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.

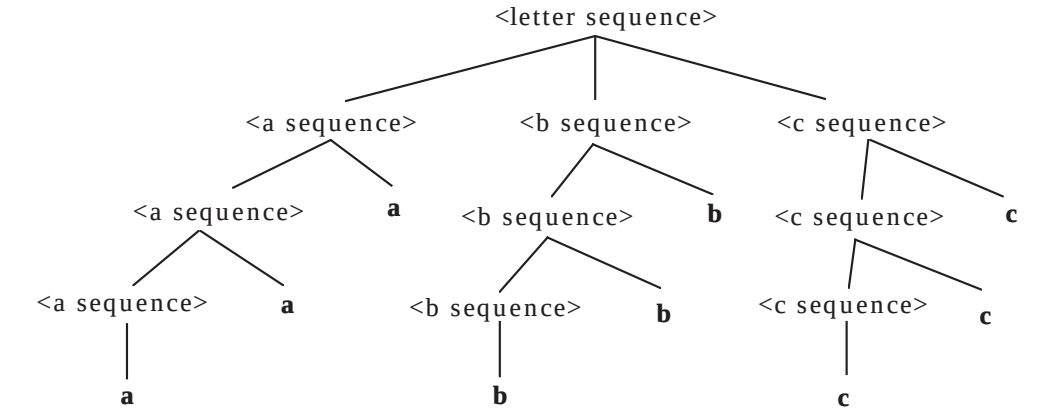

Figure 3.1: Parse Tree for the String **aaabbbccc**

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.

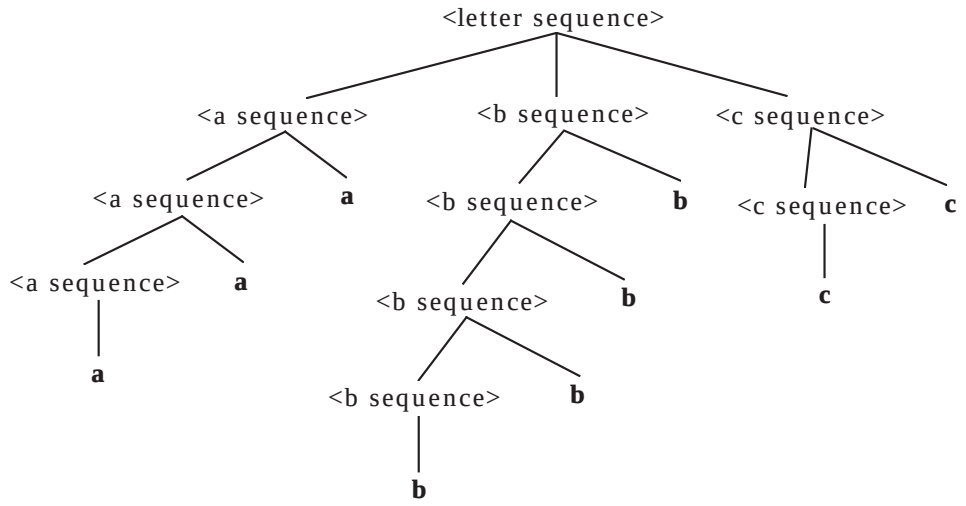

Figure 3.2: Parse Tree for the String **aaabbbbcc**

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.

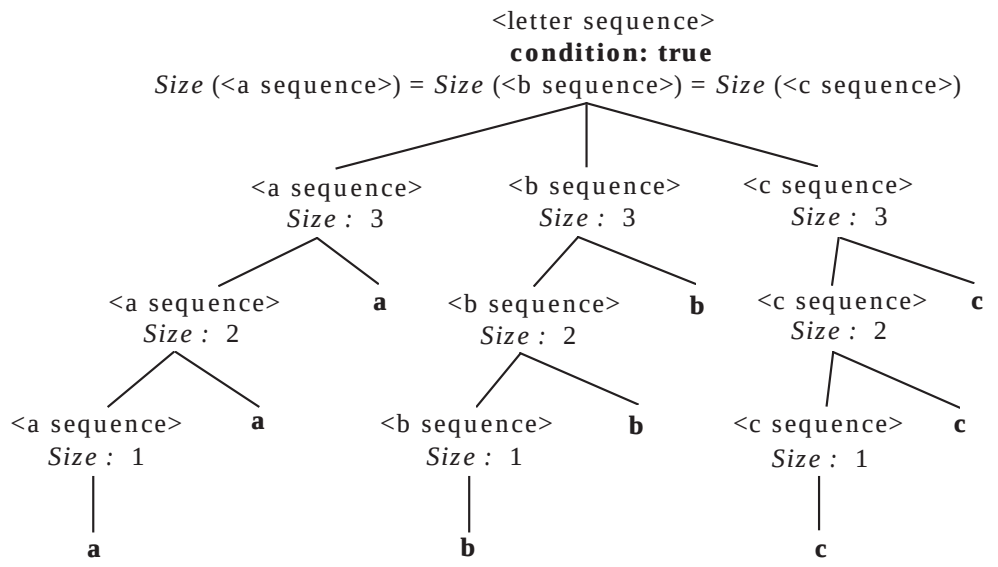

Figure 3.3: Parse Tree for **aaabbbccc** Using Synthesized Attributes

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.

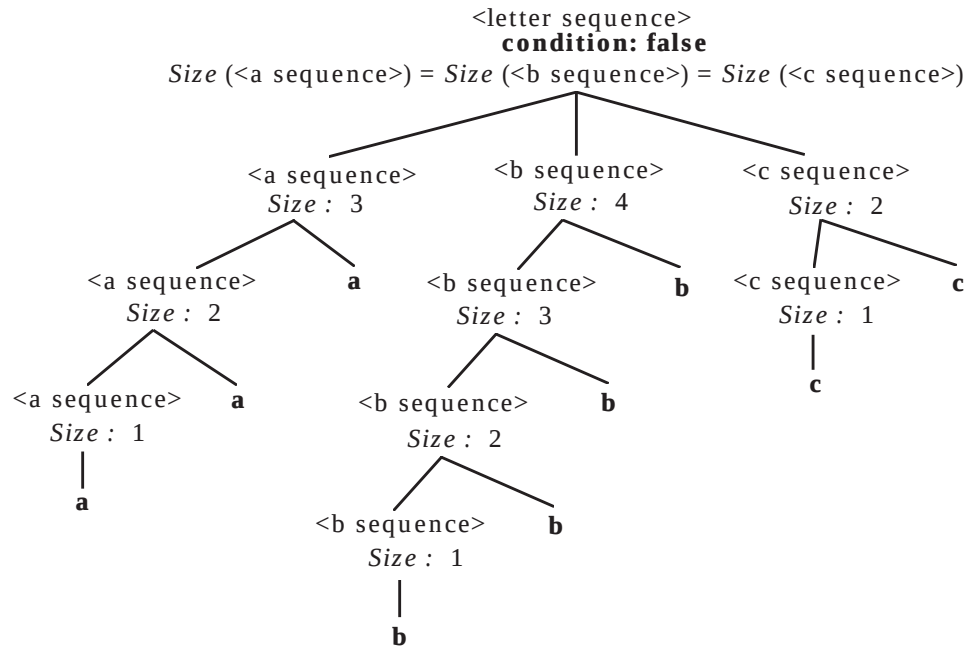

Figure 3.4: Parse Tree for **aaabbbbcc** Using Synthesized Attributes

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.

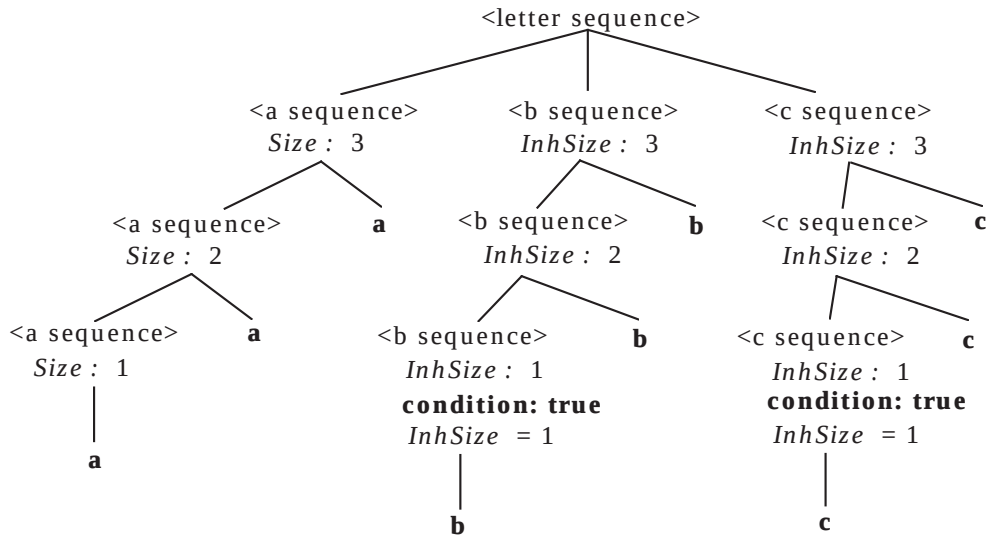

Figure 3.5: Parse Tree for **aaabbbccc** Using Inherited Attributes

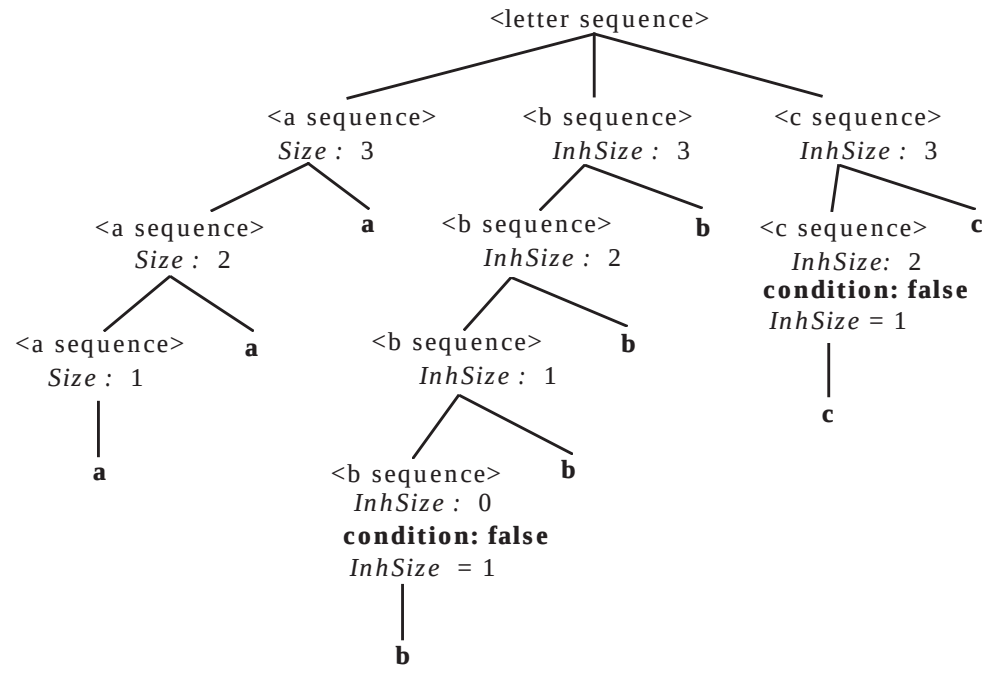

Figure 3.6: Parse Tree for **aaabbbbcc** Using Inherited Attributes

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 $X_1, X_2, \ldots, X_{n_p}$ 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.

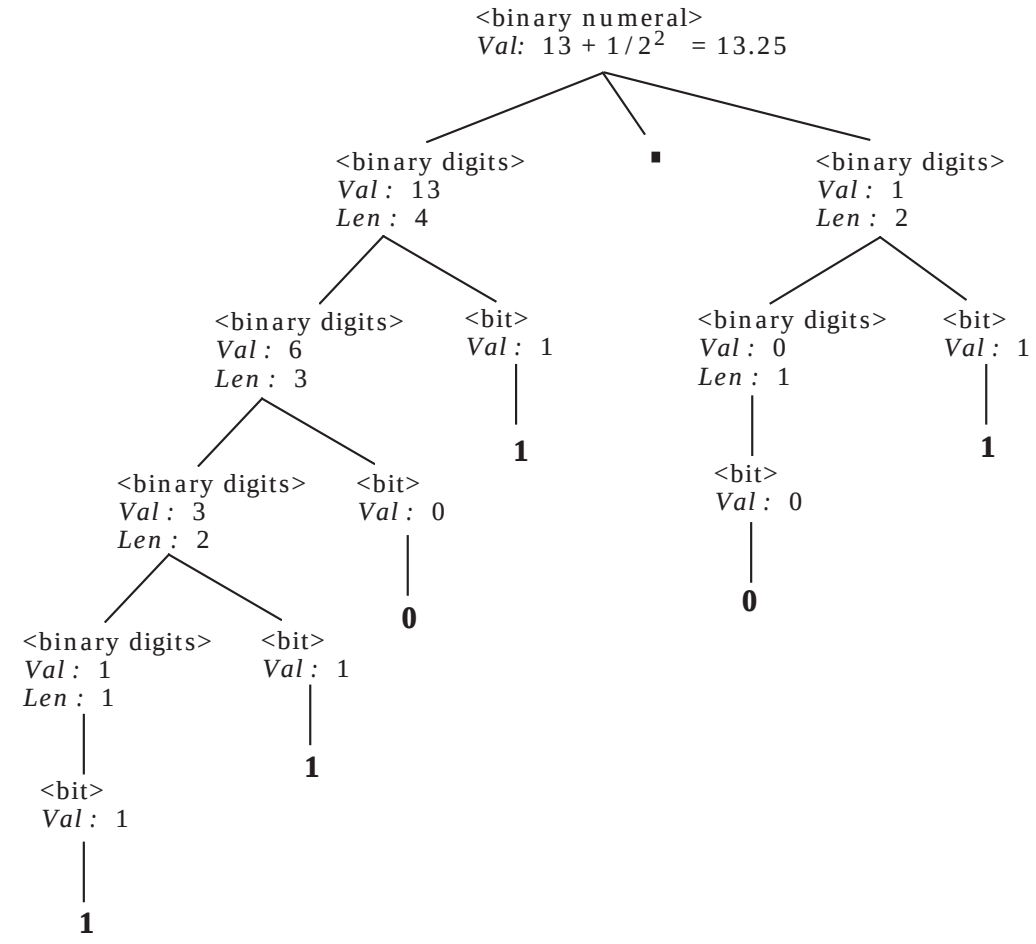

Figure 3.7: Binary Numeral Semantics Using Synthesized Attributes

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,

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

and then in base 2,

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

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.

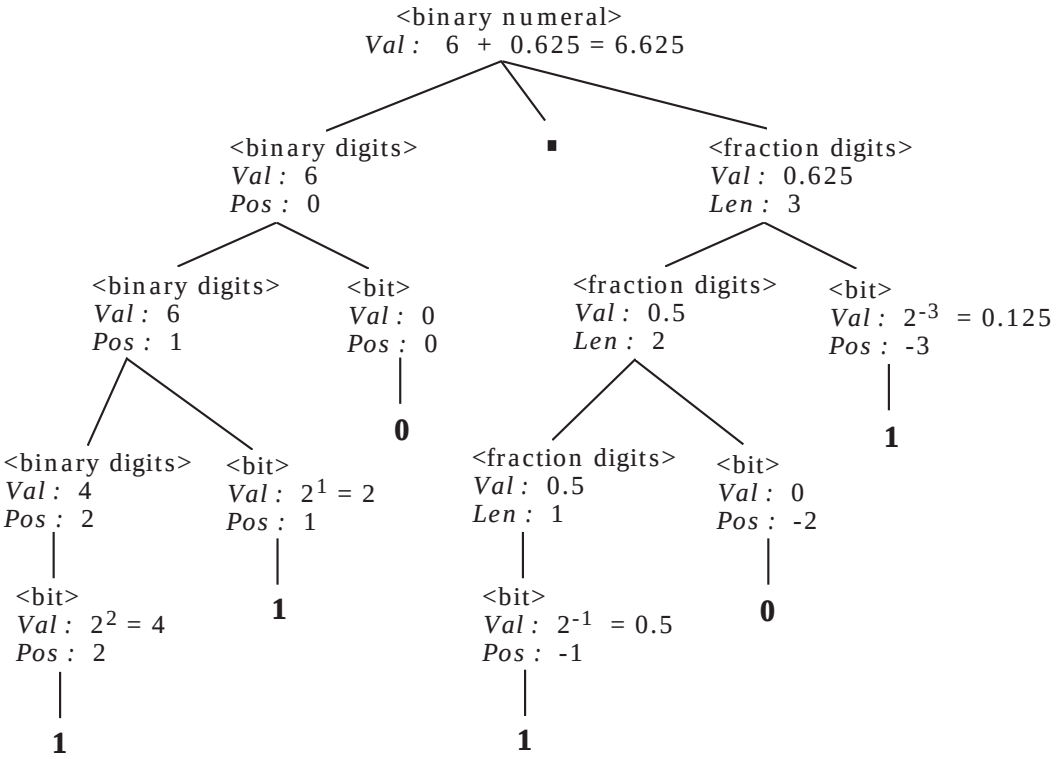

Figure 3.8: Binary Numeral Semantics Using Positional Attributes

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.