In a unit of the form [{---}], the braces may be omitted without ambiguity. The following defines the syntax of a PL/I loop specification:

specification:

expression $\left[\begin{matrix}\text{TO expression [BY expression]}\\ \text{BY expression [TO expression]}\end{matrix}\right]$ [WHILE (expression)]

An equivalent set of pure BNF rules would be:

⟨specification⟩ ::= ⟨expression⟩ | ⟨expression⟩ ⟨rest⟩

⟨rest⟩ ::= ⟨to and by part⟩ | ⟨while part⟩ |
⟨to and by part⟩ ⟨while part⟩

⟨to and by part⟩ ::= ⟨to part⟩ | ⟨to part⟩ ⟨by part⟩ |
⟨by part⟩ | ⟨by part⟩ ⟨to part⟩

⟨to part⟩ ::= TO ⟨expression⟩

⟨by part⟩ ::= BY ⟨expression⟩

⟨while part⟩ ::= WHILE (⟨expression⟩)

EXERCISE

1. Translate each of the following into pure BNF:
   (a) ⟨Fortran DO statement⟩ ::=

   [⟨label⟩] DO ⟨label⟩ ⟨variable⟩ = ⟨initial⟩ ,
   ⟨limit⟩ [, ⟨increment⟩]

   (b) ⟨Pascal var decl⟩ ::= var ⟨var decl⟩ [; ⟨var decl⟩] ;
   ⟨var decl⟩ ::= ⟨identifier⟩ [, ⟨identifier⟩] : ⟨type⟩
   (c) PL/I block:

   [label :] ... BEGIN ; $\left\{\begin{matrix}\text{declaration}\\ \text{command}\end{matrix}\right\}$ ... END [label] ;

For Further Information

Pascal (Jensen and Wirth, 1974) is a well-known example of a language which was specified with the aid of BNF extended with braces as metasymbols. The two-dimensional notation is briefly described in the book by Cleaveland and Uzgalis (1977, pp. 35–38), as well as in many other places, and is given a formal treatment by Rochester (1966). Most manuals and other reference works on Cobol and PL/I [illegible] this notation. Other variations on BNF are usually explained wherever they

2.3 ATTRIBUTE GRAMMARS

2.3.1 Concepts and Characteristics

The specification technique described in this section may be used to formalize not only the context-free aspects of the syntax of a subject language but also the context-sensitive aspects. It is thus inherently more powerful than BNF or any of its variants described in the last section. A language specification constructed using this technique is called an *attribute grammar*. After introducing the basic tools and concepts, we shall illustrate the use of the approach by constructing an attribute grammar which constitutes a complete, formal definition of the syntax of Eva.

Basically, an attribute grammar is a context-free grammar augmented with certain formal devices ("attributes," "evaluation rules," and "conditions") that enable the non-context-free aspects to be specified by means of a powerful and elegant mechanism. In this book, we shall always employ the standard, unextended BNF notation for the context-free component of an attribute grammar.

With each distinct symbol of the context-free grammar, there is associated a finite set of *attributes*, which, notationally, are just names. (We adopt the convention that words with the first letter capitalized are attribute names.) With each distinct attribute, moreover, there is associated a domain of *values*. A given attribute may be associated with any number of grammatical symbols. We may regard each node of the syntax tree of a valid program as being labeled not only by a grammatical symbol but also by a set of attribute-value pairs, one for each attribute associated with the symbol, and possibly by a logical *condition* expressing a constraint that must be satisfied by the attribute values involved. The value associated with an attribute occurrence in the tree is determined by various *evaluation rules* associated with the grammar's production rules. Obviously, an example is needed to clarify these points.

Suppose that we are defining a machine-dependent dialect of PL/I for use on a computer with 32-bit words and that, as a consequence, an unsigned integer constant is to be considered syntactically invalid if its value exceeds $2^{31} - 1$ (= 2,147,483,647). While the set of *all* unsigned numerals is easily defined by the production rules

⟨numeral⟩ ::= ⟨digit⟩ | ⟨numeral⟩ ⟨digit⟩

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

the set of numerals {0, 1, [illegible]}

In a unit of the form [{---}], the braces may be omitted without ambiguity. The following defines the syntax of a PL/I loop specification:

specification:

expression [TO expression [BY expression] / BY expression [TO expression]] [WHILE (expression)]

An equivalent set of pure BNF rules would be:

⟨specification⟩ ::= ⟨expression⟩ | ⟨expression⟩ ⟨rest⟩
⟨rest⟩ ::= ⟨to and by part⟩ | ⟨while part⟩ |
⟨to and by part⟩ ⟨while part⟩
⟨to and by part⟩ ::= ⟨to part⟩ | ⟨to part⟩ ⟨by part⟩ |
⟨by part⟩ | ⟨by part⟩ ⟨to part⟩
⟨to part⟩ ::= TO ⟨expression⟩
⟨by part⟩ ::= BY ⟨expression⟩
⟨while part⟩ ::= WHILE (⟨expression⟩)

EXERCISE

1. Translate each of the following into pure BNF:
   (a) ⟨Fortran DO statement⟩ ::=
   [⟨label⟩] DO ⟨label⟩ ⟨variable⟩ = ⟨initial⟩ ,
   ⟨limit⟩ [, ⟨increment⟩]
   (b) ⟨Pascal var decl⟩ ::= var ⟨var decl⟩ {; ⟨var decl⟩} ;
   ⟨var decl⟩ ::= ⟨identifier⟩ [, ⟨identifier⟩} : ⟨type⟩
   (c) PL/I block:

[label :] ... BEGIN ; { declaration / command } ... END [label] ;

For Further Information

Pascal (Jensen and Wirth, 1974) is a well-known example of a language which was specified with the aid of BNF extended with braces as metasymbols. The two-dimensional notation is briefly described in the book by Cleaveland and Uzgalis (1977, pp. 35–38), as well as in many other places, and is given a formal treatment by Rochester (1966). Most manuals and other reference works on Cobol and PL/I employ this notation. Other variations on BNF are usually explained wherever they are used.

2.3 ATTRIBUTE GRAMMARS

2.3.1 Concepts and Characteristics

The specification technique described in this section may be used to formalize not only the context-free aspects of the syntax of a subject language but also the context-sensitive aspects. It is thus inherently more powerful than BNF or any of its variants described in the last section. A language specification constructed using this technique is called an *attribute grammar*. After introducing the basic tools and concepts, we shall illustrate the use of the approach by constructing an attribute grammar which constitutes a complete, formal definition of the syntax of Eva.

Basically, an attribute grammar is a context-free grammar augmented with certain formal devices ("attributes," "evaluation rules," and "conditions") that enable the non-context-free aspects to be specified by means of a powerful and elegant mechanism. In this book, we shall always employ the standard, unextended BNF notation for the context-free component of an attribute grammar.

With each distinct symbol of the context-free grammar, there is associated a finite set of *attributes*, which, notationally, are just names. (We adopt the convention that words with the first letter capitalized are attribute names.) With each distinct attribute, moreover, there is associated a domain of *values*. A given attribute may be associated with any number of grammatical symbols. We may regard each node of the syntax tree of a valid program as being labeled not only by a grammatical symbol but also by a set of attribute-value pairs, one for each attribute associated with the symbol, and possibly by a logical *condition* expressing a constraint that must be satisfied by the attribute values involved. The value associated with an attribute occurrence in the tree is determined by various *evaluation rules* associated with the grammar's production rules. Obviously, an example is needed to clarify these points.

Suppose that we are defining a machine-dependent dialect of PL/I for use on a computer with 32-bit words and that, as a consequence, an unsigned integer constant is to be considered syntactically invalid if its value exceeds $2^{31} - 1$ (= 2,147,483,647). While the set of *all* unsigned numerals is easily defined by the production rules

⟨numeral⟩ ::= ⟨digit⟩ | ⟨numeral⟩ ⟨digit⟩
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

the set of numerals {0, 1, . . . , 2147483646, 2147483647} is difficult to define concisely in BNF or its variants (try it!). Instead, we associate an attribute

Val, corresponding to the domain of integers, with both of the symbols ⟨numeral⟩ and ⟨digit⟩ and write the following specifications:

⟨numeral⟩ ::= ⟨digit⟩
Val(⟨numeral⟩) ← Val(⟨digit⟩)
| ⟨numeral⟩$_2$ ⟨digit⟩
Val(⟨numeral⟩) ← 10 × Val(⟨numeral⟩$_2$) + Val(⟨digit⟩)
Condition: Val(⟨numeral⟩) ≤ 2,147,483,647
⟨digit⟩ ::= 0
Val(⟨digit⟩) ← 0
| . . .
. . .
| 9
Val(⟨digit⟩) ← 9

The notation Val(S), where S is a grammatical symbol, may be read as 'the value of the attribute Val for this occurrence of S', and a line of the form

$$\text{Val(S)} \leftarrow \text{E}$$

where E may be any integer-valued expression, is an evaluation rule that "assigns" a value to Val(S). Each alternative definition in a production rule has an associated set of evaluation rules and possibly a condition. Observe the layout conventions used—each evaluation rule or condition refers to the alternative above it. The only purpose of the subscript in the symbol ⟨numeral⟩$_2$ is to distinguish the two occurrences of ⟨numeral⟩ in the specifications.

To see how the rules work for a legal example, the basic form of the syntax tree for the terminal string 909 is that shown in Fig. 2.9. Note that we have written the attribute Val under each occurrence of a nonterminal symbol and the word Cond (for 'condition') at each fork corresponding to the second alternative of the first production rule (⟨numeral⟩ ::= ⟨numeral⟩$_2$ ⟨digit⟩). Now, using the evaluation rules associated with the production rules that were used to build the tree, we must fill in the attribute values and check that all the conditions are satisfied. We are allowed to do this in any order permitted by the evaluation rules. (The resulting attribute values should be the same regardless of the order of computation. If this is not the case, then the grammar is not well formed.) The evaluation rule

$$\text{Val(⟨numeral⟩)} \leftarrow 10 \times \text{Val(⟨numeral⟩}_2\text{)} + \text{Val(⟨digit⟩)}$$

implies that the Val value at the top node cannot be filled in until we know

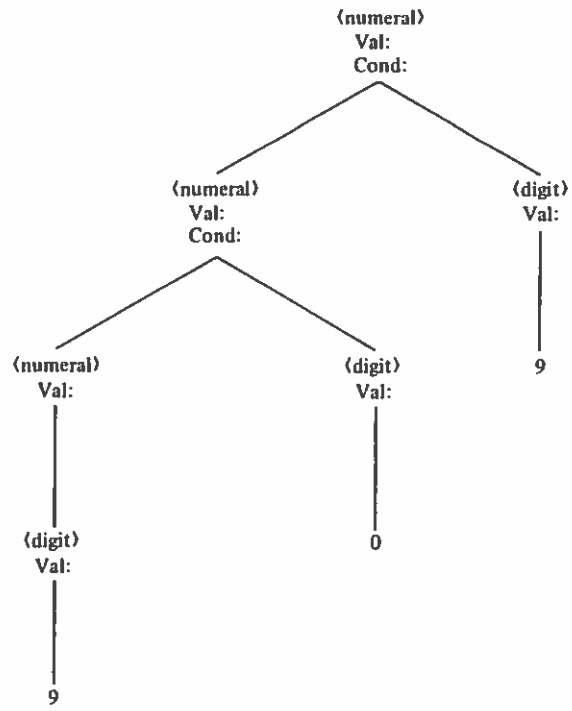


FIGURE 2.9

the values at its two offspring. Similarly, the values at the other two ⟨numeral⟩ nodes cannot be filled in until the values at their offspring are known. However, using the evaluation rules

$$\text{Val(⟨digit⟩)} \leftarrow 0$$

and

$$\text{Val(⟨digit⟩)} \leftarrow 9$$

the values at the three ⟨digit⟩ nodes can be filled (Fig. 2.10). Now, using the rule

$$\text{Val(⟨numeral⟩)} \leftarrow \text{Val(⟨digit⟩)}$$

we have Fig. 2.11. Next, the rule

$$\text{Val(⟨numeral⟩)} \leftarrow 10 \times \text{Val(⟨numeral⟩}_2\text{)} + \text{Val(⟨digit⟩)}$$

is used, resulting in Fig. 2.12, and the condition $90 \leq 2{,}147{,}483{,}647$ is seen

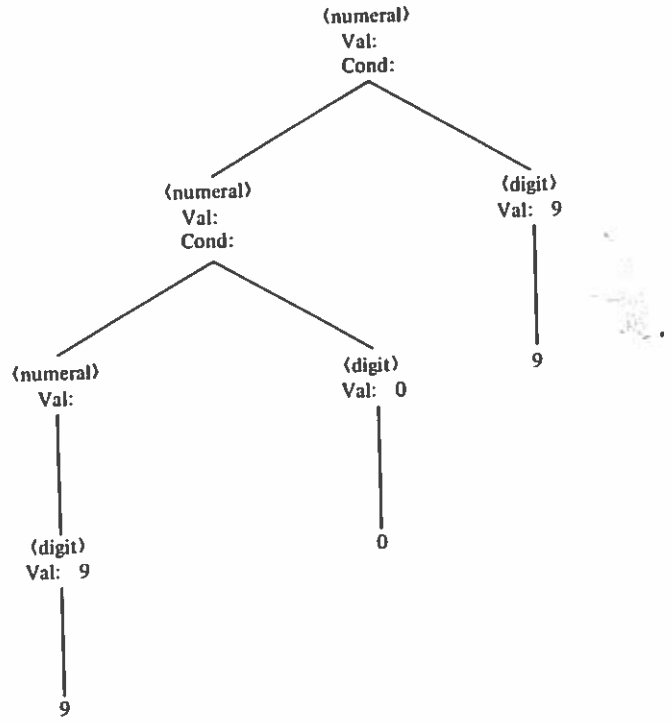


FIGURE 2.10

to be true. Repeating this process, the final tree is constructed as in Fig. 2.13. It should be clear that the tree for a numeral whose value exceeds 2,147,483,647 would contain a false condition, thus ruling out that string as one which is generated by the attribute grammar.

In this example, we have seen that the attribute value at each node in the tree is obtained from the values at the offspring of the node, so that, in a sense, we have information moving upward through the tree from the leaves to the root. Because of this, Val is said to be a *synthesized* attribute of ⟨numeral⟩ and of ⟨digit⟩. It is also possible for an attribute value at a node labeled by a symbol S to be obtained from the node's parent; the attribute is then said to be an *inherited* attribute of S. In general, a given grammatical symbol may have both synthesized and inherited attributes, and a given attribute may be synthesized with respect to one symbol and inherited with respect to another.

As a small example involving the use of an inherited attribute, consider the "Hollerith literals" of Fortran. These are essentially string constants,

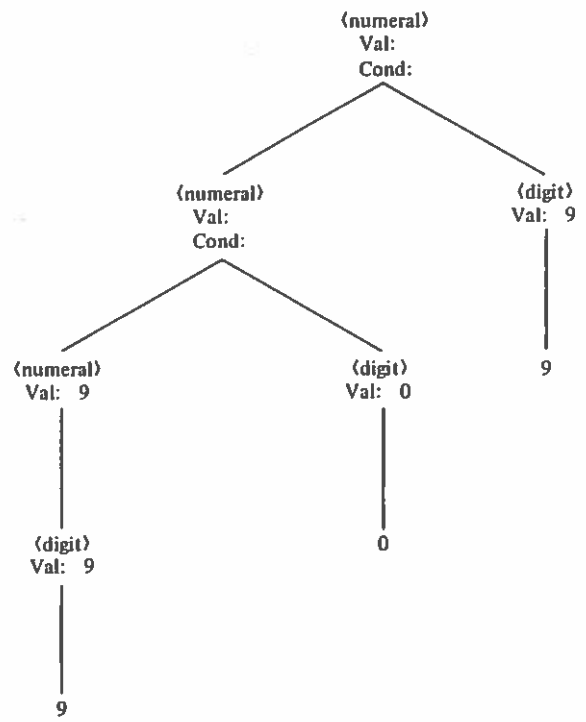


FIGURE 2.11

where the actual characters in the string are preceded by the letter H, which is itself preceded by an integer constant giving the length of the string. The following are some examples: 1HA, 6HSTRING, 15HA LONGER STRING. The number of characters following the H must be equal to the value of the integer constant preceding it. Clearly, the existence of an attribute grammar for this set of strings will demonstrate the superiority of this formalism over BNF.

We make use of two attributes, Val and Size, both corresponding to the domain of positive integer values. Val is a synthesized attribute of ⟨numeral⟩ and ⟨digit⟩, as in the previous example:

⟨numeral⟩ ::= ⟨digit⟩

Val(⟨numeral⟩) ⟵ Val(⟨digit⟩)

| ⟨numeral⟩$_2$ ⟨digit⟩

Val(⟨numeral⟩) ⟵ 10 × Val(⟨numeral⟩$_2$) + Val(⟨digit⟩)

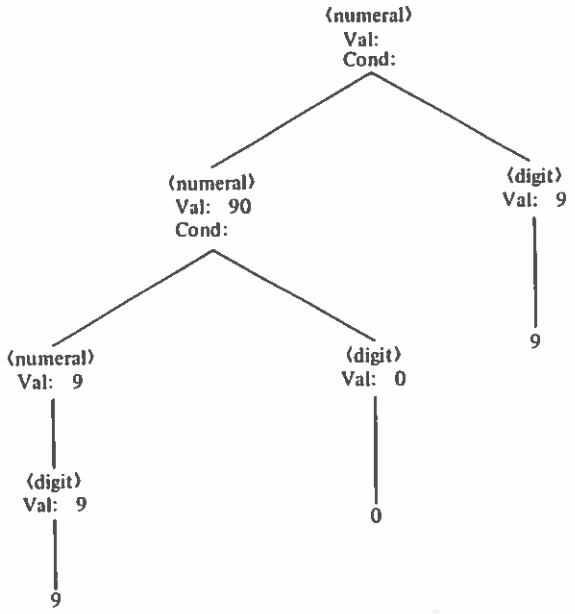


FIGURE 2.12

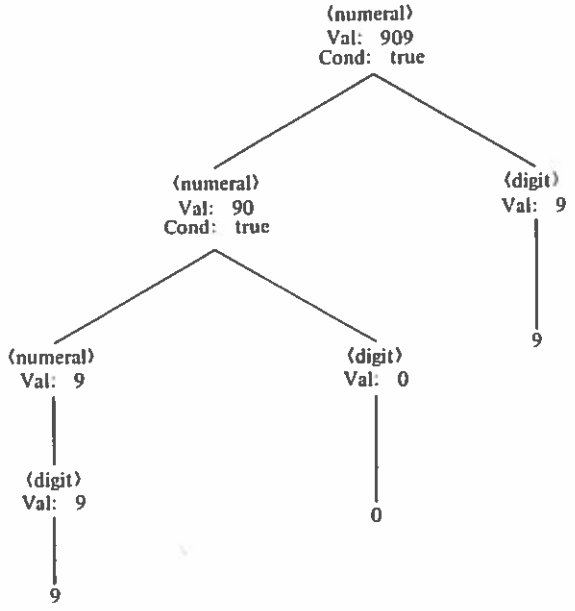


FIGURE 2.13

⟨digit⟩ ::= 0
Val(⟨digit⟩) ← 0
| . . .
. . .
| 9
Val(⟨digit⟩) ← 9

Size is an inherited attribute of the symbol ⟨string⟩, which is defined as follows:

⟨string⟩ ::= ⟨char⟩
Condition: Size(⟨string⟩) = 1
| ⟨string⟩$_2$ ⟨char⟩
Size(⟨string⟩$_2$) ← Size(⟨string⟩) − 1
⟨char⟩ ::= ⟨digit⟩ | A | B | . . .

(The symbol ⟨char⟩ has no synthesized or inherited attributes.) Now the following rules state that the size (> 0) inherited by the ⟨string⟩ following the H in a Hollerith literal is the value synthesized from the initial ⟨numeral⟩:

⟨literal⟩ ::= ⟨numeral⟩ H ⟨string⟩
Size(⟨string⟩) ← Val(⟨numeral⟩)
Condition: Val(⟨numeral⟩) > 0

Figure 2.14 shows the complete tree for the literal 2HAB. The numbers preceding the attribute occurrences and conditions indicate the order in which the values were filled in. Observe how information moves up the ⟨numeral⟩ subtree and then down the ⟨string⟩ subtree. The illegal string 2HA is disallowed because a false condition arises, as shown in Fig. 2.15. Similarly, 1HAB is disallowed (Fig. 2.16).

Intuitively, a synthesized attribute at a node corresponds to information arising from the internal constituents of that construct, while an inherited attribute corresponds to information arising from the external context of the construct. Under each alternative of a production rule, there must be an evaluation rule for each *synthesized* attribute of the symbol on the *left* (the symbol being defined) and for each *inherited* attribute of *each* symbol on the *right* (each symbol in the alternative). This provides us with a useful aid for checking the completeness of complex attribute grammars.

There is no really "standard" metalanguage for attribute grammars that has been generally adopted by all grammar writers. Although the underlying mechanism would be the same, the previous examples could well have been specified using different conventions and notations, especially with

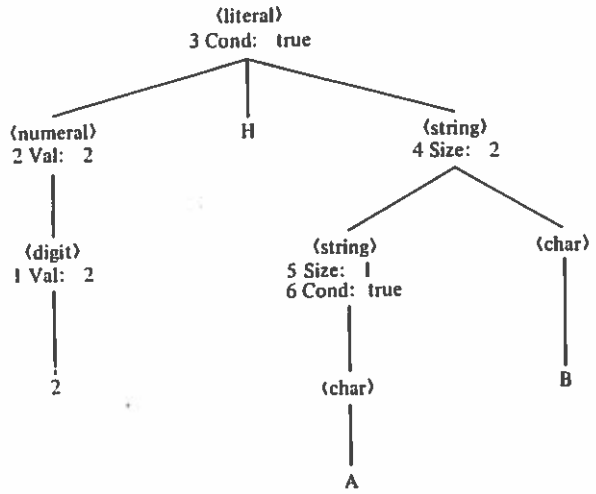


FIGURE 2.14

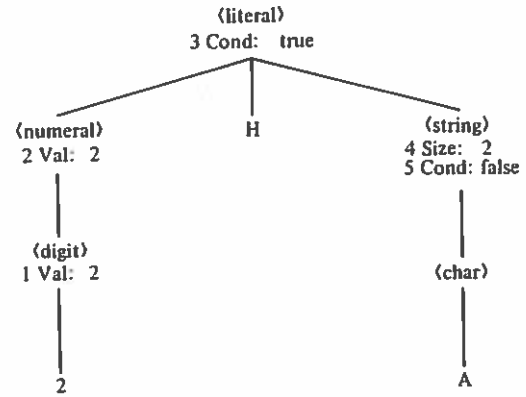


FIGURE 2.15

respect to the evaluation rules and conditions. More complicated grammars require additional notation, since it is generally necessary to deal with structured, nonnumeric domains of attribute values. For our purposes, we will make use of value domains that are *enumerations*, *sets*, *tuples*, or *sequences*, according to the following conventions:

1. The "constants" of an enumeration domain are arbitrarily chosen names enclosed in single quotes, e.g., 'full', 'empty'.

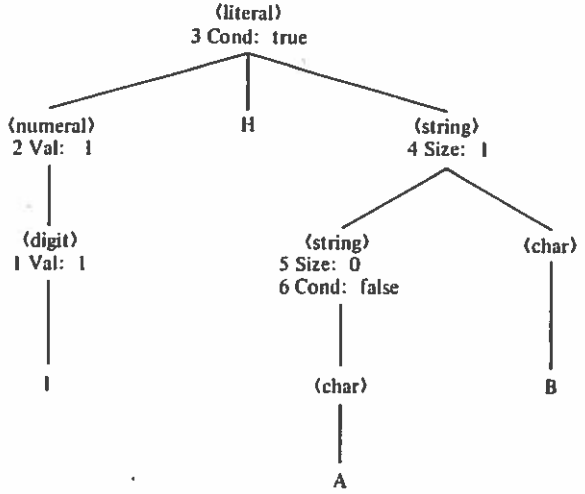


FIGURE 2.16

2. For sets, the evaluation rules will employ the usual notation, including the following symbols: { } (set brackets), ∪ (union), ∈ (membership).
3. Tuples belonging to a given domain consist of a fixed number of values, possibly of different types, in a certain order. The notation (. . . , . . .) may be used to specify a tuple value in terms of its components. For example, (17, 'a') denotes an ordered pair where the first component is the integer 17 and the second component is the character 'a'. The primitive functions $field_1$, $field_2$, . . . may be used to extract the components of a tuple.
4. A sequence is an ordered collection of any number of values of the same type. The notation ⟨. . . , . . .⟩ denotes a sequence, and ⟨ ⟩ denotes the empty sequence. The following primitive functions apply to sequences:

   - *append*(s, v). The sequence obtained by adding the value v to the end of the sequence s.
   - *concat*(s_1, s_2, . . . , s_n). The sequence obtained by joining the sequences s_1, s_2, . . . , s_n in order.
   - *length*(s). The number of elements in the sequence s.
   - *first*(s). The first element of the sequence s.
   - *last*(s). The last element of the sequence s.

- *tail*(s). The sequence obtained by deleting the first element of the sequence s.
- *allbutlast*(s). The sequence obtained by deleting the last element of the sequence s.

The usual notations for arithmetic and logic will also be used freely. Sometimes it is desirable or necessary to make use of separately defined auxiliary functions in the evaluation rules; a self-explanatory notation similar to that commonly used in mathematical descriptions will be used to define such functions.

For the sake of uniformity, we further decree that the specification of a subject language by means of an attribute grammar will consist of four parts:

1. *Attributes and values* (a list of the attributes used and their corresponding value domains).
2. *Attributes associated with nonterminal symbols* (a table showing the sets of inherited and synthesized attributes associated with each nonterminal symbol of the grammar).
3. *Production and attribute evaluation rules* (the grammar itself, laid out according to the conventions introduced earlier).
4. *Definition of auxiliary evaluation functions.*

EXERCISES

1. Determine the set of terminal strings generated by the following attribute grammar, where Size is a synthesized attribute of ⟨x string⟩ and an inherited attribute of ⟨y string⟩ and ⟨z string⟩:

⟨string⟩ ::= ⟨x string⟩ ⟨y string⟩ ⟨z string⟩
Size(⟨y string⟩) ← Size(⟨x string⟩)
Size(⟨z string⟩) ← Size(⟨x string⟩)

⟨x string⟩ ::= *x*
Size(⟨x string⟩) ← 1
| ⟨x string⟩$_2$ *x*
Size(⟨x string⟩) ← Size(⟨x string⟩$_2$) + 1

⟨y string⟩ ::= *y*
Condition: Size(⟨y string⟩) = 1
| ⟨y string⟩$_2$ *y*
Size(⟨y string⟩$_2$) ← Size(⟨y string⟩) − 1

⟨z string⟩ ::= *z*
Condition: Size(⟨z string⟩) = 1
| ⟨z string⟩$_2$ *z*
Size(⟨z string⟩$_2$) ← Size(⟨z string⟩) − 1

2. Without changing the set of terminal strings generated, modify the grammar of Exercise 1 so that Size is used as a synthesized attribute only.

3. Construct an attribute grammar that generates the set of all integer constants corresponding to values of less than 2^{31} with any of the following forms:

⟨sequence of binary digits⟩ (2)
⟨sequence of octal digits⟩ (8)
⟨sequence of hexadecimal digits⟩ (16)

[For example, the constants 11010(2), 32(8), and 1A(16) all stand for the value 26.]

2.3.2 A Complete Syntactic Specification for Eva

To be complete, an attribute grammar for the syntax of Eva must include formal conditions corresponding to the various context-sensitive aspects, such as type matching, which were characterized informally in Sec. 2.1.3.

The main idea leading to the formalization of these context conditions is to make use of an attribute Nest, which is an inherited attribute of most of the nonterminal symbols in the grammar. A Nest value records the name and type information specified by just those declaration sequences and/or parameter lists that bear upon the construct with which it is associated. Specifically, it is a sequence of values corresponding to the attribute Decs, where a Decs value records the name and type information specified by a single ⟨declaration sequence⟩ or ⟨parameter list⟩.

To define these value domains precisely, we introduce the attribute Type with values 'char', 'string', and 'proc', and the attribute Tag with values that are sequences of letter values between 'a' and 'z'. Tag is a synthesized attribute of the symbol ⟨name⟩ and serves to record the actual characters in the name:

⟨name⟩ ::= ⟨letter sequence⟩
Tag(⟨name⟩) ← Tag(⟨letter sequence⟩)

⟨letter sequence⟩ ::= ⟨letter⟩
Tag(⟨letter sequence⟩) ← Tag(⟨letter⟩)
| ⟨letter sequence⟩$_2$ ⟨letter⟩
Tag(⟨letter sequence⟩) ← *concat*(Tag(⟨letter sequence⟩$_2$), Tag(⟨letter⟩))

⟨letter⟩ ::= *a*
Tag(⟨letter⟩) ← ⟨'a'⟩
| . . .
. . .
| *z*
Tag(⟨letter⟩) ← ⟨'z'⟩

Now a Decs value is defined to be a set of triples of the form (Type, Tag, Params), where a Params value is a sequence of Type values. The Params field of a Decs triple will be a nonempty sequence of Type values only if the Type and Tag fields of the triple correspond to a procedure with parameters.

As a source of examples, consider the following Eva program:

```
begin
    char x
    proc p = (
        input x
        neq x, "z": ( output x  call p ) )
    call p
end
```

The Nest value for the ⟨block⟩ that comprises the entire program will be the empty sequence, because there are no outer declarations affecting it. The Nest value for each principal construct inside the ⟨block⟩ will be the following list containing one Decs value:

⟨[('char', ⟨'x'⟩, ⟨⟩), ('proc', ⟨'p'⟩, ⟨⟩)]⟩

If the body of *p* were a ⟨block⟩ with a local string variable *s*, the Nest value for the constructs in *p*'s body would be

⟨[('char', ⟨'x'⟩, ⟨⟩), ('proc', ⟨'p'⟩, ⟨⟩)], [('string', ⟨'s'⟩, ⟨⟩)]⟩

Because a procedure body may make reference to names declared at a textually later point in the program, the correct specification of the evaluation rules for Nest requires a little ingenuity.

First, note that a ⟨name list⟩ may be part of either a ⟨parameter list⟩ or a ⟨declaration⟩. The symbol ⟨name list⟩ thus has Params as well as Decs as a synthesized attribute, but the Params value will be used subsequently only in the context of a ⟨parameter list⟩. The two attribute values are computed in parallel:

⟨name list⟩ ::= ⟨name⟩
  Decs(⟨name list⟩) ⟵ [(Type(⟨name list⟩), Tag(⟨name⟩), ⟨⟩)]
  Params(⟨name list⟩) ⟵ ⟨Type(⟨name list⟩)⟩
| ⟨name list⟩₂, ⟨name⟩
  Decs(⟨name list⟩) ⟵ Decs(⟨name list⟩₂) ∪ [(Type(⟨name list⟩₂), Tag(⟨name⟩), ⟨⟩)]
  Params(⟨name list⟩) ⟵ *concat*(Params(⟨name list⟩₂), ⟨Type(⟨name list⟩)⟩)

  Type(⟨name list⟩₂) ⟵ Type(⟨name list⟩)
  Condition: ~(∃ d ∈ Decs(⟨name list⟩₂))(Tag(⟨name⟩) = $field_2$(d))

The last line prevents duplicate names in the same ⟨name list⟩. (The symbol '~' means 'not' and '∃' means 'there exists'.) Note that the Type values ('char' or 'string') used in constructing the Decs and Params values are inherited from the ⟨name list⟩. They are originally synthesized according to the rules

⟨declarer⟩ ::= **char**
  Type(⟨declarer⟩) ⟵ 'char'
| **string**
  Type(⟨declarer⟩) ⟵ 'string'

and are "handed over" according to the following rules, which also complete the synthesis of the Params and Decs values associated with a ⟨parameter list⟩ and the Decs value associated with a ⟨declaration⟩:

⟨parameter list⟩ ::= ⟨declarer⟩ ⟨name list⟩
  Params(⟨parameter list⟩) ⟵ Params(⟨name list⟩)
  Decs(⟨parameter list⟩) ⟵ Decs(⟨name list⟩)
  Type(⟨name list⟩) ⟵ Type(⟨declarer⟩)
| ⟨parameter list⟩₂, ⟨declarer⟩ ⟨name list⟩
  Params(⟨parameter list⟩) ⟵ *concat*(Params(⟨parameter list⟩₂), Params(⟨name list⟩))
  Decs(⟨parameter list⟩) ⟵ Decs(⟨parameter list⟩₂) ∪ Decs(⟨name list⟩)
  Type(⟨name list⟩) ⟵ Type(⟨declarer⟩)
⟨declaration⟩ ::= ⟨declarer⟩ ⟨name list⟩
  Decs(⟨declaration⟩) ⟵ Decs(⟨name list⟩)
  Type(⟨name list⟩) ⟵ Type(⟨declarer⟩)
| **proc** ⟨name⟩ = ⟨statement⟩
  Decs(⟨declaration⟩) ⟵ [('proc', Tag(⟨name⟩), ⟨⟩)]
  Nest(⟨statement⟩) ⟵ Nest(⟨declaration⟩)
| **proc** ⟨name⟩ (⟨parameter list⟩) = ⟨statement⟩
  Decs(⟨declaration⟩) ⟵ [('proc', Tag(⟨name⟩), Params(⟨parameter list⟩))]
  Nest(⟨statement⟩) ⟵ *append*(Nest(⟨declaration⟩), Decs(⟨parameter list⟩))

It can be seen that the Nest inherited by the body of a procedure with parameters is augmented with the Decs value corresponding to those parameters. The following rules complete the synthesis of the Decs value for a ⟨declaration sequence⟩, pass the relevant Nest value down to the individual declarations, and check that no name is declared more than once in the sequence:

⟨declaration sequence⟩ ::= ⟨declaration⟩
  Decs(⟨declaration sequence⟩) ⟵ Decs(⟨declaration⟩)
  Nest(⟨declaration⟩) ⟵ Nest(⟨declaration sequence⟩)
| ⟨declaration sequence⟩$_2$ ⟨declaration⟩
  Decs(⟨declaration sequence⟩) ⟵ Decs(⟨declaration sequence⟩$_2$) ∪ Decs(⟨declaration⟩)
  Nest(⟨declaration sequence⟩$_2$) ⟵ Nest(⟨declaration sequence⟩)
  Nest(⟨declaration⟩) ⟵ Nest(⟨declaration sequence⟩)
  Condition: (∀d ∈ Decs(⟨declaration sequence⟩$_2$))(~(∃d′ ∈ Decs(⟨declaration⟩))(*field*$_2$(d) = *field*$_2$(d′)))

(The symbol '∀' means 'for all'.) Augmentation of the Nest external to a ⟨block⟩ with the Decs value synthesized in the ⟨block⟩ is achieved as follows:

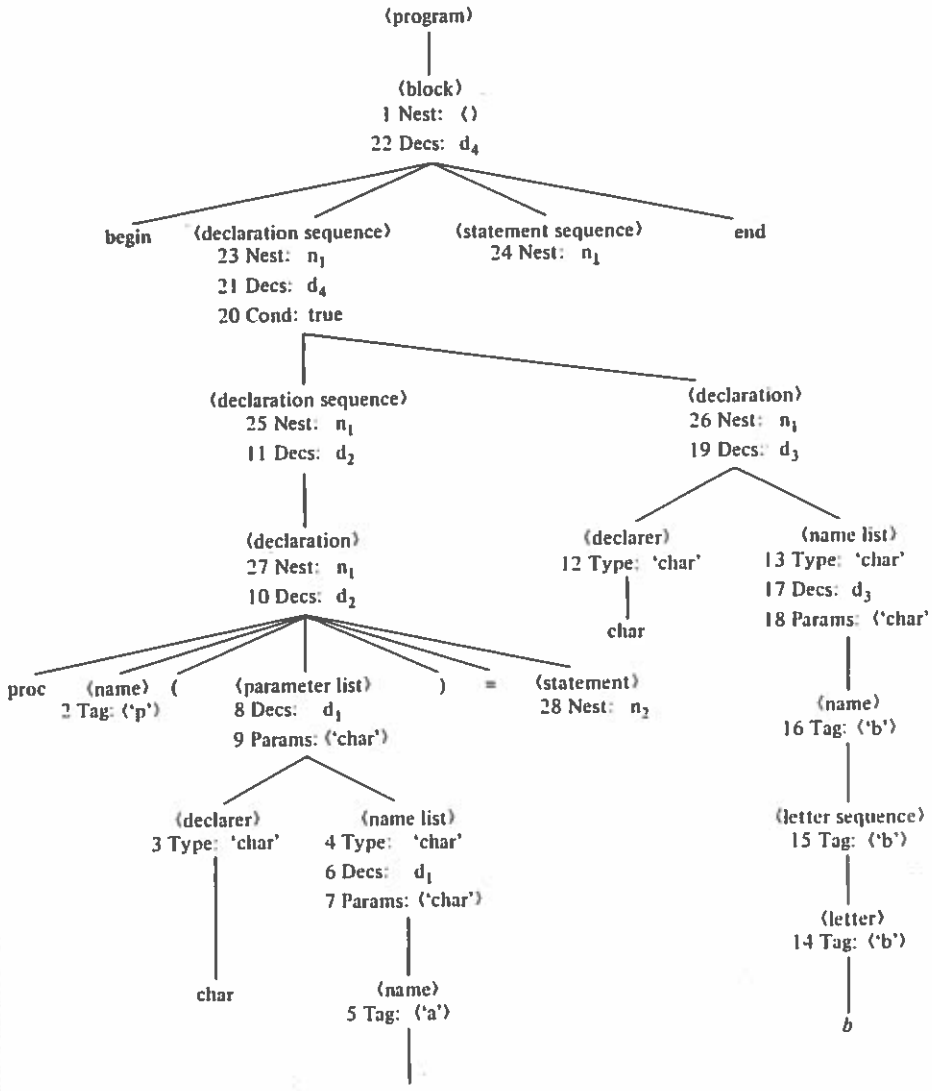
⟨block⟩ ::= **begin** ⟨declaration sequence⟩ ⟨statement sequence⟩ **end**
  Decs(⟨block⟩) ⟵ Decs(⟨declaration sequence⟩)
  Nest(⟨declaration sequence⟩) ⟵ *append*(Nest(⟨block⟩), Decs(⟨declaration sequence⟩))
  Nest(⟨statement sequence⟩) ⟵ *append*(Nest(⟨block⟩), Decs(⟨declaration sequence⟩))

⟨program⟩ ::= ⟨block⟩
  Nest(⟨block⟩) ⟵ ⟨⟩

As an example of how all these rules interact, suppose that we have a program of the following form:

```
begin
    proc p (char a) = ...
    char b
    ...
end
```

In the partial tree shown in Fig. 2.17, the numbers preceding the attributes



d_1 = {⟨'char', ⟨'a'⟩, ⟨⟩⟩}
d_2 = {⟨'proc', ⟨'p'⟩, ⟨'char'⟩⟩}
d_3 = {⟨'char', ⟨'b'⟩, ⟨⟩⟩}
$d_4 = d_2 \cup d_3$
$n_1 = \langle d_4 \rangle$
$n_2 = \langle d_4, d_1 \rangle$

FIGURE 2.17

indicate the *relative* order in which the values were filled in. Observe how the Nest values can be inserted (in a downward direction) only after all the Decs values have been inserted (in an upward direction).

The part of the grammar dealing with statements and expressions is now relatively straightforward. Many of the evaluation rules simply serve to pass down the Nest value computed in the manner described above. Type checking of names is carried out with the aid of an auxiliary function *latesttype* which, in effect, searches the Nest value from back to front for an occurrence of a specified name. Thus two more context conditions are disposed of in the following rules:

⟨char expression⟩ ::= ⟨name⟩
  Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨char expression⟩)) = 'char'
| " ⟨letter⟩ "
| **space**
| **head** ⟨string expression⟩
  Nest(⟨string expression⟩) ⟵ Nest(⟨char expression⟩)

⟨string expression⟩ ::= ⟨name⟩
  Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨string expression⟩)) = 'string'
| " "
| " ⟨letter⟩ ⟨letter sequence⟩ "
| **tail** ⟨string expression⟩$_2$
  Nest(⟨string expression⟩$_2$) ⟵ Nest(⟨string expression⟩)

(No use is made of the Tag values synthesized for these occurrences of ⟨letter⟩ and ⟨letter sequence⟩.)

Apart from the treatment of procedure calls, the following rules should be readily understood:

⟨statement sequence⟩ ::= ⟨statement⟩
  Nest(⟨statement⟩) ⟵ Nest(⟨statement sequence⟩)
| ⟨statement sequence⟩$_2$ ⟨statement⟩
  Nest(⟨statement sequence⟩$_2$) ⟵ Nest(⟨statement sequence⟩)
  Nest(⟨statement⟩) ⟵ Nest(⟨statement sequence⟩)

⟨statement⟩ ::= **input** ⟨name⟩
  Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨statement⟩)) = 'char'
| **output** ⟨char expression⟩
  Nest(⟨char expression⟩) ⟵ Nest(⟨statement⟩)

| ⟨block⟩
  Nest(⟨block⟩) ⟵ Nest(⟨statement⟩)
| (⟨statement sequence⟩)
  Nest(⟨statement sequence⟩) ⟵ Nest(⟨statement⟩)
| ⟨test⟩ ⟨pair⟩ : ⟨statement⟩$_2$
  Nest(⟨pair⟩) ⟵ Nest(⟨statement⟩)
  Nest(⟨statement⟩$_2$) ⟵ Nest(⟨statement⟩)
| **cons** ⟨char expression⟩ , ⟨name⟩
  Nest(⟨char expression⟩) ⟵ Nest(⟨statement⟩)
  Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨statement⟩)) = 'string'
| **call** ⟨name⟩
  Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨statement⟩)) =
    'proc' ∧ *parameters*(Tag(⟨name⟩), Nest(⟨statement⟩)) = ◇
| **call** ⟨name⟩ (⟨expression list⟩)
  Nest(⟨expression list⟩) ⟵ Nest(⟨statement⟩)
  Params(⟨expression list⟩) ⟵ *parameters*(Tag(⟨name⟩),
    Nest(⟨statement⟩))
  Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨statement⟩)) = 'proc'

⟨test⟩ ::= **eq**
| **neq**

⟨pair⟩ ::= ⟨char expression⟩$_1$, ⟨char expression⟩$_2$
  Nest(⟨char expression⟩$_1$) ⟵ Nest(⟨pair⟩)
  Nest(⟨char expression⟩$_2$) ⟵ Nest(⟨pair⟩)
| ⟨string expression⟩$_1$, ⟨string expression⟩$_2$
  Nest(⟨string expression⟩$_1$) ⟵ Nest(⟨pair⟩)
  Nest(⟨string expression⟩$_2$) ⟵ Nest(⟨pair⟩)

The auxiliary function *parameters* returns the Params value associated with a specified procedure name in the Nest. The argument list (⟨expression list⟩) of a call of a procedure with parameters inherits this value. In order that the type of each argument may be checked for compatibility with the type of the corresponding parameter, the symbol ⟨expression⟩ has Type as a synthesized attribute:

⟨expression⟩ ::= ⟨char expression⟩
  Type(⟨expression⟩) ⟵ 'char'
  Nest(⟨char expression⟩) ⟵ Nest(⟨expression⟩)

| ⟨string expression⟩

Type(⟨expression⟩) ⟵ 'string'

Nest(⟨string expression⟩) ⟵ Nest(⟨expression⟩)

⟨expression list⟩ ::= ⟨expression⟩

Nest(⟨expression⟩) ⟵ Nest(⟨expression list⟩)

Condition: *length*(Params(⟨expression list⟩)) = 1 ∧ Type(⟨expression⟩) = *first*(Params(⟨expression list⟩))

| ⟨expression⟩ , ⟨expression list⟩$_2$

Nest(⟨expression⟩) ⟵ Nest(⟨expression list⟩)

Nest(⟨expression list⟩$_2$) ⟵ Nest(⟨expression list⟩)

Params(⟨expression list⟩$_2$) ⟵ *tail*(Params(⟨expression list⟩))

Condition: *length*(Params(⟨expression list⟩)) ≥ 1 ∧ Type(⟨expression⟩) = *first*(Params(⟨expression list⟩))

These rules also require the number of arguments to be equal to the number of parameters.

The formal syntactic definition of Eva by means of an attribute grammar is now complete, and the specifications are summarized in Table 2.5.

TABLE 2.5 Attribute Grammar for the Syntax of Eva

Attributes and Values

Attribute	Values
Type	'char', 'string', 'proc'
Params	sequences of Type values
Tag	sequences of letters ('a', . . . , 'z')
Decs	sets of triples of the form (Type, Tag, Params)
Nest	sequences of Decs values

Attributes Associated with Nonterminal Symbols

Nonterminal	Inherited attributes	Synthesized attributes
⟨program⟩	—	—
⟨block⟩	Nest	Decs
⟨declaration sequence⟩	Nest	Decs
⟨declaration⟩	Nest	Decs
⟨parameter list⟩	—	Decs, Params
⟨declarer⟩	—	Type
⟨name list⟩	Type	Decs, Params
⟨statement sequence⟩	Nest	—
⟨statement⟩	Nest	—
⟨pair⟩	Nest	—
⟨expression list⟩	Nest, Params	—
⟨expression⟩	Nest	Type
⟨char expression⟩	Nest	—
⟨string expression⟩	Nest	—
⟨name⟩	—	Tag
⟨letter sequence⟩	—	Tag
⟨letter⟩	—	Tag

Production and Attribute Evaluation Rules

⟨program⟩ ::= ⟨block⟩

Nest(⟨block⟩) ⟵ ⟨⟩

⟨block⟩ ::= **begin** ⟨declaration sequence⟩ ⟨statement sequence⟩ **end**

Decs(⟨block⟩) ⟵ Decs(⟨declaration sequence⟩)

Nest(⟨declaration sequence⟩) ⟵ *append*(Nest(⟨block⟩), Decs(⟨declaration sequence⟩))

Nest(⟨statement sequence⟩) ⟵ *append*(Nest(⟨block⟩), Decs(⟨declaration sequence⟩))

⟨declaration sequence⟩ ::= ⟨declaration⟩

Decs(⟨declaration sequence⟩) ⟵ Decs(⟨declaration⟩)

Nest(⟨declaration⟩) ⟵ Nest(⟨declaration sequence⟩)

| ⟨declaration sequence⟩$_2$ ⟨declaration⟩

Decs(⟨declaration sequence⟩) ⟵ Decs(⟨declaration sequence⟩$_2$) ∪ Decs(⟨declaration⟩)

Nest(⟨declaration sequence⟩$_2$) ⟵ Nest(⟨declaration sequence⟩)

Nest(⟨declaration⟩) ⟵ Nest(⟨declaration sequence⟩)

Condition: (∀ d ∈ Decs(⟨declaration sequence⟩$_2$))(∼(∃ d′ ∈ Decs(⟨declaration⟩))(*field*$_2$(d) = *field*$_2$(d′)))

⟨declaration⟩ ::= ⟨declarer⟩ ⟨name list⟩

Decs(⟨declaration⟩) ⟵ Decs(⟨name list⟩)

Type(⟨name list⟩) ⟵ Type(⟨declarer⟩)

| **proc** ⟨name⟩ = ⟨statement⟩

Decs(⟨declaration⟩) ⟵ {('proc', Tag(⟨name⟩), ⟨⟩)}

Nest(⟨statement⟩) ⟵ Nest(⟨declaration⟩)

| **proc** ⟨name⟩ (⟨parameter list⟩) = ⟨statement⟩

Decs(⟨declaration⟩) ⟵ {('proc', Tag(⟨name⟩), Params(⟨parameter list⟩))}

Nest(⟨statement⟩) ⟵ *append*(Nest(⟨declaration⟩), Decs(⟨parameter list⟩))

TABLE 2.5 (Continued)

⟨parameter list⟩ ::= ⟨declarer⟩ ⟨name list⟩
Params(⟨parameter list⟩) ⟵ Params(⟨name list⟩)
Decs(⟨parameter list⟩) ⟵ Decs(⟨name list⟩)
Type(⟨name list⟩) ⟵ Type(⟨declarer⟩)

| ⟨parameter list⟩$_2$, ⟨declarer⟩ ⟨name list⟩
Params(⟨parameter list⟩) ⟵ *concat*(Params(⟨parameter list⟩$_2$), Params(⟨name list⟩))
Decs(⟨parameter list⟩) ⟵ Decs(⟨parameter list⟩$_2$) ∪ Decs(⟨name list⟩) .
Type(⟨name list⟩) ⟵ Type(⟨declarer⟩)

⟨declarer⟩ ::= **char**
Type(⟨declarer⟩) ⟵ 'char'

| **string**
Type(⟨declarer⟩) ⟵ 'string'

⟨name list⟩ ::= ⟨name⟩
Decs(⟨name list⟩) ⟵ {(Type(⟨name list⟩), Tag(⟨name⟩), ⟨⟩)}
Params(⟨name list⟩) ⟵ ⟨Type(⟨name list⟩)⟩

| ⟨name list⟩$_2$, ⟨name⟩
Decs(⟨name list⟩) ⟵ Decs(⟨name list⟩$_2$) ∪ {(Type(⟨name list⟩$_2$), Tag(⟨name⟩), ⟨⟩)}
Params(⟨name list⟩) ⟵ *concat*(Params(⟨name list⟩$_2$), ⟨Type(⟨name list⟩)⟩)
Type(⟨name list⟩$_2$) ⟵ Type(⟨name list⟩)
Condition: ~(∃ d ∈ Decs(⟨name list⟩$_2$))(Tag(⟨name⟩) = $field_2$(d))

⟨statement sequence⟩ ::= ⟨statement⟩
Nest(⟨statement⟩) ⟵ Nest(⟨statement sequence⟩)

| ⟨statement sequence⟩$_2$ ⟨statement⟩
Nest(⟨statement sequence⟩$_2$) ⟵ Nest(⟨statement sequence⟩)
Nest(⟨statement⟩) ⟵ Nest(⟨statement sequence⟩)

⟨statement⟩ ::= **input** ⟨name⟩
Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨statement⟩)) = 'char'

| **output** ⟨char expression⟩
Nest(⟨char expression⟩) ⟵ Nest(⟨statement⟩)

| ⟨block⟩
Nest(⟨block⟩) ⟵ Nest(⟨statement⟩)

| (⟨statement sequence⟩)
Nest(⟨statement sequence⟩) ⟵ Nest(⟨statement⟩)

TABLE 2.5 (Continued)

| ⟨test⟩ ⟨pair⟩ : ⟨statement⟩$_2$
Nest(⟨pair⟩) ⟵ Nest(⟨statement⟩)
Nest(⟨statement⟩$_2$) ⟵ Nest(⟨statement⟩)

| **cons** ⟨char expression⟩ , ⟨name⟩
Nest(⟨char expression⟩) ⟵ Nest(⟨statement⟩)
Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨statement⟩)) = 'string'

| **call** ⟨name⟩
Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨statement⟩)) = 'proc'
∧ *parameters*(Tag(⟨name⟩), Nest(⟨statement⟩)) = ⟨⟩

| **call** ⟨name⟩ (⟨expression list⟩)
Nest(⟨expression list⟩) ⟵ Nest(⟨statement⟩)
Params(⟨expression list⟩) ⟵ *parameters*(Tag(⟨name⟩), Nest(⟨statement⟩))
Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨statement⟩)) = 'proc'

⟨test⟩ ::= **eq**
| **neq**

⟨pair⟩ ::= ⟨char expression⟩$_1$, ⟨char expression⟩$_2$
Nest(⟨char expression⟩$_1$) ⟵ Nest(⟨pair⟩)
Nest(⟨char expression⟩$_2$) ⟵ Nest(⟨pair⟩)

| ⟨string expression⟩$_1$, ⟨string expression⟩$_2$
Nest(⟨string expression⟩$_1$) ⟵ Nest(⟨pair⟩)
Nest(⟨string expression⟩$_2$) ⟵ Nest(⟨pair⟩)

⟨expression list⟩ ::= ⟨expression⟩
Nest(⟨expression⟩) ⟵ Nest(⟨expression list⟩)
Condition: *length*(Params(⟨expression list⟩)) = 1 ∧ Type(⟨expression⟩) = *first*(Params(⟨expression list⟩))

| ⟨expression⟩ , ⟨expression list⟩$_2$
Nest(⟨expression⟩) ⟵ Nest(⟨expression list⟩)
Nest(⟨expression list⟩$_2$) ⟵ Nest(⟨expression list⟩)
Params(⟨expression list⟩$_2$) ⟵ *tail*(Params(⟨expression list⟩))
Condition: *length*(Params(⟨expression list⟩)) ≥ 1 ∧ Type(⟨expression⟩) = *first*(Params(⟨expression list⟩))

⟨expression⟩ ::= ⟨char expression⟩
Nest(⟨char expression⟩) ⟵ Nest(⟨expression⟩)
Type(⟨expression⟩) ⟵ 'char'

| ⟨string expression⟩
Nest(⟨string expression⟩) ⟵ Nest(⟨expression⟩)

TABLE 2.5 (Continued)

Type(⟨expression⟩) ⟵ 'string'

⟨char expression⟩ ::= ⟨name⟩

Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨char expression⟩)) = 'char'

| " ⟨letter⟩ "

| space

| head ⟨string expression⟩

Nest(⟨string expression⟩) ⟵ Nest(⟨char expression⟩)

⟨string expression⟩ ::= ⟨name⟩

Condition: *latesttype*(Tag(⟨name⟩), Nest(⟨string expression⟩)) = 'string'

| " "

| " ⟨letter⟩ ⟨letter sequence⟩ "

| tail ⟨string expression⟩$_2$

Nest(⟨string expression⟩$_2$) ⟵ Nest(⟨string expression⟩)

⟨name⟩ ::= ⟨letter sequence⟩

Tag(⟨name⟩) ⟵ Tag(⟨letter sequence⟩)

⟨letter sequence⟩ ::= ⟨letter⟩

Tag(⟨letter sequence⟩) ⟵ Tag(⟨letter⟩)

| ⟨letter sequence⟩$_2$ ⟨letter⟩

Tag(⟨letter sequence⟩) ⟵ *concat*(Tag(⟨letter sequence⟩$_2$), Tag(⟨letter⟩))

⟨letter⟩ ::= *a*

Tag(⟨letter⟩) ⟵ ⟨'a'⟩

| . . .

. . .

| *z*

Tag(⟨letter⟩) ⟵ ⟨'z'⟩

Definition of Auxiliary Evaluation Functions

latesttype (tag, nest) =

'undefined', if nest = ⟨⟩;

t, if (∃ d ∈ *last*(nest))(d = (t, tag, p)) for some t, p;

latesttype (tag, *allbutlast* (nest)), otherwise.

parameters (tag, nest) =

⟨'undefined'⟩, if nest = ⟨⟩;

p, if (∃ d ∈ *last*(nest))(d = (t, tag, p)) for some t, p;

parameters (tag, *allbutlast* (nest)), otherwise.

EXERCISES

1. Referring to the grammar of Table 2.5, construct the complete, decorated syntax tree for the program

```
begin
    proc p (char a, b) =
        output b
    call p ("x", "y")
end
```

2. Replacement of the constant "*y*" by "*yz*" in the program of Exercise 1 introduces a violation of a context condition. Verify that the resulting erroneous program is not derivable from the grammar.

3. For some actual programming language with which you are familiar, identify the context condition(s) (or at least some of them) associated with the agreement of data types for the left and right sides of an assignment statement. Explain in general terms how the condition(s) might be formalized in an attribute grammar for the language.

For Further Information

The basic technique of augmenting context-free grammars with inherited and synthesized attributes is due to Knuth (1968), who also investigates its mathematical aspects. The attribute-grammar formalism and its application to the problem of compiler generation are further discussed by Lewis et al. (1974) and by Bochmann (1976). There is no sharp boundary between syntax and semantics as far as the linguistic properties definable by attribute grammars are concerned, and parts of these papers deal with the concept of "translational semantics" discussed in Sec. 3.2 of this book. Watt (1979) presents a complete definition of the syntax of Pascal using an extended version of the attribute-grammar formalism.

Ledgard's (1974, 1977) formalism of *production systems* is capable of defining the context-sensitive aspects of programming language syntax in a manner somewhat analogous to that of attribute grammars. Another powerful grammatical formalism which is closely related both to attribute grammars and to the two-level grammars described in the next section is that of *affix grammars* (Koster, 1971); Crowe (1972) and Watt (1977) discuss its application to compiler generation.

2.4
TWO-LEVEL GRAMMARS

We now turn to another formalism which is inherently much more powerful than BNF and which is capable of dealing with the context-sensitive as well as the context-free aspects of a subject language. We begin by describing a

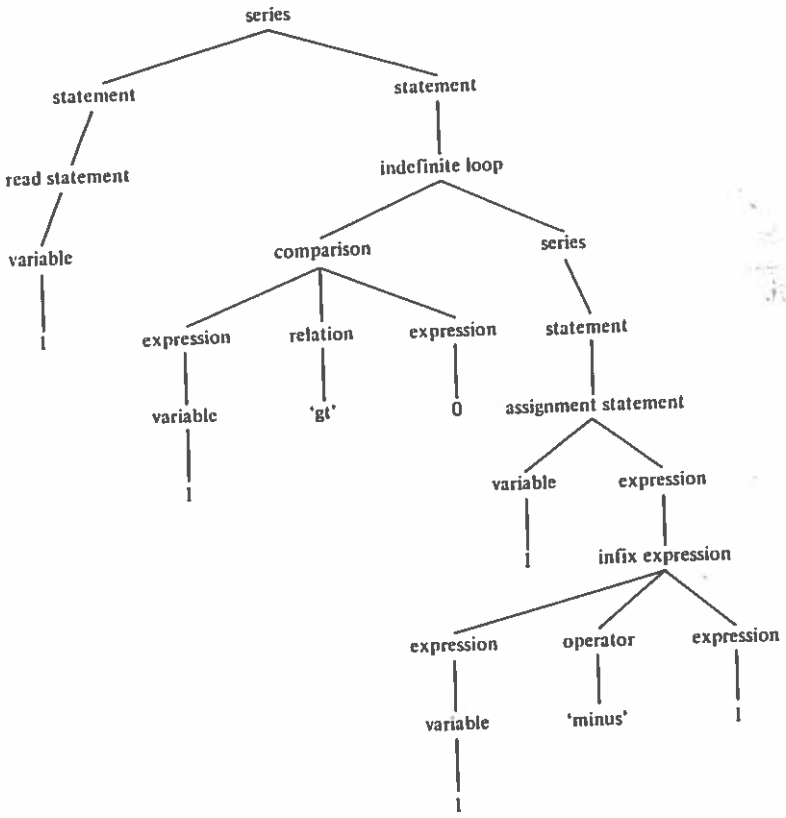


FIGURE 3.1

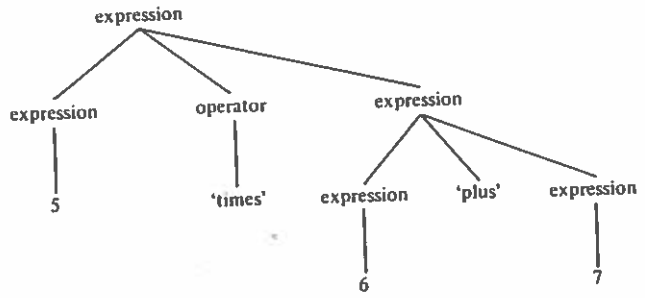


FIGURE 3.2

scarcely noticeable in the section that follows, but much of the rest of the book may be viewed as dealing with techniques for associating meaning with abstract program representations. Only the general idea of abstract syntax has been introduced here; each semantic formalism has its own conceptual refinements and notations.

EXERCISES

1. Give an informal but careful description of the abstract syntax of Eva.

2. Identify a small but usable subset of Fortran, Pascal, PL/I, or some other actual language, and give an informal but careful description of its abstract syntax.

3.2 TRANSLATIONAL SEMANTICS USING ATTRIBUTE GRAMMARS—A COMPLETE DEFINITION OF PAM

The definitional power of attribute grammars, which were introduced in Sec. 2.3, extends beyond the realm of syntax into the realm of semantics. In particular, the attribute grammar technique is well suited to the formal specification of a mapping from a subject language to some other language which, following the terminology associated with compilers, we may call an *object language*. The contention that such a mapping constitutes a complete semantic specification of the subject language rests upon the assumption that the object language either is itself defined independently or is "semantically primitive." In this section, we shall illustrate this "translational" approach to language definition by taking Pam as the subject language and a simple symbolic (assembly-like) machine language as the object language.

The machine-like object language corresponds to a very simple computer with one accumulator and a memory whose words are capable of holding integer values. There are 17 instruction types, with the following mnemonic operation codes:

LOAD	Load accumulator
STO	Store
ADD	Add
SUB	Subtract
MULT	Multiply
DIV	Divide
GET	Input a value
PUT	Output a value
J	Jump
JN	Jump on negative

JP	Jump on positive
JZ	Jump on zero
JNZ	Jump on negative or zero
JPZ	Jump on positive or zero
JNP	Jump on negative or positive
LAB	Label (no operation)
HALT	Halt execution

Each STO, GET, and PUT instruction includes one operand specifying a memory address. This may be either an address corresponding to a Pam variable, in which case the variable itself appears in the instruction, or an extra working-storage address ("temporary") represented by one of the symbols T1, T2, etc. Each LOAD, ADD, SUB, MULT, and DIV instruction includes one operand given either by a memory address or by an integer constant; the result of the operation is left in the accumulator. HALT and LAB have no operands, but the latter symbol is always preceded by a label of the form L1, L2, etc.; only LAB instructions may be so labeled. Each jump instruction includes one operand that is a label. Except in the case of a J instruction, whether the transfer of control actually occurs depends on the value currently in the accumulator.

As an example of a program expressed in this object language, the Pam program

```
read x, y ;
while x <> 99 do
    ans := (x + 1) - (y / 2) ;
    write ans ;
    read x, y
end
```

could be translated as follows:

```
    GET   x          STO   T2
    GET   y          LOAD  T1
L1  LAB              SUB   T2
    LOAD  x          STO   ans
    SUB   99         PUT   ans
    JZ    L2         GET   x
    LOAD  x          GET   y
    ADD   1          J     L1
    STO   T1    L2   LAB
    LOAD  y          HALT
    DIV   2
```


The basic strategy for constructing our attribute grammar is to associate with most of the nonterminal symbols a synthesized attribute Code, where a Code value represents a sequence of instructions such as that given above. The Code value for a given construct will represent the object-language translation of that construct, and the meaning of an entire ⟨program⟩ is therefore defined to be the Code value associated with the root of its syntax tree.

To make this precise, we introduce the attribute Num corresponding to sequences of digit characters and the attribute Tag corresponding to sequences of letter and/or digit characters. Num is a synthesized attribute of ⟨constant⟩ and Tag a synthesized attribute of ⟨variable⟩:

⟨constant⟩ ::= ⟨digit⟩

Num(⟨constant⟩) ← Num(⟨digit⟩)

| ⟨constant⟩$_2$ ⟨digit⟩

Num(⟨constant⟩) ← *concat*(Num(⟨constant⟩$_2$), Num(⟨digit⟩))

⟨variable⟩ ::= ⟨letter⟩

Tag(⟨variable⟩) ← Tag(⟨letter⟩)

| ⟨variable⟩$_2$ ⟨letter⟩

Tag(⟨variable⟩) ← *concat*(Tag(⟨variable⟩$_2$), Tag(⟨letter⟩))

| ⟨variable⟩$_2$ ⟨digit⟩

Tag(⟨variable⟩) ← *concat*(Tag(⟨variable⟩$_2$), Num(⟨digit⟩))

⟨digit⟩ ::= 0

Num(⟨digit⟩) ← ⟨'0'⟩

| ...

...

| 9

Num(⟨digit⟩) ← ⟨'9'⟩

⟨letter⟩ ::= *a*

Tag(⟨letter⟩) ← ⟨'a'⟩

| ...

...

| *z*

Tag(⟨letter⟩) ← ⟨'z'⟩

We also introduce the attribute Opcode, with values 'LOAD', 'STO', 'ADD', 'SUB', 'MULT', 'DIV', 'GET', 'PUT', 'J', 'JN', 'JP', 'JZ', 'JNZ', 'JPZ', and 'JNP'. Then the values corresponding to Code are defined as sequences of

values of the following forms:

'HALT'
(Opcode, Num)
(Opcode, Tag)
(Tag, 'LAB')

The attribute grammar, which is shown in its entirety in Table 3.1, page 92, makes use of three further attributes, Temp, Labin, and Labout, all corresponding to nonnegative integer values. The attribute Temp serves to keep track of the number of temporaries (T1, T2, etc.) which are in use at any given point in a program and which therefore must not be reused by an embedded construct. A Temp value is inherited by each construct that may require the use of additional temporaries in its translation. For example, suppose that we have an ⟨expression⟩ of the form

⟨expression⟩$_2$ − ⟨term⟩

If the ⟨term⟩ is just a variable or a constant, the code for the ⟨expression⟩ can simply take the form

code for ⟨expression⟩$_2$
SUB ⟨term⟩

but if the ⟨term⟩ contains any operators, its code will contain instructions that alter the accumulator contents, so that the overall pattern of code should be something like

code for ⟨expression⟩$_2$
STO T1
code for ⟨term⟩
STO T2
LOAD T1
SUB T2

Now if the code for the ⟨term⟩ similarly requires the use of a pair of temporaries, these will obviously have to be different ones, say T3 and T4. To make this possible, we arrange for the ⟨term⟩ to inherit a Temp value that is greater by 2 than the Temp value for the original ⟨expression⟩.

At this point, we may examine the detailed specifications for expressions:

⟨expression⟩ ::= ⟨term⟩
  Code(⟨expression⟩) ← Code(⟨term⟩)
  Temp(⟨term⟩) ← Temp(⟨expression⟩)
| ⟨expression⟩$_2$ ⟨weak operator⟩ ⟨term⟩
  Code(⟨expression⟩) ← *concat*(Code(⟨expression⟩$_2$),
    selectcode(Code(⟨term⟩), Temp(⟨expression⟩),
      Opcode(⟨weak operator⟩)))
  Temp(⟨expression⟩$_2$) ← Temp(⟨expression⟩)
  Temp(⟨term⟩) ← Temp(⟨expression⟩) + 2

Opcode is a synthesized attribute of ⟨weak operator⟩:

⟨weak operator⟩ ::= +
  Opcode(⟨weak operator⟩) ← 'ADD'
| −
  Opcode(⟨weak operator⟩) ← 'SUB'

If the ⟨term⟩ contains no operators, then, as will be seen shortly, Code (⟨term⟩) will consist of a single LOAD instruction. In that case, the auxiliary function *selectcode* returns a single ADD or SUB instruction in place of the LOAD, but with the same operand. Otherwise, it returns the longer pattern of instructions involving the use of temporaries. The following rules are completely analogous to those given above:

⟨term⟩ ::= ⟨element⟩
  Code(⟨term⟩) ← Code(⟨element⟩)
  Temp(⟨element⟩) ← Temp(⟨term⟩)
| ⟨term⟩$_2$ ⟨strong operator⟩ ⟨element⟩
  Code(⟨term⟩) ← *concat*(Code(⟨term⟩$_2$), *selectcode*(Code(⟨element⟩),
    Temp(⟨term⟩), Opcode(⟨strong operator⟩)))
  Temp(⟨term⟩$_2$) ← Temp(⟨term⟩)
  Temp(⟨element⟩) ← Temp(⟨term⟩) + 2
⟨strong operator⟩ ::= *
  Opcode(⟨strong operator⟩) ← 'MULT'
| /
  Opcode(⟨strong operator⟩) ← 'DIV'

The specifications for ⟨element⟩ are straightforward:

⟨element⟩ ::= ⟨constant⟩
  Code(⟨element⟩) ← ⟨('LOAD', Num(⟨constant⟩))⟩
| ⟨variable⟩
  Code(⟨element⟩) ← ⟨('LOAD', Tag(⟨variable⟩))⟩
| (⟨expression⟩)
  Code(⟨element⟩) ← Code(⟨expression⟩)
  Temp(⟨expression⟩) ← Temp(⟨element⟩)

As a specific example of the operation of these rules, Fig. 3.3 shows the order in which the attribute values are filled in for $a + b * c$. Here we have assumed that the Temp value inherited by the top ⟨expression⟩ is 0. Evidently, we are not going out of our way to specify "optimized" code—a better translation of $a + b * c$ would be

```
LOAD b
MULT c
ADD  a
```

It should be noted in passing that the same temporaries may be used for different purposes in disjoint constructs. For example, it may be verified that the expresssion $(x + y * z) + b * c$ is mapped into the code

```
LOAD  x        STO   T1
STO   T1       LOAD  b
LOAD  y        MULT  c
MULT  z        STO   T2
STO   T2       LOAD  T1
LOAD  T1       ADD   T2
ADD   T2
```

whereas $a + (x + y * z)$ is mapped into

```
LOAD  a        STO   T4
STO   T1       LOAD  T3
LOAD  x        ADD   T4
STO   T3       STO   T2
LOAD  y        LOAD  T1
MULT  z        ADD   T2
```

Turning now to the higher constructs of Pam, the attribute Labin, which is inherited by each construct whose code may contain one or more labels, serves to keep track of the number of labels generated so far. Each such construct also has a synthesized attribute Labout, whose value will be the Labin value plus the number of labels contained in the code for the construct. This scheme prevents the appearance of duplicate labels in the code.

At the beginning of a ⟨program⟩, no temporaries or labels have been used, and the code for the ⟨program⟩ is simply the code for the ⟨series⟩ of which it is composed followed by a HALT instruction:

⟨program⟩ ::= ⟨series⟩
    Code(⟨program⟩) ← *append*(Code(⟨series⟩), 'HALT')
    Temp(⟨series⟩) ← 0
    Labin(⟨series⟩) ← 0

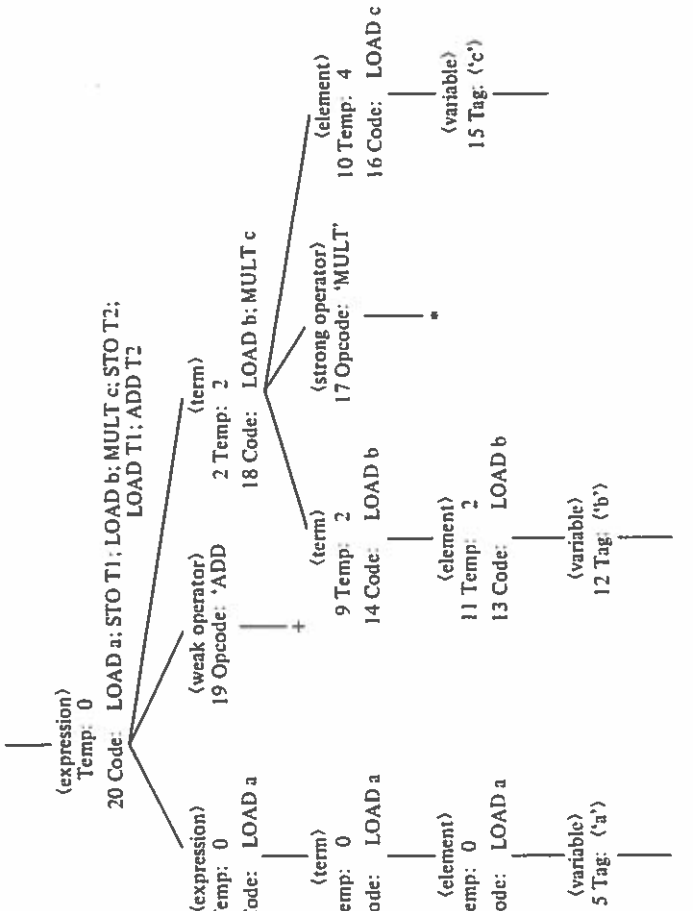


FIGURE 3.3

The situation where a ⟨series⟩ consists of a single ⟨statement⟩ is trivial:

⟨series⟩ ::= ⟨statement⟩
Code(⟨series⟩) ⟵ Code(⟨statement⟩)
Labout(⟨series⟩) ⟵ Labout(⟨statement⟩)
Temp(⟨statement⟩) ⟵ Temp(⟨series⟩)
Labin(⟨statement⟩) ⟵ Labin(⟨series⟩)

In the next part, note that a semicolon does not give rise to any code, and that the label count is "handed over" from one statement to the next by the last evaluation rule:

⟨series⟩ ::= ⟨series⟩$_2$; ⟨statement⟩
Code(⟨series⟩) ⟵ *concat*(Code(⟨series⟩$_2$), Code(⟨statement⟩))
Labout(⟨series⟩) ⟵ Labout(⟨statement⟩)
Temp(⟨series⟩$_2$) ⟵ Temp(⟨series⟩)
Labin(⟨series⟩$_2$) ⟵ Labin(⟨series⟩)
Temp(⟨statement⟩) ⟵ Temp(⟨series⟩)
Labin(⟨statement⟩) ⟵ Labout(⟨series⟩$_2$)

All types of ⟨statement⟩ synthesize code and a final label count (Labout); for input, output, and assignment statements, the latter is simply the inherited Labin value. Input and output statements do not inherit a Temp value because their code never involves the use of temporaries:

⟨statement⟩ ::= ⟨input statement⟩
Code(⟨statement⟩) ⟵ Code(⟨input statement⟩)
Labout(⟨statement⟩) ⟵ Labin(⟨statement⟩)
| ⟨output statement⟩
Code(⟨statement⟩) ⟵ Code(⟨output statement⟩)
Labout(⟨statement⟩) ⟵ Labin(⟨statement⟩)
| ⟨assignment statement⟩
Code(⟨statement⟩) ⟵ Code(⟨assignment statement⟩)
Labout(⟨statement⟩) ⟵ Labin(⟨statement⟩)
Temp(⟨assignment statement⟩) ⟵ Temp(⟨statement⟩)
| ⟨conditional statement⟩
Code(⟨statement⟩) ⟵ Code(⟨conditional statement⟩)
Labout(⟨statement⟩) ⟵ Labout(⟨conditional statement⟩)
Temp(⟨conditional statement⟩) ⟵ Temp(⟨statement⟩)
Labin(⟨conditional statement⟩) ⟵ Labin(⟨statement⟩)

| ⟨definite loop⟩
Code(⟨statement⟩) ⟵ Code(⟨definite loop⟩)
Labout(⟨statement⟩) ⟵ Labout(⟨definite loop⟩)
Temp(⟨definite loop⟩) ⟵ Temp(⟨statement⟩)
Labin(⟨definite loop⟩) ⟵ Labin(⟨statement⟩)
| ⟨indefinite loop⟩
Code(⟨statement⟩) ⟵ Code(⟨indefinite loop⟩)
Labout(⟨statement⟩) ⟵ Labout(⟨indefinite loop⟩)
Temp(⟨indefinite loop⟩) ⟵ Temp(⟨statement⟩)
Labin(⟨indefinite loop⟩) ⟵ Labin(⟨statement⟩)

Input and output statements synthesize an Opcode value of 'GET' and 'PUT', respectively, and pass it to the constituent ⟨variable list⟩, where a sequence of GET or PUT instructions is generated:

⟨input statement⟩ ::= **read** ⟨variable list⟩
Code(⟨input statement⟩) ⟵ Code(⟨variable list⟩)
Opcode(⟨variable list⟩) ⟵ 'GET'
⟨output statement⟩ ::= **write** ⟨variable list⟩
Code(⟨output statement⟩) ⟵ Code(⟨variable list⟩)
Opcode(⟨variable list⟩) ⟵ 'PUT'
⟨variable list⟩ ::= ⟨variable⟩
Code(⟨variable list⟩) ⟵ ⟨(Opcode(⟨variable list⟩), Tag(⟨variable⟩))⟩
| ⟨variable list⟩$_2$, ⟨variable⟩
Code(⟨variable list⟩) ⟵ *append*(Code(⟨variable list⟩$_2$),
(Opcode(⟨variable list⟩), Tag(⟨variable⟩)))
Opcode(⟨variable list⟩$_2$) ⟵ Opcode(⟨variable list⟩)

In the code for an assignment statement, the variable at the beginning of the statement appears as the last operand:

⟨assignment statement⟩ ::= ⟨variable⟩ := ⟨expression⟩
Code(⟨assignment statement⟩) ⟵ *append*(Code(⟨expression⟩),
('STO', Tag(⟨variable⟩)))
Temp(⟨expression⟩) ⟵ Temp(⟨assignment statement⟩)

For a ⟨conditional statement⟩ of the form

if ⟨expression⟩$_1$ ⟨relation⟩ ⟨expression⟩$_2$ **then** ⟨series⟩ **fi**

a suitable pattern of code is

```
        code for ⟨expression⟩1
        STO  Tn
        code for ⟨expression⟩2
        SUB  Tn
        conditional jump to Lm
        code for ⟨series⟩
Lm      LAB
```

where m is the numeral whose value is one more than the current label count, n is the numeral whose value is one more than the current temporary count, and the identity of the conditional jump instruction depends on the ⟨relation⟩. To handle the latter feature, ⟨relation⟩ has Opcode as a synthesized attribute:

```
⟨relation⟩ ::= =
        Opcode(⟨relation⟩) ← 'JNP'
    | =<
        Opcode(⟨relation⟩) ← 'JN'
    | <
        Opcode(⟨relation⟩) ← 'JNZ'
    | >
        Opcode(⟨relation⟩) ← 'JPZ'
    | >=
        Opcode(⟨relation⟩) ← 'JP'
    | <>
        Opcode(⟨relation⟩) ← 'JZ'
```

Because it occurs in three different code patterns, the subpattern

```
code for ⟨expression⟩1
STO  Tn
code for ⟨expression⟩2
SUB  Tn
conditional jump to Lm
```

is synthesized by ⟨comparison⟩, which inherits the value of m via Labin:

```
⟨comparison⟩ ::= ⟨expression⟩1 ⟨relation⟩ ⟨expression⟩2
    Code(⟨comparison⟩) ← concat(
        Code(⟨expression⟩1),
        ⟨('STO', temporary(Temp(⟨comparison⟩) + 1))⟩,
        Code(⟨expression⟩2),
        ⟨('SUB', temporary(Temp(⟨comparison⟩) + 1))⟩,
        ⟨(Opcode(⟨relation⟩), label(Labin(⟨comparison⟩)))⟩)
    Temp(⟨expression⟩1) ← Temp(⟨comparison⟩) + 1
    Temp(⟨expression⟩2) ← Temp(⟨comparison⟩) + 1
```


Given a specified integer, the auxiliary functions *temporary* and *label* return Tag values of the form Tk and Lk, respectively, where k is the decimal representation of the integer. Now we may give the rules

```
⟨conditional statement⟩ ::= if ⟨comparison⟩ then ⟨series⟩ fi
    Code(⟨conditional statement⟩) ← concat(Code(⟨comparison⟩),
        Code(⟨series⟩), ⟨(label(Labin(⟨conditional statement⟩) + 1), 'LAB')⟩)
    Labout(⟨conditional statement⟩) ← Labout(⟨series⟩)
    Temp(⟨comparison⟩) ← Temp(⟨conditional statement⟩)
    Labin(⟨comparison⟩) ← Labin(⟨conditional statement⟩) + 1
    Temp(⟨series⟩) ← Temp(⟨conditional statement⟩)
    Labin(⟨series⟩) ← Labin(⟨conditional statement⟩) + 1
```

For a conditional statement of the form

```
if ⟨expression⟩1 ⟨relation⟩ ⟨expression⟩2 then ⟨series⟩1
    else ⟨series⟩2 fi
```

a suitable code pattern is

```
        code for ⟨expression⟩1
        STO  Tn
        code for ⟨expression⟩2
        SUB  Tn
        conditional jump to Lm
        code for ⟨series⟩1
        J    Lm+1
Lm      LAB
        code for ⟨series⟩2
Lm+1    LAB
```

Thus, taking into account the other necessary actions for attribute evaluation, we have

```
⟨conditional statement⟩ ::= if ⟨comparison⟩ then ⟨series⟩1
    else ⟨series⟩2 fi
        Code(⟨conditional statement⟩) ← concat(
```

Code(⟨comparison⟩),
Code(⟨series⟩$_1$),
⟨('J', *label*(Labin(⟨conditional statement⟩) + 2))⟩,
⟨(*label*(Labin(⟨conditional statement⟩) + 1), 'LAB')⟩,
Code(⟨series⟩$_2$),
⟨(*label*(Labin(⟨conditional statement⟩) + 2), 'LAB')⟩)
Labout(⟨conditional statement⟩) ⟵ Labout(⟨series⟩$_2$)
Temp(⟨comparison⟩) ⟵ Temp(⟨conditional statement⟩)
Labin(⟨comparison⟩) ⟵ Labin(⟨conditional statement⟩) + 1
Temp(⟨series⟩$_1$) ⟵ Temp(⟨conditional statement⟩)
Labin(⟨series⟩$_1$) ⟵ Labin(⟨conditional statement⟩) + 2
Temp(⟨series⟩$_2$) ⟵ Temp(⟨conditional statement⟩)
Labin(⟨series⟩$_2$) ⟵ Labout(⟨series⟩$_1$)

For a statement of the form

while ⟨expression⟩$_1$ ⟨relation⟩ ⟨expression⟩$_2$ **do** ⟨series⟩ **end**

the code pattern will be

```
Lm      LAB
        code for ⟨expression⟩1
        STO   Tn
        code for ⟨expression⟩2
        SUB   Tn
        conditional jump to Lm+1
        code for ⟨series⟩
        J     Lm
Lm+1    LAB
```

Hence we have

⟨indefinite loop⟩ ::= **while** ⟨comparison⟩ **do** ⟨series⟩ **end**
Code(⟨indefinite loop⟩) ⟵ *concat*(
⟨(*label*(Labin(⟨indefinite loop⟩) + 1), 'LAB')⟩,
Code(⟨comparison⟩),
Code(⟨series⟩),
⟨('J', *label*(Labin(⟨indefinite loop⟩) + 1))⟩,
⟨(*label*(Labin(⟨indefinite loop⟩) + 2), 'LAB')⟩)
Labout(⟨indefinite loop⟩) ⟵ Labout(⟨series⟩)
Temp(⟨comparison⟩) ⟵ Temp(⟨indefinite loop⟩)

Labin(⟨comparison⟩) ⟵ Labin(⟨indefinite loop⟩) + 2
Temp(⟨series⟩) ⟵ Temp(⟨indefinite loop⟩)
Labin(⟨series⟩) ⟵ Labin(⟨indefinite loop⟩) + 2

Finally, for a loop of the form

to ⟨expression⟩ **do** ⟨series⟩ **end**

a suitable code pattern is

```
        code for ⟨expression⟩
        STO   Tn
Lm      LAB
        LOAD  Tn
        SUB   1
        JN    Lm+1
        STO   Tn
        code for ⟨series⟩
        J     Lm
Lm+1    LAB
```

so that the following rules complete our definition of Pam:

⟨definite loop⟩ ::= **to** ⟨expression⟩ **do** ⟨series⟩ **end**
Code(⟨definite loop⟩) ⟵ *concat*(
Code(⟨expression⟩),
⟨('STO', *temporary*(Temp(⟨definite loop⟩) + 1))⟩,
⟨(*label*(Labin(⟨definite loop⟩) + 1), 'LAB')⟩,
⟨('LOAD', *temporary*(Temp(⟨definite loop⟩) + 1))⟩,
⟨('SUB', ⟨'1'⟩)⟩,
⟨('JN', *label*(Labin(⟨definite loop⟩) + 2))⟩,
⟨('STO', *temporary*(Temp(⟨definite loop⟩) + 1))⟩,
Code(⟨series⟩),
⟨('J', *label*(Labin(⟨definite loop⟩) + 1))⟩,
⟨(*label*(Labin(⟨definite loop⟩) + 2), 'LAB')⟩)
Labout(⟨definite loop⟩) ⟵ Labout(⟨series⟩)
Temp(⟨expression⟩) ⟵ Temp(⟨definite loop⟩) + 1
Temp(⟨series⟩) ⟵ Temp(⟨definite loop⟩) + 1
Labin(⟨series⟩) ⟵ Labin(⟨definite loop⟩) + 2

The complete specifications are collected together in Table 3.1. It may be noted that, unlike the previous attribute grammar for the syntax of Eva,

this grammar contains no conditions. If Pam had a non-context-free syntax, these would be present, along with extra attributes playing syntactic roles. The Eva grammar could be extended to include semantics using the general, translational approach taken in this section, but it would be necessary to employ a different object language.

TABLE 3.1 Attribute Grammar Mapping Pam into a Simple Symbolic Machine Language

Attributes and Values

Attribute	Values
Num	sequences of digits
Tag	sequences of letters and digits
Temp	non-negative integers
Labin	non-negative integers
Labout	non-negative integers
Opcode	'ADD', 'SUB', 'MULT', 'DIV', 'GET', 'PUT', 'JN', 'JP', 'JZ', 'JNZ', 'JPZ', 'JNP', 'J', 'LOAD', 'STO'
Code	sequences of values of the form 'HALT', (Opcode, Num), (Opcode, Tag), or (Tag, 'LAB')

Attributes Associated with Nonterminal Symbols

Nonterminal	Inherited attributes	Synthesized attributes
⟨program⟩	—	Code
⟨series⟩	Temp, Labin	Code, Labout
⟨statement⟩	Temp, Labin	Code, Labout
⟨input statement⟩	—	Code
⟨output statement⟩	—	Code
⟨variable list⟩	Opcode	Code
⟨assignment statement⟩	Temp	Code
⟨conditional statement⟩	Temp, Labin	Code, Labout
⟨definite loop⟩	Temp, Labin	Code, Labout
⟨indefinite loop⟩	Temp, Labin	Code, Labout
⟨comparison⟩	Temp, Labin	Code
⟨expression⟩	Temp	Code
⟨term⟩	Temp	Code
⟨element⟩	Temp	Code
⟨constant⟩	—	Num
⟨variable⟩	—	Tag
⟨relation⟩	—	Opcode
⟨weak operator⟩	—	Opcode
⟨strong operator⟩	—	Opcode
⟨digit⟩	—	Num

TABLE 3.1 (Continued)

Production and Attribute Evaluation Rules

⟨program⟩ ::= ⟨series⟩
- Code(⟨program⟩) ⟵ *append*(Code(⟨series⟩), 'HALT')
- Temp(⟨series⟩) ⟵ 0
- Labin(⟨series⟩) ⟵ 0

⟨series⟩ ::= ⟨statement⟩
- Code(⟨series⟩) ⟵ Code(⟨statement⟩)
- Labout(⟨series⟩) ⟵ Labout(⟨statement⟩)
- Temp(⟨statement⟩) ⟵ Temp(⟨series⟩)
- Labin(⟨statement⟩) ⟵ Labin(⟨series⟩)

| ⟨series⟩$_2$; ⟨statement⟩
- Code(⟨series⟩) ⟵ *concat*(Code(⟨series⟩$_2$), Code(⟨statement⟩))
- Labout(⟨series⟩) ⟵ Labout(⟨statement⟩)
- Temp(⟨series⟩$_2$) ⟵ Temp(⟨series⟩)
- Labin(⟨series⟩$_2$) ⟵ Labin(⟨series⟩)
- Temp(⟨statement⟩) ⟵ Temp(⟨series⟩)
- Labin(⟨statement⟩) ⟵ Labout(⟨series⟩$_2$)

⟨statement⟩ ::= ⟨input statement⟩
- Code(⟨statement⟩) ⟵ Code(⟨input statement⟩)
- Labout(⟨statement⟩) ⟵ Labin(⟨statement⟩)

| ⟨output statement⟩
- Code(⟨statement⟩) ⟵ Code(⟨output statement⟩)
- Labout(⟨statement⟩) ⟵ Labin(⟨statement⟩)

| ⟨assignment statement⟩
- Code(⟨statement⟩) ⟵ Code(⟨assignment statement⟩)
- Labout(⟨statement⟩) ⟵ Labin(⟨statement⟩)
- Temp(⟨assignment statement⟩) ⟵ Temp(⟨statement⟩)

| ⟨conditional statement⟩
- Code(⟨statement⟩) ⟵ Code(⟨conditional statement⟩)
- Labout(⟨statement⟩) ⟵ Labout(⟨conditional statement⟩)
- Temp(⟨conditional statement⟩) ⟵ Temp(⟨statement⟩)
- Labin(⟨conditional statement⟩) ⟵ Labin(⟨statement⟩)

| ⟨definite loop⟩
- Code(⟨statement⟩) ⟵ Code(⟨definite loop⟩)
- Labout(⟨statement⟩) ⟵ Labin(⟨statement⟩)

TABLE 3.1 (Continued)

```
            Temp(⟨definite loop⟩) ⟵ Temp(⟨statement⟩)
            Labin(⟨definite loop⟩) ⟵ Labin(⟨statement⟩)
    | ⟨indefinite loop⟩
            Code(⟨statement⟩) ⟵ Code(⟨indefinite loop⟩)
            Labout(⟨statement⟩) ⟵ Labout(⟨indefinite loop⟩)
            Temp(⟨indefinite loop⟩) ⟵ Temp(⟨statement⟩)
            Labin(⟨indefinite loop⟩) ⟵ Labin(⟨statement⟩)
⟨input statement⟩ ::= read ⟨variable list⟩
            Code(⟨input statement⟩) ⟵ Code(⟨variable list⟩)
            Opcode(⟨variable list⟩) ⟵ 'GET'
⟨output statement⟩ ::= write ⟨variable list⟩
            Code(⟨output statement⟩) ⟵ Code(⟨variable list⟩)
            Opcode(⟨variable list⟩) ⟵ 'PUT'
⟨variable list⟩ ::= ⟨variable⟩
            Code(⟨variable list⟩) ⟵ ⟨(Opcode(⟨variable list⟩), Tag(⟨variable⟩))⟩
    | ⟨variable list⟩₂ , ⟨variable⟩
            Code(⟨variable list⟩) ⟵ append(Code(⟨variable list⟩₂),
                                        (Opcode(⟨variable list⟩), Tag(⟨variable⟩)))
            Opcode(⟨variable list⟩₂) ⟵ Opcode(⟨variable list⟩)
⟨assignment statement⟩ ::= ⟨variable⟩ := ⟨expression⟩
            Code(⟨assignment statement⟩) ⟵ append(Code(⟨expression⟩),
                                                ('STO', Tag(⟨variable⟩)))
            Temp(⟨expression⟩) ⟵ Temp(⟨assignment statement⟩)
⟨conditional statement⟩ ::= if ⟨comparison⟩ then ⟨series⟩ fi
            Code(⟨conditional statement⟩) ⟵ concat(Code(⟨comparison⟩),
                Code(⟨series⟩), ⟨(label(Labin(⟨conditional statement⟩) + 1), 'LAB')⟩)
            Labout(⟨conditional statement⟩) ⟵ Labout(⟨series⟩)
            Temp(⟨comparison⟩) ⟵ Temp(⟨conditional statement⟩)
            Labin(⟨comparison⟩) ⟵ Labin(⟨conditional statement⟩) + 1
            Temp(⟨series⟩) ⟵ Temp(⟨conditional statement⟩)
            Labin(⟨series⟩) ⟵ Labin(⟨conditional statement⟩) + 1
    | if ⟨comparison⟩ then ⟨series⟩₁ else ⟨series⟩₂ fi
            Code(⟨conditional statement⟩) ⟵ concat(
                Code(⟨comparison⟩),
                Code(⟨series⟩₁),
```

TABLE 3.1 (Continued)

```
                ⟨('J', label(Labin(⟨conditional statement⟩) + 2))⟩,
                ⟨(label(Labin(⟨conditional statement⟩) + 1), 'LAB')⟩,
                Code(⟨series⟩₂),
                ⟨(label(Labin(⟨conditional statement⟩) + 2), 'LAB')⟩)
            Labout(⟨conditional statement⟩) ⟵ Labout(⟨series⟩₂)
            Temp(⟨comparison⟩) ⟵ Temp(⟨conditional statement⟩)
            Labin(⟨comparison⟩) ⟵ Labin(⟨conditional statement⟩) + 1
            Temp(⟨series⟩₁) ⟵ Temp(⟨conditional statement⟩)
            Labin(⟨series⟩₁) ⟵ Labin(⟨conditional statement⟩) + 2
            Temp(⟨series⟩₂) ⟵ Temp(⟨conditional statement⟩)
            Labin(⟨series⟩₂) ⟵ Labout(⟨series⟩₁)
⟨definite loop⟩ ::= to ⟨expression⟩ do ⟨series⟩ end
            Code(⟨definite loop⟩) ⟵ concat(
                Code(⟨expression⟩),
                ⟨('STO', temporary(Temp(⟨definite loop⟩) + 1))⟩,
                ⟨(label(Labin(⟨definite loop⟩) + 1), 'LAB')⟩,
                ⟨('LOAD', temporary(Temp(⟨definite loop⟩) + 1))⟩,
                ⟨('SUB', ⟨'1'⟩)⟩,
                ⟨('JN', label(Labin(⟨definite loop⟩) + 2))⟩,
                ⟨('STO', temporary(Temp(⟨definite loop⟩) + 1))⟩,
                Code(⟨series⟩),
                ⟨('J', label(Labin(⟨definite loop⟩) + 1))⟩,
                ⟨(label(Labin(⟨definite loop⟩) + 2), 'LAB')⟩)
            Labout(⟨definite loop⟩) ⟵ Labout(⟨series⟩)
            Temp(⟨expression⟩) ⟵ Temp(⟨definite loop⟩) + 1
            Temp(⟨series⟩) ⟵ Temp(⟨definite loop⟩) + 1
            Labin(⟨series⟩) ⟵ Labin(⟨definite loop⟩) + 2
⟨indefinite loop⟩ ::= while ⟨comparison⟩ do ⟨series⟩ end
            Code(⟨indefinite loop⟩) ⟵ concat(
                ⟨(label(Labin(⟨indefinite loop⟩) + 1), 'LAB')⟩,
                Code(⟨comparison⟩),
                Code(⟨series⟩),
                ⟨('J', label(Labin(⟨indefinite loop⟩) + 1))⟩,
                ⟨(label(Labin(⟨indefinite loop⟩) + 2), 'LAB')⟩)
            Labout(⟨indefinite loop⟩) ⟵ Labout(⟨series⟩)
```

TABLE 3.1 (Continued)

```
            Temp(⟨comparison⟩) ⟵ Temp(⟨indefinite loop⟩)
            Labin(⟨comparison⟩) ⟵ Labin(⟨indefinite loop⟩) + 2
            Temp(⟨series⟩) ⟵ Temp(⟨indefinite loop⟩)
            Labin(⟨series⟩) ⟵ Labin(⟨indefinite loop⟩) + 2
⟨comparison⟩ ::= ⟨expression⟩1 ⟨relation⟩ ⟨expression⟩2
            Code(⟨comparison⟩) ⟵ concat(
                Code(⟨expression⟩1),
                ⟨('STO', temporary(Temp(⟨comparison⟩) + 1))⟩,
                Code(⟨expression⟩2),
                ⟨('SUB', temporary(Temp(⟨comparison⟩) + 1))⟩,
                ⟨(Opcode(⟨relation⟩), label(Labin(⟨comparison⟩)))⟩)
            Temp(⟨expression⟩1) ⟵ Temp(⟨comparison⟩) + 1
            Temp(⟨expression⟩2) ⟵ Temp(⟨comparison⟩) + 1
⟨expression⟩ ::= ⟨term⟩
            Code(⟨expression⟩) ⟵ Code(⟨term⟩)
            Temp(⟨term⟩) ⟵ Temp(⟨expression⟩)
    | ⟨expression⟩2 ⟨weak operator⟩ ⟨term⟩
            Code(⟨expression⟩) ⟵ concat(Code(⟨expression⟩2),
                selectcode(Code(⟨term⟩), Temp(⟨expression⟩),
                                                Opcode(⟨weak operator⟩)))
            Temp(⟨expression⟩2) ⟵ Temp(⟨expression⟩)
            Temp(⟨term⟩) ⟵ Temp(⟨expression⟩) + 2
⟨term⟩ ::= ⟨element⟩
            Code(⟨term⟩) ⟵ Code(⟨element⟩)
            Temp(⟨element⟩) ⟵ Temp(⟨term⟩)
    | ⟨term⟩2 ⟨strong operator⟩ ⟨element⟩
            Code(⟨term⟩) ⟵ concat(Code(⟨term⟩2),
                selectcode(Code(⟨element⟩), Temp(⟨term⟩), Opcode(⟨strong operator⟩)))
            Temp(⟨term⟩2) ⟵ Temp(⟨term⟩)
            Temp(⟨element⟩) ⟵ Temp(⟨term⟩) + 2
⟨element⟩ ::= ⟨constant⟩
            Code(⟨element⟩) ⟵ ⟨('LOAD', Num(⟨constant⟩))⟩
    | ⟨variable⟩
            Code(⟨element⟩) ⟵ ⟨('LOAD', Tag(⟨variable⟩))⟩
    | ( ⟨expression⟩ )
```

TABLE 3.1 (Continued)

```
            Code(⟨element⟩) ⟵ Code(⟨expression⟩)
            Temp(⟨expression⟩) ⟵ Temp(⟨element⟩)
⟨constant⟩ ::= ⟨digit⟩
            Num(⟨constant⟩) ⟵ Num(⟨digit⟩)
    | ⟨constant⟩2 ⟨digit⟩
            Num(⟨constant⟩) ⟵ concat(Num(⟨constant⟩2), Num(⟨digit⟩))
⟨variable⟩ ::= ⟨letter⟩
            Tag(⟨variable⟩) ⟵ Tag(⟨letter⟩)
    | ⟨variable⟩2 ⟨letter⟩
            Tag(⟨variable⟩) ⟵ concat(Tag(⟨variable⟩2), Tag(⟨letter⟩))
    | ⟨variable⟩2 ⟨digit⟩
            Tag(⟨variable⟩) ⟵ concat(Tag(⟨variable⟩2), Num(⟨digit⟩))
⟨relation⟩ ::= =
            Opcode(⟨relation⟩) ⟵ 'JNP'
    | =<
            Opcode(⟨relation⟩) ⟵ 'JN'
    | <
            Opcode(⟨relation⟩) ⟵ 'JNZ'
    | >
            Opcode(⟨relation⟩) ⟵ 'JPZ'
    | >=
            Opcode(⟨relation⟩) ⟵ 'JP'
    | <>
            Opcode(⟨relation⟩) ⟵ 'JZ'
⟨weak operator⟩ ::= +
            Opcode(⟨weak operator⟩) ⟵ 'ADD'
    | -
            Opcode(⟨weak operator⟩) ⟵ 'SUB'
⟨strong operator⟩ ::= *
            Opcode(⟨strong operator⟩) ⟵ 'MULT'
    | /
            Opcode(⟨strong operator⟩) ⟵ 'DIV'
⟨digit⟩ ::= 0
            Num(⟨digit⟩) ⟵ ⟨'0'⟩
    | ...
```

TABLE 3.1 (Continued)

```
    ...
  | 9
        Num(⟨digit⟩) ⟵ ⟨'9'⟩
⟨letter⟩ ::= a
        Tag(⟨letter⟩) ⟵ ⟨'a'⟩
  | ...
    ...
  | z
        Tag(⟨letter⟩) ⟵ ⟨'z'⟩
```

Definition of Auxiliary Evaluation Functions

label (int) =
  concat (⟨'L'⟩, *string* (int))

temporary (int) =
  concat (⟨'T'⟩, *string* (int))

string (n) =
  ⟨'0'⟩, if $n = 0$;
  ...
  ⟨'9'⟩, if n = 9;
  concat (*string* (n ÷ 10), *string* (n mod 10)), otherwise.

selectcode (code, temp, opcode) =
  ⟨(opcode, $field_2$ (*first* (code)))⟩, if *length* (code) = 1;
  concat (⟨('STO', temp + 1)⟩, code, ⟨('STO, temp + 2)⟩, ⟨('LOAD', temp + 1)⟩,
    ⟨(opcode, temp + 2)⟩), otherwise.

EXERCISES

1. Verify that the grammar of Table 3.1 maps the expressions

$$(x + y * z) + b * c$$
$$a + (x + y * z)$$

   into the code sequences given earlier in this section.

2. Construct the complete, decorated tree for the program

read *a* ;
if *a* = 0 **then read** *b* ; **write** *b* **else write** *a* **fi**

3. If you are familiar with reverse Polish (Polish postfix) notation, determine what modifications to the grammar of Table 3.1 would be required in order to define the semantics of Pam in terms of a suitable reverse Polish language.

For Further Information

The use of attribute grammars for defining translations is described in more general terms by Lewis et al. (1974), Bochmann (1976), Cohen and Harry (1979), and Kennedy and Ramanathan (1979). Marcotty et al. (1976) take a rather different approach to the specification of semantics using attribute grammars.

The concept of defining semantics translationally appears in a variety of approaches that do not involve attribute grammars. Feldman (1966), for example, describes a Formal Semantic Language, a metalanguage for describing abstract translators, and its application to the problem of compiler generation. Landin (1965) formalizes some of the semantics of Algol 60 by defining a mapping from that language into a modified form of λ-notation.

3.3 INTERPRETIVE SEMANTICS USING TWO-LEVEL GRAMMARS

Using Pam and Eva as examples, this section examines an ingenious technique for formalizing semantics using the syntactic methods provided by two-level grammars. The philosophy on which this technique rests is that the "meaning" of a program should be described essentially in terms of the correspondence it defines between its input data and its output data and that a formal system therefore constitutes a complete semantic formalization if it generates triples consisting of (a) a syntactically valid program, (b) an input file, and (c) an output file such that the execution of the program with the input file produces the output file. Since input and output files are, like programs, basically sequences of characters, a program-input-output triple can be represented as a single character string of a form such as

⟨program⟩ **eof** ⟨input file⟩ **eof** ⟨output file⟩ **eof**

This suggests the possibility of constructing a grammar that generates all and only these sequences of symbols. In the case of Pam, for example, the grammar should imply a syntax tree for

read *x*, *y* ; **write** *y* **eof** −2. 7. 1. **eof** 7. **eof**

but not for

read *x*, *y* ; **write** *y* **eof** −2. 7. **eof** 3. **eof**