start symbol it is applied. In order to implement the precedence rule stated earlier, we include the following productions in our calculator grammar:

\[
\begin{align*}
\text{<expression>} & ::= [\text{<expression> } \text{<addoperator>} ] \text{<term>} \\
\text{<term>} & ::= [\text{<term> } \text{<multoperator>} ] \text{<value>} \\
\text{<addoperator>} & ::= + \mid - \\
\text{<multoperator>} & ::= * \mid / \\
\end{align*}
\]

The derivation of \(4+2*3=\) using this grammar is given in Figure 2.8c. Notice that by using left recursion in each case, we have also specified left-to-right order among operators of the same precedence. It would, of course, be possible to reverse the order for some precedence levels to be evaluated right-to-left by making the corresponding productions right recursive. This is often the case with the exponentiation operator, which in many languages is evaluated right-to-left, even though all other operators are left-to-right.

**Discuss:** What are some context-sensitive rules found in most programming languages?

**Reinforce:** Consider the following ambiguous grammar:

\[
\text{<goal>} ::= \text{<expression>} \\
\text{<expression>} ::= \text{<item>} \\
\quad | \text{<expression>} + \text{<expression>} \\
\quad | \text{<expression>} * \text{<expression>}
\]

Rewrite this as an unambiguous grammar in which

(a) + and * have equal precedence and both are performed left to right.
(b) * has precedence over + and both are performed right to left.
(c) + has precedence over *, + is performed left to right, and * is performed right to left.

**Reinforce:** The following EBNF specification is ambiguous:

\[
\text{<stmt>} ::= \text{if } \text{<expr>} \text{ then } \text{<stmt>} \\
\quad | \text{if } \text{<expr>} \text{ then } \text{<stmt> else } \text{<stmt>} \\
\quad | \text{<others>}
\]

Assume that \text{<expr>} and \text{<others>} are defined elsewhere and that they describe a boolean expression and other statements, respectively.

(a) Show that the specification is ambiguous.
(b) Give a different specification that describes the identical language, but that is unambiguous. Do not give further definitions to \text{<expr>} and \text{<others>}.

**Reinforce:** Consider this language specification:

\[
\text{<S>} ::= \text{<E>} \\
\text{<E>} ::= \text{<E>} * \text{<E>}
\]
\[
\begin{align*}
<E> & := [E] ^ <E> \\
<E> & := c
\end{align*}
\]

The set of terminal symbols is \{c, *, ^\} and \(<E>\) and \(<S>\) are the nonterminal symbols. The start symbol is obviously \(<S>\).

(a) Give a derivation and parse tree for the string

\[c \ast c \wedge c \ast \wedge c\]

(b) The preceding specification is ambiguous. Give a proof of its ambiguity.
Figure 2.8a First derivation tree for 4+2*3=

```
4 + 2 * 3 =
  |   |   |   |
digit operator digit operator digit
  |   |   |   |
unsigned unsigned unsigned
  |   |   |
value value value
  |   |
expression expression expression
  |
expression

expression

expression
calculation
```
Figure 2.8b Second derivation tree for 4+2*3=

```
4   +   2   *   3   =
     |     |     |     |
    digit operator digit operator digit
     |     |     |     |
    unsigned unsigned unsigned
     |     |     |
    value value value
     |     |
    expression expression expression
     |     |
    expression calculation
```
2.2.2 Semantics Specification

Although the syntax of a language can be nicely described in a formal manner by the use of tools such as BNF, the semantics is a different matter. Even though tools are available, which we will briefly describe in this section, these tools are too complex to describe the semantics of a practical language in an easily interpreted way. Therefore, semantics is usually described in English, a tool that is understandable enough but unfortunately lacks the formalism we desire.

In spite of its complexity, formal semantic specification is desirable, because it allows the actions of programs to be described in an unambiguous, machine-independent way, just as formal
syntactic specifications describe legal program syntax. In addition, formal semantics provide a standard and a reference for the testing or verification of language translation systems.

In this section, we briefly describe three approaches to formal semantics specification, each of which corresponds to one of the language models presented in this book. The first approach, operational semantics, which is based on the imperative model of languages, describes the semantics of a language by mapping programs in that language into equivalent programs implemented on an abstract machine. Denotational semantics uses the functional model of languages to map programs in a language into equivalent functions. The final approach, axiomatic semantics, is based on the logic model of programming languages. It describes the action of a program by determining logical assertions that hold before and after its execution.

Each of the three approaches is illustrated by application to a variation of the calculator grammar used in the previous section. That variation is given in Figure 2.9.

Figure 2.9 Calculator grammar for semantic specification

```
<calculation> ::= <expression> =
<expression> ::= <value> | <value> <operator> <expression>
=value
<value> ::= [<sign>]<integer-part> | [<sign>]<integer-part>.<digit>
<integer-part> ::= <digit> | <integer-part> <digit>
<decimal-part> ::= <digit> | <digit> <decimal-part>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<sign> ::= + | -
<operator> ::= + | - | * | /
```

2.2.2.1 Operational Semantics

The operational approach to semantic specification defines the meaning of any statement in a language by providing a means of translating that statement into an equivalent statement in another language. In order for that technique to be useful, the new language must be one whose semantics is unambiguous and completely understood by the user.

As an example, if we already understand the Pascal programming language, an operational semantic specification for Ada might consist of a program that translates any Ada program into an equivalent Pascal program. But this reduces the problem to a semantic specification of Pascal, which, since Pascal is approximately the same level of complexity as Ada, is nearly as difficult to accomplish.

Therefore, the most useful operational semantic specification would automate a translation from the source language to a lower-level language -- that is, one whose semantics can be defined more simply. The lowest-level language in this sense is machine language, because its semantics is completely and unambiguously defined by the architecture of the machine. An operational semantic specifier that generates machine language is also known as a compiler, and for those who are fluent in machine language, a compiler fulfills the role of a semantic specification. However, the machine language of a real computer may be more detailed than we wish for the purpose of semantic specification. Therefore, operational semantic specifications are usually based on translation to the language of an abstract machine that contains only the essential parts of a
machine language. The implementation of a compiler for the source language can then be reduced to producing a translator from the abstract machine language to the target machine language.

The most famous abstract machine used for this purpose was the Vienna Definition Language developed by Wegner (1972) for the semantic description of PL/I. This abstract machine definition was important from a theoretical point of view but did not come into popular use because the semantic descriptions were very complex.

To illustrate the operational approach to semantic specification, we use a simple stack machine as our abstract machine. Its machine language will contain six instructions described below:

- **push value**: push the given value on the stack
- **pop**: pop the top value from the stack
- **add**: add the top two values, replacing them with their sum
- **sub**: subtract the top two values, replacing them with their difference
- **mult**: multiply the top two values, replacing them with their product
- **div**: divide the top value by the next, replacing them with their quotient

The semantic description of the calculator language now consists of the stack machine code that is generated for each syntax rule in the grammar. For example, the syntax rule

\[
<\text{integer-part}> ::= <\text{integer-part}> <\text{digit}>
\]

would be replaced by the following stack machine code:

1  [the code generated by the right-hand-side <integer-part>]
2  push 10
3  mult
4  [the code generated by <digit> which leaves that result on top of the stack]
5  add

This requires some explanation. The code is a recursive definition of the code for <integer-part> in that the code for another <integer-part> is included within it. The second and third steps multiply that value by 10, whereas the fourth and fifth steps add the value of <digit>.

Let us assume the code generated by the right-hand-side <integer-part> results in a value of 28 being placed on top of the stack, and the code generated by <digit> results in 7 being placed on top. Then the preceding code would be equivalent to the following with the stack shown to the right.

1  [push 28] 28
2  push 10 10 28
3  mult 280
4  [push 7] 7 280
5  add 287
A complete definition of the semantics of the calculator language is found in Figure 2.10. Here the code generated by a syntactic structure is represented by that structure's name.

We now examine the abstract machine program generated by the calculation

\[ 1.5 \times -40 = \]

Figure 2.11 shows the stack machine code generated by our model along with the accompanying syntax structures. This diagram is the parse tree for the calculation and illustrates the process by which the stack machine program is generated as you examine it from right to left. For example, the goal calculation is semantically represented by

expression
pop

where expression is then further parsed into

expression
value

and pop is a statement in our stack machine language.

The items in parentheses in Figure 2.11 represent the portion of the source string represented by each syntactic component. The left column is the stack machine code generated that is equivalent to the original expression.

Reinforce: Write the semantic description of the following programs in the stack machine language, using the semantic description in Figure 2.10:

(a) \(-17 + 2.41 / 5 = \)
(b) \(1.2 / 3 + 2 = \)

Laboratory: Write a simulator for the stack machine described in this chapter. Then implement the calculator language on that simulated machine.
Figure 2.10 Operational semantics for calculator language

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;calculation&gt;</code></td>
<td>::= <code>&lt;expression&gt; =</code> <code>&lt;expression&gt;</code> pop</td>
</tr>
<tr>
<td><code>&lt;expression&gt;</code></td>
<td>::= <code>&lt;value&gt;</code></td>
</tr>
<tr>
<td></td>
<td>::= <code>&lt;value&gt; &lt;operator&gt; &lt;expression&gt;</code> <code>&lt;value&gt;</code> <code>&lt;operator&gt;</code></td>
</tr>
<tr>
<td><code>&lt;value&gt;</code></td>
<td>::= <code>[&lt;sign&gt;]</code> <code>&lt;integer-part&gt;</code> <code>&lt;integer-part&gt;</code></td>
</tr>
<tr>
<td></td>
<td>::= <code>[&lt;sign&gt;]</code> <code>&lt;integer-part&gt;.&lt;digit&gt; &lt;integer-part&gt;</code> <code>&lt;decimal-part&gt;</code> add</td>
</tr>
<tr>
<td><code>&lt;integer-part&gt;</code></td>
<td>::= <code>&lt;digit&gt;</code></td>
</tr>
<tr>
<td></td>
<td>::= <code>&lt;integer-part&gt; &lt;digit&gt;</code> push 10</td>
</tr>
<tr>
<td><code>&lt;decimal-part&gt;</code></td>
<td>::= <code>&lt;digit&gt;</code> push 10</td>
</tr>
<tr>
<td></td>
<td>::= <code>&lt;digit&gt; &lt;decimal-part&gt;</code> <code>&lt;decimal-part&gt;</code> push 10</td>
</tr>
<tr>
<td><code>&lt;digit&gt;</code></td>
<td>::= 0 push 0</td>
</tr>
<tr>
<td></td>
<td>::= 1 push 1</td>
</tr>
<tr>
<td></td>
<td>::= ⋮ push 9</td>
</tr>
<tr>
<td></td>
<td>::= 9 push 9</td>
</tr>
<tr>
<td><code>&lt;sign&gt;</code></td>
<td>::= + nop</td>
</tr>
<tr>
<td></td>
<td>::= - push -1 mult</td>
</tr>
</tbody>
</table>


### 2.2.2.2 Denotational Semantics

While the operational approach maps each syntax production into a sequence of abstract machine language statements, the denotational approach maps each production into a function. For each nonterminal symbol on the right-hand side of a production, the generated function includes a call on the function defined for that nonterminal. The theory behind this approach was

---

<table>
<thead>
<tr>
<th>&lt;operator&gt; ::=</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>sub</td>
</tr>
<tr>
<td>*</td>
<td>mult</td>
</tr>
<tr>
<td>/</td>
<td>div</td>
</tr>
</tbody>
</table>

**Figure 2.11 Stack machine program for 1.5*-40**

```
digit(4)  int-part(4)  
           |                   |
           |                   |
           |                   |
int-part(40) value(-40) expr(-40)

digit(0)

sign(-)

digit(1)  int-part(1)

digit(5)  dec-part(5)  value(1.5)
```

```
originally developed by Scott and Strachey (1971).

The final function generated for the start symbol accepts as its parameter a program written in the language being described and has as its result the output that the given program should generate. Specifically, if the final function for the calculator grammar was given by \( F_C \), the application of this function to a syntactically valid string would give the correct result of the calculation—for example,

\[
F_C(\text{‘2+3=’}) = ‘5’
\]

Let us first examine how a function can be derived from a production. We choose the production

\[
\text{<integer-part> ::= <integer-part> <digit>}
\]

from our calculator grammar. We specify the function \( IP \) defined by this production to be

\[
IP(\text{<integer-part><digit>}) = 10 \times IP(\text{<integer-part>}) + D(\text{<digit>})
\]

where \( D \) is the function associated with \(<digit>\). Note that this is a recursive definition of function \( IP \), but a nonrecursive alternative definition is associated with the production

\[
\text{<integer-part> ::= <digit>}
\]

namely,

\[
IP(\text{<digit>}) = D(\text{<digit>})
\]

Therefore, if we expand the definition of ‘245’, we see that

\[
\begin{align*}
IP(‘245’) &= 10 \times IP(‘24’) + D(‘5’) \\
&= 10 \times (10 \times IP(‘2’) + D(‘4’)) + D(‘5’) \\
&= 10 \times (10 \times D(‘2’) + D(‘4’)) + D(‘5’)
\end{align*}
\]

where the first two lines apply the recursive definition of \( IP \) and the last line applies the nonrecursive one.

A complete denotational description of our calculator grammar is given in Figure 2.12. The function definitions found there are self-explanatory. Figure 2.13 contains the evaluation of the function \( C \) for the string ‘1.5*-40=’. Each line of this evaluation is the application of one or more functions from the specification of Figure 2.12.

**Figure 2.12 Denotational semantic description of calculator language**

\[
C(<expression> =) = E(<expression>)
\]

\[
E(<value>) = V(<value>)
\]

\[
E(<value>+<expression>) = V(<value>) + E(<expression>)
\]
\[ E(<\text{value}>-<\text{expression}>) = V(<\text{value}> - E(<\text{expression}>) \]
\[ E(<\text{value}>*<\text{expression}>) = V(<\text{value}> * E(<\text{expression}>) \]
\[ E(<\text{value}>/<\text{expression}>) = V(<\text{value}> / E(<\text{expression}>) \]
\[ V(\text{+} \ <\text{value}> ) = V(<\text{value}> ) \]
\[ V(\text{-} \ <\text{value}> ) = - V(<\text{value}> ) \]
\[ V(<\text{integer-part}> ) = \text{IP}(<\text{integer-part}> ) \]
\[ V(<\text{integer-part}.<\text{decimal-part}>)= \text{IP}(<\text{integer-part}> )+\text{DP}(<\text{decimal-part}> ) \]
\[ \text{IP}(<\text{digit}> ) = \text{D}(<\text{digit}> ) \]
\[ \text{IP}(<\text{integer-part}> <\text{digit}> ) = 10*\text{IP}(<\text{integer-part}> )+\text{D}(<\text{digit}> ) \]
\[ \text{DP}(<\text{digit}> ) = 0.1*\text{D}(<\text{digit}> ) \]
\[ \text{DP}(<\text{digit}> <\text{decimal-part}> ) = 0.1*(\text{DP}(<\text{decimal-part}> )+\text{D}(<\text{digit}> )) \]
\[ D('0') = 0 \]
\[ D('1') = 1 \]
\[ \ldots \]
\[ D('9') = 9 \]

Figure 2.13 Derivation of denotational function for 1.5*-40

\[ C('1.5*-40=' ) = \]
\[ E('1.5*-40') = \]
\[ V('1.5')*E('40') = \]
\[ (\text{IP}('1')+\text{DP}('5'))*E('40') = \]
\[ (1 + 0.1*5)*E('40') = \]
\[ 1.5*V('40') = \]
\[ 1.5*(-V('40')) = \]
\[ 1.5*(-\text{IP}('40')) = \]
\[ 1.5*(-(10*\text{IP}('4')+\text{D}('0'))) = \]
\[ 1.5*(-(10*\text{D}('4')+\text{D}('0'))) = \]
\[ 1.5*(-(10*4+0)) = \]
\[ 1.5*(-40) = \]
\[ -60 \]

Note that this semantic description is simplified by two properties of our language. First, the language admits no input, so all of our functions are constant functions. Suppose, for example, a construct reads an integer and outputs its double. Its function could then be defined as

\[ D(<\text{double}>) = [f(x) = 2*x] \]

The second simplifying property of our language is that no construct produces side-effects that
change the computing environment. In cases where side-effects can be produced, the computing environment, such as the data store, must be included as input and output to the specified function, allowing the syntactic structure to modify the environment as well.

*Reinforce:* Derive the denotational function for each of the following calculations in the extended calculator language of Figure 2.9.

(a) \(-17 + 2.41/5\) = 
(b) \(1.2/3 + 2\) = 

2.2.2.3 Axiomatic Semantics

The axiomatic method of specifying semantics is based on the logic model of computing. It makes use of logical **assertions**, which are defined as statements that are either true or false. For each syntactic definition, the axiomatic method specifies the assertions that are assumed to be true before the execution of the syntactic unit, known as the **precondition**, and the assertion that then must be true after the unit's execution, called the **postcondition**. The first major work on this approach was done by Hoare (1969).

Like the logic model, the axiomatic approach to semantic specification is nonprocedural in nature. Rather than specifying the actions that a syntactic unit performs, as the operational and denotational approaches do, this approach describes the effect of the action in the form of the postcondition.

Although the axiomatic approach requires extensive background in predicate calculus and its use in the semantic description of programming languages can be quite difficult to understand, it is easy to apply this approach to the semantic definition of our calculator language. In order to do this, we introduce some helpful notation. First, we will express the assertion that a syntactic structure \(S\) evaluates to a given value \(v\) by the notation

\[
<S> \rightarrow v
\]

We will then express precondition and postcondition pairs by placing the precondition on top of the postcondition and dividing them by a line.

As an example, consider the axiomatic semantic description of the syntax rule

\[
<integer-part> ::= <integer-part> <digit>
\]

We would write this in our notation as follows:

\[
<integer-part>_r \rightarrow v_1 \text{ and } <digit> \rightarrow v_2 \\
\frac{<integer-part>_l \rightarrow 10*v_1 + v_2 }
\]

This notation indicates that if the precondition is that the two constituent syntactic units evaluate to \(v_1\) and \(v_2\), the postcondition is that the defined unit evaluates to \(10*v_1 + v_2\). Note the use of the subscripts \(r\) and \(l\) to differentiate between the occurrences of \(<integer-part>\) on the left
and right sides of the production.

A complete axiomatic definition of our calculator language is given in Figure 2.14. Productions with no precondition are represented by leaving the expression above the line blank. See definitions of <digit> and <sign> in Figure 2.14.

Let us examine the second definition of <value> in Figure 2.14. It states that when <value> is represented by

\[
[<\text{sign}>] \ <\text{int-part}> \ <\text{dec-part}>
\]

The semantics rule is stated in English as follows:

If <int-part> is the value \(v_1\) and <dec-part> is the value \(v_2\) and <sign>, if present, is \(v_3\), then <value> has the value \(v_1 + v_2\) if <sign> is not present or \(v_3 \times (v_1 + v_2)\) if <sign> is present.

Discuss: Compare the three methods for semantic description of languages in terms of understandability and usefulness.

Discuss: Characterize the semantics of a language that you know. What would be the best formal method to describe this language? Explain this.

Reinforce: Suppose the following production is added to the calculator grammar as a monadic square operator:

\[
<sqr\text{-}op> ::= @
\]

This is interpreted to mean that \(@x\) would represent \(x^2\).

a. How would this change the operational semantics specification in Figure 2.10?  
b. How would this change the denotational semantics specification in Figure 2.12?  
c. How would this change the axiomatic semantics specification in Figure 2.14?
Figure 2.14 Axiomatic semantic description of calculator language

\[
\begin{align*}
\text{<calculation>} &::= \text{<expression>} \\
\text{<expression>} &::= \text{<value>} \\
\text{<value>} &::= [\text{<sign>}]\text{<int-part>} \\
\text{<int-part>} &::= \text{<digit>} \\
\text{<dec-part>} &::= \text{<digit>} \\
\text{<digit>} &::= 0 \\
\text{<sign>} &::= + \\
\end{align*}
\]
2.3 Language Translation

The purpose of any language is communication between two parties, a sender and a receiver. With natural language, the communication is between two people who alternate between the sending and receiving roles. With programming languages, the communication is between a programmer and a language translation program. Communication in a programming language is sent by the programmer and received by the translation program. There is also communication in the opposite direction in the form of diagnostic messages and other information about the translation; however, this communication is not in the programming language itself but usually in English, some cryptic codes, or a combination of the two.

The purpose of a language translation program is to accept a set of instructions written in a programming language and to cause the activities specified by these instructions to be carried out by the receiving computer. Therefore, a language translation program accepts as input a program in a programming language and makes it possible for the activities specified by this program to be carried out, either by translating the program into an equivalent program in a language that is already executable or by directly carrying out the activities specified by the translation program.

In order to understand a language and judge its effectiveness, we must have some understanding of the capabilities and needs of both the sender and the receiver. This text is concerned primarily with languages from the point of view of the sender, i.e. the programmer. Other books examine languages from the point of view of the receiver, the language translation program. Such books are useful in courses on compiler analysis and design.

In this section, we will briefly describe the properties of translation programs. Our intent is that as you study programming languages from the perspective of the programmer, you might gain some understanding of the perspective of the translation program located at the receiving end of this communication.

2.3.1 Compilers and Interpreters

There are two fundamental approaches to language translation: compilation and interpretation. In order to help explain the difference between these two approaches, we will examine an analogy.

Suppose that you employ a German gardener who speaks no English, and you speak no German. Every day you prepare a list of instructions in English of the gardener’s assigned tasks for that day. You present this list to him upon his arrival in the morning. The gardener must then use a German-English dictionary that you provide to translate your instructions so he can perform the requested tasks.

With the second approach—the compilation approach—the gardener first translates all the instructions on the list from English to German, recording all the translated instructions. He then takes this translated list of instructions in German and begins performing the requested tasks.

As we examine these two approaches, we note some advantages of each. Interpretation has the advantage that if for some reason the gardener does not complete all of the jobs specified,
he has not wasted time translating descriptions of tasks he will not do. Also, if one of the instructions is not executable, the gardener is able to point out the troublesome instruction to you on your English list, since he knows where he is translating at the moment. This may not be possible under compilation.

Compilation is advantageous if there is a need to repeat the same activity several times, because the translated version is saved and can be referred to again later. For example, if the instructions included

1. Mow the front lawn.
2. Prune shrubs in front lawn.
3. Rake front lawn.
4. Repeat steps 1-3 for the back lawn.

the interpretive gardener would have to retranslate instructions 1 to 3 when he reached instruction 4 because we assume he does not save the translated version either on paper or in his head. The compiling gardener would not need to retranslate because he would have saved the complete set of translated instructions. Another case of repetition of a translated instruction would occur if the instructions for this Monday read, "Repeat the jobs you did last Monday." We assume the compiling gardener, in anticipation of such a circumstance, saves each day's translated instructions and this Monday needs only to retrieve the list from last Monday rather than perform any translation at all. The interpreting gardener would not have recorded the translated instructions and would need to repeat the translation process.

The application of this analogy to programming language translation is straightforward. We will refer to translation programs using these two techniques as interpreters and compilers. An interpreter translates one statement of a program at a time and then calls a routine to complete the execution of that statement. A compiler produces from the given input program another program that is equivalent to the original but in a language that is executable. This resulting program may be in a language that is directly executable, such as machine language, or indirectly executable, such as another language for which a translator already exists.

The relative advantages of interpreters and compilers are similar to those of the gardener's two processes. An interpreter has the advantages of not translating statements that are never executed and of being able to relate back to the corresponding instruction in the programming language from every point of the execution. The compiler, on the other hand, only needs to translate each statement once, no matter how many times the statement is executed. This is applicable in both the case of iteration and the case of repeated executions of the same program.

The advantages of a compiler generally outweigh those of an interpreter in practice, and this form of translation is by far the one most frequently used. For this reason, and because compilation is a more complex process, we will focus on the activities of a compiler in the remainder of this section.

### 2.3.2 Overview of the Compilation Process

The purpose of a compiler is to translate a program written in a source language into an equivalent program expressed in a language that is executable directly by the machine. These two programs are called the source program and the object program. The language of the object program is called the target language. Figure 2.15 shows an overview of the compilation process.
where the object program is directly executable. The time during which the compiler is executing is known as compile time. The time during which the object program is executing is called run time.

**Figure 2.15 Compilation process overview**

The compilation can be broken down into several phases. These phases are conceptual and specify activities that all compilers perform, although frequently the activities of several phases may be combined and performed simultaneously. These phases and their corresponding inputs and outputs are illustrated in Figure 2.16.
Figure 2.16 Phases of compilation

- Source Program (String of Characters)
  - Lexical Analysis
    - Token String
      - Syntactic Analysis
        - Parse Tree
          - Semantic Analysis
            - Abstract Program
              - Code Generation
                - Object Program
Under this model, the program takes on three intermediate forms, the token string, the parse tree, and the abstract program. Each of these will be discussed in later sections describing the steps that produce them.

An additional important data structure in the compilation process is the symbol table. The symbol table contains an entry for every user-defined symbol or identifier included in the source program. This table is used to communicate between phases. It associates each identifier's name with its attributes or properties. The attributes included vary from compiler to compiler and from phase to phase in a single compiler. They may include specification of type, location, and status. Symbol table entries are created during lexical analysis, but the table is modified and referenced by both semantic analysis and code generation.

Discuss: Under what circumstances is an interpreter preferable to a compiler? What circumstances make a compiler preferable?

Discuss: Identify languages you have used. Can you classify the whether the implementation you used was compiled or interpreted?

Laboratory: Write two programs that do the same thing: one in a scripting or interpreted language and one in a compiled language. Test each program and explain the performance results.

2.3.2.1 Lexical Analysis

The purpose of the lexical analyzer (also known as the scanner) is to transform the source program from a character string into a string of tokens. A token is a symbol that expresses the nature of a language element, as abstracted from the string of characters that represents it. Some characters may translate directly into tokens—for example, operators and punctuation marks. Other tokens may be formed from strings of characters such as reserved words, statement labels, variables, and constants. The lexical analyzer must determine the appropriate token for a string of characters by examining the string itself and its context.

Some tokens, such as identifiers, may have names associated with them. In this case, there will be a reference to an entry in the symbol table attached to the token that represents that identifier. The first time a given identifier is encountered in the source program, it is entered into the symbol table. Subsequent occurrences will result in references to that symbol table entry being included in the token.

Some characters in the source program will result in no token entries. In particular, this will include characters that serve as separators between token elements in the source program. The output from lexical analysis is, therefore, a string of pairs where one element of the pair identifies the class of the token and the other element points to the entry in the symbol table where attributes of this particular token will be found. Some tokens, such as those for operators and reserved words, will contain an empty pointer because no further attributes are required.

For example, the Java statement

\[ x = x . \text{substring}(1,4) + \text{...}; \]
could, after lexical analysis, be represented by the following sequence of pairs:

```
(identifier, ptr1)
(equal, nil)
(identifier, ptr1)
(dot, nil)
(identifier, ptr2)
(leftparen, nil)
(integer, ptr3)
(comma, nil)
(integer, ptr4)
(rightparen, nil)
(plus, nil)
(string, ptr5)
(semicolon, nil)
```

All identifiers and constants contain pointers to corresponding symbol table entries. The two references to the identifier x both point to the same entry in the symbol table, ptr1. In practice, abbreviated codes are used to identify token types instead of the more descriptive names used here.

### 2.3.2.2 Syntactic Analysis

**Syntactic analysis** is the process of applying a grammar to form the derivation tree for a program from the sequence of tokens generated by the lexical analysis process. This process is called *parsing*. There are two strategies that can be applied for this purpose: top-down and bottom-up.

*Top-down parsing* begins with the start symbol and replaces that symbol with one of possibly several alternative definitions. A definition will be chosen that has the potential to match the string of tokens, where nonterminals within the definition match substrings of the original token string. These nonterminals are in turn replaced by one of their definitions, and so on, until no unmatched nonterminals remain. This top-down process corresponds to the generation of a valid string, as shown in Figure 2.2. However, in this case, the generation is directed toward a specific string of terminal symbols, and the steps are retained to form the derivation tree.

Frequently, a sequence of definition substitutions leads to a dead end in the generation process, at which point the top-down parser must resort to *backtracking*. Backtracking refers to the process of backing up to the most recent substitution where there are untested alternative definitions and attempting to use one of those alternatives in the generation process.

*Bottom-up parsing* starts with a string of tokens and attempts to match its substrings to the right-hand-sides of productions and, substituting the defined nonterminal for the string, continues until the entire string is replaced by the start symbol. This process proceeds as in Figure 2.3, with the derivation tree as its result.

### 2.3.2.3 Semantic Analysis

**Semantic analysis** uses the parse tree generated during syntactic analysis to generate a
program in some abstract programming language, a form of operational semantic specification. This abstract language is typically a machine language for some simple, hypothetical machine designed to be compatible with the data types and operations of the source language. This abstract language is intended to be an intermediate step between the source and target languages of the compiler, but some languages, Java for instance, stop at this stage and send the abstract language version to an interpreter.

One of the primary purposes of this abstract language is to enhance portability of the language. By portability we mean the ability to easily translate the programming language into multiple target languages. With a suitable choice for the abstract language, compilers sharing the same source language but having different target languages could share the same code for the first three phases of the compilation process. Once a compiler is written for a given source language, writing one for the same source language but a different target language would only require developing a new code generation phase for mapping the abstract language into the object language.

Intermediate languages usually resemble a high-level assembly language. The semantic analyzer calls a translation procedure that is identified with the syntactic class of the statement as determined by the syntax analysis. Each syntactic class has a corresponding translation procedure that analyzes the constituent parts of the parse tree and constructs the equivalent intermediate language form of the statement.

Another job of the semantic analyzer is the detection of context-sensitive errors such as mismatched types and undeclared variables. It also completes symbol table entries.

### 2.3.2.4 Code Generation

**Code generation** consists of replacing abstract language statements by parameterized object code templates. Therefore, the two steps in the code generation process are (1) matching a statement or sequence of statements, in the abstract language to some pattern, and (2) from that pattern, extracting a number of elements that will serve as parametric values. Next, the object code template that matches that pattern is obtained, and the parameter values are substituted. This filled-in template is then passed on as a part of the generated object program.

Let us consider a simple example to illustrate this use of templates. Suppose our intermediate language uses a three-address instruction set, and one such instruction is represented by the template

\[
\text{ADD a,b,c}
\]

This specifies that the operand located in \( a \) is to be added to the operand located in \( b \) and the result is to be stored in the location specified by operand \( c \). Such a template may be translated by the code generator into the following sequence of instructions in a two address target language:

\[
\begin{align*}
\text{MOVE loc(a),R1} \\
\text{ADD loc(b),R1} \\
\text{MOVE R1,loc(c)}
\end{align*}
\]

where \( R1 \) represents register 1 and \( \text{loc} \) is a function that returns the address of its identifier operand as it is found in the symbol table.
2.3.2.5 Optimization

Compiler optimization refers to the process of transforming a program into an equivalent program that will result in a more efficient execution as measured by a shorter run time. This modification might be performed on the source program, on the object program, or on any of the intermediate forms the program takes during the compilation process. The goal is to produce a program that will execute faster while still matching the functionality of the original source program. The word optimization is actually too strong, because it implies that the best of all possible equivalent programs will be produced. In practice, we can only hope to improve the run-time efficiency, not to optimize it.

Two forms of optimization we will discuss are object-independent optimization and object-dependent optimization. Object-independent optimization can be performed on the source program prior to the compilation process or on any of the intermediate forms up to the abstract program. It involves the identification of unnecessarily long or redundant run-time activities. Simple examples are expressions computed at compile-time whose values are known at that time, repeated identical computations that are calculated once and saved in storage, and operators that can be replaced by equivalent but faster operators.

Object-dependent optimization can only be performed on the object program and uses properties of the target language itself to reduce execution time. One such transformation is the use of registers to retain values that are reused later, thus avoiding recalculation. Another common technique is replacing operations by those having faster target instructions. Examples of this include the replacement of multiplication by a power of 2 with a shift instruction or using an increment instruction in place of adding 1. Such optimization might also be used to perform independent computations to take advantage of parallel processing capabilities of the target language.

Reinforce: Indicate how the following Java code fragment could be optimized by a compiler. Do this by rewriting the fragment in optimized form.

```java
u = 10;
x = 3 * y - 7;
while (x>0)
{
  if (3*y > z)
    z++;
  else
    x+=u;
}
```

2.4 Language Design Characteristics

The primary goal of a programming language is to assist the programmer in the software-development process. This includes assistance in the design, implementation, testing, verification, and maintenance of the software.

There are a number of characteristics of a language that contribute to this goal. All of these characteristics are present in each language in varying degrees, and there is often a trade-off among two or more of them. In this section, we will discuss five of these characteristics.
Although this list is by no means exhaustive, it includes those the authors believe to be the most significant for programming language effectiveness.

2.4.1 Simplicity

A programming language should strive for simplicity in both semantics and syntax. A language has semantic simplicity if that language contains a minimal number of concepts and structures. These concepts should be natural, quickly learned, and easily understood with little danger of misinterpretation.

Syntactic simplicity requires that the syntax represent each concept in one and only one way and that this representation is as easily understood as possible. This does not necessarily imply that the syntax is as concise as possible, because conciseness is often counterproductive to readability. It does exclude multiple representations of the same semantic concept and syntactic representations that are easily confused.

2.4.2 Abstraction

An abstraction is a representation of an object that includes only the relevant attributes of the original object, ignoring those attributes that are irrelevant to the purpose at hand. For example, a box score for a baseball game contains a summary of the performance of each player and is an abstraction of the game itself, including only those details relevant to those interested in its contents. A further abstraction of the game would be just the score.

A programming language's ability to express and use abstractions is important at both the data and procedural level. At the data level, the programmer is able to work more effectively by using simpler abstractions that do not include many irrelevant details of the data objects. At the procedural level, abstractions facilitate good design practices and modularity.

The level to which a language implements the hiding of irrelevant details in the creation and use of abstractions is important to its ability to support effective design, implementation, and modification of programs.

2.4.3 Expressiveness

Expressiveness refers to the ease with which an object can be represented. In relation to programming languages, this means the language should permit the natural representation of both data objects and procedures. Appropriate data structures and control structures are corresponding examples of this.

Expressiveness may conflict with simplicity, as more expressiveness will frequently result in greater complexity in the language. In addition, expressiveness is related to the problem domain. For example, language features that are expressive for artificial intelligence are probably not expressive for engineering applications. In order to resolve this conflict between expressiveness and simplicity, the current trend is for languages to restrict their problem domain to provide both characteristics.

2.4.4 Orthogonality

Simplicity requires a language to incorporate as few concepts as possible. Expressiveness
requires that the concepts closely match the objects they represent, while abstraction eliminates irrelevant details. **Orthogonality** refers to the interaction between concepts—namely, the degree to which different concepts can be combined with each other in a consistent manner.

Violations of orthogonality occur when two concepts cannot interact with each other or when they interact with each other in a manner inconsistent with their other interactions. For example, if a language does not allow a string to be passed as a parameter, the two concepts "string" and "parameter" may not interact, and hence there is a lack of orthogonality. If a language uses the operator := for integer assignment and <- for string assignment, then the concept "assignment" interacts inconsistently with the concepts of "integer" and "string."

Orthogonality reduces the number of exceptions to the rules of a language and makes the language easier to learn and remember. It is easier to remember that = means assignment for all types than to remember a handful of exceptions to this rule.

Orthogonality can lead to problems, however, in situations where certain combinations of concepts are difficult to implement. This is particularly true when the combination is unlikely to occur or is completely impossible.

### 2.4.5 Portability

The ability to maintain programs is affected by the first four characteristics in the sense that they all encourage ease of program understanding and modification. One major maintenance problem that these characteristics do not address is the movement of a program from one computing environment to another. The **portability** of a language is much greater when a machine-independent standard exists for that language. For some languages, the standard is an official standard constructed and approved by a standards organization such as the American National Standards Institute (ANSI). For others, the standard may be a **de facto** standard that has informally developed through the use of the language. Some languages for which formal standards are defined are FORTRAN, COBOL, Ada, Common LISP, C, and Pascal.

*Discuss: List other important language design characteristics in addition to the given in this section.*

### 2.5 Choice of Language

Because there are a large number of programming languages in existence, a programmer with a given application is faced with the choice of the language to use. In this section, we describe seven criteria that are important considerations when making this decision.

#### 2.5.1 Implementation

The implementation of the language refers to the language translator that is used. There are two important considerations related to the implementation: its availability and its efficiency.

The availability of a language obviously impacts the decision as to whether or not to use it for a given application. For example, a language with no translator available for many small computers is, therefore, not a feasible choice if programs must be executed on such machines. Even when a translator exists, its cost may make it impossible to use.

The efficiency of an implementation refers to the speed of execution of the object pro-
grams created by the translator. Historically, FORTRAN was frequently chosen for applications because FORTRAN compilers often contained many optimization features and therefore produced very efficient object code. On the other hand, languages translated by interpreters are notorious for the slow execution of their programs. Efficiency is a factor when the application has specific speed requirements.

2.5.2 Programmer Knowledge

Although it would be nice to assume that all programmers (and especially those who have read this book) are equally adept at programming in every language, this is certainly not the case. Primarily as a result of the programmer's education and experience, he or she typically favors one or two languages and programs best when using those languages. Although learning a new, more appropriate language for a given application would certainly be a broadening experience, it is one that an employer is often unwilling to support financially.

An even more common factor is the knowledge of other programmers within the organization. Those other programmers may be responsible for validating, testing, modifying, or maintaining the program, so the cost of using an unfamiliar language would include training costs for many people. This factor in the choice of a language is a major reason why languages like FORTRAN and COBOL continue to be popular in spite of the availability of many more effective languages.

2.5.3 Portability

When the ability to run an application on a variety of computers is important, portability is a significant criterion in language choice. Languages that adhere to stable standards, such as FORTRAN, COBOL, and Ada, are much safer choices in this situation, than are languages whose implementations are machine dependent or rapidly changing, and result in more time and expense when porting to a new system.

2.5.4 Syntax

Some applications are better accommodated by the syntax of one language than the syntax of other languages. For example, the syntax of FORTRAN was designed to meet the requirements of expressing mathematical programs, whereas the English-like syntax of COBOL makes it a good choice when the program must be understandable to non-programmers. Other languages such as Pascal and Ada have syntax that makes the expression of control structures easier.

2.5.5 Semantics

The semantics of a language can be a significant factor in its selection for a given application. If an application requires or is facilitated by a certain language feature, a language that provides that capability might be chosen. For example, if concurrent processing is required for a given application, a language with concurrent features, such as Java, might be chosen. Often processing an application is greatly simplified by the use of recursion, making languages that support recursion strong candidates.
2.5.6 Programming Environment

The presence of a rich environment to support software development can be a factor in the choice of a language. If the language resides in an environment that provides a context-sensitive editor, a symbolic debugger, a source code control system, windowing, or any of a number of other software development tools, the effort required to produce the software can be significantly reduced. Although some of these features are available in language-independent environments such as operating systems, often they are specific to a single language implementation. The availability of libraries of programs that interface with programs written in a given language is also an important factor.

2.5.7 Model of Computation

A final consideration in the selection of a programming language is the model of computation on which the language is based. Section 1.1 discusses the models emphasized in this text, and certain applications adapt most easily to one model over the others. For example, if the application requires a significant amount of heuristic searching, a language adhering to the logic-oriented model would be appropriate. Simulations are usually most easily implemented using the object-oriented model.

Discuss: Select a language that you have used to write programs. Discuss the design of that language in terms of the five criteria given in Section 2.5.

2.5.8 An Example

Consider the following real-life situation.

A researcher requires a program that will generate all possible combinations of a collection of objects, apply a metric to each combination, and report the results of the application. The application of the metric involves comparing each object to each of the other objects in every combination. Thus, this task is iterative at several levels and could be coded to exploit recursion. The first question the researcher must answer is "which programming language should be used?"

We consider the seven criteria discussed above. The researcher is really only concerned with the resulting measurements; thus the speed of the implementation is not an issue. This is to be a locally written program with a specific purpose and a short lifetime; therefore programmer knowledge and portability concerns can be ignored. The researcher likes to program but not in cryptic languages, so syntax is an issue. A language whose semantics provides easy iteration and/or natural recursion would allow for a simpler and more easily understandable program. In addition, a model of computation that allows a simple, abstract representation of the objects and the metric is desirable.

In this situation, the researcher chooses the language Prolog as the programming language for the project. Prolog implements the logic-oriented model of computation (see chapter 14) and provides a simple and natural mechanism for recursively generating the combinations. The objects can be represented at a very abstract level, and the metric can be simplified because of
the simpler object representation. The Prolog program is quite short (40 lines of semimodular code) and easy to understand. Because Prolog is typically interpreted, and the interpreter on the researcher's computer is not very fast, the program will take a long time to compute the necessary results. The researcher will begin data collection on a Friday and, returning on Monday, will find it takes more than two days to complete data collection.

Now, we reconsider the seven criteria under different circumstances. The researcher wants to work this metric application into a collection of programs that will be used by other people. Performance has become an issue; that is, 52 hours is not an acceptable time to wait for a program to collect metric data. After investigating, the researcher discovers that the speed of the Prolog interpreter at the work site cannot be improved and another interpreter cannot be obtained. In addition, the researcher wants to share the code with others and wants someone else (a research assistant) to support the code. Thus, a different language must be chosen that compiles to much faster code, that is portable, and that is known by many other programmers.

The new language chosen for the project is C. The big advantages to C are that it compiles to very fast code and it is very portable. It also provides natural support for both complex iteration and recursion. The new program is much longer than the Prolog version (because the objects cannot be abstractly represented), but it is also much faster. The program is now 200 lines long, but runs in less than two hours.

_Each: Identify four computers or computer functions and describe the languages they accept for input. Make sure you select at least one printer and one other computer peripheral (e.g., scanner [using TWAIN] or graphics interface card)._ 

_Each: Look up some languages that are used in handheld devices and evaluate them. Use the criteria from Section 2.5. Some suggestions are: OVAL from Symbian (http://www.symbian.com/...) scripting language for digital cameras (Kodak web site) language for running Lego Mindstorms robots_ 

Chapter 2 - Terms

syntax
semantics
grammar
terminal symbol
nonterminal symbol
production
start symbol
metalanguage
Backus-Naur Form (BNF)
Extended BNF (EBNF)
alternation
optionality
repetition
syntax diagrams
ambiguous grammar
ambiguous language
operational semantics
abstract machine
denotational semantics
axiomatic semantics
Vienna Definition Language
assertion
precondition
postcondition
interpreter
compiler
compile time
run time
token string
parse tree
abstract program
symbol table
lexical analysis
syntactic analysis
semantic analysis
code generation
optimization
abstraction
orthogonality
portability