Chapter 3
ATTRIBUTE GRAMMARS

In Chapter 1 we discussed the hierarchy of formal grammars proposed by Noam Chomsky. We mentioned that context-sensitive conditions, such as ensuring the same value for n in a string $a^n b^n c^n$, cannot be tested using a context-free grammar. Although we showed a context-sensitive grammar for this particular problem, these grammars in general are impractical for specifying the context conditions for a programming language. In this chapter and the next we investigate two different techniques for augmenting a context-free grammar in order to verify context-sensitive conditions.

Attribute grammars can perform several useful functions in specifying the syntax and semantics of a programming language. An attribute grammar can be used to specify the context-sensitive aspects of the syntax of a language, such as checking that an item has been declared and that the use of the item is consistent with its declaration. As we will see in Chapter 7, attribute grammars can also be used in specifying an operational semantics of a programming language by defining a translation into lower-level code based on a specific machine architecture.

Attribute grammars were first developed by Donald Knuth in 1968 as a means of formalizing the semantics of a context-free language. Since their primary application has been in compiler writing, they are a tool mostly used by programming language implementers. In the first section, we use examples to introduce attribute grammars. We then provide a formal definition for an attribute grammar followed by additional examples. Next we develop an attribute grammar for Wren that is sensitive to the context conditions discussed in Chapter 1 (see Figure 1.11). Finally, as a laboratory activity, we develop a context-sensitive parser for Wren.

3.1 CONCEPTS AND EXAMPLES

An attribute grammar may be informally defined as a context-free grammar that has been extended to provide context sensitivity using a set of attributes, assignment of attribute values, evaluation rules, and conditions. A finite, possibly empty set of attributes is associated with each distinct symbol in the grammar. Each attribute has an associated domain of values, such as

integers, character and string values, or more complex structures. Viewing the input sentence (or program) as a parse tree, attribute grammars can pass values from a node to its parent, using a synthesized attribute, or from the current node to a child, using an inherited attribute. In addition to passing attribute values up or down the parse tree, the attribute values may be assigned, modified, and checked at any node in the derivation tree. The following examples should clarify some of these points.

Examples of Attribute Grammars

We will attempt to write a grammar to recognize sentences of the form $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$. The sentences **aaabbbccc** and **abc** belong to this grammar but the sentences **aaabbbbcc** and **aabbbcc** do not. Consider this first attempt to describe the language using a context-free grammar:

<letter sequence> ::= <a sequence> <b sequence> <c sequence>

<asequence> ::= **a** | <a sequence>**a**

<bsequence> ::= **b** | <bsequence> **b**

<csequence> ::= **c** | <csequence> **c**

As seen in Figure 3.1, this grammar can generate the string **aaabbbccc** . It can also generate the string **aaabbbbcc** , as seen in Figure 3.2.

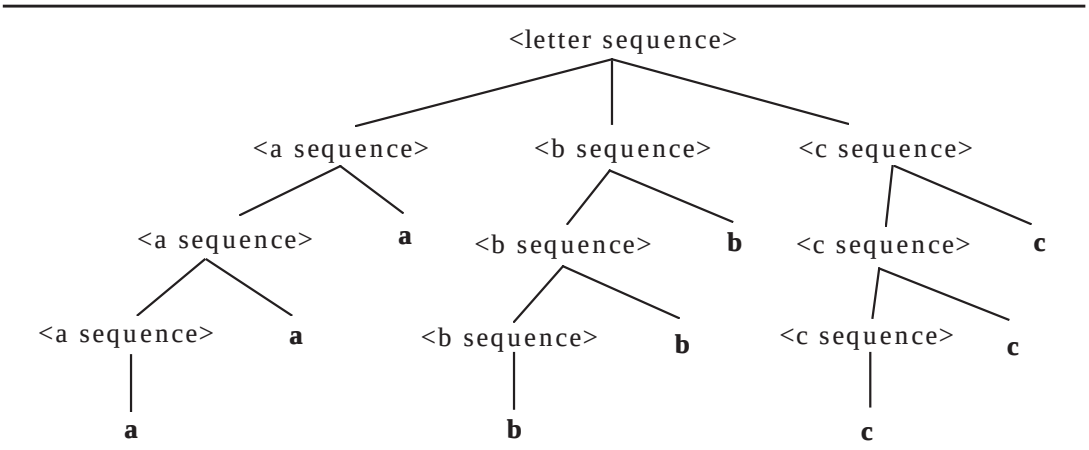

Figure 3.1: Parse Tree for the String **aaabbbccc**

As has already been noted in Chapter 1, it is impossible to write a context-free grammar to generate only those sentences of the form $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$. However, it is possible to write a context-sensitive grammar for sentences of this form. Attribute grammars provide another approach for defining context-sensitiv-

ity. If we augment our grammar with an attribute describing the length of aletter sequence, we can use these values to ensur e that the sequences of **a**'s, **b**'s, and **c**'s all have the same length.

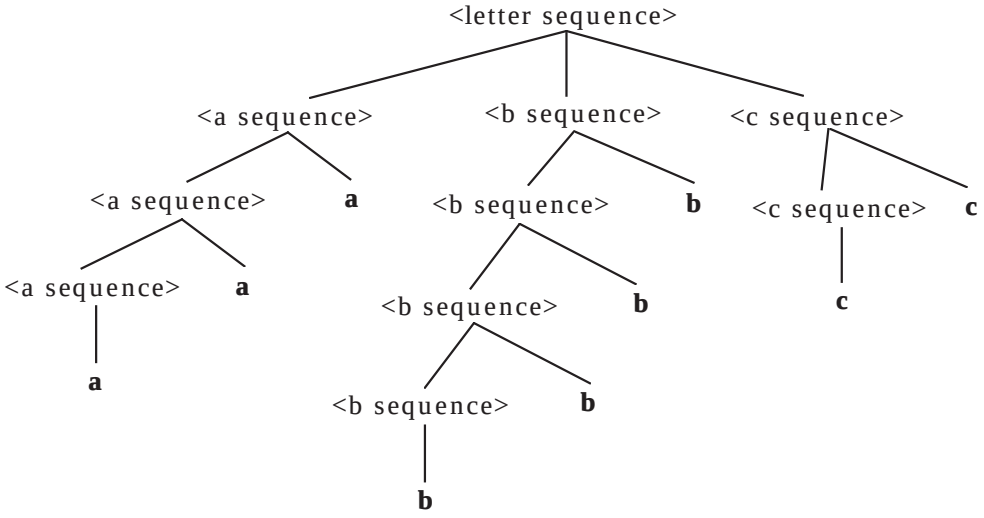

Figure 3.2: Parse Tree for the String **aaabbbbcc**

The first solution involves a synthesized attribute *Size* that is associated with the nonterminals <asequence>, <bsequence>, and <csequence>. W e add the condition that, at the root of the tree, the *Size* attribute for each of the letter sequences has the same value. If a character sequence consists of a single character, *Size* is set to 1; if it consists of a character sequence followed by a single character, *Size* for the parent character sequence is the *Size* of the child character sequence plus one. We have added the necessary attribute assignments and conditions to the grammar shown below. Notice that we differentiate a parent sequence from a child sequence by adding subscripts to the nonterminal symbols.

<lettersequence> ::= <asequence> <bsequence> <csequence>
condition :
Size (<asequence>) = *Size* (<bsequence>) = *Size* (<csequence>)

<asequence> ::= **a**
Size (<asequence>) ← 1
| <asequence> $_2$ **a**
Size (<asequence>) ← *Size* (<asequence> $_2$) + 1

<bsequence> ::= **b**
Size (<bsequence>) ← 1
| <bsequence> $_2$ **b**
Size (<bsequence>) ← *Size* (<bsequence> $_2$) + 1

<csequence> ::= **c**
Size (<csequence>) ← 1
| <csequence> $_2$ **c**
Size (<csequence>) ←*Size* (<csequence> $_2$) + 1

This attribute grammar successfully parses the sequence **aaabbbccc** since the sequence obeys the BNF and satisfies all conditions in the attribute grammar. The complete, decorated parse tree is shown in Figure 3.3.

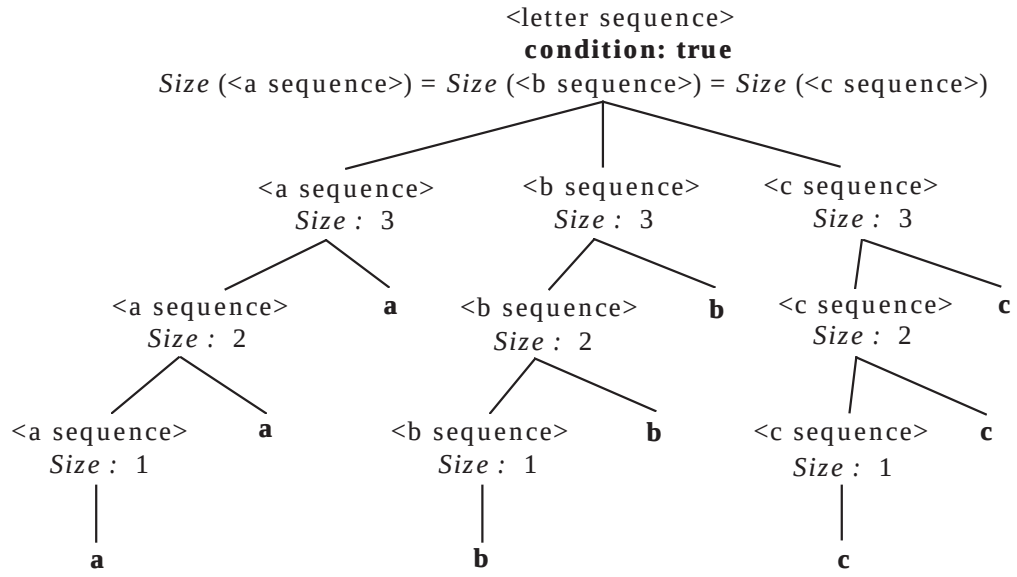

Figure 3.3: Parse Tree for **aaabbbccc** Using Synthesized Attributes

On the other hand, this attribute grammar cannot parse the sequence **aaabbbbcc** . Although this sequence satisfies the BNF part of the grammar, it does not satisfy the condition required of the attribute values, as shown in Figure 3.4.

When using only synthesized attributes, all of the relevant information is passed up to the root of the parse tree where the checking takes place. However, it is often more convenient to pass information up from one part of a tree, transfer it at some specified node, and then have it inherited down into other parts of the tree.

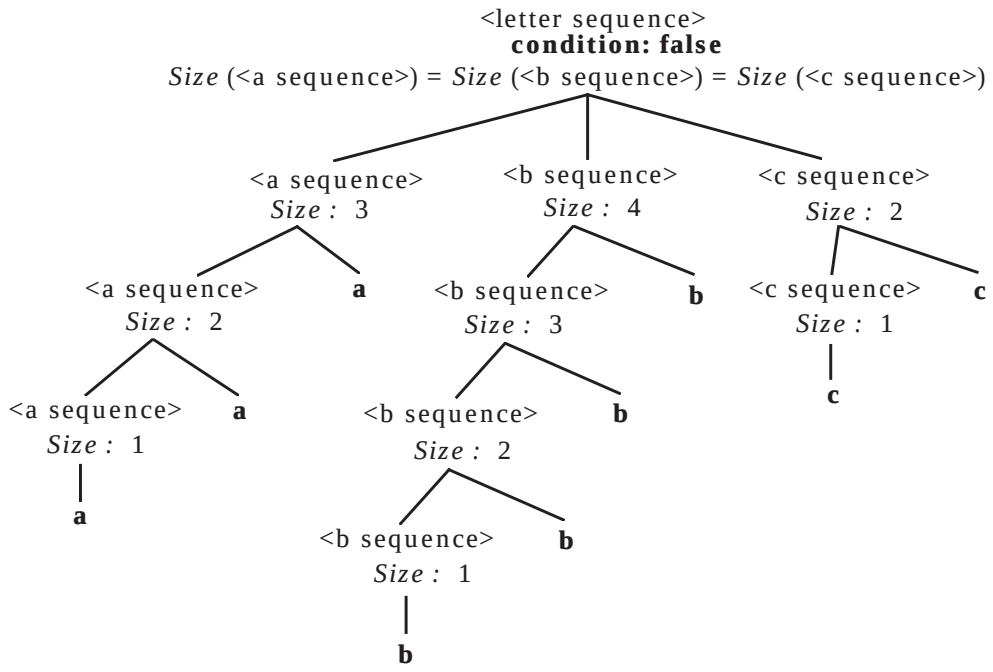

Figure 3.4: Parse Tree for **aaabbbbcc** Using Synthesized Attributes

Reconsider the problem of recognizing sequences of the form $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$. In this solution, we use the attribute *Size* as a synthesized attribute for the sequence of **a**'s and *InhSize* as inherited attributes for the sequences of **b**'s and **c**'s. As we have already seen, we can synthesize the size of the sequence of **a**'s to the root of the parse tree. In this solution we set the *InhSize* attribute for the **b** sequence and the **c** sequence to this value and inherit it down the tree, decrementing the value by one every time we see another character in the sequence. When we reach the node where the sequence has a child consisting of a single character, we check if the inherited *InhSize* attribute equals one. If so, the size of the sequence must be the same as the size of the sequences of **a**'s; otherwise, the two sizes do not match and the parse is unsuccessful. These ideas are expressed in the following attribute grammar:

<lettersequence> ::= <asequence> <bsequence> <csequence>
InhSize (<bsequence>) ← *Size* (<asequence>)
InhSize (<csequence>) ← *Size* (<asequence>)

<asequence> ::= **a**
Size (<asequence>) ← 1
| <asequence> $_2$ **a**
Size (<asequence>) ← *Size* (<asequence> $_2$) + 1

<bsequence> ::= **b**
condition: *InhSize* (<bsequence>) = 1
| <bsequence> $_2$ **b**
InhSize (<bsequence> $_2$) ← *InhSize* (<bsequence>) – 1

<csequence> ::= **c**
condition: *InhSize* (<csequence>) = 1
| <csequence> $_2$ **c**
InhSize (<csequence> $_2$) ← *InhSize* (<csequence>) – 1

For the nonterminal <asequence>, *Size* is a synthesized attribute, as we can see from the attribute assignment

Size (<asequence>) ← *Size* (<asequence> $_2$) + 1.

Here the value of the child is incremented by one and passed to the parent. For the nonterminals <bsequence> and <csequence>, *InhSize* is an inherited attribute that is passed from parent to child. The assignment

InhSize (<bsequence> $_2$) ← *InhSize* (<bsequence>) – 1

shows that the value is decremented by one each time it is passed from the parent sequence to the child sequence. When the sequence is a single character, we check that the inherited size attribute value is one. Figure 3.5 shows a decorated attribute parse tree for the sequence **aaabbbccc** , which satisfies the attribute grammar since it satisfies the BNF and all attribute conditions are true. *Size* is synthesized up the left branch, passed over to the center and right branches at the root, inherited down the center branch, and inherited down the right branch as *InhSize*.

As before, we demonstrate that the attribute grammar cannot parse the sequence **aaabbbbcc** . Although this sequence satisfies the BNF part of the grammar, it does not satisfy all conditions associated with attribute values, as shown in Figure 3.6. In this case, the parse fails on two conditions. It only takes one false condition anywhere in the decorated parse tree to make the parse fail.

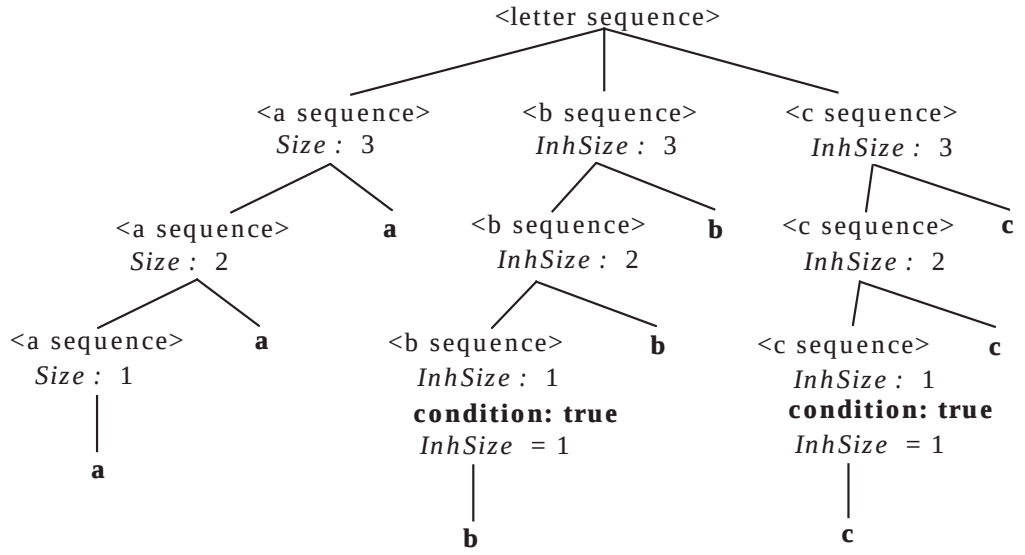

Figure 3.5: Parse Tree for **aaabbbccc** Using Inherited Attributes

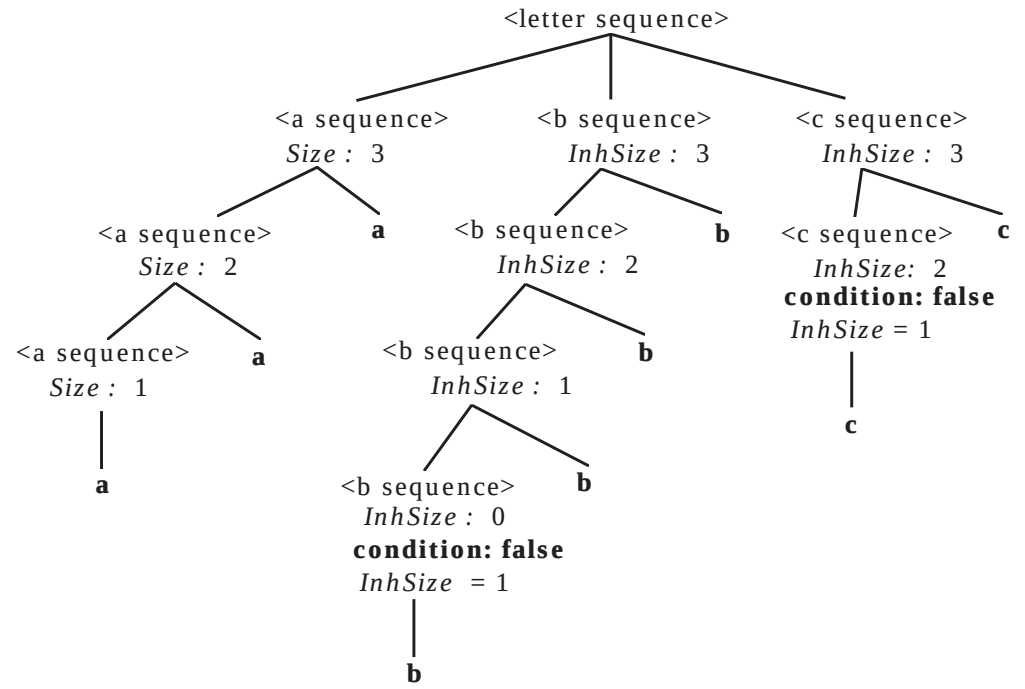

Figure 3.6: Parse Tree for **aaabbbbcc** Using Inherited Attributes

In this grammar the sequence of **a**'s determines the "desired" length against which the other sequences are checked. Consider the sequence **aabbbccc** . It might be argued that the sequence of **a**'s is "at fault" and not the other two sequences. However, in a programming language with declarations, we use the declarations to determine the "desired" types against which the remainder of the program is checked. The declaration information is synthesized up to the root of the tree and passed into the entire program for checking. Using this approach makes it easier to localize errors that cause the parse to fail. Also, if both synthesized and inherited attributes are used, an attribute value may be threaded throughout a tree. We will see this mechanism in Chapter 7 when an attribute grammar is used to help determine label names in the generation of code. Before developing the complete attribute grammar for Wren, we provide some formal definitions associated with attribute grammars and examine one more example where attributes are used to determine the semantics of binary numerals.

Formal Definitions

Although the above examples were introduced in an informal way, attribute grammars furnish a formal mechanism for specifying a context-sensitive grammar, as indicated by the following definitions.

Definition : An **attribute grammar** is a context-free grammar augmented with attributes, semantic rules, and conditions.

Let G = <N,Σ,P,S> be a context-free grammar (see Chapter 1).
Write a production p ∈ P in the form:

$$p:\ X_0 ::= X_1 X_2 \ldots X_{n_p}$$

where $n_p \geq 1$, $X_0 \in N$ and $X_k \in N \cup \Sigma$ for $1 \leq k \leq n_p$.

A derivation tree for a sentence in a context-free language, as defined in Chapter 1, has the property that each of its leaf nodes is labeled with a symbol from Σ and each interior node t corresponds to a production p ∈ P such that t is labeled with X_0 and t has n_p children labeled with X_1, X_2, ..., X_{n_p} in left-to-right order.

For each syntactic category X ∈ N in the grammar, there are two finite disjoint sets I(X) and S(X) of **inherited** and **synthesized attributes** . For X = S, the start symbol, I(X) = ∅.

Let A(X) = I(X) ∪ S(X) be the set of attributes of X. Each attribute *Atb* ∈ A(X) takes a value from some semantic domain (such as the integers, strings of characters, or structures of some type) associated with that attribute. These values are defined by **semantic functions** or **semantic rules** associated with the productions in P.

Consider again a production p ∈ P of the form $X_0 ::= X_1 X_2 \ldots X_{n_p}$ Each synthesized attribute *Atb* ∈ $S(X_0)$ has its value defined in terms of the at-

tributes in $A(X_1) \cup A(X_2) \cup \ldots \cup A(X_{n_p}) \cup I(X_0)$. Each inherited attribute $Atb \in I(X_k)$ for $1 \leq k \leq n_p$ has its value defined in terms of the attributes in $A(X_0) \cup S(X_1) \cup S(X_2) \cup \ldots \cup S(X_{n_p})$.

Each production may also have a set of conditions on the values of the attributes in $A(X_0) \cup A(X_1) \cup A(X_2) \cup \ldots \cup A(X_{n_p})$ that further constrain an application of the production. The derivation (or parse) of a sentence in the attribute grammar is satisfied if and only if the context-free grammar is satisfied and all conditions are true. The semantics of a nonterminal can be considered to be a distinguished attribute evaluated at the root node of the derivation tree of that nonterminal. ∎

Semantics via Attribute Grammars

We illustrate the use of attribute grammars to specify meaning by developing the semantics of binary numerals. A binary numeral is a sequence of binary digits followed by a binary point (a period) and another sequence of binary digits—for example, 100.001 and 0.001101. For simplicity, we require at least one binary digit, which may be 0, for each sequence of binary digits. It is possible to relax this assumption—for example 101 or .11—but this flexibility adds to the complexity of the grammar without altering the semantics of binary numerals. Therefore we leave this modification as an exercise. We define the semantics of a binary numeral to be the real number value *Val* associated with the numeral, expressed in base-ten notation. For example, the semantics of the numeral 100.001 is 4.125.

The first version of an attribute grammar defining the meaning of binary numerals involves only synthesized attributes.

Nonterminals	Synthesized Attributes	Inherited Attributes
<binary numeral>	*Val*	—
<binary digits>	*Val, Len*	—
<bit>	*Val*	—

<binary numeral> ::= <binary digits>$_1$. <binary digits>$_2$
Val (<binary numeral>) ← *Val* (<binary digits>$_1$) +
Val (<binary digits>$_2$) / $2^{Len\ (<\text{binary digits}>_2)}$

<binary digits> ::=
<binary digits>$_2$ <bit>
Val (<binary digits>) ← 2 • *Val* (<binary digits>$_2$) + *Val* (<bit>)
Len (<binary digits>) ← *Len* (<binary digits>$_2$) + 1
| <bit>
Val (<binary digits>) ← *Val* (<bit>)
Len (<binary digits>) ← 1

<bit> ::=

0

Val (<bit>) ← 0

| **1**

Val (<bit>) ← 1

The derivation tree in Figure 3.7 illustrates the use of attributes that give the semantics for the binary numeral 1101.01 to be the real number 13.25.

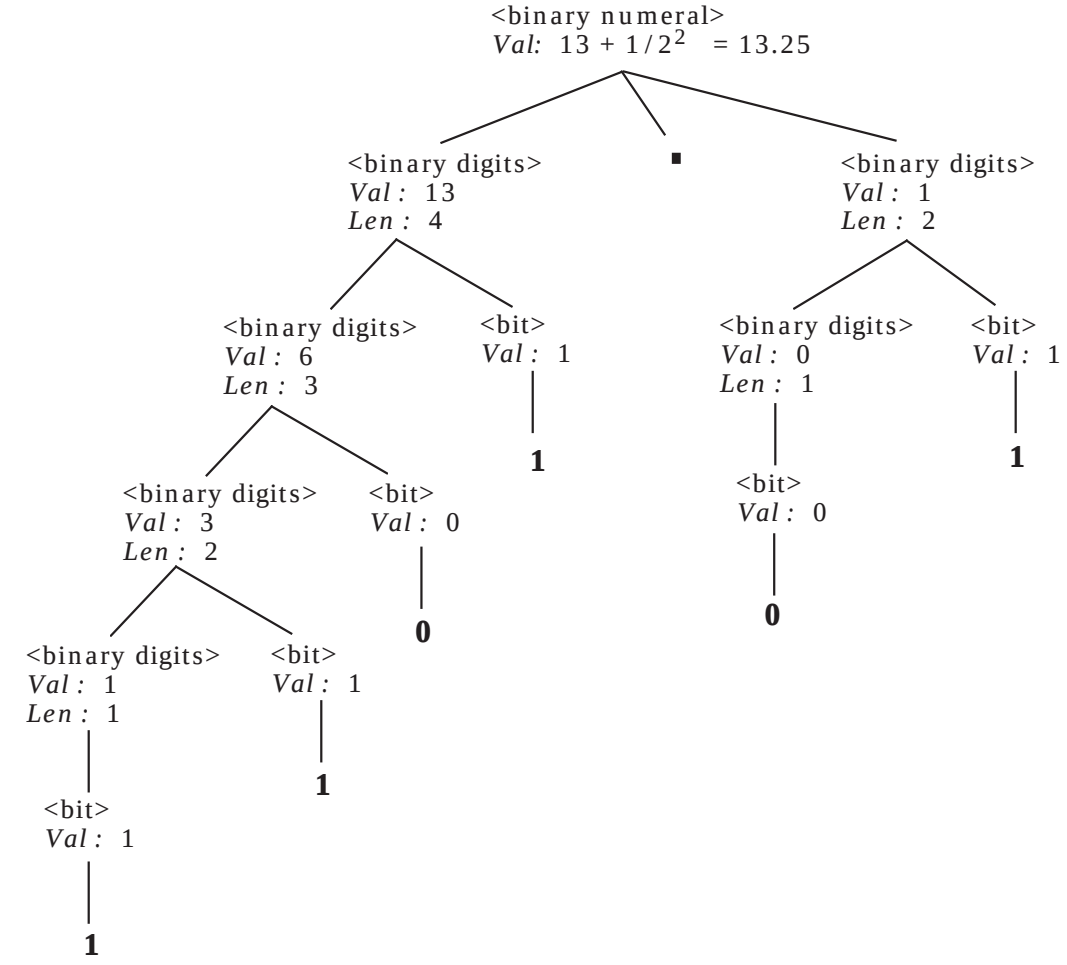

Figure 3.7: Binary Numeral Semantics Using Synthesized Attributes

The previous specification for the semantics of binary numerals was not based on positional information. As a result, the attribute values below the root do not represent the semantic meaning of the digits at the leaves. We now present an approach based on positional semantics, illustrated first in base 10,

$$123.45 = 1 \bullet 10^2 + 2 \bullet 10^1 + 3 \bullet 10^0 + 4 \bullet 10^{-1} + 5 \bullet 10^{-2}$$

and then in base 2,

$$\begin{aligned} 110.101 &= 1 \bullet 2^2 + 1 \bullet 2^1 + 0 \bullet 2^0 + 1 \bullet 2^{-1} + 0 \bullet 2^{-2} + 1 \bullet 2^{-3} \\ &= 6.625 \text{ (base 10).} \end{aligned}$$

We develop a positional semantics in which an inherited attribute called *Pos* is introduced. It is convenient to separate the sequence of binary digits to the left of the binary point, identified by the nonterminal <binary digits>, from the fractional binary digits to the right of the binary point, identified by the nonterminal <fraction digits>.

Nonterminals	Synthesized Attributes	Inherited Attributes
<binary numeral>	*Val*	—
<binary digits>	*Val*	*Pos*
<fraction digits>	*Val, Len*	—
<bit>	*Val*	*Pos*

We write our grammar in left recursive form, which means that the leftmost binary digit in a sequence of digits is "at the bottom" of the parse tree, as shown in Figure 3.7. For the binary digits to the left of the binary point, we initialize the *Pos* attribute to zero and increment it by one as we go down the tree structure. This technique provides the correct positional information for the binary digits in the integer part, but a different approach is needed for the fractional binary digits since the exponents from left to right are -1, -2, -3, ... Notice that this exponent information can be derived from the length of the binary sequence of digits from the binary point up to, and including, the digit itself. Therefore we add a length attribute for fractional digits that is transformed into a positional attribute for the individual bit. Notice that the *Val* attribute at any point in the tree contains the absolute value for the portion of the binary numeral in that subtree. Therefore the value of a parent node is the sum of the values for the children nodes. These ideas are implemented in the following attribute grammar:

<binary numeral> ::= <binary digits> **.** <fraction digits>

Val (<binary numeral>) ← *Val* (<binary digits>)+*Val* (<fraction digits>)

Pos (<binary digits>) ← 0

<binary digits> ::=
    <binary digits>$_2$ <bit>
        Val (<binary digits>) ← *Val* (<binary digits>$_2$) + *Val* (<bit>)
        Pos (<binary digits>$_2$) ← *Pos* (<binary digits>) + 1
        Pos (<bit>) ← *Pos* (<binary digits>)
    | <bit>
        Val (<binary digits>) ← *Val* (<bit>)
        Pos (<bit>) ← *Pos* (<binary digits>)

<fraction digits> ::=
    <fraction digits>$_2$ <bit>
        Val (<fraction digits>) ← *Val* (<fraction digits>$_2$) + *Val* (<bit>)
        Len (<fraction digits>) ← *Len* (<fraction digits>$_2$) + 1
        Pos (<bit>) ← - *Len* (<fraction digits>)
    | <bit>
        Val (<fraction digits>) ← *Val* (<bit>)
        Len (<fraction digits>) ← 1
        Pos (<bit>) ← - 1

<bit> ::=
    0
        Val (<bit>) ← 0
    | **1**
        Val (<bit>) ← $2^{Pos\ (<bit>)}$

The parse tree in Figure 3.8 illustrates the use of positional attributes to generate the semantics of the binary numeral 110.101 to be the real number 6.625.

The two attribute grammars for binary numerals do not involve conditions. If we limit the size of binary numerals to match a particular machine architecture, conditionals can be introduced to ensure that the binary numerals are of proper size. Actually, this situation is fairly complex since real number representations in most computers are based on scientific notation, not the fractional notation that has been illustrated above. We examine this problem of checking the size of binary numerals in the exercises.

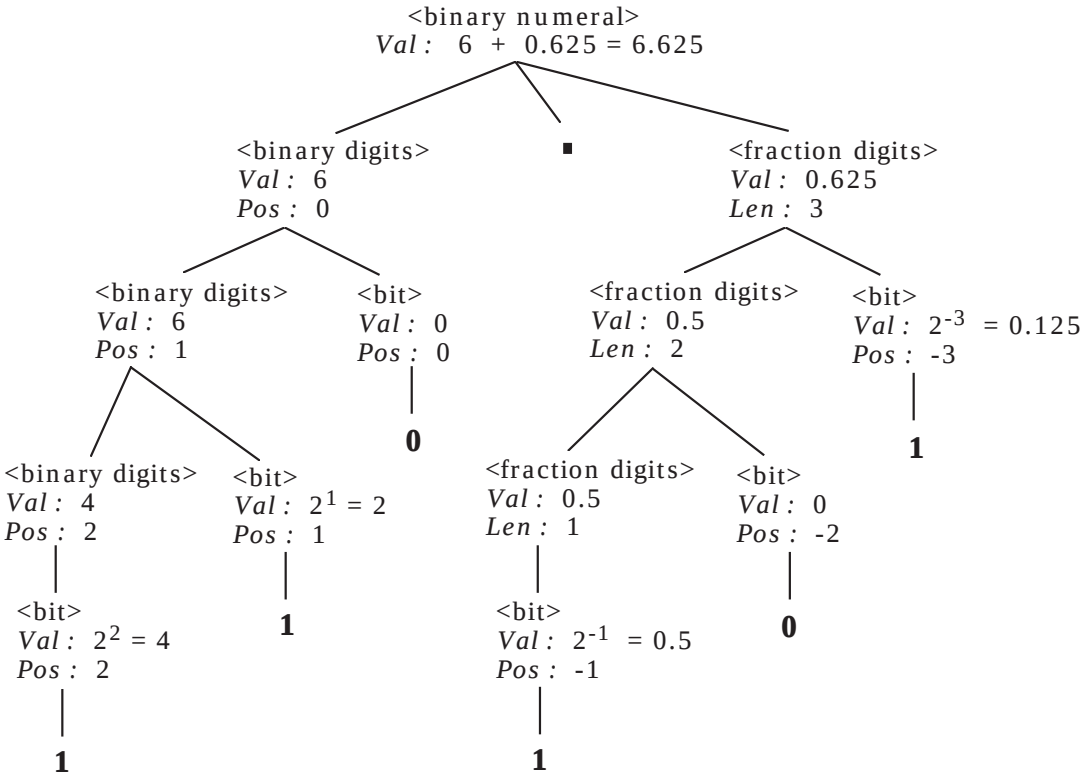

Figure 3.8: Binary Numeral Semantics Using Positional Attributes

Exercises

1. In old versions of Fortran that did not have the character data type, character strings were expressed in the following format:

   <string literal> ::= <numeral> H <string>

   where the <numeral> is a base-ten integer (≥ 1), H is a keyword (named after Herman Hollerith), and <string> is a sequence of characters. The semantics of this string literal is correct if the numeric value of the base-ten numeral matches the length of the string. Write an attribute grammar using only synthesized attributes for the nonterminals in the definition of <string literal>.

2. Repeat exercise 1, using a synthesized attribute for <numeral> and an inherited attribute for <string>.

3. Repeat exercise 1, using an inherited attribute for <numeral> and a synthesized attribute for <string>.

4. The following BNF specification defines the language of Roman numerals less than 1000:

   <roman> ::= <hundreds> <tens> <units>

   <hundreds> ::= <low hundreds> | **CD** | **D** <low hundreds> | **CM**

   <low hundreds> ::= ε | <low hundreds> **C**

   <tens> ::= <low tens> | **XL** | **L** <low tens> | **XC**

   <low tens> ::= ε | <low tens> **X**

   <units> ::= <low units> | **IV** | **V** <low units> | **IX**

   <low units> ::= ε | <low units> **I**

   Define attributes for this grammar to carry out two tasks:

   a) Restrict the number of X's in <low tens>, the I's in <low units>, and the C's in <low hundreds> to no more than three.

   b) Provide an attribute for <roman> that gives the decimal value of the Roman numeral being defined.

   Define any other attributes needed for these tasks, but do not change the BNF grammar.

5. Expand the binary numeral attribute grammar (either version) to allow for binary numerals with no binary point (1101), binary fractions with no fraction part (101.), and binary fractions with no whole number part (.101).

6. Develop an attribute grammar for integers that allows a leading sign character (+ or -) and that ensures that the value of the integer does not exceed the capacity of the machine. Assume a two's complement representation; if the word-size is n bits, the values range from -2^{n-1} to $2^{n-1}-1$.

7. Develop an attribute grammar for binary numerals that represents signed integers using two's complement. Assume that a word-size attribute is inherited by the two's complement binary numeral. The meaning of the binary numeral should be present at the root of the tree.

8. Assume that we have a 32-bit machine where real numbers are represented in scientific notation with a 24-bit mantissa and an 8-bit exponent with 2 as the base. Both mantissa and exponent are two's complement binary numerals. Using the results from exercise 7, write an attribute grammar for <binary real number> where the meaning of the binary numeral is at the root of the tree in base-10 notation—for example, $0.5 \bullet 2^5$.

9. Assuming that we allow the left side of a binary fraction to be left recursive and the fractional part to be right recursive, simplify the positional attribute grammar for binary fractions.

10. Consider a language of expressions with only the variables a, b, and c and formed using the binary infix operators

    +, –, *, /, and ↑ (for exponentiation)

    where ↑ has the highest precedence, * and / have the same next lower precedence, and + and – have the lowest precedence. ↑ is to be right associative and the other operations are to be left associative. Parentheses may be used to override these rules. Provide a BNF specification of this language of expressions. Add attributes to your BNF specification so that the following (unusual) conditions are satisfied by every valid expression accepted by the attribute grammar:

    a) The maximum depth of parenthesis nesting is three.

    b) No valid expression has more than eight applications of operators.

    c) If an expression has more divisions than multiplications, then subtractions are forbidden.

11. A binary tree consists of a root containing a value that is an integer, a (possibly empty) left subtree, and a (possibly empty) right subtree. Such a binary tree can be represented by a triple (Left subtree, Root, Right subtree). Let the symbol nil denote an empty tree. Examples of binary trees include:

    (nil,13,nil)
    represents a tree with one node labeled with the value 13.

    ((nil,3,nil),8,nil)
    represents a tree with 8 at the root, an empty right subtree, and a nonempty left subtree with root labeled by 3 and empty subtrees.

    The following BNF specification describes this representation of binary trees.

    <binary tree> ::= nil | (<binary tree> <value> <binary tree>)

    <value> ::= <digit> | <value> <digit>

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

    Augment this grammar with attributes that carry out the following tasks:

    a) A binary tree is balanced if the heights of the subtrees at each interior node are within one of each other. Accept only balanced binary trees.

    b) A binary search tree is a binary tree with the property that all the values in the left subtree of any node N are less than the value at N, and all the value in the right subtree of N are greater than or equal to the value at node N. Accept only binary search trees.

3.2 AN ATTRIBUTE GRAMMAR FOR WREN

In this section we develop an attribute grammar for Wren that performs context checking that is the same as that done by a compiler. We concentrate on context-sensitive conditions for programs that obey the BNF of Wren, as summarized in Figure 1.11.

Wren, as we have defined it, is a flat language in the sense that there is only one block in a program. As a consequence, all declarations belong to a single declaration sequence at the main program level. In the exercises we extend Wren and investigate nested blocks, but for the moment we concentrate on developing an attribute grammar for our current version of Wren. It should be noted that there is one small exception to our single set of declarations: The program name itself is not part of the block structure. It is a language design decision whether an object can have the same name as the program name; at this point we have elected to require that the program name be unique and not be used elsewhere in the program.

The Symbol Table

We build our declaration information in an attribute called *Symbol-table*. This attribute is synthesized from the declaration sequence and inherited into the command sequence of a program. The attribute value is transferred at the block level, at which time the program name is added to the *Symbol-table* attribute. *Symbol-table* contains a set of pairs each associating a name with a type. All variables are of type *integer* or *boolean*, and we introduce a pseudo-type, called *program*, for the program name identifier, and a default value *undefined* to represent the absence of a type. Since all declarations in our current version of Wren are global, there is a single *Symbol-table* that is passed down to the command sequence. We will develop a number of utility operations to manipulate the *Symbol-table* attribute.

Since variable names and types cannot magically appear in the symbol table attribute at the internal nodes of our parse tree, all of this information must be synthesized into the tree using attributes such as *Name*, *Type*, and *Var-list*. Figure 3.9 contains a complete list of the attributes and associated value types. We have added the pseudo-type value of *program* to the attribute *Type* so that the program name is uniquely identified. A *Name* value is a string of one or more letters or digits. A *Var-list* value is a sequence of *Name* values. The *Symbol-table* attribute consists of a set of pairs containing a name and a type. The nonterminals and their associated attributes for the grammar are listed in Figure 3.10. Next we introduce our attribute grammar rules and associated conditions by first focusing on the declaration portion of a Wren program.

Attribute	Value Types
Type	{ *integer*, *boolean*, *program*, *undefined* }
Name	String of letters or digits
Var-list	Sequence of Name values
Symbol-table	Set of pairs of the form [Name, Type]

Figure 3.9: Attributes and Values

Nonterminals	Synthesized Attributes	Inherited Attributes
<block>	—	*Symbol-table*
<declarationsequence>	*Symbol-table*	—
<declaration>	*Symbol-table*	—
<variable list>	*Var-list*	—
<type>	*Type*	—
<commandsequence>	—	*Symbol-table*
<command>	—	*Symbol-table*
<expr>	—	*Symbol-table*, *Type*
<integer expr>	—	*Symbol-table*, *Type*
<term>	—	*Symbol-table*, *Type*
<element>	—	*Symbol-table*, *Type*
<boolean expr>	—	*Symbol-table*, *Type*
<boolean term>	—	*Symbol-table*, *Type*
<boolean element>	—	*Symbol-table*, *Type*
<comparison>	—	*Symbol-table*
<variable>	*Name*	—
<identifier>	*Name*	—
<letter>	*Name*	—
<digit>	*Name*	—

Figure 3.10: Attributes Associated with Nonterminal Symbols

Consider the short program fragment:

```
program p is
    var x, y : integer;
    var a : boolean;
begin
    :
end
```

The *Symbol-table* attribute value passed to the command sequence will be

[[‘p’, *program*], [‘x’, *integer*], [‘y’, *integer*], [‘a’, *boolean*]].

We have chosen to use list-like notation for both sets and sequences; however, we assume no significance for the ordering in the case of sets. The decorated parse tree for this program fragment appears in Figure 3.11.

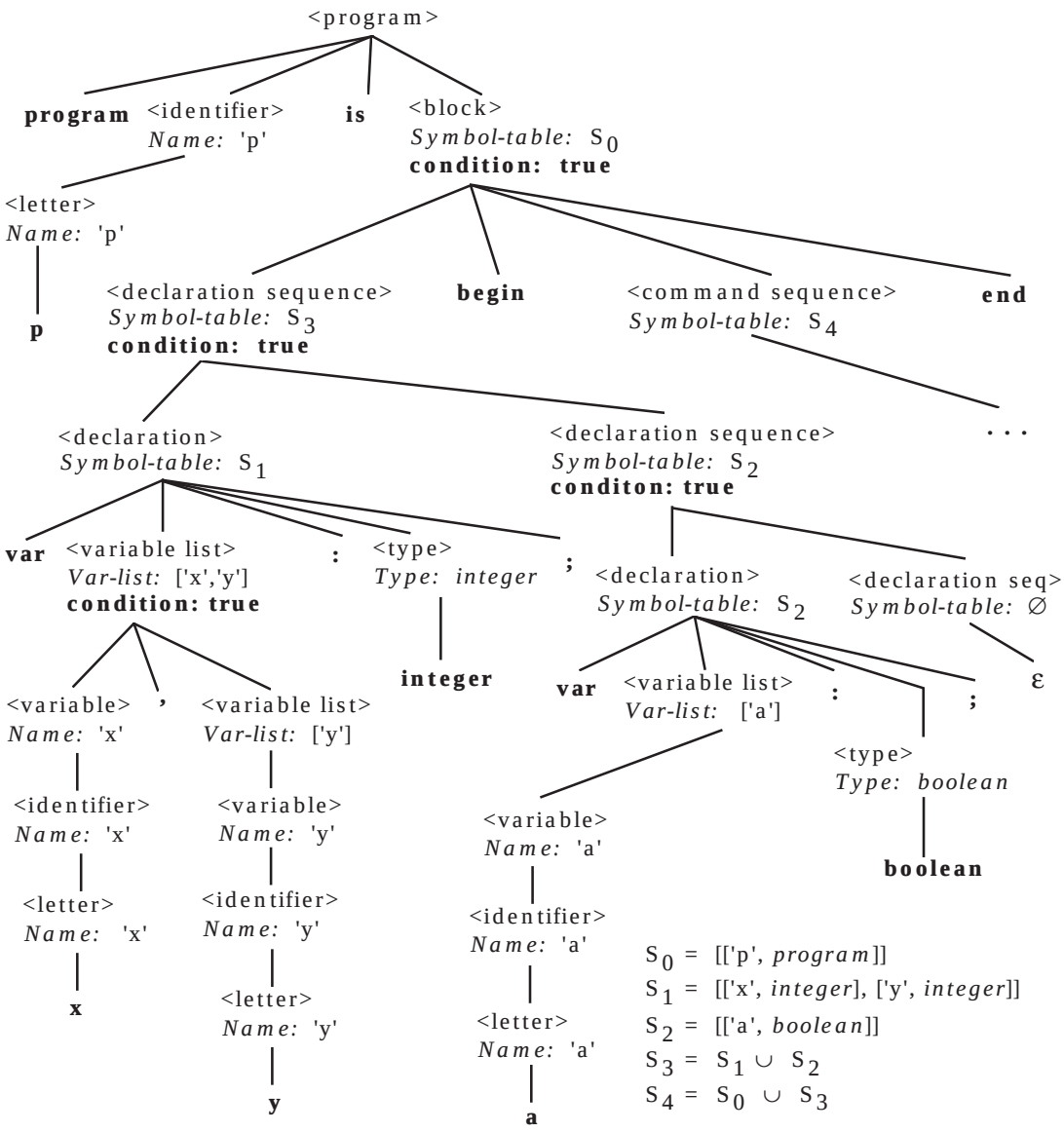

Figure 3.11: Decorated Parse Tree for Wren Program Fragment

The attribute *Symbol-table* is initialized with a pair of values: the *Name* value for the program identifier and the pseudo-type *program*. This *Symbol-table* attribute is inherited into <block>.

<program> ::= **program** <identifier> **is** <block>
Symbol-table(<block>) ←
add-item((*Name*(<identifier>), *program*), empty-table)

For the example in Figure 3.11, the *Symbol-table* attribute for <block> has the value [['p', *program*]]. A single declaration has the form

var <var-list> **:** <type>**;**

The attribute grammar must construct a *Symbol-table* attribute value in which each variable in the list is entered separately in the table with the associated type. For example,

var x**,** y **:** integer**;**

results in a symbol table value of [['x', *integer*], ['y', *integer*]]. In order to accomplish this, we need a synthesized attribute *Var-list* that collects a list of *Name* values, ['x', 'y'] in this case, and a utility function "build-symbol-table" to construct the required symbol table value.

<declaration> ::= **var** <variable list> **:** <type>**;**
Symbol-table(<declaration>) ←
build-symbol-table(*Var-list*(<variable list>), *Type*(<type>))

We first look at how the *Var-list* value is synthesized. Observe that the Lisp-like function "cons" builds the lists of variables.

<variable list> ::=
<variable>
Var-list(<variable list>) ←
cons(*Name*(<variable>), empty-list)
| <variable> **,** <variable list>$_2$
Var-list(<variable list>) ←
cons(*Name*(<variable>),*Var-list*(<variable list>$_2$))
condition:
if *Name*(<variable>) is not a member of *Var-list*(<variable list>$_2$)
then error("")
else error("Duplicate variable in declaration list")

Every time we add a new *Name* value to the *Var-list* attribute, we must verify that it is not already present in the synthesized attribute value for variables that appear to the right. In the attribute grammars for strings of the form $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$, the conditions are either true or false. In this attribute grammar for Wren, we use a slightly different strategy that provides more precise information about any condition check that fails. We assume the existence of an error routine with a string parameter. Calling the error routine with an empty string means that the condition is true. Calling the error routine with a nonempty error message indicates that the condition is false and that the error message provides specific information about the nature of the error encountered.

For the nonterminal <type>, *Type* is a synthesized attribute with the values *integer* or *boolean*, depending on the declared type.

<type> ::= **integer**

Type(<type>) ← *integer*

| **boolean**

Type(<type>) ← *boolean*

We complete our discussion of <declaration> by looking at the utility functions involved. We have assumed some basic list manipulation functions, such as head, tail, and cons. These utility functions are described later, at the end of Figure 3.12, using pattern matching with Prolog-like list structures. The "build-symbol-table" utility function removes names from the variable listing one at a time and adds the pair [name, type] to the symbol table. The utility function "add-item" does the actual appending. This process continues until the entire symbol table is built.

```
build-symbol-table(var-list, type) =
    if empty(var-list)
        then empty-table
        else add-item(head(var-list), type,
                                  build-symbol-table(tail(var-list),type))

add-item(name, type, table) = cons([name,type], table)
```

In situations where a declaration sequence is empty, the empty symbol table value is returned. When the declaration sequence is one declaration followed by another declaration sequence, the union of the two table values is passed up to the parent provided that the intersection of the two table values is empty; otherwise, an error condition occurs and the parse fails.

```
<declarationsequence> ::=
      ε
        Symbol-table(<declarationsequence>) ← empty-table
    | <declaration> <declarationsequence> 2
        Symbol-table(<declarationsequence>) ←
            table-union(Symbol-table(<declaration>),
                                Symbol-table(<declarationsequence> 2))
        condition:
         if table-intersection(Symbol-table(<declaration>),
                    Symbol-table(<declarationsequence> 2)) = empty
            then error("")
            else error("Duplicate declaration of an identifier")
```

The utility function "table-union" glues the symbol tables together. Compare it with the Prolog function concat in Appendix A.

```
table-union(table1, table2) =
    if empty(table1)
       then table2
       else if lookup-type(first-name(table1),table2) = undefined
              then cons(head(table1), table-union(tail(table1), table2))
              else table-union(tail(table1), table2))
```

The utility function "table-intersection" does not perform a set intersection, rather it returns only one of two values, *empty* or *nonempty*, as appropriate. This task is accomplished by removing items from $table_1$, one at a time, and looking up the type associated with the name in $table_2$. If the type is *undefined*, then the intersection process continues with the rest of $table_1$. However, if any other type is returned, the table intersection must be *nonempty* and this value is returned immediately without continuing the search.

```
table-intersection(table1, table2) =
    if empty(table1)
        then empty
        else if lookup-type(first-name(table1),table2) ≠ undefined
                  then nonempty
                  else table-intersection(tail(table1), table2)
```

The utility function "lookup-type" proceeds down a list using recursion, checking the first item as it goes to see if it matches the given name. If it does, the corresponding type is returned; if it does not, the search continues with the

tail of the table. If the empty table is reached, the value *undefined* is returned.

```
lookup-type(name, table) =
    if empty(table)
        then undefined
        else if head(table) = [name, type]
                    then type
                    else lookup-type(name, tail(table))
```

Commands

The grammar rule for <block> is very similar to <declarationsequence> except that one of the symbol tables contains the program identifier. The union of these two tables is passed to <commandsequence> in the rule for <block>, as shown in Figure 3.12. In the command section, *Symbol-table* is an inherited attribute that is passed down from <commandsequence> to the various instances of <command>, except for **skip** which does not require any declaration or type checking.

```
<commandsequence> ::=
      <command>
        Symbol-table(<command>) ←
                                    Symbol-table(<commandsequence>)
    | <command> ; <commandsequence> 2
        Symbol-table(<command>) ←
                                    Symbol-table(<commandsequence>)
        Symbol-table(<commandsequence> 2) ←
                                    Symbol-table(<commandsequence>)
```

A **read** command requires an integer variable. Two context-sensitive errors are possible: The variable is not declared or the variable is not of type integer. In the condition check, the function lookup-type retrieves the variable type, which may be *undefined* if the variable is not found in *Symbol-table*; thus the type either satisfies the condition of being an integer or fails because it is not declared or not of the proper type.

```
<command> ::= read <variable>
    condition:
    case lookup-type(Name(<variable>), Symbol-table(<command>)) is
        integer      : error("")
        undefined    : error("Variable not declared")
        boolean, program  : error("Integer variable expected for read")
```

A **write** command requires an integer expression. One way of specifying this is through a BNF production:

<command> ::= **write** <integer expr>.

However, since <integer expr> is only one alternative for <expr>, we have elected to show a more relaxed BNF that expands to expression and to pass an inherited attribute *Type* to <expr> so that it can check that the expression is an integer expression. This attribute will be passed to each kind of expression so that the type consistency of variables is maintained. The symbol table is also inherited down to the <integer expr> nonterminal. The attribute grammar for <integer expr> ensures that any variables occurring in the expression are of type integer.

<command> ::= **write** <expr>

Symbol-table(<expr>) ← *Symbol-table*(<command>)

Type(<expr>) ← *integer*

If the language has other types of expressions, such as character and string expressions, having output commands pass an inherited attribute *Type* provides a way of type checking the expressions.

In an assignment command, the *Symbol-table* and the type of the target variable are passed to the expression. We also look up the target variable in the *Symbol-table*. If the type of the target variable is *undefined* or *program*, an error occurs.

<command> ::= <variable> **:=** <expr>

Symbol-table(<expr>) ← *Symbol-table*(<command>)

Type(<expr>) ←

lookup-type(*Name*(<variable>),*Symbol-table*(<command>))

condition:

case lookup-type(*Name*(<variable>),*Symbol-table*(<command>)) is

integer, boolean : error(“”)

undefined : error(“Target variable not declared”)

program : error(“Target variable same as program name”).

The control commands **while** and **if** pass the *Symbol-table* attribute to the <boolean expression> and <commandsequence> levels and the expected type to <boolean expr>. Notice that in this case we have only allowed for <boolean expr> (and not <expr>) in the BNF since, even if other types are added such as character or string, the conditional still allows only a Boolean expression.

<command> ::=
while <boolean expr> **do** <commandsequence> **end while**
Symbol-table(<boolean expr>) ← *Symbol-table*(<command>)
Symbol-table(<commandsequence>) ←
Symbol-table(<command>)
Type(<boolean expr>) ← *boolean*

<command> ::=
if <boolean expr> **then** <cmdsequence> **end if**
Symbol-table(<boolean expr>) ← *Symbol-table*(<command>)
Symbol-table(<commandsequence>) ←
Symbol-table(<command>)
Type(<boolean expr>) ← *boolean*
| **if** <boolean expr> **then** <commandsequence> $_1$
else <commandsequence> $_2$ **end if**
Symbol-table(<boolean expr>) ← *Symbol-table*(<command>)
Symbol-table(<commandsequence> $_1$) ←
Symbol-table(<command>)
Symbol-table(<commandsequence> $_2$) ←
Symbol-table(<command>)
Type(<boolean expr>) ← *boolean*

Expressions

The *Symbol-table* and *Type* attributes of <expr> are passed to the two kinds of expressions in Wren. To ensure that the proper alternative for expression is chosen, a guard (condition) on each rule stops the derivation if the types are not consistent. Other errors are handled at a lower level in the derivation. If more sorts of data are available, the sets in the conditions can be expanded.

<expr> ::=
<integer expr>
Symbol-table(<integer expr>) ← *Symbol-table*(<expr>)
Type(<integer expr>) ← *Type*(<expr>)
condition : *Type*(<expr>) ∉ { *boolean* }
| <boolean expr>
Symbol-table(<boolean expr>) ← *Symbol-table*(<expr>)
Type(<boolean expr>) ← *Type*(<expr>)
condition : *Type*(<expr>) ∉ { *integer* }

The nonterminals <integer expr> and <term> pass the *Symbol-table* and *Type* attributes down to the children nodes, except for <weak op> and <strong op>, which require no context checking.

<integer expr> ::=
    <term>
        Symbol-table(<term>) ← *Symbol-table*(<integer expr>)
        Type(<term>) ← *Type*(<integer expr>)
  | <integer expr>$_2$ <weak op> <term>
        Symbol-table(<integer expr>$_2$) ← *Symbol-table*(<integer expr>)
        Symbol-table(<term>) ← *Symbol-table*(<integer expr>)
        Type(<integer expr>$_2$) ← *Type*(<integer expr>)
        Type(<term>) ← *Type*(<integer expr>)

<term> ::=
    <element>
        Symbol-table(<element>) ← *Symbol-table*(<term>)
        Type(<element>) ← *Type*(<term>)
  | <term>$_2$ <strong op> <element>
        Symbol-table(<term>$_2$) ← *Symbol-table*(<term>)
        Symbol-table(<element>) ← *Symbol-table*(<term>)
        Type(<term>$_2$) ← *Type*(<term>)
        Type(<element>) ← *Type*(<term>)

The nonterminal <element> can expand to <numeral>, which requires no context checking, a parenthesized or negated expression, which receives *Symbol-table* and *Type*, or a variable, which is looked up in the symbol table. Normally, we expect this variable to be declared (not *undefined*) and to have type integer. On the other hand, if the inherited *Type* attribute is *undefined*, we have no expectations for the type of the variable, so no error is reported, thereby avoiding certain spurious errors.

<element> ::=
    <numeral>
  | <variable>
        condition:
        case lookup-type(*Name*(<variable>), *Symbol-table*(<element>)) is
          integer : error("")
          undefined : error("Variable not declared")
          boolean, *program* : if Type(<element>)=*undefined*
                then error("")
                else error("Integer variable expected")

| **(** <expr> **)**
Symbol-table(<expr>) ← *Symbol-table*(<element>)
Type(<expr>) ← *Type*(<element>)
| **-** <element>$_2$
Symbol-table(<element>$_2$) ← *Symbol-table*(<element>)
Type(<element>$_2$) ← *Type*(<element>)

The attribute grammar definitions for <boolean expr>, <boolean term>, and <boolean element> are similar to their integer counterparts and are shown in Figure 3.12. A comparison passes the *Symbol-table* and *Type* attributes down to both integer expressions.

<comparison> ::= <integer expr>$_1$ <relation> <integer expr>$_2$
Symbol-table(<integer expr>$_1$) ← *Symbol-table*(<comparison>)
Symbol-table(<integer expr>$_2$) ← *Symbol-table*(<comparison>)
Type(<integer expr>$_1$) ← *integer*
Type(<integer expr>$_2$) ← *integer*

Note that we have restricted comparisons to integer expressions only. Other alternatives are presented in the exercises.

This completes the context checking attribute grammar for Wren, except for the productions for <identifier>, <variable>, <letter>, and <digit>, which appear in the complete grammar in Figure 3.12.

<program> ::= **program** <identifier> **is** <block>
Symbol-table(<block>) ←
add-item((*Name*(<identifier>), *program*), empty-table)

<block> ::= <declarationsequence> **begin** <commandsequence> **end**
Symbol-table(<commandsequence>) ←
table-union(*Symbol-table*(<block>),
Symbol-table(<declarationsequence>))
condition:
if table-intersection(*Symbol-table*(<block>),
Symbol-table(<declarationsequence>)) = *empty*
then error(“”)
else error(“Program name used as a variable”)

<declaration> ::= **var** <variable list> **:** <type>**;**
Symbol-table(<declaration>) ←
build-symbol-table(*Var-list*(<variable list>), *Type*(<type>))

Figure 3.12: Context Checking Attribute Grammar for Wren (Part 1)

```
<declarationsequence> ::=
        ε
            Symbol-table(<declarationsequence>) ← empty-table
      | <declaration> <declaration sequence>2
            Symbol-table(<declarationsequence>) ←
                    table-union(Symbol-table(<declaration>),
                            Symbol-table(<declarationsequence> 2))
            condition:
            if table-intersection(Symbol-table(<declaration>),
                    Symbol-table(<declarationsequence> 2)) = empty
                    then error("")
                    else error("Duplicate declaration of identifier")
<variable list> ::=
         <variable>
            Var-list(<variable list>) ← cons(Name(<variable>), empty-list)
       | <variable> , <variable list>2
            Var-list(<variable list>) ←
                    cons(Name(<variable>),Var-list(<variable list>2))
            condition:
            if Name(<variable>) is not a member of Var-list(<variable list>2)
                    then error("")
                    else error("Duplicate variable in declaration list")
 <type> ::=
        integer
            Type(<type>) ← integer
       | boolean
            Type(<type>) ← boolean
 <commandsequence> ::=
         <command>
            Symbol-table(<command>) ← Symbol-table(<commandsequence>)
       | <command> ; <command sequence>2
            Symbol-table(<command>) ← Symbol-table(<commandsequence>)
            Symbol-table(<cmdsequence> 2) ← Symbol-table(<commandsequence>)
 <command> ::=
     skip
   | read <variable>
            condition:
            case lookup-type(Name(<variable>), Symbol-table(<command>)) is
                    integer : error("")
                    undefined : error("Variable not declared")
                    boolean, program : error("Integer variable expected for read")
```

Figure 3.12: Context Checking Attribute Grammar for Wren (Part 2)

| **write** <expr>
Symbol-table(<expr>) ← *Symbol-table*(<command>)
Type(<expr>) ← *integer*

| <variable> **:=** <expr>
Symbol-table(<expr>) ← *Symbol-table*(<command>)
Type(<expr>) ←
lookup-type(*Name*(<variable>),*Symbol-table*(<command>))
condition:
case lookup-type(*Name*(<variable>), *Symbol-table*(<command>)) is
integer, boolean: error("")
undefined : error("Target variable not declared")
program : error("Target variable same as program name")

| **while** <boolean expr> **do** <commandsequence> **end while**
Symbol-table(<boolean expr>) ← *Symbol-table*(<command>)
Symbol-table(<commandsequence>) ← *Symbol-table*(<command>)
Type(<boolean expr>) ← *boolean*

| **if** <boolean expr> **then** <commandsequence> $_1$
else <commandsequence> $_2$ **end if**
Symbol-table(<boolean expr>) ← *Symbol-table*(<command>)
Symbol-table(<commandsequence> $_1$) ← *Symbol-table*(<command>)
Symbol-table(<commandsequence> $_2$) ← *Symbol-table*(<command>)
Type(<boolean expr>) ← *boolean*

| **if** <boolean expr> **then** <commandsequence> **end if**
Symbol-table(<boolean expr>) ← *Symbol-table*(<command>)
Symbol-table(<commandsequence>) ← *Symbol-table*(<command>)
Type(<boolean expr>) ← *boolean*

<expr> ::=
<integer expr>
Symbol-table(<integer expr>) ← *Symbol-table*(<expr>)
Type(<integer expr>) ← *Type*(<expr>)
condition : *Type*(<expr>) ∉ { *boolean* }

| <boolean expr>
Symbol-table(<boolean expr>) ← *Symbol-table*(<expr>)
Type(<boolean expr>) ← *Type*(<expr>)
condition : *Type*(<expr>) ∉ { *integer* }

<integer expr> ::=
<term>
Symbol-table(<term>) ← *Symbol-table*(<integer expr>)
Type(<term>) ← *Type*(<integer expr>)

| <integer expr>$_2$ <weak op> <term>
Symbol-table(<integer expr>$_2$) ← *Symbol-table*(<integer expr>)
Symbol-table(<term>) ← *Symbol-table*(<integer expr>)
Type(<integer expr>$_2$) ← *Type*(<integer expr>)
Type(<term>) ← *Type*(<integer expr>)

Figure 3.12: Context Checking Attribute Grammar for Wren (Part 3)

```
<term> ::=
      <element>
        Symbol-table(<element>) ← Symbol-table(<term>)
        Type(<element>) ← Type(<term>)
    | <term>2 <strong op> <element>
        Symbol-table(<term>2) ← Symbol-table(<term>)
        Symbol-table(<element>) ← Symbol-table(<term>)
        Type(<term>2) ← Type(<term>)
        Type(<element>) ← Type(<term>)
<weak op> ::= + | –
<strong op> ::= * | /
<element> ::=
        <numeral>
      | <variable>
            condition:
            case lookup-type(Name(<variable>), Symbol-table(<element>)) is
                integer : error("")
                undefined : error("Variable not declared")
                boolean, program : if Type(<element>)=undefined
                                       then error("")
                                       else error("Integer variable expected")
      | ( <expr> )
            Symbol-table(<expr>) ← Symbol-table(<element>)
            Type(<expr>) ← Type(<element>)
      | - <element>2
            Symbol-table(<element>2) ← Symbol-table(<element>)
            Type(<element>2) ← Type(<element>)
<boolean expr> ::=
        <boolean term>
            Symbol-table(<boolean term>) ← Symbol-table(<boolean expr>)
            Type(<boolean term>) ← Type(<boolean expr>)
      | <boolean expr>2 or <boolean term>
            Symbol-table(<boolean expr>2) ←Symbol-table(<boolean expr>)
            Symbol-table(<boolean term>) ← Symbol-table(<boolean expr>)
            Type(<boolean expr>2) ← Type(<boolean expr>)
            Type(<boolean term>) ← Type(<boolean expr>)
<boolean term> ::=
        <boolean element>
            Symbol-table(<boolean element>) ← Symbol-table(<boolean term>)
            Type(<boolean element>) ← Type(<boolean term>)
```

Figure 3.12: Context Checking Attribute Grammar for Wren (Part 4)

```
    | <boolean term>2 and <boolean element>
        Symbol-table(<boolean term>2) ← Symbol-table(<boolean term>)
        Symbol-table(<boolean element>) ← Symbol-table(<boolean term>)
        Type(<boolean term>2) ← Type(<boolean term>)
        Type(<boolean element>) ← Type(<boolean term>)
<boolean element> ::=
        true
      | false
      | <variable>
            condition:
            case lookup-type(Name(<variable>),Symbol-table(<boolean element>)) is
                boolean : error("")
                undefined : error("Variable not declared")
                integer, program : if Type(<boolean element>) = undefined
                                       then error("")
                                       else error("Boolean variable expected")
      | <comparison>
            Symbol-table(<comparison>) ← Symbol-table(<boolean element>)
      | not ( <boolean expr> )
            Symbol-table(<boolean expr>) ← Symbol-table(<boolean element>)
            Type(<boolean expr>) ← Type(<boolean element>)
      | ( <boolean expr> )
            Symbol-table(<boolean expr>) ← Symbol-table(<boolean element>)
            Type(<boolean expr>) ← Type(<boolean element>)
<comparison> ::= <integer expr>1 <relation> <integer expr>2
            Symbol-table(<integer expr>1) ← Symbol-table(<comparison>)
            Symbol-table(<integer expr>2) ← Symbol-table(<comparison>)
            Type(<integer expr>1) ← integer
            Type(<integer expr>2) ← integer
<relation> ::= = | < > | < | < = | > | > =
<variable> ::= <identifier>
            Name(<variable>) ← Name(<identifier>)
<identifier> ::=
        <letter>
            Name(<identifier>) ← Name(<letter>)
      | <identifier>2 <letter>
            Name(<identifier>) ← str-concat(Name(<identifier>2),Name(<letter>))
      | <identifier>2 <digit>
            Name(<identifier>) ← str-concat(Name(<identifier>2),Name(<digit>))
<letter> ::=
        a
            Name(<letter>) ← 'a'
      :       :         :
      | z
            Name(<letter>) ← 'z'
```

Figure 3.12: Context Checking Attribute Grammar for Wren (Part 5)

<numeral> ::= <digit> | <numeral> <digit>

<digit> ::=

0

Name(<digit>) ← '0'

: : :

| **9**

Name(<digit>) ← '9'

Auxiliary Functions

```
build-symbol-table(var-list, type) =
    if empty(var-list)
        then empty-table
        else add-item(head(var-list),type,build-symbol-table(tail(var-list), type))

add-item(name, type, table) = cons([name,type], table)

table-union(table1, table2) =
    if empty(table1)
        then table2
        else if lookup-type(first-name(table1),table2) = undefined
            then cons(head(table1), table-union(tail(table1), table2))
            else table-union(tail(table1), table2))

table-intersection(table1, table2) =
    if empty(table1)
        then empty
        else if lookup-type(first-name(table1),table2) ≠ undefined
            then nonempty else table-intersection(tail(table1),table2)

lookup-type(name, table) =
    if empty(table)
        then undefined
        else if head(table) = [name, type]
            then type else lookup-type(name,tail(table))

head([ first | rest ]) = head

tail([ first | rest ]) = rest

cons(first, rest) = [ first | rest ]

first-name([ [name,type] | restTable ]) = name

empty-table = empty-list = [ ]

empty([ ]) = true

empty([ first | rest ]) = false

str-concat(char-sequence1, char-sequence2) returns the
        concatenation of char-sequence1 followed by char-sequence2

error(string) prints nonempty strings
```

Figure 3.12: Context Checking Attribute Grammar for Wren (Part 6)

Exercises

1 Draw the parse tree decorated with attributes for the following Wren program:

```
program p is
    var b: boolean;
    var m, n: integer;
begin
    read m; read n;
    b := m < n;
    if b  then write m
          else write n
    end if
end
```

2. Suppose the declarations in the above program are replaced by

```
var b, m, n: integer;
```

Show the changes in the parse tree from exercise 1.

3. Modify the attribute grammar for Wren to allow for checking equality or inequality of Boolean expressions in comparisons, but none of the other relations.

4. Add the declaration types character and string to Wren. Allow the input of an integer and character (use **readch**), but not Boolean and string. Allow output of integer, character, and string (use **writech** and **writestr**), but not Boolean. Restrict a string literal to a sequence of lowercase alphabetic characters, digit characters, and the space character. Modify the attribute grammar to enforce the related context conditions. Overloading **read** and **write** makes this problem more difficult.

5. After completing exercise 4, add the following string expressions, character expressions, and additions to integer expressions.

String Expressions:

concat(<str expr>,<str expr>)
substr(<str expr>,<int expr>,<int expr>)
where the first integer expression is the start position and the second expression is the length
toStr(<char expr>)
“example of a string literal”

Character Expressions:
toChar(<str expr>,<int expr>)
where the integer expression is the position of the character in the string
char(<int expr>)
'X' character literal

Additions to Integer Expressions:
ord(<char expr>)
length(<str expr>)

After carefully specifying the BNF for these operations, add the appropriate context checking using attributes.

6. Suppose that we extend Wren to allow for the following alternative in declarations:

<declaration> ::= **procedure** <identifier> **is** <block>

This alternative results in a new value for *Type*, which we name *procedure*. We also add a call command:

<command> ::= **call** <identifier>

These changes allow nested blocks with local declarations. Modify the attribute grammar for Wren to accommodate these changes. Follow Pascal scope rules by requiring that an identifier must be declared before it is used. Furthermore, remove the first context condition concerning the program identifier and relax the second and third context conditions:

2. All identifiers that appear in a block must be declared in that block or in an enclosing block.

3. No identifier may be declared more than once at the top level of a block.

Hint: One attribute should synthesize declarations and a different attribute should inherit declarations since the declaration information has to be inherited into the declaration section itself because of the occurrence of a <block> in a procedure declaration.

7. Recall the language of expressions formed as lists of integers in exercise 9 in section 1.2. Augment the BNF grammar for the language with attributes that enforce the conformity of lists given to the arithmetic operations +, –, and *.

3.3 LABORATORY: CONTEXT CHECKING WREN

We have already seen how logic grammars in Prolog can be used to construct an abstract syntax tree for a Wren program. Using several utility predicates, we constructed a scanner that converts a text file containing a program into a sequence of tokens. We utilize this same "front-end" software for the current laboratory activity; however, we extend the parser using attributes to perform context-sensitive declaration and type checking.

Before proceeding directly into the development of the attribute grammar in Prolog, we need to make some important design decisions about the expected output from our context checker. The scanning and parsing front-end program from Chapter 2 assumes that the input program obeys the BNF for Wren. With attribute grammars, we have additional context-sensitive conditions at selected nodes that must be satisfied for the parse to succeed. The first question we need to address is what should be the output of the parser for a program that obeys the BNF but fails one of the context condition checks. We can elect to have the entire parse fail, with Prolog simply reporting "no", but this response seems less than satisfactory. Another alternative, the one we develop, is to allow the parse to succeed, provided the BNF is satisfied, and to insert error messages in cases where context-sensitive checking fails.

The second design decision we have to make is the form of the output of the parser in cases where the parse succeeds but may contain context checking errors. In Chapter 2 Wren programs were transformed into a compact form that contained only the relevant syntactic information—namely, abstract syntax trees. For example, the assignment statement in Wren

```
x := 3 + 2 * y
```

was tokenized by the scanner to:

```
[ide(x),assign,num(3),plus,num(2),times,ide(y)]
```

and then parsed to produce the abstract syntax tree:

```
assign(x,exp(plus,num(3),exp(times,num(2),ide(y)))).
```

This latter form will be useful when we develop an interpreter for Wren in later chapters. Since the current project deals with the detection of context condition violations, we elect to retain the stream of tokens output from the scanner with the possible error messages inserted to indicate any context condition violations. This approach is best illustrated by an example. The program below does not perform any useful function; it simply demonstrates a variety of commands and types.

```
?- go.
>>> Checking Context Constraints in Wren <<<
Enter name of source file: prog1.wren
    program prog1 is
      var x,y: integer;
      var b,c: boolean;
    begin
      read x;  read y;  write x+y;
      b := x < y;
      if x = y
        then c := x <= y
        else c := x > y end if;
      while c do x := x + 1 end while;
      b := b and (b or c)
    end
Scan successful
[program,ide(prog1),is,
   var,ide(x),comma,ide(y),colon,integer,semicolon,
   var,ide(b),comma,ide(c),colon,boolean,semicolon,
   begin,
     read,ide(x),semicolon,read,ide(y),semicolon,
     write,ide(x),plus,ide(y),semicolon,
     ide(b),assign,ide(x),less,ide(y),semicolon,
     if,ide(x),equal,ide(y),
       then,ide(c),assign,ide(x),lteq,ide(y),
       else,ide(c),assign,ide(x),grtr,ide(y),
     end,if,semicolon,
     while,ide(c),do,
       ide(x),assign,ide(x),plus,num(1),
     end,while,semicolon,
     ide(b),assign,ide(b),and,lparen,ide(b),or,ide(c),rparen,
 end,
eop]
Parse successful
[program,ide(prog1),is,
   var,ide(x),comma,ide(y),colon,integer,semicolon,
   var,ide(b),comma,ide(c),colon,boolean,semicolon,
   begin,
     read,ide(x),semicolon,read,ide(y),semicolon,
     write,ide(x),plus,ide(y),semicolon,
     ide(b),assign,ide(x),less,ide(y),semicolon,
     if,ide(x),equal,ide(y),
```

```
       then,ide(c),assign,ide(x),lteq,ide(y),
       else,ide(c),assign,ide(x),grtr,ide(y),
     end,if,semicolon,
     while,ide(c),do,
       ide(x),assign,ide(x),plus,num(1),
     end,while,semicolon,
     ide(b),assign,ide(b),and,lparen,ide(b),or,ide(c),rparen,
 end]
```

For readability, we have inserted line feeds and spacing for indentation in the listing shown above. The test program obeys the BNF and all context-sensitive conditions, so the output of the parser is the same as the output from the scanner, except for the removal of the final eop token. It may seem that we have done a lot of work to accomplish nothing, but introducing some context-sensitive errors will illustrate what the parser is doing for us.

```
?- go.
>>> Checking Context Constraints in Wren <<<
Enter name of source file: prog2.wren
    program prog2 is
      var x,y,b: integer;
      var b,c: boolean;
    begin
      read x;  read c;  write x+a;
      b := x < c;
      if x = y
         then c := x <= y
         else y := x > y
      end if;
      while c > b do x := x + 1 end while;
      b := b and (y or z)
    end
Scan successful
[program,ide(prog2),is,
  var,ide(x),comma,ide(y),comma,ide(b),colon,integer,semicolon,
   var,ide(b),comma,ide(c),colon,boolean,semicolon,
   begin,
     read,ide(x),semicolon,read,ide(c),semicolon,
     write,ide(x),plus,ide(a),semicolon,
     ide(b),assign,ide(x),less,ide(c),semicolon,
     if,ide(x),equal,ide(y),
       then,ide(c),assign,ide(x),lteq,ide(y),
       else,ide(y),assign,ide(x),grtr,ide(y),
     end,if,semicolon,
     while,ide(c),grtr,ide(b),do,
```

```
       ide(x),assign,ide(x),plus,num(1),
     end,while,semicolon,
     ide(b),assign,ide(b),and,lparen,ide(y),or,ide(z),rparen,
 end, eop]

Parse successful
[program,ide(prog2),is,
    var,ide(x),comma,ide(y),comma,ide(b),colon,integer,semicolon,
       ERROR: Duplicate declaration of an identifier,
     var,ide(b),comma,ide(c),colon,boolean,semicolon,
 begin,
   read,ide(x),semicolon,
   read,ide(c),
     ERROR: Integer variable expected for read,semicolon,
   write,ide(x),plus,ide(a),
     ERROR: Variable not declared,semicolon,
   ide(b),assign,ide(x),less,ide(c),
     ERROR: Integer variable expected,
     ERROR: Integer expression expected,
   semicolon,
   if,ide(x),equal,ide(y),
        then,ide(c),assign,ide(x),lteq,ide(y),
        else,ide(y),assign,ide(x),grtr,ide(y),
     ERROR: Integer expression expected,
   end,if,semicolon,
   while,ide(c),
     ERROR: Integer variable expected,grtr,ide(b),do,
       ide(x),assign,ide(x),plus,num(1),end,while,semicolon,
       ide(b),assign,ide(b),
     ERROR: Boolean variable expected,and,lparen,ide(y),
     ERROR: Boolean variable expected,or,ide(z),
     ERROR: Variable not declared,rparen,
     ERROR: Integer expression expected,
 end]
```

Again, we have formatted the output for readability. It should be noted that the error messages appear near the locations where the errors occur. As mentioned previously, for programs that obey the BNF, we allow the parse to succeed, although there may or may not be context-sensitive errors. The following strategy is implemented: An error variable is placed at all locations when a context-sensitive check is made. This variable is bound either to the atom noError or to an appropriate error message entered as an atom (a string inside apostrophes). During the final stage of processing, we flatten the parse tree into a linear list and strip away all noError values using a predicate called flattenplus, so only the real error messages remain.

Declarations

Now that we have formulated a goal for this laboratory exercise, we can proceed to develop the parser using stepwise refinement. Enough code is presented to introduce the idea of implementing the attribute grammar in Prolog. Those portions of code that are not detailed here are left as exercises.

```
program(TokenList) -->
        [program], [ide(I)], [is],
        { addItem(I,program,[ ],InitialSymbolTable) },
        block(Block, InitialSymbolTable),
        { flattenplus([program, ide(I), is, Block], TokenList) }.
```

After getting the program identifier name, we add it with the pseudo-type program to the InitialSymbolTable, which is passed to the predicate block. This predicate returns a structure (Block) that is used to build the list

```
[program, ide(I), is, Block],
```

which is flattened into a modified token list and given as the result of the context checker.

The utility functions in our attribute grammar use functional notation that can only be simulated in Prolog. We adopt the strategy that the return value is the last term in the parameter list, so addItem in Prolog becomes

```
addItem(Name, Type, Table, [[Name,Type] | Table]).
```

The code for block is the first place we do context checking to ensure that the program name is not declared elsewhere in the program.

```
block([ErrorMsg, Decs, begin, Cmds, end],InitialSymbolTable) -->
    decs(Decs,DecsSymbolTable),
    { tableIntersection(InitialSymbolTable, DecsSymbolTable,Result),
      tableUnion(InitialSymbolTable, DecsSymbolTable, SymbolTable),
     ( Result=nonEmpty,
         ErrorMsg='ERROR: Program name used as a variable'
      ; Result=empty, ErrorMsg=noError) },
    [begin], cmds(Cmds,SymbolTable), [end].
```

A block parses simply as

```
decs(Decs,DecsSymbolTable), [begin], cmds(Cmds,SymbolTable), [end],
```

but we have added some Prolog code to perform a table intersection, which returns one of two results: empty or nonEmpty. We bind the variable ErrorMsg to the atom noError if the intersection is empty or to an appropriate error message (another atom) if the program name appears in the declarations. We also form the union of the InitialSymbolTable and the DecsSymbolTable producing a value to be passed to the command sequence as SymbolTable.

The utility predicate tableIntersection follows directly from the definition of the utility function in the attribute grammar. Notice the use of lookupType that returns the value undefined if the identifier is not found in the table, or the associated type if the identifier is found. The predicate tableUnion also follows directly from the definition in the attribute grammar; its definition is left as an exercise.

```
tableIntersection([ ], Table2, empty).

tableIntersection(Table1, [ ], empty).

tableIntersection([[Name, Type1] | RestTable ], Table2, nonEmpty) :-
        lookupType(Name, Table2,Type2), (Type2=integer; Type2=boolean).

tableIntersection([[Name, Type] | RestTable], Table2, Result) :-
        tableIntersection(RestTable, Table2, Result).

lookupType(Name, [ ], undefined).

lookupType(Name, [[Name,Type] | RestTable], Type).

lookupType(Name, [Name1 | RestTable], Type) :-
        lookupType(Name, RestTable, Type).
```

Observe that many of the variables in the heads of these clauses do not appear in the bodies of the clauses. Anonymous variables such as the following can be used in this situation:

```
tableIntersection([ ], _, empty).

tableIntersection(_, [ ], empty).

tableIntersection([[Name, _] | _ ], Table2, nonEmpty) :-
        lookupType(Name, Table2,Type2), (Type2=integer; Type2=boolean).

tableIntersection([[_ , _] | RestTable], Table2, Result) :-
        tableIntersection(RestTable, Table2, Result).
```

We prefer using variable names instead of anonymous variables because suggestive variable names make the clause definitions more intelligible. Substituting variable names in place of anonymous variables may result in warning messages from some Prolog systems, but the program still functions correctly.

Two types of multiple declarations may occur in Wren: duplicates within the same declaration, as in

```
var x, y, z, x : boolean ;
```

and duplicates between two different declarations, as in

```
var x, y, z: boolean ;
var u, v, w, x: integer ;
```

The context checker needs to recognize both errors. A variable list is a single variable followed by a list of variables, which may or may not be empty. In either case, we check if the current head of the variable list is a member of the list of remaining variables. If so, we have a duplicate variable error; otherwise, we pass forward the error message generated by the remainder of the list. Note that commas are inserted into the variable list that is returned since we want to construct the original token sequence.

```
varlist(Vars,ErrorMsg) --> [ide(Var)], restvars(ide(Var),Vars,ErrorMsg).

restvars(ide(Var),[ide(Var), comma |Vars],ErrorMsg) -->
                        [comma], varlist(Vars,ErrorMsg1),
                        { member(ide(Var),Vars),
                          ErrorMsg='ERROR: Duplicate variable in listing'
                        ; ErrorMsg = ErrorMsg1 }.

restvars(ide(Var),[ide(Var)],ErrorMsg) --> [ ], { ErrorMsg=noError }.
```

Once we have determined there are no duplicate variables within a single declaration, we check between declarations. The strategy is much the same: A sequence of declarations is a single declaration followed by any remaining declarations, which may or may not be empty. In each case, we check if the table intersection of the symbol table associated with the current declaration is disjoint from the symbol table of the remaining declarations. If it is not, an error message is generated. The code shown below is incomplete, as the table intersection test and the ERROR message are missing. Completing this code is left as an exercise.

```
decs(Decs,SymbolTable) --> dec(Dec,SymbolTable1),
                restdecs(Dec,SymbolTable1,Decs,SymbolTable).

decs([ ],[ ]) --> [ ].

restdecs(Dec,SymbolTable1,[Dec,ErrorMsg|Decs],SymbolTable) -->
                decs(Decs,SymbolTable2),
                { tableUnion(SymbolTable1,SymbolTable2,SymbolTable),
                  (ErrorMsg=noError) }.

restdecs(Dec,SymbolTable,[Dec],SymbolTable) --> [ ].
```

A single declaration results in a symbol table that is constructed by the utility predicate buildSymbolTable, which takes a list of variables and a single type and inserts a [Var, Type] pair into an initially empty symbol table for each variable name in the list. Observe that we remove commas from the variable list before passing it to buildSymbolTable. A predicate delete needs to be defined to perform this task.

```
dec([var, Vars, ErrorMsg, colon, Type, semicolon],SymbolTable) -->
        [var], varlist(Vars, ErrorMsg), [colon], type(Type), [semicolon],
        { delete(comma,Vars,NewVars),
          buildSymbolTable(NewVars, Type, SymbolTable) }.

type(integer) --> [integer].

type(boolean) --> [boolean].

buildSymbolTable([ ], Type, [ ]).

buildSymbolTable([ide(Var)|RestVars], Type, SymbolTable):-
                buildSymbolTable(RestVars,Type,SymbolTable1),
                addItem(Var, Type, SymbolTable1, SymbolTable).
```

Commands

We now turn our attention to the context checking within command sequences. A command sequence is a single command followed by the remaining commands, which may or may not be empty. We pass the symbol table attribute down the derivation tree to both the first command and to the remaining commands.

```
cmds(Cmds,SymbolTable) -->
        command(Cmd,SymbolTable), restcmds(Cmd,Cmds,SymbolTable).

restcmds(Cmd,[Cmd, semicolon|Cmds],SymbolTable) -->
                        [semicolon], cmds(Cmds,SymbolTable).

restcmds(Cmd,[Cmd],SymbolTable) --> [ ].
```

The **skip** command is very simple; it needs no type checking. The **read** command requires the associated variable to be of type integer. Two possible errors may occur in a **read** command: The variable has not been declared or the variable is of the wrong type.

```
command(skip,SymbolTable) --> [skip].

command([read, ide(I), ErrorMsg], SymbolTable) -->
    [read], [ide(I)],
    { lookupType(I,SymbolTable,Type),
        ( Type = integer, ErrorMsg=noError
        ; Type = undefined, ErrorMsg='ERROR: Variable not declared'
        ; (Type = boolean; Type = program),
                ErrorMsg='ERROR: Integer variable expected') }.
```

The **write** command requests an integer expression by passing the value integer as an inherited attribute to the expression. This task is left as an exercise.

A correct assignment command has one of two forms: An integer variable is assigned the result of an integer expression or a Boolean variable is assigned the result of a Boolean expression. Two potential errors can occur: The target variable is not declared or the target variable and the expression are not the same type. The decision to have a successful parse whenever the BNF is satisfied complicates the code for the assignment command. No matter which errors occur, we must consume the symbols in the expression on the right-hand side. View the definition below as a case command controlled by the type of the target variable. Each case selection includes a call to parse the expression.

```
command([ide(V), assign, E, ErrorMsg], SymbolTable) -->
  [ide(V)], [assign],
  { lookupType(V,SymbolTable,VarType) },
   ({ VarType = integer },
        (expr(E,SymbolTable,integer), { ErrorMsg=noError }
         ; expr(E,SymbolTable,boolean),
                    { ErrorMsg='ERROR: Integer expression expected' }) ;
   { VarType = boolean },
        (expr(E,SymbolTable,boolean), { ErrorMsg=noError }
        ;  expr(E,SymbolTable,integer),
                    { ErrorMsg='ERROR: Boolean expression expected' }) ;
   { VarType = undefined, ErrorMsg='ERROR: Target of assign not declared' ;
     VarType = program,
                    ErrorMsg='ERROR: Program name used as a variable' },
          expr(E,SymbolTable,undefined)).
```

The **if** and **while** commands do no type checking directly; rather they pass the SymbolTable and Type attributes to their constituent parts. The **if-then-else** command is given; the **while** command is left as an exercise.

```
command([if,Test,then,Then,Else],SymbolTable) -->
                              [if], boolexpr(Test,SymbolTable,boolean), [then],
                    cmds(Then,SymbolTable), restif(Else,SymbolTable).

restif([else,Else,end,if],SymbolTable) -->
                    [else], cmds(Else,SymbolTable), [end], [if].

restif([end,if],SymbolTable) --> [end], [if].
```

Expressions

The inherited attribute passed from <expr> to <int expr> and <bool expr> may have the value undefined. We cannot let such a value cause failure in the parsing, so four clauses are needed in the logic grammar.

```
expr(E,SymbolTable,integer) --> intexpr(E,SymbolTable,integer).
expr(E,SymbolTable,boolean) --> boolexpr(E,SymbolTable,boolean).
expr(E,SymbolTable,undefined) --> intexpr(E,SymbolTable,undefined).
expr(E,SymbolTable,undefined) --> boolexpr(E,SymbolTable,undefined).
```

In the attribute grammar, we made expression and term left recursive since this matches the left associativity of the additive and multiplicative operations. Since we cannot use left recursion in logic grammars, we need to adopt a different strategy for producing the same parse tree. When we studied BNF, we learned that

```
<int expr> ::= <int expr> <weak op> <term>
```

can also be expressed as

```
<int expr> ::= <term> { <weak op> <term> }
```

where the braces mean zero or more occurrences. We use this technique to develop our logic grammar (see Chapter 2 for more on this issue).

```
intexpr(E,SymbolTable,Type) -->
        term(T,SymbolTable,Type), restintexpr(T,E,SymbolTable,Type).
restintexpr(T, E, SymbolTable,Type) -->
                weakop(Op), term(T1, SymbolTable,Type),
                restintexpr([T,Op,T1], E, SymbolTable,Type).
restintexpr(E,E,SymbolTable,Type) --> [ ].
weakop(plus) --> [plus].
weakop(minus) --> [minus].
```

A term is an element, possibly followed by more elements separated by multiplication or division (strong operators). The code for term, restterm, and strongop is left as an exercise.

An element may be a constant number, in which case no type checking is required. If the element is a variable, it is looked up in the symbol table. Two errors are possible: The variable is not declared or it is the wrong type. No error occurs if it is an integer and we are expecting an integer or if the variable is defined, but we are not expecting any type in particular (the inherited attribute Type has the value undefined because the target variable in an assignment command was undeclared).

```
element([num(N)],SymbolTable,Type) --> [num(N)].

element([ide(I),ErrorMsg],SymbolTable,Type) -->
    [ide(I)],
    { lookupType(I,SymbolTable,VarType),
        (VarType = integer, Type = integer, ErrorMsg=noError
        ; VarType = undefined, ErrorMsg='ERROR: Variable not declared'
        ; Type = undefined, ErrorMsg=noError
        ; (VarType = boolean; VarType = program),
                ErrorMsg='ERROR: Integer variable expected') }.

element([lparen, E, rparen], SymbolTable,Type) -->
                [lparen], intexpr(E,SymbolTable,Type), [rparen].

element([minus|E],SymbolTable,Type) -->
                [minus], element(E, SymbolTable,Type).
```

We complete the discussion of the Prolog implementation of the attribute grammar for context checking by focusing on the code for Boolean expressions and for comparisons. Boolean expression, which handles the **or** operator, and Boolean term, which handles the **and** operator, are very similar to integer expression and term. A Boolean element may be a constant, **true** or **false**, a variable, whose declaration and type must be checked, a comparison, a parenthesized Boolean expression, or the unary Boolean operator **not**. Except for comparison, which is given below, this code is left as an exercise.

```
comparison([E1,R,E2],SymbolTable) -->
    intexpr(E1,SymbolTable,integer), rel(R), intexpr(E2,SymbolTable,integer).

rel(equal) --> [equal].  rel(neq) --> [neq].  rel(less) --> [less].

rel(grtr) --> [grtr].  rel(gteq) --> [gteq].  rel(lteq) --> [lteq].
```

This completes the discussion and partial implementation of our context checking attribute grammar. When the omitted code has been developed, the program will produce the output given at the start of the section.

Exercises

1. Complete the code for the following predicates that were omitted from the text:
   - the tableUnion utility function
   - the predicate restdecs by adding the tableIntersection test
   - the **write** command

- the **while** command
- term, restterm, and strongop
- boolexpr, boolterm, and boolelement
- a flatten utility predicate flattenplus that also removes noError

2. Modify the Prolog implementation of our Wren attribute grammar to allow checking equality or inequality of Boolean expressions in comparisons, but none of the other relations.

3. Following exercise 4 in Section 3.2, add the declaration types character and string to Wren. Implement the changes to the attribute grammar in Prolog.

4. Following exercise 5 in Section 3.2, add the commands for character and string manipulations. Use attributes to add any appropriate context checking.

5. Following exercise 6 in Section 3.2, add the declaration and calling of parameterless procedures.

3.4 FURTHER READING

The seminal paper in attribute grammars has been written by Donald Knuth [Knuth68]. Other papers have explored the mathematical semantics of attribute grammars [Mayoh81] or developed new concepts, such as ordered attribute grammars [Kastens80]. David Watt presents an extended attribute grammar for Pascal [Watt79].

The primary application of attribute grammars is in compiler construction [Bochman78]. Attribute grammars can be used both for type checking, as we have seen in this chapter, and code generation, as we will see in Chapter 7. Many automated tools have been written to aid in compiler construction. Kennedy and Warren discuss the generation of attribute grammar evaluators [Kennedy76]. Those familiar with Unix software may have used LEX, an automated lexical analyzer [Lesk75], and YACC, "Yet Another Compiler-Compiler" [Johnson78]. Automated tools can help generate production level compilers [Farrow84]. Readers wanting to explore the application of attribute grammars in compiler construction can consult any number of references, including [Aho86], [Fischer91], [Parsons92], and [Pittman92].

Recent research in attribute grammars includes work in attribute propagation by message passing [Demers85] and using attribute grammars to build language-based editors [Johnson85]. The Synthesizer-Generator [Reps89] is

a modern software tool to build context-sensitive editors. This sophisticated, windows-based product is built on top of LEX and YACC (or equivalent tools). Editors are available for languages such as Pascal and C. We have used the Synthesizer-Generator as a teaching tool in a compiler class by asking students to build a context-sensitive editor for Wren. Uses of the Synthesizer-Generator include many diverse context-sensitive situations, such as calculations in a spreadsheet or balancing chemical equations.

Attribute grammars can also be used for type inferencing. It is possible to have a strongly typed language without requiring explicit declarations. ML is one such language. The first time an identifier appears in a program, its type is inferred from the usage. The type can be synthesized to the root of the parse tree. Other usage of the same identifier must be type consistent. Reps and Teitelbaum [Reps89] demonstrate type inferencing by using the Synthesizer-Generator to build a language editor that automatically inserts type declarations in a program based on the usage of identifiers.