3

Describing Syntax and Semantics

This chapter covers the following topics. First, the terms *syntax* and *semantics* are defined. Then, a detailed discussion of the most common method of describing syntax, context-free grammars (also known as Backus-Naur Form), is presented. Included in this discussion are derivations, parse trees, ambiguity, descriptions of operator precedence and associativity, and extended Backus-Naur Form. Attribute grammars, which can be used to describe both the syntax and static semantics of programming languages, are discussed next. In the last section, three formal methods of describing semantics—operational, axiomatic, and denotational semantics—are introduced. Because of the inherent complexity of the semantics description methods, our discussion of them is brief. One could easily write an entire book on just one of the three (as several authors have).

3.1 Introduction

The task of providing a concise yet understandable description of a programming language is difficult but essential to the language's success. ALGOL 60 and ALGOL 68 were first presented using concise formal descriptions; in both cases, however, the descriptions were not easily understandable, partly because each used a new notation. The levels of acceptance of both languages suffered as a result. On the other hand, some languages have suffered the problem of having many slightly different dialects, a result of a simple but informal and imprecise definition.

One of the problems in describing a language is the diversity of the people who must understand the description. Among these are initial evaluators, implementors, and users. Most new programming languages are subjected to a period of scrutiny by potential users, often people within the organization that employs the language's designer, before their designs are completed. These are the initial evaluators. The success of this feedback cycle depends heavily on the clarity of the description.

Programming language implementors obviously must be able to determine how the expressions, statements, and program units of a language are formed, and also their intended effect when executed. The difficulty of the implementors' job is, in part, determined by the completeness and precision of the language description.

Finally, language users must be able to determine how to encode software solutions by referring to a language reference manual. Textbooks and courses enter into this process, but language manuals are usually the only authoritative printed information source about a language.

The study of programming languages, like the study of natural languages, can be divided into examinations of syntax and semantics. The **`syntax`** of a programming language is the form of its expressions, statements, and program units. Its **semantics** is the meaning of those expressions, statements, and program units. For example, the syntax of a Java **`while`** statement is

`while` (boolean_expr) statement

The semantics of this statement form is that when the current value of the Boolean expression is true, the embedded statement is executed. Otherwise, control continues after the **`while`** construct. Then control implicitly returns to the Boolean expression to repeat the process.

Although they are often separated for discussion purposes, syntax and semantics are closely related. In a well-designed programming language, semantics should follow directly from syntax; that is, the appearance of a statement should strongly suggest what the statement is meant to accomplish.

Describing syntax is easier than describing semantics, partly because a concise and universally accepted notation is available for syntax description, but none has yet been developed for semantics.

3.2 The General Problem of Describing Syntax

A language, whether natural (such as English) or artificial (such as Java), is a set of strings of characters from some alphabet. The strings of a language are called **sentences** or statements. The syntax rules of a language specify which strings of characters from the language's alphabet are in the language. English, for example, has a large and complex collection of rules for specifying the syntax of its sentences. By comparison, even the largest and most complex programming languages are syntactically very simple.

Formal descriptions of the syntax of programming languages, for simplicity's sake, often do not include descriptions of the lowest-level syntactic units. These small units are called **lexemes**. The description of lexemes can be given by a lexical specification, which is usually separate from the syntactic description of the language. The lexemes of a programming language include its numeric literals, operators, and special words, among others. One can think of programs as strings of lexemes rather than of characters.

Lexemes are partitioned into groups—for example, the names of variables, methods, classes, and so forth in a programming language form a group called *identifiers*. Each lexeme group is represented by a name, or token. So, a **token** of a language is a category of its lexemes. For example, an identifier is a token that can have lexemes, or instances, such as `sum` and `total`. In some cases, a token has only a single possible lexeme. For example, the token for the arithmetic operator symbol + has just one possible lexeme. Consider the following Java statement:

```
index = 2 * count + 17;
```

The lexemes and tokens of this statement are

Lexemes	*Tokens*
`index`	identifier
`=`	equal_sign
`2`	int_literal

`*`	mult_op
`count`	identifier
`+`	plus_op
`17`	int_literal
`;`	semicolon

The example language descriptions in this chapter are very simple, and most include lexeme descriptions.

3.2.1 Language Recognizers

In general, languages can be formally defined in two distinct ways: by **recognition** and by **generation** (although neither provides a definition that is practical by itself for people trying to learn or use a programming language). Suppose we have a language L that uses an alphabet Σ of characters. To define L formally using the recognition method, we would need to construct a mechanism R, called a recognition device, capable of reading strings of characters from the alphabet Σ. R would indicate whether a given input string was or was not in L. In effect, R would either accept or reject the given string. Such devices are like filters, separating legal sentences from those that are incorrectly formed. If R, when fed any string of characters over Σ, accepts it only if it is in L, then R is a description of L. Because most useful languages are, for all practical purposes, infinite, this might seem like a lengthy and ineffective process. Recognition devices, however, are not used to enumerate all of the sentences of a language—they have a different purpose.

The syntax analysis part of a compiler is a recognizer for the language the compiler translates. In this role, the recognizer need not test all possible strings of characters from some set to determine whether each is in the language. Rather, it need only determine whether given programs are in the language. In effect then, the syntax analyzer determines whether the given programs are syntactically correct. The structure of syntax analyzers, also known as parsers, is discussed in Chapter 4.

3.2.2 Language Generators

A language generator is a device that can be used to generate the sentences of a language. We can think of the generator as having a button that produces a sentence of the language every time it is pushed. Because the particular sentence that is produced by a generator when its button is pushed is unpredictable, a generator seems to be a device of limited usefulness as a language descriptor. However, people prefer certain forms of generators over recognizers because they can more easily read and understand them. By contrast, the syntax-checking portion of a compiler (a language recognizer) is not as useful a language description for a programmer because it can be used only in trial-and-error mode. For example, to determine the correct syntax of a particular statement using a compiler, the programmer can only submit a speculated version and note whether

the compiler accepts it. On the other hand, it is often possible to determine whether the syntax of a particular statement is correct by comparing it with the structure of the generator.

There is a close connection between formal generation and recognition devices for the same language. This was one of the seminal discoveries in computer science, and it led to much of what is now known about formal languages and compiler design theory. We return to the relationship of generators and recognizers in the next section.

3.3 Formal Methods of Describing Syntax

This section discusses the formal language-generation mechanisms, usually called **grammars**, that are commonly used to describe the syntax of programming languages.

3.3.1 Backus-Naur Form and Context-Free Grammars

In the middle to late 1950s, two men, Noam Chomsky and John Backus, in unrelated research efforts, developed the same syntax description formalism, which subsequently became the most widely used method for programming language syntax.

3.3.1.1 Context-Free Grammars

In the mid-1950s, Chomsky, a noted linguist (among other things), described four classes of generative devices or grammars that define four classes of languages (Chomsky, 1956, 1959). Two of these grammar classes, named *context-free* and *regular*, turned out to be useful for describing the syntax of programming languages. The forms of the tokens of programming languages can be described by regular grammars. The syntax of whole programming languages, with minor exceptions, can be described by context-free grammars. Because Chomsky was a linguist, his primary interest was the theoretical nature of natural languages. He had no interest at the time in the artificial languages used to communicate with computers. So it was not until later that his work was applied to programming languages.

3.3.1.2 Origins of Backus-Naur Form

Shortly after Chomsky's work on language classes, the ACM-GAMM group began designing ALGOL 58. A landmark paper describing ALGOL 58 was presented by John Backus, a prominent member of the ACM-GAMM group, at an international conference in 1959 (Backus, 1959). This paper introduced a new formal notation for specifying programming language syntax. The new notation was later modified slightly by Peter Naur for the description of

ALGOL 60 (Naur, 1960). This revised method of syntax description became known as **Backus-Naur Form**, or simply **BNF**.

BNF is a natural notation for describing syntax. In fact, something similar to BNF was used by Panini to describe the syntax of Sanskrit several hundred years before Christ (Ingerman, 1967).

Although the use of BNF in the ALGOL 60 report was not immediately accepted by computer users, it soon became and is still the most popular method of concisely describing programming language syntax.

It is remarkable that BNF is nearly identical to Chomsky's generative devices for context-free languages, called **context-free grammars**. In the remainder of the chapter, we refer to context-free grammars simply as grammars. Furthermore, the terms BNF and grammar are used interchangeably.

3.3.1.3 Fundamentals

A **metalanguage** is a language that is used to describe another language. BNF is a metalanguage for programming languages.

BNF uses abstractions for syntactic structures. A simple Java assignment statement, for example, might be represented by the abstraction <assign> (pointed brackets are often used to delimit names of abstractions). The actual definition of <assign> can be given by

<assign> → <var> = <expression>

The text on the left side of the arrow, which is aptly called the **left-hand side** (LHS), is the abstraction being defined. The text to the right of the arrow is the definition of the LHS. It is called the **right-hand side** (RHS) and consists of some mixture of tokens, lexemes, and references to other abstractions. (Actually, tokens are also abstractions.) Altogether, the definition is called a **rule**, or **production**. In the example rule just given, the abstractions <var> and <expression> obviously must be defined for the <assign> definition to be useful.

This particular rule specifies that the abstraction <assign> is defined as an instance of the abstraction <var>, followed by the lexeme =, followed by an instance of the abstraction <expression>. One example sentence whose syntactic structure is described by the rule is

```
total = subtotal1 + subtotal2
```

The abstractions in a BNF description, or grammar, are often called **nonterminal symbols**, or simply **nonterminals**, and the lexemes and tokens of the rules are called **terminal symbols**, or simply **terminals**. A BNF description, or **grammar**, is a collection of rules.

Nonterminal symbols can have two or more distinct definitions, representing two or more possible syntactic forms in the language. Multiple definitions can be written as a single rule, with the different definitions separated by

the symbol |, meaning logical OR. For example, a Java **if** statement can be described with the rules

<if_stmt> → **if** (<logic_expr>) <stmt>
<if_stmt> → **if** (<logic_expr>) <stmt> **else** <stmt>

or with the rule

<if_stmt> → **if** (<logic_expr>) <stmt>
| **if** (<logic_expr>) <stmt> **else** <stmt>

In these rules, <stmt> represents either a single statement or a compound statement.

Although BNF is simple, it is sufficiently powerful to describe nearly all of the syntax of programming languages. In particular, it can describe lists of similar constructs, the order in which different constructs must appear, and nested structures to any depth, and even imply operator precedence and operator associativity.

3.3.1.4 Describing Lists

Variable-length lists in mathematics are often written using an ellipsis (. . .); 1, 2, . . . is an example. BNF does not include the ellipsis, so an alternative method is required for describing lists of syntactic elements in programming languages (for example, a list of identifiers appearing on a data declaration statement). For BNF, the alternative is recursion. A rule is **recursive** if its LHS appears in its RHS. The following rules illustrate how recursion is used to describe lists:

<ident_list> → identifier
| identifier, <ident_list>

This defines <ident_list> as either a single token (identifier) or an identifier followed by a comma and another instance of <ident_list>. Recursion is used to describe lists in many of the example grammars in the remainder of this chapter.

3.3.1.5 Grammars and Derivations

A grammar is a generative device for defining languages. The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the **start symbol**. This sequence of rule applications is called a **derivation**. In a grammar for a complete programming language, the start symbol represents a complete program and is often named <program>. The simple grammar shown in Example 3.1 is used to illustrate derivations.

EXAMPLE 3.1 A Grammar for a Small Language

```
<program> → begin <stmt_list> end

<stmt_list> → <stmt>
            | <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
             | <var> – <var>
             | <var>
```

The language described by the grammar of Example 3.1 has only one statement form: assignment. A program consists of the special word **begin**, followed by a list of statements separated by semicolons, followed by the special word **end**. An expression is either a single variable or two variables separated by either a + or - operator. The only variable names in this language are A, B, and C.

A derivation of a program in this language follows:

```
<program> => begin <stmt_list> end
          => begin <stmt> ; <stmt_list> end
          => begin <var> = <expression> ; <stmt_list> end
          => begin A = <expression> ; <stmt_list> end
          => begin A = <var> + <var> ; <stmt_list> end
          => begin A = B + <var> ; <stmt_list> end
          => begin A = B + C ; <stmt_list> end
          => begin A = B + C ; <stmt> end
          => begin A = B + C ; <var> = <expression> end
          => begin A = B + C ; B = <expression> end
          => begin A = B + C ; B = <var> end
          => begin A = B + C ; B = C end
```

This derivation, like all derivations, begins with the start symbol, in this case <program>. The symbol => is read "derives." Each successive string in the sequence is derived from the previous string by replacing one of the nonterminals with one of that nonterminal's definitions. Each of the strings in the derivation, including <program>, is called a **sentential form**.

In this derivation, the replaced nonterminal is always the leftmost nonterminal in the previous sentential form. Derivations that use this order of replacement are called **leftmost derivations**. The derivation continues until the sentential form contains no nonterminals. That sentential form, consisting of only terminals, or lexemes, is the generated sentence.

In addition to leftmost, a derivation may be rightmost or in an order that is neither leftmost nor rightmost. Derivation order has no effect on the language generated by a grammar.

By choosing alternative RHSs of rules with which to replace nonterminals in the derivation, different sentences in the language can be generated. By exhaustively choosing all combinations of choices, the entire language can be generated. This language, like most others, is infinite, so one cannot generate *all* the sentences in the language in finite time.

Example 3.2 is another example of a grammar for part of a typical programming language.

EXAMPLE 3.2 A Grammar for Simple Assignment Statements

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
         | <id> * <expr>
         | ( <expr> )
         | <id>
```

The grammar of Example 3.2 describes assignment statements whose right sides are arithmetic expressions with multiplication and addition operators and parentheses. For example, the statement

```
A = B * ( A + C )
```

is generated by the leftmost derivation:

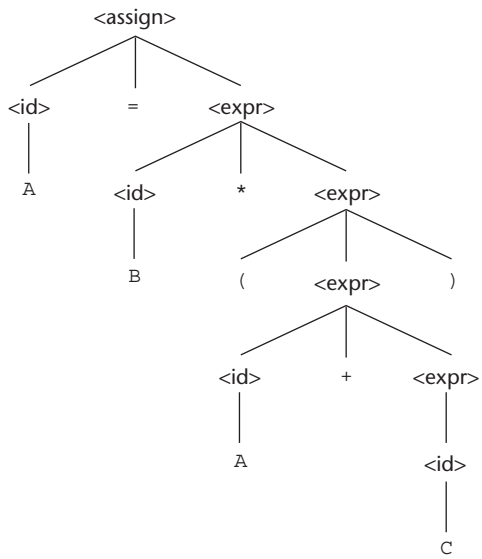
```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <id> * <expr>
         => A = B * <expr>
         => A = B * ( <expr> )
         => A = B * ( <id> + <expr> )
         => A = B * ( A + <expr> )
         => A = B * ( A + <id> )
         => A = B * ( A + C )
```

3.3.1.6 Parse Trees

One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the languages they define. These hierarchical structures are called **parse trees**. For example, the parse tree in Figure 3.1 shows the structure of the assignment statement derived previously.

Figure 3.1

A parse tree for the simple statement `A = B * (A + C)`



Every internal node of a parse tree is labeled with a nonterminal symbol; every leaf is labeled with a terminal symbol. Every subtree of a parse tree describes one instance of an abstraction in the sentence.

3.3.1.7 Ambiguity

A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be **ambiguous**. Consider the grammar shown in Example 3.3, which is a minor variation of the grammar shown in Example 3.2.

EXAMPLE 3.3 An Ambiguous Grammar for Simple Assignment Statements

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
         | <expr> * <expr>
         | ( <expr> )
         | <id>
```

The grammar of Example 3.3 is ambiguous because the sentence

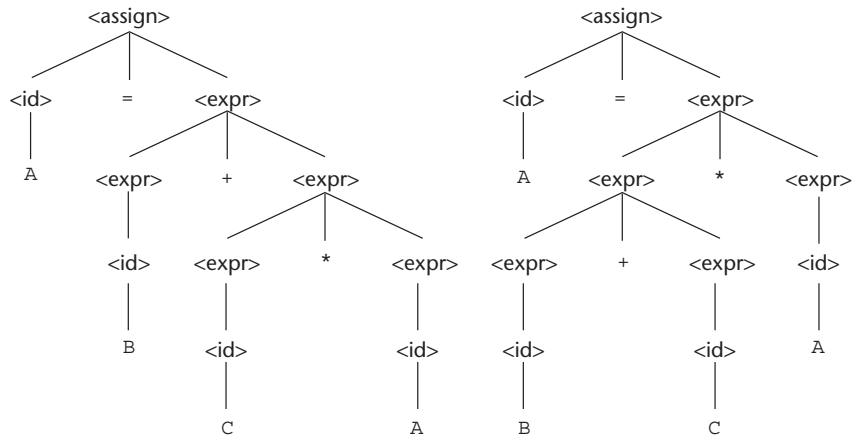
```
A = B + C * A
```

has two distinct parse trees, as shown in Figure 3.2. The ambiguity occurs because the grammar specifies slightly less syntactic structure than does the grammar of

Example 3.2. Rather than allowing the parse tree of an expression to grow only on the right, this grammar allows growth on both the left and the right.

Figure 3.2

Two distinct parse trees for the same sentence, `A = B + C * A`



Syntactic ambiguity of language structures is a problem because compilers often base the semantics of those structures on their syntactic form. Specifically, the compiler chooses the code to be generated for a statement by examining its parse tree. If a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely. This problem is discussed in two specific examples in the following subsections.

There are several other characteristics of a grammar that are sometimes useful in determining whether a grammar is ambiguous.[1] They include the following: (1) if the grammar generates a sentence with more than one leftmost derivation and (2) if the grammar generates a sentence with more than one rightmost derivation.

Some parsing algorithms can be based on ambiguous grammars. When such a parser encounters an ambiguous construct, it uses nongrammatical information provided by the designer to construct the correct parse tree. In many cases, an ambiguous grammar can be rewritten to be unambiguous but still generate the desired language.

3.3.1.8 Operator Precedence

When an expression includes two different operators, for example, `x + y * z`, one obvious semantic issue is the order of evaluation of the two operators (for example, in this expression is it add and then multiply, or vice versa?). This semantic question can be answered by assigning different precedence levels to operators. For example, if `*` has been assigned higher precedence than `+` (by the language

1. Note that it is mathematically impossible to determine whether an arbitrary grammar is ambiguous.

designer), multiplication will be done first, regardless of the order of appearance of the two operators in the expression.

As stated previously, a grammar can describe a certain syntactic structure so that part of the meaning of the structure can be determined from its parse tree. In particular, the fact that an operator in an arithmetic expression is generated lower in the parse tree (and therefore must be evaluated first) can be used to indicate that it has precedence over an operator produced higher up in the tree. In the first parse tree of Figure 3.2, for example, the multiplication operator is generated lower in the tree, which could indicate that it has precedence over the addition operator in the expression. The second parse tree, however, indicates just the opposite. It appears, therefore, that the two parse trees indicate conflicting precedence information.

Notice that although the grammar of Example 3.2 is not ambiguous, the precedence order of its operators is not the usual one. In this grammar, a parse tree of a sentence with multiple operators, regardless of the particular operators involved, has the rightmost operator in the expression at the lowest point in the parse tree, with the other operators in the tree moving progressively higher as one moves to the left in the expression. For example, in the expression `A + B * C`, `*` is the lowest in the tree, indicating it is to be done first. However, in the expression `A * B + C`, `+` is the lowest, indicating it is to be done first.

A grammar can be written for the simple expressions we have been discussing that is both unambiguous and specifies a consistent precedence of the `+` and `*` operators, regardless of the order in which the operators appear in an expression. The correct ordering is specified by using separate nonterminal symbols to represent the operands of the operators that have different precedence. This requires additional nonterminals and some new rules. Instead of using <expr> for both operands of both `+` and `*`, we could use three nonterminals to represent operands, which allows the grammar to force different operators to different levels in the parse tree. If <expr> is the root symbol for expressions, `+` can be forced to the top of the parse tree by having <expr> directly generate only `+` operators, using the new nonterminal, <term>, as the right operand of `+`. Next, we can define <term> to generate `*` operators, using <term> as the left operand and a new nonterminal, <factor>, as its right operand. Now, `*` will always be lower in the parse tree, simply because it is farther from the start symbol than `+` in every derivation. The grammar of Example 3.4 is such a grammar.

EXAMPLE 3.4 An Unambiguous Grammar for Expressions

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
         | <term>
<term> → <term> * <factor>
         | <factor>
<factor> → ( <expr> )
           | <id>
```

The grammar in Example 3.4 generates the same language as the grammars of Examples 3.2 and 3.3, but it is unambiguous and it specifies the usual precedence order of multiplication and addition operators. The following derivation of the sentence `A = B + C * A` uses the grammar of Example 3.4:

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <expr> + <term>
         => A = <term> + <term>
         => A = <factor> + <term>
         => A = <id> + <term>
         => A = B + <term>
         => A = B + <term> * <factor>
         => A = B + <factor> * <factor>
         => A = B + <id> * <factor>
         => A = B + C * <factor>
         => A = B + C * <id>
         => A = B + C * A
```

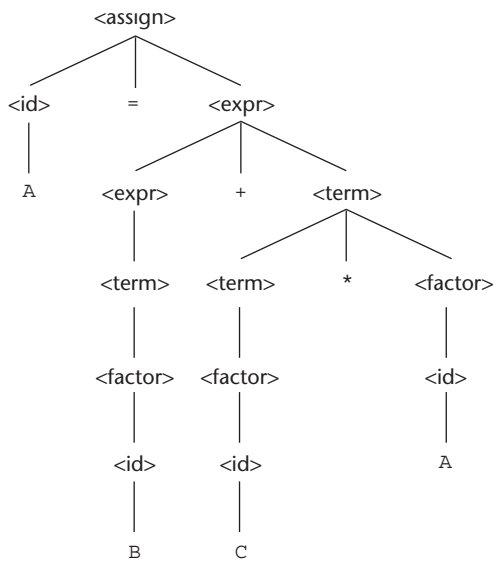
The unique parse tree for this sentence, using the grammar of Example 3.4, is shown in Figure 3.3.

The connection between parse trees and derivations is very close: Either can easily be constructed from the other. Every derivation with an unambiguous grammar has a unique parse tree, although that tree can be represented by different derivations. For example, the following derivation of the sentence `A = B + C * A` is different from the derivation of the same sentence given previously. This is a rightmost derivation, whereas the previous one is leftmost. Both of these derivations, however, are represented by the same parse tree.

```
<assign> => <id> = <expr>
         => <id> = <expr> + <term>
         => <id> = <expr> + <term> * <factor>
         => <id> = <expr> + <term> * <id>
         => <id> = <expr> + <term> * A
         => <id> = <expr> + <factor> * A
         => <id> = <expr> + <id> * A
         => <id> = <expr> + C * A
         => <id> = <term> + C * A
         => <id> = <factor> + C * A
         => <id> = <id> + C * A
         => <id> = B + C * A
         => A = B + C * A
```

Figure 3.3

The unique parse tree for `A = B + C * A` using an unambiguous grammar



3.3.1.9 Associativity of Operators

When an expression includes two operators that have the same precedence (as * and / usually have)—for example, `A / B * C`—a semantic rule is required to specify which should have precedence.[2] This rule is named *associativity*.

2. An expression with two occurrences of the same operator has the same issue; for example, `A / B / C`.

As was the case with precedence, a grammar for expressions may correctly imply operator associativity. Consider the following example of an assignment statement:

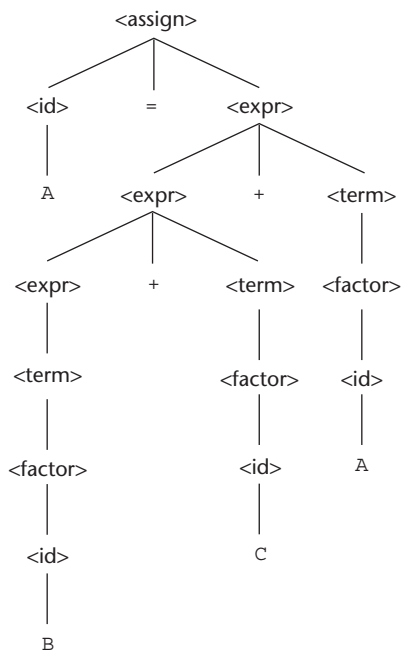
```
A = B + C + A
```

The parse tree for this sentence, as defined with the grammar of Example 3.4, is shown in Figure 3.4.

The parse tree of Figure 3.4 shows the left addition operator lower than the right addition operator. This is the correct order if addition is meant to be left associative, which is typical. In most cases, the associativity of addition in a computer is irrelevant. In mathematics, addition is associative, which means that left and right associative orders of evaluation mean the same thing. That is, `(A + B) + C = A + (B + C)`. Floating-point addition in a computer, however, is not necessarily associative. For example, suppose floating-point values store seven digits of accuracy. Consider the problem of adding 11 numbers together, where one of the numbers is 10^7 and the other ten are 1. If the small numbers (the 1's) are each added to the large number, one at a time, there is no effect on that number, because the small numbers occur in the eighth digit of the large number. However, if the small numbers are first added together and the result is added to the large number, the result in seven-digit accuracy is $1.000001 * 10^7$. Subtraction and division are not associative, whether in mathematics or in a computer. Therefore, correct associativity may be essential for an expression that contains either of them.

Figure 3.4

A parse tree for `A = B + C + A` illustrating the associativity of addition



When a grammar rule has its LHS also appearing at the beginning of its RHS, the rule is said to be **left recursive**. This left recursion specifies left associativity. For example, the left recursion of the rules of the grammar of Example 3.4 causes it to make both addition and multiplication left associative. Unfortunately, left recursion disallows the use of some important syntax analysis algorithms. When such algorithms are to be used, the grammar must be modified to remove the left recursion. This, in turn, disallows the grammar from precisely specifying that certain operators are left associative. Fortunately, left associativity can be enforced by the compiler, even though the grammar does not dictate it.

In most languages that provide it, the exponentiation operator is right associative. To indicate right associativity, right recursion can be used. A grammar rule is **right recursive** if the LHS appears at the right end of the RHS. Rules such as

<factor> → <exp> ** <factor>
| <exp>
<exp> → (<expr>)
| id

could be used to describe exponentiation as a right-associative operator.

3.3.1.10 An Unambiguous Grammar for `if-then-else`

The BNF rules for an Ada **`if-then-else`** statement are as follows:

<if_stmt> → **`if`** <logic_expr> **`then`** <stmt>
`if` <logic_expr> **`then`** <stmt> **`else`** <stmt>

If we also have <stmt> → <if_stmt>, this grammar is ambiguous. The simplest sentential form that illustrates this ambiguity is

`if` <logic_expr> **`then if`** <logic_expr> **`then`** <stmt> **`else`** <stmt>

The two parse trees in Figure 3.5 show the ambiguity of this sentential form. Consider the following example of this construct:

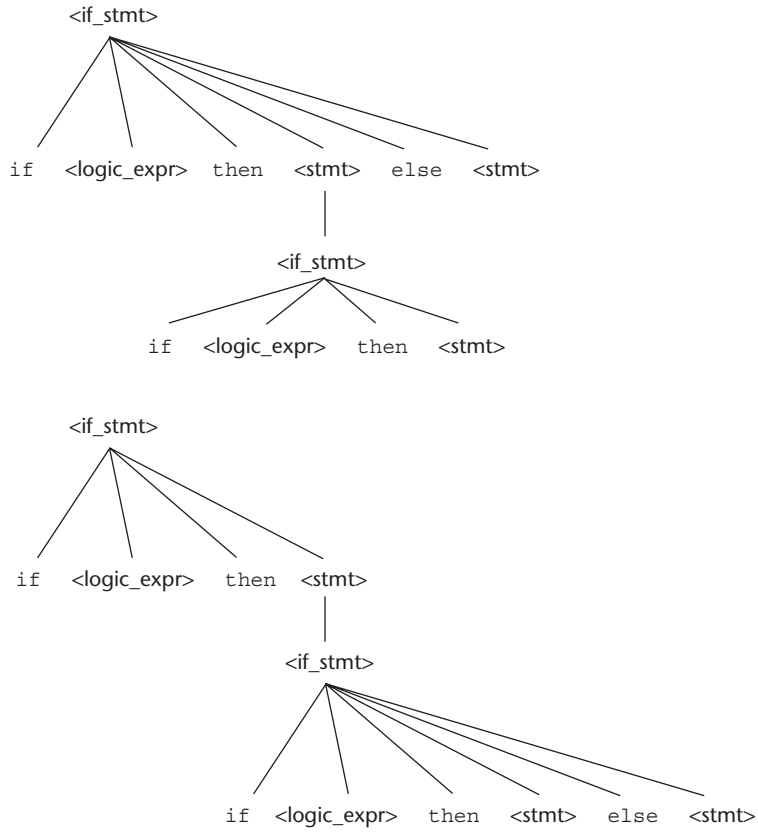
```
if done == true
  then if denom == 0
    then quotient = 0;
    else quotient = num / denom;
```

The problem is that if the upper parse tree in Figure 3.5 is used as the basis for translation, the else clause would be executed when `done` is not true, which probably is not what was intended by the author of the construct. We will examine the practical problems associated with this else-association problem in Chapter 8.

We will now develop an unambiguous grammar that describes this **`if`** statement. The rule for **`if`** constructs in many languages is that an else clause, when present, is matched with the nearest previous unmatched **`then`**.

Figure 3.5

Two distinct parse trees for the same sentential form



Therefore, there cannot be an **if** statement without an **else** between a **then** and its matching **else**. So, for this situation, statements must be distinguished between those that are matched and those that are unmatched, where unmatched statements are **else**-less **if**s and all other statements are matched. The problem with the earlier grammar is that it treats all statements as if they had equal syntactic significance—that is, as if they were all matched.

To reflect the different categories of statements, different abstractions, or nonterminals, must be used. The unambiguous grammar based on these ideas follows:

<stmt> → <matched> | <unmatched>

<matched> → **if** <logic_expr> **then** <matched> **else** <matched>
| any non-if statement

<unmatched> → **if** <logic_expr> **then** <stmt>
| **if** <logic_expr> **then** <matched> **else** <unmatched>

There is just one possible parse tree, using this grammar, for the following sentential form:

if <logic_expr> **then if** <logic_expr> **then** <stmt> **else** <stmt>

3.3.2 Extended BNF

Because of a few minor inconveniences in BNF, it has been extended in several ways. Most extended versions are called Extended BNF, or simply EBNF, even though they are not all exactly the same. The extensions do not enhance the descriptive power of BNF; they only increase its readability and writability.

Three extensions are commonly included in the various versions of EBNF. The first of these denotes an optional part of an RHS, which is delimited by brackets. For example, a C **`if-else`** statement can be described as

<if_stmt> → **`if`** (<expression>) <statement> [**`else`** <statement>]

Without the use of the brackets, the syntactic description of this statement would require the following two rules:

<if_stmt> → **`if`** (<expression>) <statement>
| **`if`** (<expression>) <statement> **`else`** <statement>

The second extension is the use of braces in an RHS to indicate that the enclosed part can be repeated indefinitely or left out altogether. This extension allows lists to be built with a single rule, instead of using recursion and two rules. For example, lists of identifiers separated by commas can be described by the following rule:

<ident_list> → <identifier> {, <identifier>}

This is a replacement of the recursion by a form of implied iteration; the part enclosed within braces can be iterated any number of times.

The third common extension deals with multiple-choice options. When a single element must be chosen from a group, the options are placed in parentheses and separated by the OR operator, |. For example,

<term> → <term> (* | / | %) <factor>

In BNF, a description of this <term> would require the following three rules:

<term> → <term> * <factor>
| <term> / <factor>
| <term> % <factor>

The brackets, braces, and parentheses in the EBNF extensions are **metasymbols**, which means they are notational tools and not terminal symbols in the syntactic entities they help describe. In cases where these metasymbols are also terminal symbols in the language being described, the instances that are terminal symbols can be underlined or quoted. Example 3.5 illustrates the use of braces and multiple choices in an EBNF grammar.

EXAMPLE 3.5 BNF and EBNF Versions of an Expression Grammar

BNF:

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
<factor> → <exp> ** <factor>
         | <exp>
<exp> → (<expr>)
      | id
```

EBNF:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
      | id
```

The BNF rule

<expr> → <expr> + <term>

clearly specifies—in fact forces—the + operator to be left associative. However, the EBNF version,

<expr> → <term> {+ <term>}

does not imply the direction of associativity. This problem is overcome in a syntax analyzer based on an EBNF grammar for expressions by designing the syntax analysis process to enforce the correct associativity. This is further discussed in Chapter 4.

Some versions of EBNF allow a numeric superscript to be attached to the right brace to indicate an upper limit to the number of times the enclosed part can be repeated. Also, some versions use a plus (+) superscript to indicate one or more repetitions. For example,

<compound> → **begin** <stmt> {<stmt>} **end**

and

<compound> → **begin** {<stmt>}$^{+}$ **end**

are equivalent.

In recent years, some variations on BNF and EBNF have appeared. Among these are the following:

- In place of the arrow, a colon is used and the RHS is placed on the next line.
- Instead of a vertical bar to separate alternative RHSs, they are simply placed on separate lines.
- In place of square brackets to indicate something being optional, the subscript opt is used. For example,

  Constructor Declarator → SimpleName (FormalParameterList$_{opt}$)
- Rather than using the | symbol in a parenthesized list of elements to indicate a choice, the words "one of" are used. For example,

AssignmentOperator → one of = *= /= %= += -=
<<= >>= &= ^= |=

There is a standard for EBNF, ISO/IEC 14977:1996(1996), but it is rarely used. The standard uses the equal sign (=) instead of an arrow in rules, terminates each RHS with a semicolon, and requires quotes on all terminal symbols. It also specifies a host of other notational rules.

3.3.3 Grammars and Recognizers

Earlier in this chapter, we suggested that there is a close relationship between generation and recognition devices for a given language. In fact, given a context-free grammar, a recognizer for the language generated by the grammar can be algorithmically constructed. A number of software systems have been developed that perform this construction. Such systems allow the quick creation of the syntax analysis part of a compiler for a new language and are therefore quite valuable. One of the first of these syntax analyzer generators is named yacc[3] (Johnson, 1975). There are now many such systems available.

3.4 Attribute Grammars

An **attribute grammar** is a device used to describe more of the structure of a programming language than can be described with a context-free grammar. An attribute grammar is an extension to a context-free grammar. The extension

3. The term yacc is an acronym for "yet another compiler compiler."

history note

Attribute grammars have been used in a wide variety of applications. They have been used to provide complete descriptions of the syntax and static semantics of programming languages (Watt, 1979); they have been used as the formal definition of a language that can be input to a compiler generation system (Farrow, 1982); and they have been used as the basis of several syntax-directed editing systems (Teitelbaum and Reps, 1981; Fischer et al., 1984). In addition, attribute grammars have been used in natural-language processing systems (Correa, 1992).

allows certain language rules to be conveniently described, such as type compatibility. Before we formally define the form of attribute grammars, we must clarify the concept of static semantics.

3.4.1 Static Semantics

There are some characteristics of the structure of programming languages that are difficult to describe with BNF, and some that are impossible. As an example of a syntax rule that is difficult to specify with BNF, consider type compatibility rules. In Java, for example, a floating-point value cannot be assigned to an integer type variable, although the opposite is legal. Although this restriction can be specified in BNF, it requires additional nonterminal symbols and rules. If all of the typing rules of Java were specified in BNF, the grammar would become too large to be useful, because the size of the grammar determines the size of the syntax analyzer.

As an example of a syntax rule that cannot be specified in BNF, consider the common rule that all variables must be declared before they are referenced. It has been proven that this rule cannot be specified in BNF.

These problems exemplify the categories of language rules called static semantics rules. The **static semantics** of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics). Many static semantic rules of a language state its type constraints. Static semantics is so named because the analysis required to check these specifications can be done at compile time.

Because of the problems of describing static semantics with BNF, a variety of more powerful mechanisms has been devised for that task. One such mechanism, attribute grammars, was designed by Knuth (1968a) to describe both the syntax and the static semantics of programs.

Attribute grammars are a formal approach both to describing and checking the correctness of the static semantics rules of a program. Although they are not always used in a formal way in compiler design, the basic concepts of attribute grammars are at least informally used in every compiler (see Aho et al., 1986).

Dynamic semantics, which is the meaning of expressions, statements, and program units, is discussed in Section 3.5.

3.4.2 Basic Concepts

Attribute grammars are context-free grammars to which have been added attributes, attribute computation functions, and predicate functions. **Attributes**, which are associated with grammar symbols (the terminal and nonterminal symbols), are similar to variables in the sense that they can have values assigned to them. **Attribute computation functions**, sometimes called semantic

functions, are associated with grammar rules. They are used to specify how attribute values are computed. **Predicate functions**, which state the static semantic rules of the language, are associated with grammar rules.

These concepts will become clearer after we formally define attribute grammars and provide an example.

3.4.3 Attribute Grammars Defined

An attribute grammar is a grammar with the following additional features:

- Associated with each grammar symbol X is a set of attributes A(X). The set A(X) consists of two disjoint sets S(X) and I(X), called synthesized and inherited attributes, respectively. **Synthesized attributes** are used to pass semantic information up a parse tree, while **inherited attributes** pass semantic information down and across a tree.
- Associated with each grammar rule is a set of semantic functions and a possibly empty set of predicate functions over the attributes of the symbols in the grammar rule. For a rule $X_0 \rightarrow X_1 \ldots X_n$, the synthesized attributes of X_0 are computed with semantic functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$. So the value of a synthesized attribute on a parse tree node depends only on the values of the attributes on that node's children nodes. Inherited attributes of symbols X_j, $1 \leq j \leq n$ (in the rule above), are computed with a semantic function of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$. So the value of an inherited attribute on a parse tree node depends on the attribute values of that node's parent node and those of its sibling nodes. Note that, to avoid circularity, inherited attributes are often restricted to functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_{(j-1)}))$. This form prevents an inherited attribute from depending on itself or on attributes to the right in the parse tree.
- A predicate function has the form of a Boolean expression on the union of the attribute set $\{A(X_0), \ldots, A(X_n)\}$ and a set of literal attribute values. The only derivations allowed with an attribute grammar are those in which every predicate associated with every nonterminal is true. A false predicate function value indicates a violation of the syntax or static semantics rules of the language.

A parse tree of an attribute grammar is the parse tree based on its underlying BNF grammar, with a possibly empty set of attribute values attached to each node. If all the attribute values in a parse tree have been computed, the tree is said to be **fully attributed**. Although in practice it is not always done this way, it is convenient to think of attribute values as being computed after the complete unattributed parse tree has been constructed by the compiler.

3.4.4 Intrinsic Attributes

Intrinsic attributes are synthesized attributes of leaf nodes whose values are determined outside the parse tree. For example, the type of an instance of a variable in a program could come from the symbol table, which is used to store variable names

and their types. The contents of the symbol table are set based on earlier declaration statements. Initially, assuming that an unattributed parse tree has been constructed and that attribute values are needed, the only attributes with values are the intrinsic attributes of the leaf nodes. Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attribute values.

3.4.5 Examples of Attribute Grammars

As a very simple example of how attribute grammars can be used to describe static semantics, consider the following fragment of an attribute grammar that describes the rule that the name on the **`end`** of an Ada procedure must match the procedure's name. (This rule cannot be stated in BNF.) The string attribute of <proc_name>, denoted by <proc_name>.string, is the actual string of characters that were found immediately following the reserved word **`procedure`** by the compiler. Notice that when there is more than one occurrence of a nonterminal in a syntax rule in an attribute grammar, the nonterminals are subscripted with brackets to distinguish them. Neither the subscripts nor the brackets are part of the described language.

Syntax rule: <proc_def> → **`procedure`** <proc_name>[1]
<proc_body> **`end`** <proc_name>[2] ;
Predicate: <proc_name>[1] string == <proc_name>[2].string

In this example, the predicate rule states that the name string attribute of the <proc_name> nonterminal in the subprogram header must match the name string attribute of the <proc_name> nonterminal following the end of the subprogram.

Next, we consider a larger example of an attribute grammar. In this case, the example illustrates how an attribute grammar can be used to check the type rules of a simple assignment statement. The syntax and static semantics of this assignment statement are as follows: The only variable names are `A`, `B`, and `C`. The right side of the assignments can be either a variable or an expression in the form of a variable added to another variable. The variables can be one of two types: int or real. When there are two variables on the right side of an assignment, they need not be the same type. The type of the expression when the operand types are not the same is always real. When they are the same, the expression type is that of the operands. The type of the left side of the assignment must match the type of the right side. So the types of operands in the right side can be mixed, but the assignment is valid only if the target and the value resulting from evaluating the right side have the same type. The attribute grammar specifies these static semantic rules.

The syntax portion of our example attribute grammar is

<assign> → <var> = <expr>
<expr> → <var> + <var>
| <var>
<var> → A | B | C

The attributes for the nonterminals in the example attribute grammar are described in the following paragraphs:

- *actual_type*—A synthesized attribute associated with the nonterminals <var> and <expr>. It is used to store the actual type, int or real, of a variable or expression. In the case of a variable, the actual type is intrinsic. In the case of an expression, it is determined from the actual types of the child node or children nodes of the <expr> nonterminal.
- *expected_type*—An inherited attribute associated with the nonterminal <expr>. It is used to store the type, either int or real, that is expected for the expression, as determined by the type of the variable on the left side of the assignment statement.

The complete attribute grammar follows in Example 3.6.

EXAMPLE 3.6 An Attribute Grammar for Simple Assignment Statements

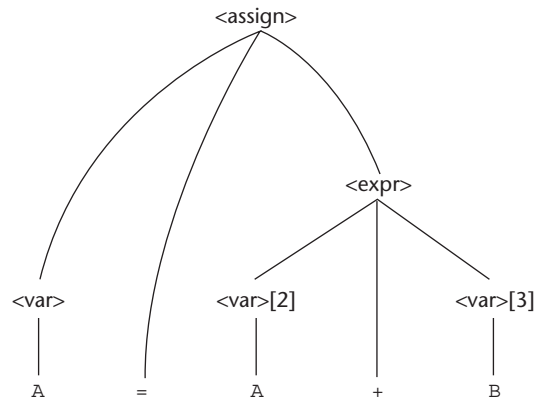
1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → `A | B | C`
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

The look-up function looks up a given variable name in the symbol table and returns the variable's type.

A parse tree of the sentence `A = A + B` generated by the grammar in Example 3.6 is shown in Figure 3.6. As in the grammar, bracketed numbers are added after the repeated node labels in the tree so they can be referenced unambiguously.

Figure 3.6

A parse tree for
`A = A + B`



3.4.6 Computing Attribute Values

Now, consider the process of computing the attribute values of a parse tree, which is sometimes called **decorating** the parse tree. If all attributes were inherited, this could proceed in a completely top-down order, from the root to the leaves. Alternatively, it could proceed in a completely bottom-up order, from the leaves to the root, if all the attributes were synthesized. Because our grammar has both synthesized and inherited attributes, the evaluation process cannot be in any single direction. The following is an evaluation of the attributes, in an order in which it is possible to compute them:

1. <var>.actual_type ← look-up(`A`) (Rule 4)
2. <expr>.expected_type ← <var>.actual_type (Rule 1)
3. <var>[2].actual_type ← look-up(`A`) (Rule 4)
   <var>[3].actual_type ← look-up(`B`) (Rule 4)
4. <expr>.actual_type ← either int or real (Rule 2)
5. <expr>.expected_type == <expr>.actual_type is either TRUE or FALSE (Rule 2)

The tree in Figure 3.7 shows the flow of attribute values in the example of Figure 3.6. Solid lines are used for the parse tree; dashed lines show attribute flow in the tree.

The tree in Figure 3.8 shows the final attribute values on the nodes. In this example, `A` is defined as a real and `B` is defined as an int.

Determining attribute evaluation order for the general case of an attribute grammar is a complex problem, requiring the construction of a dependency graph to show all attribute dependencies.

Figure 3.7

The flow of attributes in the tree

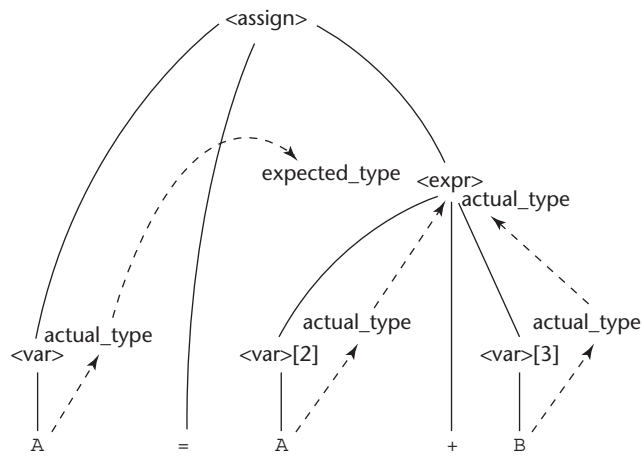
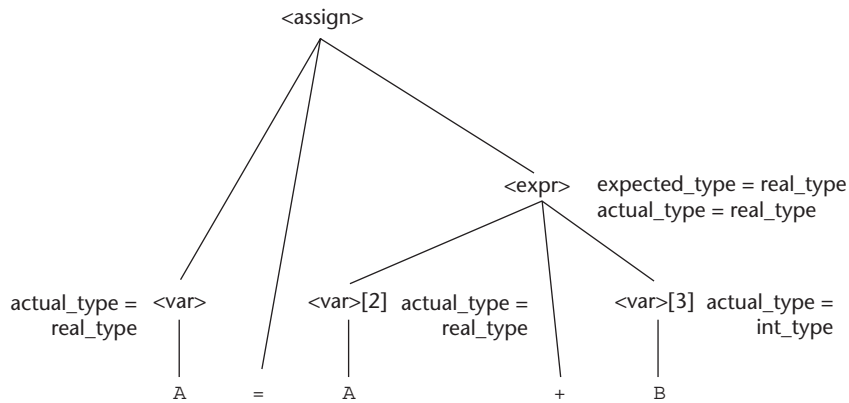


Figure 3.8

A fully attributed parse tree



3.4.7 Evaluation

Checking the static semantic rules of a language is an essential part of all compilers. Even if a compiler writer has never heard of an attribute grammar, he or she would need to use their fundamental ideas to design the checks of static semantics rules for his or her compiler.

One of the main difficulties in using an attribute grammar to describe all of the syntax and static semantics of a real contemporary programming language is the size and complexity of the attribute grammar. The large number of attributes and semantic rules required for a complete programming language make such grammars difficult to write and read. Furthermore, the attribute values on a large parse tree are costly to evaluate. On the other hand, less formal attribute

grammars are a powerful and commonly used tool for compiler writers, who are more interested in the process of producing a compiler than they are in formalism.

3.5 Describing the Meanings of Programs: Dynamic Semantics

We now turn to the difficult task of describing the **dynamic semantics**, or meaning, of the expressions, statements, and program units of a programming language. Because of the power and naturalness of the available notation, describing syntax is a relatively simple matter. On the other hand, no universally accepted notation or approach has been devised for dynamic semantics. In this section, we briefly describe several of the methods that have been developed. For the remainder of this section, when we use the term *semantics*, we mean dynamic semantics.

There are several different reasons underlying the need for a methodology and notation for describing semantics. Programmers obviously need to know precisely what the statements of a language do before they can use them effectively in their programs. Compiler writers must know exactly what language constructs mean to design implementations for them correctly. If there were a precise semantics specification of a programming language, programs written in the language potentially could be proven correct without testing. Also, compilers could be shown to produce programs that exhibited exactly the behavior given in the language definition; that is, their correctness could be verified. A complete specification of the syntax and semantics of a programming language could be used by a tool to generate a compiler for the language automatically. Finally, language designers, who would develop the semantic descriptions of their languages, could in the process discover ambiguities and inconsistencies in their designs.

Software developers and compiler designers typically determine the semantics of programming languages by reading English explanations in language manuals. Because such explanations are often imprecise and incomplete, this approach is clearly unsatisfactory. Due to the lack of complete semantics specifications of programming languages, programs are rarely proven correct without testing, and commercial compilers are never generated automatically from language descriptions.

Scheme, a functional language described in Chapter 15, is one of only a few programming languages whose definition includes a formal semantics description. However, the method used is not one described in this chapter, as this chapter is focused on approaches that are suitable for imperative languages.

3.5.1 Operational Semantics

The idea behind **operational semantics** is to describe the meaning of a statement or program by specifying the effects of running it on a machine. The effects on the machine are viewed as the sequence of changes in its

state, where the machine's state is the collection of the values in its storage. An obvious operational semantics description, then, is given by executing a compiled version of the program on a computer. Most programmers have, on at least one occasion, written a small test program to determine the meaning of some programming language construct, often while learning the language. Essentially, what such a programmer is doing is using operational semantics to determine the meaning of the construct.

There are several problems with using this approach for complete formal semantics descriptions. First, the individual steps in the execution of machine language and the resulting changes to the state of the machine are too small and too numerous. Second, the storage of a real computer is too large and complex. There are usually several levels of memory devices, as well as connections to enumerable other computers and memory devices through networks. Therefore, machine languages and real computers are not used for formal operational semantics. Rather, intermediate-level languages and interpreters for idealized computers are designed specifically for the process.

There are different levels of uses of operational semantics. At the highest level, the interest is in the final result of the execution of a complete program. This is sometimes called **natural operational semantics**. At the lowest level, operational semantics can be used to determine the precise meaning of a program through an examination of the complete sequence of state changes that occur when the program is executed. This use is sometimes called **structural operational semantics**.

3.5.1.1 The Basic Process

The first step in creating an operational semantics description of a language is to design an appropriate intermediate language, where the primary characteristic of the language is clarity. Every construct of the intermediate language must have an obvious and unambiguous meaning. This language is at the intermediate level, because machine language is too low-level to be easily understood and another high-level language is obviously not suitable. If the semantics description is to be used for natural operational semantics, a virtual machine (an interpreter) must be constructed for the intermediate language. The virtual machine can be used to execute either single statements, code segments, or whole programs. The semantics description can be used without a virtual machine if the meaning of a single statement is all that is required. In this use, which is structural operational semantics, the intermediate code can be visually inspected.

The basic process of operational semantics is not unusual. In fact, the concept is frequently used in programming textbooks and programming language reference manuals. For example, the semantics of the C `for` construct can be described in terms of simpler statements, as in

C Statement	*Meaning*

`for (expr1; expr2; expr3) {`
`  ...`
`}`
	`expr1;`
	`loop: if expr2 == 0 goto out`
	`...`
	`expr3;`
	`goto loop`
	`out: ...`

The human reader of such a description is the virtual computer and is assumed to be able to "execute" the instructions in the definition correctly and recognize the effects of the "execution."

The intermediate language and its associated virtual machine used for formal operational semantics descriptions are often highly abstract. The intermediate language is meant to be convenient for the virtual machine, rather than for human readers. For our purposes, however, a more human-oriented intermediate language could be used. As such an example, consider the following list of statements, which would be adequate for describing the semantics of the simple control statements of a typical programming language:

ident = var
ident = ident + 1
ident = ident – 1
goto label
if var relop var **goto** label

In these statements, relop is one of the relational operators from the set {=, <>, >, <, >=, <=}, ident is an identifier, and var is either an identifier or a constant. These statements are all simple and therefore easy to understand and implement.

A slight generalization of these three assignment statements allows more general arithmetic expressions and assignment statements to be described. The new statements are

ident = var bin_op var
ident = un_op var

where bin_op is a binary arithmetic operator and un_op is a unary operator. Multiple arithmetic data types and automatic type conversions, of course, complicate this generalization. Adding just a few more relatively simple instructions would allow the semantics of arrays, records, pointers, and subprograms to be described.

In Chapter 8, the semantics of various control statements are described using this intermediate language.

3.5.1.2 Evaluation

The first and most significant use of formal operational semantics was to describe the semantics of PL/I (Wegner, 1972). That particular abstract machine and the translation rules for PL/I were together named the Vienna Definition Language (VDL), after the city where IBM designed it.

Operational semantics provides an effective means of describing semantics for language users and language implementors, as long as the descriptions are kept simple and informal. The VDL description of PL/I, unfortunately, is so complex that it serves no practical purpose.

Operational semantics depends on programming languages of lower levels, not mathematics. The statements of one programming language are described in terms of the statements of a lower-level programming language. This approach can lead to circularities, in which concepts are indirectly defined in terms of themselves. The methods described in the following two sections are much more formal, in the sense that they are based on mathematics and logic, not programming languages.

3.5.2 Denotational Semantics

Denotational semantics is the most rigorous and most widely known formal method for describing the meaning of programs. It is solidly based on recursive function theory. A thorough discussion of the use of denotational semantics to describe the semantics of programming languages is necessarily long and complex. It is our intent to provide the reader with an introduction to the central concepts of denotational semantics, along with a few simple examples that are relevant to programming language specifications.

The process of constructing a denotational semantics specification for a programming language requires one to define for each language entity both a mathematical object and a function that maps instances of that language entity onto instances of the mathematical object. Because the objects are rigorously defined, they model the exact meaning of their corresponding entities. The idea is based on the fact that there are rigorous ways of manipulating mathematical objects but not programming language constructs. The difficulty with this method lies in creating the objects and the mapping functions. The method is named *denotational* because the mathematical objects denote the meaning of their corresponding syntactic entities.

The mapping functions of a denotational semantics programming language specification, like all functions in mathematics, have a domain and a range. The domain is the collection of values that are legitimate parameters to the function; the range is the collection of objects to which the parameters are mapped. In denotational semantics, the domain is called the **syntactic domain**, because it is syntactic structures that are mapped. The range is called the **semantic domain**.

Denotational semantics is related to operational semantics. In operational semantics, programming language constructs are translated into simpler programming language constructs, which become the basis of the meaning of the

construct. In denotational semantics, programming language constructs are mapped to mathematical objects, either sets or, more often, functions. However, unlike operational semantics, denotational semantics does not model the step-by-step computational processing of programs.

3.5.2.1 Two Simple Examples

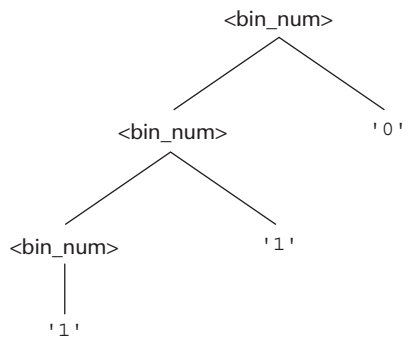
We use a very simple language construct, character string representations of binary numbers, to introduce the denotational method. The syntax of such binary numbers can be described by the following grammar rules:

```
<bin_num> → '0'
          | '1'
          | <bin_num> '0'
          | <bin_num> '1'
```

A parse tree for the example binary number, `110`, is shown in Figure 3.9. Notice that we put apostrophes around the syntactic digits to show they are not mathematical digits. This is similar to the relationship between ASCII coded digits and mathematical digits. When a program reads a number as a string, it must be converted to a mathematical number before it can be used as a value in the program.

Figure 3.9

A parse tree of the binary number `110`



The syntactic domain of the mapping function for binary numbers is the set of all character string representations of binary numbers. The semantic domain is the set of nonnegative decimal numbers, symbolized by N.

To describe the meaning of binary numbers using denotational semantics, we associate the actual meaning (a decimal number) with each rule that has a single terminal symbol as its RHS.

In our example, decimal numbers must be associated with the first two grammar rules. The other two grammar rules are, in a sense, computational rules, because they combine a terminal symbol, to which an object can be associated, with a nonterminal, which can be expected to represent some construct. Presuming an evaluation that progresses upward in the parse tree,

the nonterminal in the right side would already have its meaning attached. So, a syntax rule with a nonterminal as its RHS would require a function that computed the meaning of the LHS, which represents the meaning of the complete RHS.

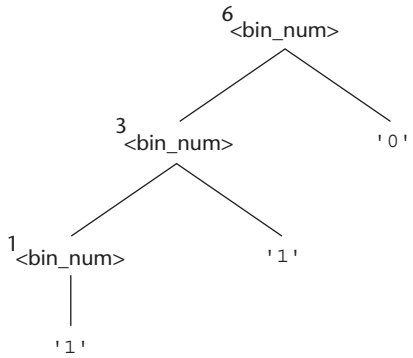
The semantic function, named M_{bin}, maps the syntactic objects, as described in the previous grammar rules, to the objects in N, the set of non-negative decimal numbers. The function M_{bin} is defined as follows:

M_{bin}('0') = 0
M_{bin}('1') = 1
M_{bin}(<bin_num> '0') = 2 * M_{bin}(<bin_num>)
M_{bin}(<bin_num> '1') = 2 * M_{bin}(<bin_num>) + 1

The meanings, or denoted objects (which in this case are decimal numbers), can be attached to the nodes of the parse tree shown on the previous page, yielding the tree in Figure 3.10. This is syntax-directed semantics. Syntactic entities are mapped to mathematical objects with concrete meaning.

Figure 3.10

A parse tree with denoted objects for 110



In part because we need it later, we now show a similar example for describing the meaning of syntactic decimal literals. In this case, the syntactic domain is the set of character string representations of decimal numbers. The semantic domain is once again the set N.

<dec_num> → '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7''8'|'9'
|<dec_num> ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')

The denotational mappings for these syntax rules are

M_{dec}('0') = 0, M_{dec}('1') = 1, M_{dec}('2') = 2, . . ., M_{dec}('9') = 9
M_{dec}(<dec_num> '0') = 10 * M_{dec}(<dec_num>)
M_{dec}(<dec_num> '1') = 10 * M_{dec}(<dec_num>) + 1
. . .
M_{dec}(<dec_num> '9') = 10 * M_{dec}(<dec_num>) + 9

In the following sections, we present the denotational semantics descriptions of a few simple constructs. The most important simplifying assumption made here is that both the syntax and static semantics of the constructs are correct. In addition, we assume that only two scalar types are included: integer and Boolean.

3.5.2.2 The State of a Program

The denotational semantics of a program could be defined in terms of state changes in an ideal computer. Operational semantics are defined in this way, and denotational semantics are defined in nearly the same way. In a further simplification, however, denotational semantics is defined in terms of only the values of all of the program's variables. So, denotational semantics uses the state of the program to describe meaning, whereas operational semantics uses the state of a machine. The key difference between operational semantics and denotational semantics is that state changes in operational semantics are defined by coded algorithms, written in some programming language, whereas in denotational semantics, state changes are defined by mathematical functions.

Let the state s of a program be represented as a set of ordered pairs, as follows:

$$s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$$

Each i is the name of a variable, and the associated v's are the current values of those variables. Any of the v's can have the special value **undef**, which indicates that its associated variable is currently undefined. Let VARMAP be a function of two parameters: a variable name and the program state. The value of VARMAP (i_j, s) is v_j (the value paired with i_j in state s). Most semantics mapping functions for programs and program constructs map states to states. These state changes are used to define the meanings of programs and program constructs. Some language constructs—for example, expressions—are mapped to values, not states.

3.5.2.3 Expressions

Expressions are fundamental to most programming languages. We assume here that expressions have no side effects. Furthermore, we deal with only very simple expressions: The only operators are + and *, and an expression can have at most one operator; the only operands are scalar integer variables and integer literals; there are no parentheses; and the value of an expression is an integer. Following is the BNF description of these expressions:

<expr> → <dec_num> | <var> | <binary_expr>
<binary_expr> → <left_expr> <operator> <right_expr>
<left_expr> → <dec_num> | <var>
<right_expr> → <dec_num> | <var>
<operator> → + | *

The only error we consider in expressions is a variable having an undefined value. Obviously, other errors can occur, but most of them are machine-dependent. Let Z be the set of integers, and let **error** be the error value. Then Z ∪ {**error**} is the semantic domain for the denotational specification for our expressions.

The mapping function for a given expression E and state s follows. To distinguish between mathematical function definitions and the assignment statements of programming languages, we use the symbol Δ= to define mathematical functions. The implication symbol, =>, used in this definition connects the form of an operand with its associated case (or switch) construct. Dot notation is used to refer to the child nodes of a node. For example, <binary_expr>.<left_expr> refers to the left child node of <binary_expr>.

```
Me(<expr>, s) Δ= case <expr> of
                  <dec_num>=>Mdec(<dec_num>, s)
                 <var> =>if VARMAP(<var>, s) == undef
                             then error
                             else VARMAP(<var>, s)
                 <binary_expr> =>
                  if(Me(<binary_expr>.<left_expr>,s) == undef OR
                     Me(<binary_expr>.<right_expr>, s) == undef)
                   then error
                   else if (<binary_expr>.<operator> == '+')
                           then Me(<binary_expr>.<left_expr>, s) +
                                  Me(<binary_expr>.<right_expr>, s)
                           else Me(<binary_expr>.<left_expr>, s) *
                                  Me(<binary_expr>.<right_expr>, s)
```

3.5.2.4 Assignment Statements

An assignment statement is an expression evaluation plus the setting of the target variable to the expression's value. In this case, the meaning function maps a state to a state. This function can be described with the following:

```
Ma(x = E, s) Δ= if Me(E, s) == error
                  then error
                  else s′ = {<i1, v1′>, <i2, v2′>, . . . , <in, vn′>}, where
                         for j = 1, 2, . . . , n
                           if ij == x
                             then vj′ = Me(E, s)
                             else vj′ = VARMAP(ij, s)
```

Note that the comparison in the third last line above, i_j == x, is of names, not values.

3.5.2.5 Logical Pretest Loops

The denotational semantics of a logical pretest loop is deceptively simple. To expedite the discussion, we assume that there are two other existing mapping functions, M_{sl} and M_b, that map statement lists and states to states and Boolean expressions to Boolean values (or **error**), respectively. The function is

$$
\begin{aligned}
M_l(\texttt{while}\ B\ \texttt{do}\ L, s)\ \Delta= &\ \text{if } M_b(B, s) == \textbf{undef} \\
&\ \text{then } \textbf{error} \\
&\ \text{else if } M_b(B, s) == \text{false} \\
&\quad \text{then } s \\
&\quad \text{else if } M_{sl}(L, s) == \textbf{error} \\
&\qquad \text{then } \textbf{error} \\
&\qquad \text{else } M_l(\texttt{while}\ B\ \texttt{do}\ L, M_{sl}(L, s))
\end{aligned}
$$

The meaning of the loop is simply the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors. In essence, the loop has been converted from iteration to recursion, where the recursion control is mathematically defined by other recursive state mapping functions. Recursion is easier to describe with mathematical rigor than iteration.

One significant observation at this point is that this definition, like actual program loops, may compute nothing because of nontermination.

3.5.2.6 Evaluation

Objects and functions, such as those used in the earlier constructs, can be defined for the other syntactic entities of programming languages. When a complete system has been defined for a given language, it can be used to determine the meaning of complete programs in that language. This provides a framework for thinking about programming in a highly rigorous way.

As stated previously, denotational semantics can be used as an aid to language design. For example, statements for which the denotational semantic description is complex and difficult may indicate to the designer that such statements may also be difficult for language users to understand and that an alternative design may be in order.

Because of the complexity of denotational descriptions, they are of little use to language users. On the other hand, they provide an excellent way to describe a language concisely.

Although the use of denotational semantics is normally attributed to Scott and Strachey (1971), the general denotational approach to language description can be traced to the nineteenth century (Frege, 1892).

3.5.3 Axiomatic Semantics

Axiomatic semantics, thus named because it is based on mathematical logic, is the most abstract approach to semantics specification discussed in this chapter. Rather than directly specifying the meaning of a program, axiomatic semantics specifies what can be proven about the program. Recall that one of the possible uses of semantic specifications is to prove the correctness of programs.

In axiomatic semantics, there is no model of the state of a machine or program or model of state changes that take place when the program is executed. The meaning of a program is based on relationships among program variables and constants, which are the same for every execution of the program.

Axiomatic semantics has two distinct applications: program verification and program semantics specification. This section focuses on program verification in its description of axiomatic semantics.

Axiomatic semantics was defined in conjunction with the development of an approach to proving the correctness of programs. Such correctness proofs, when they can be constructed, show that a program performs the computation described by its specification. In a proof, each statement of a program is both preceded and followed by a logical expression that specifies constraints on program variables. These, rather than the entire state of an abstract machine (as with operational semantics), are used to specify the meaning of the statement. The notation used to describe constraints—indeed, the language of axiomatic semantics—is predicate calculus. Although simple Boolean expressions are often adequate to express constraints, in some cases they are not.

When axiomatic semantics is used to specify formally the meaning of a statement, the meaning is defined by the statement's effect on assertions about the data affected by the statement.

3.5.3.1 Assertions

The logical expressions used in axiomatic semantics are called predicates, or **assertions**. An assertion immediately preceding a program statement describes the constraints on the program variables at that point in the program. An assertion immediately following a statement describes the new constraints on those variables (and possibly others) after execution of the statement. These assertions are called the **precondition** and **postcondition**, respectively, of the statement. For two adjacent statements, the postcondition of the first serves as the precondition of the second. Developing an axiomatic description or proof of a given program requires that every statement in the program has both a precondition and a postcondition.

In the following sections, we examine assertions from the point of view that preconditions for statements are computed from given postconditions, although it is possible to consider these in the opposite sense. We assume all variables are integer type. As a simple example, consider the following assignment statement and postcondition:

```
sum = 2 * x + 1 {sum > 1}
```

Precondition and postcondition assertions are presented in braces to distinguish them from parts of program statements. One possible precondition for this statement is {x > 10}.

In axiomatic semantics, the meaning of a specific statement is defined by its precondition and its postcondition. In effect, the two assertions specify precisely the effect of executing the statement.

In the following subsections, we focus on correctness proofs of statements and programs, which is a common use of axiomatic semantics. The more general concept of axiomatic semantics is to state precisely the meaning of statements and programs in terms of logic expressions. Program verification is one application of axiomatic descriptions of languages.

3.5.3.2 Weakest Preconditions

The **weakest precondition** is the least restrictive precondition that will guarantee the validity of the associated postcondition. For example, in the statement and postcondition given in Section 3.5.3.1, {x > 10}, {x > 50}, and {x > 1000} are all valid preconditions. The weakest of all preconditions in this case is {x > 0}.

If the weakest precondition can be computed from the most general postcondition for each of the statement types of a language, then the processes used to compute these preconditions provide a concise description of the semantics of that language. Furthermore, correctness proofs can be constructed for programs in that language. A program proof is begun by using the characteristics of the results of the program's execution as the postcondition of the last statement of the program. This postcondition, along with the last statement, is used to compute the weakest precondition for the last statement. This precondition is then used as the postcondition for the second last statement. This process continues until the beginning of the program is reached. At that point, the precondition of the first statement states the conditions under which the program will compute the desired results. If these conditions are implied by the input specification of the program, the program has been verified to be correct.

An **inference rule** is a method of inferring the truth of one assertion on the basis of the values of other assertions. The general form of an inference rule is as follows:

$$\frac{\text{S1, S2, } \ldots \text{ , S}n}{\text{S}}$$

This rule states that if S1, S2, . . . , and S*n* are true, then the truth of S can be inferred. The top part of an inference rule is called its **antecedent**; the bottom part is called its **consequent**.

An **axiom** is a logical statement that is assumed to be true. Therefore, an axiom is an inference rule without an antecedent.

For some program statements, the computation of a weakest precondition from the statement and a postcondition is simple and can be specified by an

axiom. In most cases, however, the weakest precondition can be specified only by an inference rule.

To use axiomatic semantics with a given programming language, whether for correctness proofs or for formal semantics specifications, either an axiom or an inference rule must exist for each kind of statement in the language. In the following subsections, we present an axiom for assignment statements and inference rules for statement sequences, selection statements, and logical pretest loop statements. Note that we assume that neither arithmetic nor Boolean expressions have side effects.

3.5.3.3 Assignment Statements

The precondition and postcondition of an assignment statement together define precisely its meaning. To define the meaning of an assignment statement, given a postcondition, there must be a way to compute its precondition from that postcondition.

Let x = E be a general assignment statement and Q be its postcondition. Then, its precondition, P, is defined by the axiom

$$P = Q_{x \rightarrow E}$$

which means that P is computed as Q with all instances of x replaced by E. For example, if we have the assignment statement and postcondition

```
a = b / 2 - 1 {a < 10}
```

the weakest precondition is computed by substituting `b / 2 - 1` for `a` in the postcondition {`a < 10`}, as follows:

```
b / 2 - 1 < 10
b < 22
```

Thus, the weakest precondition for the given assignment statement and postcondition is {`b < 22`}. Remember that the assignment axiom is guaranteed to be correct only in the absence of side effects. An assignment statement has a side effect if it changes some variable other than its target.

The usual notation for specifying the axiomatic semantics of a given statement form is

$$\{P\}S\{Q\}$$

where P is the precondition, Q is the postcondition, and S is the statement form. In the case of the assignment statement, the notation is

$$\{Q_{x \rightarrow E}\}\ x = E\{Q\}$$

As another example of computing a precondition for an assignment statement, consider the following:

```
x = 2 * y - 3 {x > 25}
```

The precondition is computed as follows:

```
2 * y - 3 > 25
y > 14
```

So {y > 14} is the weakest precondition for this assignment statement and postcondition.

Note that the appearance of the left side of the assignment statement in its right side does not affect the process of computing the weakest precondition. For example, for

```
x = x + y - 3 {x > 10}
```

the weakest precondition is

```
x + y - 3 > 10
y > 13 - x
```

Recall that axiomatic semantics was developed to prove the correctness of programs. In light of that, it is natural at this point to wonder how the axiom for assignment statements can be used to prove anything. Here is how: A given assignment statement with both a precondition and a postcondition can be considered a logical statement, or theorem. If the assignment axiom, when applied to the postcondition and the assignment statement, produces the given precondition, the theorem is proved. For example, consider the logical statement

```
{x > 3} x = x - 3 {x > 0}
```

Using the assignment axiom on

```
x = x - 3 {x > 0}
```

produces {x > 3}, which is the given precondition. Therefore, we have proven the example logical statement.

Next, consider the logical statement

```
{x > 5} x = x - 3 {x > 0}
```

In this case, the given precondition, {x > 5}, is not the same as the assertion produced by the axiom. However, it is obvious that {x > 5} implies {x > 3}.

To use this in a proof, an inference rule, named the **rule of consequence**, is needed. The form of the rule of consequence is

$$\frac{\{P\}\ S\ \{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

The => symbol means "implies," and S can be any program statement. The rule can be stated as follows: If the logical statement {P} S {Q} is true, the assertion P′ implies the assertion P, and the assertion Q implies the assertion Q′, then it can be inferred that {P′} S {Q′}. In other words, the rule of consequence says that a postcondition can always be weakened and a precondition can always be strengthened. This is quite useful in program proofs. For example, it allows the completion of the proof of the last logical statement example above. If we let P be {x > 3}, Q and Q′ be {x > 0}, and P′ be {x > 5}, we have

$$\frac{\{x>3\}x = x-3\{x>0\},\ (x>5) \Rightarrow \{x>3\},\ (x>0) \Rightarrow (x>0)}{\{x>5\}x = x-3\{x>0\}}$$

The first term of the antecedent ({x > 3} x = x − 3 {x > 0}) was proven with the assignment axiom. The second and third terms are obvious. Therefore, by the rule of consequence, the consequent is true.

3.5.3.4 Sequences

The weakest precondition for a sequence of statements cannot be described by an axiom, because the precondition depends on the particular kinds of statements in the sequence. In this case, the precondition can only be described with an inference rule. Let S1 and S2 be adjacent program statements. If S1 and S2 have the following pre- and postconditions

{P1} S1 {P2}
{P2} S2 {P3}

the inference rule for such a two-statement sequence is

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1, S2\ \{P3\}}$$

So, for our example, {P1} S1; S2 {P3} describes the axiomatic semantics of the sequence S1; S2. The inference rule states that to get the sequence precondition, the precondition of the second statement is computed. This new assertion is then used as the postcondition of the first statement, which can then be used to compute the precondition of the first statement, which is also the precondition of the whole sequence. If S1 and S2 are the assignment statements

x1 = E1

and

x2 = E2

then we have

$\{P3_{x2 \rightarrow E2}\}$ x2 = E2 $\{P3\}$
$\{(P3_{x2 \rightarrow E2})_{x1 \rightarrow E1}\}$ x1 = E1 $\{P3_{x2 \rightarrow E2}\}$

Therefore, the weakest precondition for the sequence x1 = E1; x2 = E2 with postcondition P3 is $\{(P3_{x2 \rightarrow E2})_{x1 \rightarrow E1}\}$.

For example, consider the following sequence and postcondition:

```
y = 3 * x + 1;
x = y + 3;
{x < 10}
```

The precondition for the second assignment statement is

```
y < 7
```

which is used as the postcondition for the first statement. The precondition for the first assignment statement can now be computed:

```
3 * x + 1 < 7
x < 2
```

So, `{x < 2}` is the precondition of both the first statement and the two-statement sequence.

3.5.3.5 Selection

We next consider the inference rule for selection statements, the general form of which is

`if` B **`then`** S1 **`else`** S2

We consider only selections that include **`else`** clauses. The inference rule is

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$$

This rule indicates that selection statements must be proven both when the Boolean control expression is true and when it is false. The first logical statement above the line represents the **`then`** clause; the second represents the **`else`**

clause. According to the inference rule, we need a precondition P that can be used in the precondition of both the **then** and **else** clauses.

Consider the following example of the computation of the precondition using the selection inference rule. The example selection statement is

```
if x > 0 then
  y = y - 1
else
  y = y + 1
```

Suppose the postcondition, Q, for this selection statement is {y > 0}. We can use the axiom for assignment on the **then** clause

```
y = y - 1 {y > 0}
```

This produces {y - 1 > 0} or {y > 1}. It can be used as the P part of the precondition for the **then** clause. Now we apply the same axiom to the **else** clause

```
y = y + 1 {y > 0}
```

which produces the precondition {y + 1 > 0} or {y > -1}. Because {y > 1} => {y > -1}, the rule of consequence allows us to use {y > 1} for the precondition of the whole selection statement.

history note

A significant amount of work has been done on the possibility of using denotational language descriptions to generate compilers automatically (Jones, 1980; Milos et al., 1984; Bodwin et al., 1982). These efforts have shown that the method is feasible, but the work has never progressed to the point where it can be used to generate useful compilers.

3.5.3.6 Logical Pretest Loops

Another essential construct of imperative programming languages is the logical pretest, or **while** loop. Computing the weakest precondition for a **while** loop is inherently more difficult than for a sequence, because the number of iterations cannot always be predetermined. In a case where the number of iterations is known, the loop can be unrolled and treated as a sequence.

The problem of computing the weakest precondition for loops is similar to the problem of proving a theorem about all positive integers. In the latter case, induction is normally used, and the same inductive method can be used for some loops. The principal step in induction is finding an inductive hypothesis. The corresponding step in the axiomatic semantics of a **while** loop is finding an assertion called a **loop invariant**, which is crucial to finding the weakest precondition.

The inference rule for computing the precondition for a **while** loop is

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end}\ \{I \text{ and (not } B)\}}$$

where I is the loop invariant. This seems simple, but it is not. The complexity lies in finding an appropriate loop invariant.

The axiomatic description of a **while** loop is written as

{P} **while** B **do** S **end** {Q}

The loop invariant must satisfy a number of requirements to be useful. First, the weakest precondition for the **while** loop must guarantee the truth of the loop invariant. In turn, the loop invariant must guarantee the truth of the postcondition upon loop termination. These constraints move us from the inference rule to the axiomatic description. During execution of the loop, the truth of the loop invariant must be unaffected by the evaluation of the loop-controlling Boolean expression and the loop body statements. Hence, the name *invariant*.

Another complicating factor for **while** loops is the question of loop termination. A loop that does not terminate cannot be correct, and in fact computes nothing. If Q is the postcondition that holds immediately after loop exit, then a precondition P for the loop is one that guarantees Q at loop exit and also guarantees that the loop terminates.

The complete axiomatic description of a **while** construct requires all of the following to be true, in which I is the loop invariant:

P => I
{I and B} S {I}
(I and (not B)) => Q
the loop terminates

If a loop computes a sequence of numeric values, it may be possible to find a loop invariant using an approach that is used for determining the inductive hypothesis when mathematical induction is used to prove a statement about a mathematical sequence. The relationship between the number of iterations and the precondition for the loop body is computed for a few cases, with the hope that a pattern emerges that will apply to the general case. It is helpful to treat the process of producing a weakest precondition as a function, wp. In general

wp(statement, postcondition) = precondition

A wp function is often called a **predicate transformer**, because it takes a predicate, or assertion, as a parameter and returns another predicate.

To find I, the loop postcondition Q is used to compute preconditions for several different numbers of iterations of the loop body, starting with none. If the loop body contains a single assignment statement, the axiom for assignment statements can be used to compute these cases. Consider the example loop:

```
while y <> x do y = y + 1 end {y = x}
```

Remember that the equal sign is being used for two different purposes here. In assertions, it means mathematical equality; outside assertions, it means the assignment operator.

For zero iterations, the weakest precondition is, obviously,

```
{y = x}
```

For one iteration, it is

wp(y = y + 1, {y = x}) = {y + 1 = x}, or {y = x - 1}

For two iterations, it is

wp(y = y + 1, {y = x - 1})={y + 1 = x - 1}, or {y = x - 2}

For three iterations, it is

wp(y = y + 1, {y = x - 2})={y + 1 = x - 2}, or {y = x – 3}

It is now obvious that {y < x} will suffice for cases of one or more iterations. Combining this with {y = x} for the zero iterations case, we get {y <= x}, which can be used for the loop invariant. A precondition for the **while** statement can be determined from the loop invariant. In fact, I can be used as the precondition, P.

We must ensure that our choice satisfies the four criteria for I for our example loop. First, because P = I, P => I. The second requirement is that it must be true that

{I and B} S {I}

In our example, we have

```
{y <= x and y <> x} y = y + 1 {y <= x}
```

Applying the assignment axiom to

```
y = y + 1 {y <= x}
```

we get {y + 1 <= x}, which is equivalent to {y < x}, which is implied by {y <= x and y <> x}. So, the earlier statement is proven.

Next, we must have

{I and (not B)} => Q

In our example, we have

```
{(y <= x) and not (y <> x)} => {y = x}
{(y <= x) and (y = x)} => {y = x}
{y = x} => {y = x}
```

So, this is obviously true. Next, loop termination must be considered. In this example, the question is whether the loop

```
{y <= x} while y <> x do y = y + 1 end {y = x}
```

terminates. Recalling that x and y are assumed to be integer variables, it is easy to see that this loop does terminate. The precondition guarantees that y initially is not larger than x. The loop body increments y with each iteration, until y is equal to x. No matter how much smaller y is than x initially, it will eventually become equal to x. So the loop will terminate. Because our choice of I satisfies all four criteria, it is a satisfactory loop invariant and loop precondition.

The previous process used to compute the invariant for a loop does not always produce an assertion that is the weakest precondition (although it does in the example).

As another example of finding a loop invariant using the approach used in mathematical induction, consider the following loop statement:

```
while s > 1 do s = s / 2 end {s = 1}
```

As before, we use the assignment axiom to try to find a loop invariant and a precondition for the loop. For zero iterations, the weakest precondition is {s = 1}. For one iteration, it is

wp(s = s / 2, {s = 1}) = {s / 2 = 1}, or {s = 2}

For two iterations, it is

wp(s = s / 2, {s = 2}) = {s / 2 = 2}, or {s = 4}

For three iterations, it is

wp(s = s / 2, {s = 4}) = {s / 2 = 4}, or {s = 8}

From these cases, we can see clearly that the invariant is

{s is a nonnegative power of 2}

Once again, the computed I can serve as P, and I passes the four requirements. Unlike our earlier example of finding a loop precondition, this one clearly is not a weakest precondition. Consider using the precondition {s > 1}. The logical statement

```
{s > 1} while s > 1 do s = s / 2 end {s = 1}
```

can easily be proven, and this precondition is significantly broader than the one computed earlier. The loop and precondition are satisfied for any positive value for s, not just powers of 2, as the process indicates. Because of the rule of consequence, using a precondition that is stronger than the weakest precondition does not invalidate a proof.

Finding loop invariants is not always easy. It is helpful to understand the nature of these invariants. First, a loop invariant is a weakened version of the loop postcondition and also a precondition for the loop. So, I must be weak enough to be satisfied prior to the beginning of loop execution, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition.

Because of the difficulty of proving loop termination, that requirement is often ignored. If loop termination can be shown, the axiomatic description of the loop is called **total correctness**. If the other conditions can be met but termination is not guaranteed, it is called **partial correctness**.

In more complex loops, finding a suitable loop invariant, even for partial correctness, requires a good deal of ingenuity. Because computing the precondition for a **`while`** loop depends on finding a loop invariant, proving the correctness of programs with **`while`** loops using axiomatic semantics can be difficult.

3.5.3.7 Program Proofs

This section provides validations for two simple programs. The first example of a correctness proof is for a very short program, consisting of a sequence of three assignment statements that interchange the values of two variables.

```
{x = A AND y = B}
t = x;
x = y;
y = t;
{x = B AND y = A}
```

Because the program consists entirely of assignment statements in a sequence, the assignment axiom and the inference rule for sequences can be used to prove its correctness. The first step is to use the assignment axiom on the last statement and the postcondition for the whole program. This yields the precondition

```
{x = B AND t = A}
```

Next, we use this new precondition as a postcondition on the middle statement and compute its precondition, which is

```
{y = B AND t = A}
```

Next, we use this new assertion as the postcondition on the first statement and apply the assignment axiom, which yields

```
{y = B AND x = A}
```

which is the same as the precondition on the program, except for the order of operands on the `AND` operator. Because `AND` is a symmetric operator, our proof is complete.

The following example is a proof of correctness of a pseudocode program that computes the factorial function.

```
{n >= 0}
count = n;
fact = 1;
while count <> 0 do
    fact = fact * count;
    count = count - 1;
end
{fact = n!}
```

The method described earlier for finding the loop invariant does not work for the loop in this example. Some ingenuity is required here, which can be aided by a brief study of the code. The loop computes the factorial function in order of the last multiplication first; that is, `(n - 1) * n` is done first, assuming `n` is greater than `1`. So, part of the invariant can be

```
fact = (count + 1) * (count + 2) *...* (n - 1) * n
```

But we must also ensure that `count` is always nonnegative, which we can do by adding that to the assertion above, to get

```
I = (fact = (count + 1) *...* n) AND (count >= 0)
```

Next, we must confirm that this I meets the requirements for invariants. Once again we let I also be used for P, so P clearly implies I. The next question is

{I and B} S {I}

I and B is

```
((fact = (count + 1) *...* n) AND (count >= 0)) AND
   (count <> 0)
```

which reduces to

```
(fact = (count + 1) *...* n) AND (count > 0)
```

In our case, we must compute the precondition of the body of the loop, using the invariant for the postcondition. For

{P} `count = count - 1` {I}

we compute P to be

```
{(fact = count * (count + 1) *...* n) AND
    (count >= 1)}
```

Using this as the postcondition for the first assignment in the loop body,

```
{P} fact = fact * count {(fact = count * (count + 1)
                          *...* n) AND (count >= 1)}
```

In this case, P is

```
{(fact = (count + 1) *...* n) AND (count >= 1)}
```

It is clear that I and B implies this P, so by the rule of consequence,

{I AND B} S {I}

is true. Finally, the last test of I is

I AND (NOT B) => Q

For our example, this is

```
((fact = (count + 1) *...* n) AND (count >= 0)) AND
   (count = 0)) => fact = n!
```

This is clearly true, for when `count = 0`, the first part is precisely the definition of factorial. So, our choice of I meets the requirements for a loop invariant. Now we can use our P (which is the same as I) from the **`while`** as the postcondition on the second assignment of the program

```
{P} fact = 1 {(fact = (count + 1) *...* n) AND
    (count >= 0)}
```

which yields for P

```
(1 = (count + 1) *...* n) AND (count >= 0))
```

Using this as the postcondition for the first assignment in the code

```
{P} count = n {(1 = (count + 1) *...* n) AND
    (count >= 0))}
```

produces for P

```
{(n + 1) *...* n = 1) AND (n >= 0)}
```

The left operand of the AND operator is true (because `1 = 1`) and the right operand is exactly the precondition of the whole code segment, {`n >= 0`}. Therefore, the program has been proven to be correct.

3.5.3.8 Evaluation

As stated previously, to define the semantics of a complete programming language using the axiomatic method, there must be an axiom or an inference rule for each statement type in the language. Defining axioms or inference rules for

some of the statements of programming languages has proven to be a difficult task. An obvious solution to this problem is to design the language with the axiomatic method in mind, so that only statements for which axioms or inference rules can be written are included. Unfortunately, such a language would necessarily leave out some useful and powerful parts.

Axiomatic semantics is a powerful tool for research into program correctness proofs, and it provides an excellent framework in which to reason about programs, both during their construction and later. Its usefulness in describing the meaning of programming languages to language users and compiler writers is, however, highly limited.

SUMMARY

Backus-Naur Form and context-free grammars are equivalent metalanguages that are well suited for the task of describing the syntax of programming languages. Not only are they concise descriptive tools, but also the parse trees that can be associated with their generative actions give graphical evidence of the underlying syntactic structures. Furthermore, they are naturally related to recognition devices for the languages they generate, which leads to the relatively easy construction of syntax analyzers for compilers for these languages.

An attribute grammar is a descriptive formalism that can describe both the syntax and static semantics of a language. Attribute grammars are extensions to context-free grammars. An attribute grammar consists of a grammar, a set of attributes, a set of attribute computation functions, and a set of predicates, which together describe static semantics rules.

This chapter provides a brief introduction to three methods of semantic description: operational, denotational, and axiomatic. Operational semantics is a method of describing the meaning of language constructs in terms of their effects on an ideal machine. In denotational semantics, mathematical objects are used to represent the meanings of language constructs. Language entities are converted to these mathematical objects with recursive functions. Axiomatic semantics, which is based on formal logic, was devised as a tool for proving the correctness of programs.

BIBLIOGRAPHIC NOTES

Syntax description using context-free grammars and BNF are thoroughly discussed in Cleaveland and Uzgalis (1976).

Research in axiomatic semantics was begun by Floyd (1967) and further developed by Hoare (1969). The semantics of a large part of Pascal was described by Hoare and Wirth (1973) using this method. The parts they did not complete involved functional side effects and goto statements. These were found to be the most difficult to describe.