# Attribute Grammars

## Objectives

+ Discuss attribute grammars, which are used to describe both the syntax and static semantics of programming languages.

## Introduction

– An attribute grammar is a device used to describe more of the structure of a programming language than can be described with a context-free grammar. An attribute grammar is an extension to a context-free grammar. The extension allows certain language rules to be conveniently described, such as type compatibility.
– Before we formally define the form of attribute grammars, we must clarify the concept of static semantics.

## 1. Static Semantics

– There are some characteristics of programming languages that are difficult to describe with BNF, and some that are impossible. As an example of a syntax rule that is difficult to specify with BNF, consider type compatibility rules.
– In Java, for example, a floating-point value cannot be assigned to an integer type variable, although the opposite is legal. Although this restriction can be specified in BNF, it requires additional nonterminal symbols and rules.
– If all of the typing rules of Java were specified in BNF, the grammar would become too large to be useful, because the size of the grammar determines the size of the syntax analyzer.
– As an example of a syntax rule that cannot be specified in BNF, consider the common rule that all variables must be declared before they are referenced. It has been proven that this rule cannot be specified in BNF.
– These problems exemplify the categories of language rules called static semantics rules. The **static semantics** of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics). Many static semantic rules of a language state its type constraints.
– Static semantics is so named because the analysis required to check these specifications can be done at compile time. Because of the problems of describing static semantics with BNF, a variety of more powerful mechanisms has been devised for that task. One such mechanism, attribute grammars, was designed by Knuth to describe both the syntax and the static semantics of programs.
– Attribute grammars are a formal approach both to describing and checking the correctness of the static semantics rules of a program. Although they are not always used in a formal way in compiler design, the basic concepts of attribute grammars are at least informally used in every compiler.

## 2. Basic Concepts

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

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

## 3. Attribute Grammars Defined

– An attribute grammar is a grammar with the following additional features:
  - Associated with each grammar symbol X is a set of attributes A(X). The set A(X) consists of two disjoint sets S(X) and I(X), called synthesized and inherited attributes, respectively. **Synthesized attributes** are used to pass semantic information up a parse tree, while **inherited attributes** pass semantic information down and across a tree.
  - Associated with each grammar rule is a set of semantic functions and a possibly empty set of predicate functions over the attributes of the symbols in the grammar rule. For a rule the $X_0 \rightarrow X_1 \ldots X_n$, synthesized attributes of $X_0$ are computed with semantic functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$. So the value of a synthesized attribute on a parse tree node depends only on the values of the attributes on that node's children nodes. Inherited attributes of symbols $X_j$, $1 \leq j \leq n$ (in the rule above), are computed with a semantic function of the form $I(X_j = f(A(X_1), \ldots, A(X_n))$. So the value of an inherited attribute on a parse tree node depends on the attribute values of that node's parent node and those of its sibling nodes. Note that, to avoid circularity, inherited attributes are often restricted to functions of the form $I(X_j) = f(A(X_0), \ldots, A(X(_{j-1})))$. This form prevents an inherited attribute from depending on itself or on attributes to the right in the parse tree.
  - A predicate function has the form of a Boolean expression on the union of the attribute set $\{A(X_0), \ldots, A(X_n)\}$ and a set of literal attribute values. The only derivations allowed with an attribute grammar are those in which every predicate associated with every nonterminal is true. A false predicate function value indicates a violation of the syntax or static semantics rules of the language.
– A parse tree of an attribute grammar is the parse tree based on its underlying BNF grammar, with a possibly empty set of attribute values attached to each node.
– If all the attribute values in a parse tree have been computed, the tree is said to be fully attributed.
– Although in practice it is not always done this way, it is convenient to think of attribute values as being computed after the complete unattributed parse tree has been constructed by the compiler.

## 4. Intrinsic Attributes

– Intrinsic attributes are synthesized attributes of leaf nodes whose values are determined outside the parse tree. For example, the type of an instance of a variable in a program could come from the symbol table, which is used to store variable names and their types.
– The contents of the symbol table are set based on earlier declaration statements.
– Initially, assuming that an unattributed parse tree has been constructed and that attribute values are needed, the only attributes with values are the intrinsic attributes of the leaf nodes.
– Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attribute values.

## 5. Examples of Attribute Grammars

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

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

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

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

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

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

– The complete attribute grammar follows in Example 1.

**Example 1 An Attribute Grammar for Simple Assignment Statements**

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

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

– A parse tree of the sentence A = A + B generated by the grammar in Example 1 is shown in Figure 1. As in the grammar, bracketed numbers are added after the repeated node labels in the tree so they can be referenced unambiguously.

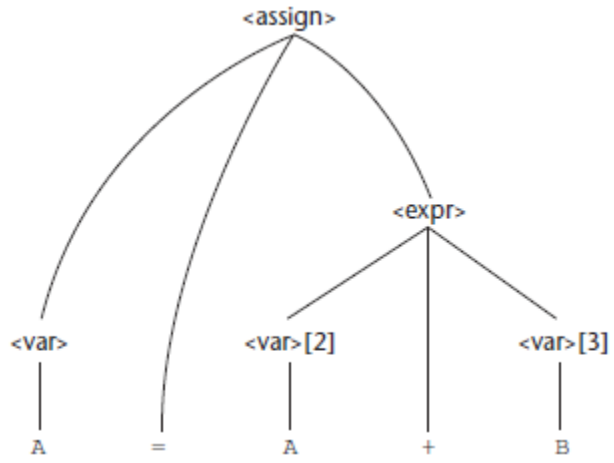


**Figure 1 A parse tree for A = A + B**

## 6. Computing Attribute Values

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

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

- The tree in Figure 2 shows the flow of attribute values in the example of Figure 1. Solid lines show the parse tree; dashed lines show attribute flow in the tree.
- The tree in Figure 3 shows the final attribute values on the nodes. In this example, A is defined as a real and B is defined as an int.
- Determining attribute evaluation order for the general case of an attribute grammar is a complex problem, requiring the construction of a dependency graph to show all attribute dependencies.

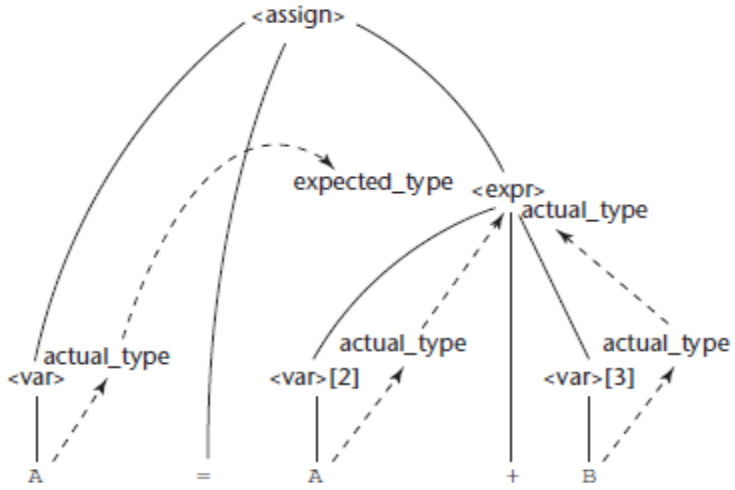


**Figure 2 The flow of attributes in the tree**

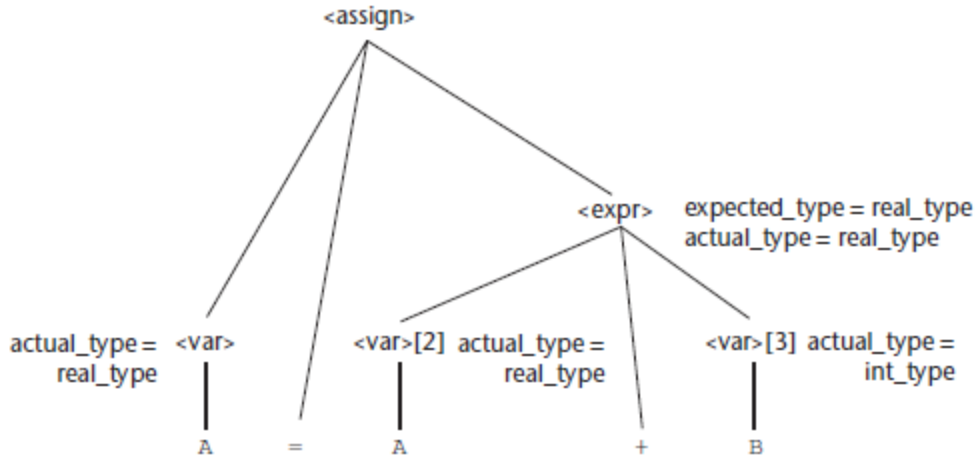


**Figure 3 A fully attributed parse tree**

## 7. Evaluation

- Checking the static semantic rules of a language is an essential part of all compilers. Even if a compiler writer has never heard of an attribute grammar, he or she would need to use the fundamental ideas of attribute grammars to design the checks of static semantics rules for his or her compiler.
- One of the main difficulties in using an attribute grammar to describe all of the syntax and static semantics of a real contemporary programming language is the size and complexity of the attribute grammar.
- The large number of attributes and semantic rules required for a complete programming language make such grammars difficult to write and read.
- Furthermore, the attribute values on a large parse tree are costly to evaluate. On the other hand, less formal attribute grammars are a powerful and commonly used tool for compiler writers, who are more interested in the process of producing a compiler than they are in formalism.