9

Semantic Analysis: Simple Declarations and Expressions

There are three major sections to this chapter. The first section presents some basic concepts related to processing declarations, including the ideas that a symbol table is used to represent information about names and that recursive traversal of an abstract syntax tree can be used to interpret declarations and use the information they provide to check types. The second section describes the tree-traversal "semantic" actions necessary to process a simple declarations of variables and types as found in commonly used programming languages. The third section includes similar semantic action descriptions for type checking abstract syntax trees corresponding to simple assignment statements. We continue to use semantic action descriptions in the next three chapters as a mechanism to discuss the compilation of more complex language features. While this presentation is based on the features of particular languages, the techniques introduced are general enough for use in compiling a wide variety of languages.

9.1 Declaration Processing Fundamentals

9.1.1 Attributes in the Symbol Table

In Chapter 8, symbol tables were presented as a means of associating names with some *attribute information*. We did not specify what kind of information was included in the attributes associated with a name or how it was represented. These topics are considered in this section.

The attributes of a name generally include anything the compiler knows about it. Because a compiler's main source of information about names is declarations, attributes can be thought of as internal representations of declarations. Compilers do generate some attribute information internally, typically from the context in which the declaration of a name appears or, when use is an implicit declaration, from a use of the name. Names are used in many different ways in a modern programming language, including as variables, constants, types, classes, and procedures. Every name, therefore, will not have the same set of attributes associated with it. Rather, it will have a set of attributes corresponding to its usage and thus to its declaration.

Attributes differing by the kind of name represented are easily implemented in either object-oriented or conventional programming languages. In a language like Java, we can declare an Attributes class to serve as the generalized type and then create subclasses of Attributes to represent the details of each specialized version needed. In Pascal and Ada, we can accomplish the same thing using variant records; in C, unions provide the necessary capability.

Figure 9.1 illustrates the style we will use to diagram data structures needed by a compiler to represent attributes and other associated information. In these diagrams, type names are shown in italics (e.g., *VariableAttributes*), while instance variable names and values are shown in the regular form of our program font (e.g., VariableName). This figure illustrates the information needed to represent the attributes of names used for variables and types. It includes two different subclasses of Attributes necessary for describing what the compiler knows about these two different kinds of names.

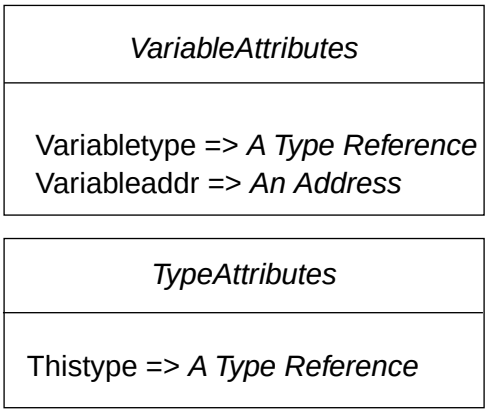

Figure 9.1 Attribute Descriptor Objects

Arbitrarily complex structures, determined by the complexity of the information to be stored, may be used for attributes. Even the simple examples used here include references to other structures that are used to represent type information.

As we outline the semantic actions for other language features, we will define a similar attribute structures for each.

9.1.2 Type Descriptor Structures

Among the attributes of almost any name is a type, which is represented by a *type reference*. A type reference is a reference to a TypeDescriptor object. Representing types presents a compiler writer with much the same problem as representing attributes: There are many different types whose descriptions require different information. As illustrated by the structures in Figure 9.2 by several subtypes of TypeDescriptor, we handle this requirement just as we did for the Attributes structures in the previous figure, with references to other type descriptor structures or additional information (such as the symbol table used to define the fields of a record), as necessary.

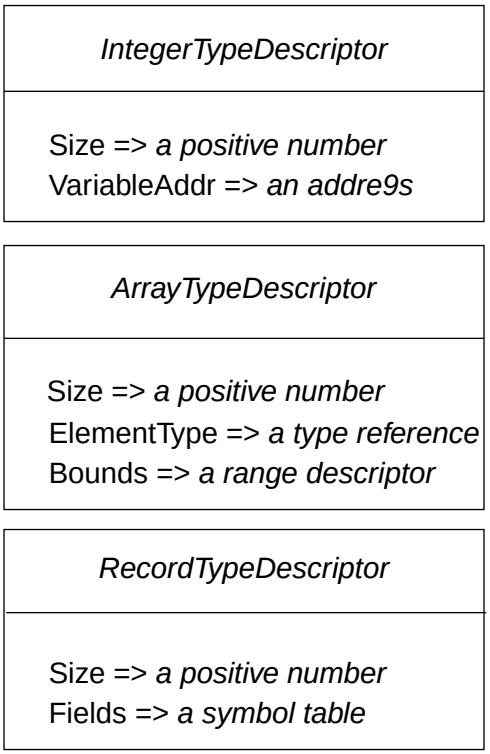

Figure 9.2 Type Descriptor Objects

The kinds of representation shown in this figure are crucial in handling the features of virtually all modern programming languages, which allow types to be constructed using powerful composition rules. Using this technique rather than some kind of fixed tabular representation also makes the compiler much more flexible in what it can allow a programmer to declare. For example, using this approach, there is no reason to have an upper bound on the number of dimensions allowed for an array or the number of fields allowed in a record or class. Such limitations in early languages like FORTRAN stem

purely from implementation considerations. We generally favor techniques that enable a compiler to avoid rejecting a legal program because of the size of the program or some part of it. Dynamic linked structures like the type descriptor structure are the basis of such techniques, as will be seen in this chapter and Chapter 14.

9.1.3 Type Checking Using an Abstract Syntax Tree

Given that we have a program represented by an abstract syntax tree, we can implement declaration processing and type checking (the semantic analysis task of our compiler) by a recursive walk through the tree. The attraction of this approach it that such a tree traversal can be specified quite easily through simple description of the actions to be performed when each kind of tree node is encountered. When each tree node is represented as an object, the necessary action can be expressed as a commonly-named method defined for all alternative subtypes of the node type. We will use the name Semantics for this method that performs the semantic analysis task.

If the implementation language does not support objects, this recursive traversal can be performed by a single routine called Semantics. Its body is most likely organized as a large case or switch statement with an alternative for each kind of node in an abstract syntax tree.

In the rest of this chapter and the three following chapters, we will describe the actions to be performed for each kind of tree node used to represent individual language features. Our actions can be interpreted as method bodies or switch/case alternatives, depending on the implementation most familiar to the reader.

Once execution of the tree traversal defined by Semantics is completed, the analysis phase of the compiler is complete. A second traversal can then be performed to synthesize intermediate code or target code. (A tree optimization pass could be done prior to code generation, particularly in the case where target code is to be generated.) We will define this second traversal similarly via methods called CodeGen. It has a structure much like that of the Semantics pass and we will similarly define it by defining the individual actions for the nodes corresponding to each language features.

Assume that a program is represented by an abstract syntax tree rooted in a Program_Node that includes as components references to two subtrees: one for a list of declarations and another for a list of statement., The common structure of the Semantics and CodeGen actions for this Program_Node are seen in Figure 9.3. Each simply invokes an equivalent operation on each of its subtrees.

```
PROGRAM_NODE.SEMANTICS( )
 1.  Declarations.Semantics()
 2.  Statements.Semantics()

PROGRAM_NODE.CODEGEN( )
 1.  Declarations.CodeGen()
 2.  Statements.CodeGen()
```

Figure 9.3 Traversing Program Trees

The Semantics and CodeGen actions for the lists of declarations and statements would likewise invoke the Semantics and CodeGen actions for each of the declarations and statements in the lists. Our primary focus in the rest of this chapter and the following chapters will be the actions needed to implement individual declarations and statements, as well as their components, particularly expressions.

9.2 Semantic Processing for Simple Declarations

We begin by examining the techniques necessary to handle declarations of variables and scalar types. Structured types will be considered in Chapter 13.

9.2.1 Simple Variable Declarations

Our study of declaration processing begins with simple variable declarations, those that include only variable names and types. (We will later consider the possibility of attaching various attributes to variables and providing an initial value.) Regardless of the exact syntactic form used in a particular language, the meaning of such declarations is quite straightforward: the identifiers given as variable names are to be entered into the symbol table for the current scope as variables. The type that is part of the declaration must be a component of the attributes information associated with each of the variables in the symbol table.

The abstract syntax tree built to represent a single variable declaration with the type defined by a type name is shown in Figure 9.4. (We will consider more general type specifications later.) Regardless of language syntax details, like whether the type comes before or after the variable name, this abstract syntax tree could be used to represent such declarations. If the declaration syntax includes a list of identifiers instead of only a single one, we can simply build a list of VarDecl_Nodes.

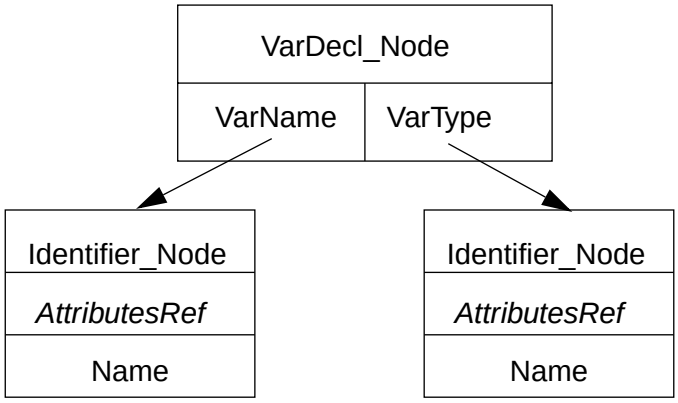

Figure 9.4 AST for a Variable Declaration

The Semantics action for variable declarations illustrates use of specialized semantic routines to process parts of the abstract syntax tree. As seen in this simple case, an Identifier_Node can be used in multiple contexts in a syntax tree. So, to define some context-specific processing to be done on it (rather than "default" actions), we specify invocation of a specialized actions like TypeSemantics. TypeSemantics attempts to interpret the Identifier_Node as describing a type. In fact, we will use TypeSemantics to access the type specified by any arbitrary subtree used where we expect a type specification.

The Semantics action on a VarDecl_Node, shown in Chapter 9.5, implements the meaning that we have already described for a variable declaration. Namely, it enters a into the current symbol table and associates an Attributes descriptor with it, indicating that the name is a variable and recording its type.

```
VARDECL_NODE.SEMANTICS( )
1.  Enter VarName.Name in the current symbol table
2.  if it is already there
3.      then  Produce an error message indicating a duplicate declaration
4.      else  Associate a VariableAttributes descriptor with the Id
              indicating:
                - its type that provided by invoking
                  VarType.TypeSemantics ()
```

Figure 9.5 VARDECL_NODE.SEMANTICS

The two routines in Chapter 9.6 process the type name. The first, TypeSemantics, is invoked in step 4 of VarDecl_Node.Semantics to interpret the type name provided as part of the variable declaration. The second is just the basic semantic action that accesses the attribute information associated with any name in the symbol table.

9.2.2 Type Definitions, Declarations, and References

In the previous section, we saw how type name references were processed, since they were used in the semantic processing for variable declarations. We now consider how type declarations are processed to create the structures accessed by type references. A type declaration in any programming language includes a name and a description of the corresponding type. The abstract syntax tree shown in Figure 9.4 can be used to represent such a declaration regardless of its source-level syntax. The Semantics action to declare a type (Figure 9.8) is similar for that used for declaring a variable: the type identifier must be entered into the current symbol table and an Attributes descriptor associated with it. In this case the Attributes descriptor must indicate that the identifier names a type and must contain a reference to a TypeDescriptor for the type it names.

```
IDENTIFIER_NODE.TYPE.SEMANTICS( )
1.  this_node.Semantics ()
2.  if AttributesRef indicates that Name is a type name
3.      then returnthe value of AttributesRef.ThisType()
4.      else Produce an error message indicating that Name cannot be interpreted as a type
5.           returnErrorType

IDENTIFIER_NODE.SEMANTICS( )
1.  Look Name up in the symbol table
2.  if it is found there
3.      then Set AttributesRef to the Attributes descriptor associated with Name
4.      else Produce an error message indicating that Name has not been declared
5.           Set AttributesRef to null
```

Figure 9.6 IDENTIFIER_NODE.TYPE.SEMANTICS

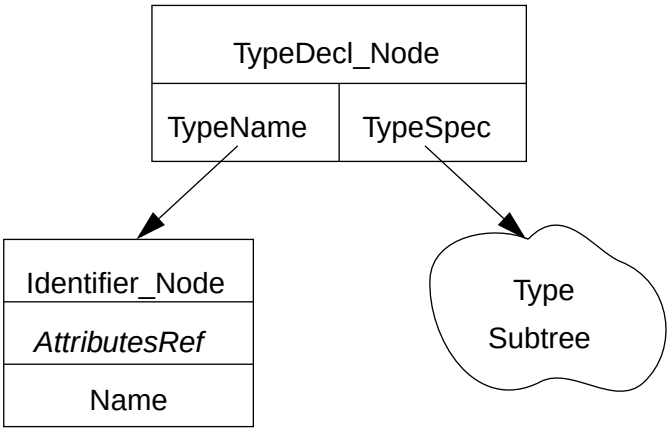

Figure 9.7 AST for a Type Declaration

Figure 9.7 and the psuedocode in Figure 9.8 leave unanswered the obvious question of what kind of subtree is pointed to by TypeSpec. This part of the abstract syntax tree defines the type represented by TypeName. Since type names are given to various types defined by a programmer, TypeSpec can point to a subtree that represents any form of type constructor allowed by the language being compiled. In Section 9.2.4, we will see how a simple type definition is processed. In Chapter 14, we will see how to compile to very commonly used type constructors: definitions of record and array types.

```
TYPEDECL_NODE.SEMANTICS( DeclType) )
1.  Enter the TypeName.Name in the current symbol table
2.  if it is already there
3.      then  Produce an error message indicating a duplicate declaration
4.      else  Associate a TypeAttributes descriptor with the
              Name indicating:
                - the type referenced is that provided by invoking TypeSpec.TypeSemantics ()
```

Figure 9.8 TYPEDECL_NODE.SEMANTICS

By using TypeSemantics (rather than Semantics) to process the subtree referenced by TypeSpec, we require that the semantic processing for a type definition result in the construction of a TypeDescriptor. As we have just seen, the Semantics action for TypeDecl_Node then associates a reference to that TypeDescriptor with a type name. Notice, however, that things work just as well if a type definition rather than a type name is used in a variable declaration. In the previous section, we assumed that the type for a variable declaration was given by a type name. However, since we processed that name by calling TypeSemantics, it is actually irrelevant to the variable declaration psuedocode whether the type is described by a type name or by a type definition. In either case, processing the type subtree with TypeSemantics will result in a reference to a TypeDescriptor being returned. The only difference is that in one case (a name), the reference is obtained from the symbol table, while in the other (a type definition), the reference is to a descriptor created by the call to TypeSemantics.

Handling Type Declaration and Type Reference Errors. In the psuedocode for the TypeSemantics action for an identifier, we introduce the idea of a *static semantic check*. That is, we define a situation in which an error can be recognized in a program that is syntactically correct. The error in this case is a simple one: a name used as a type name does not actually name a type. In such a situation, we want our compiler to generate an error message. Ideally, we want to generate only one error message per error, even if the illegal type name is used many times. (Many compilers fall far short of this ideal!). The simplest way to accomplish this goal is to immediately stop the compilation, though it is also at least approachable for a compiler that tries to detect as many errors as possible in a source program.

Whenever TypeSemantics finds that a type error, it returns a reference to a special TypeDescriptor that we refer to as ErrorType. That particular value is a signal to the code that called TypeSemantics that an error has been detected and that an error message has already been generated. In this calling context, an explicit check may be made for ErrorType or it may be treated like any other TypeDescriptor. In the specific instance of the variable declaration psuedocode, the possibility of an ErrorType return can be ignored. It works perfectly well to declare variables with ErrorType as their type. In fact, doing so prevents later, unnecessary error messages. That is, if a variable is left

undeclared because of a type error in its declaration, each time it is used, the compiler will generate an undeclared variable error message. It is clearly better to declare the variable with ErrorType as its type in order to avoid these perhaps confusing messages. In later sections, we will see other uses of ErrorType to avoid the generation of extraneous error messages.

Type Compatibility. One question remains to be answered: Just what does it mean for types to be the same or for a constraint (as used in Ada) to be compatible with a type? Ada, Pascal, and Modula-2 have a strict definition of type equivalence that says that every type definition defines a new, distinct type that is incompatible with all other types. This definition means that the declarations

```
A, B : ARRAY (1..10) OF Integer;
C, D : ARRAY (1..10) OF Integer;
```

are equivalent to

```
type Type1 is ARRAY (1..10) OF Integer;
A, B : Type1;
type Type2 is ARRAY (1..10) OF Integer;
C, D : Type2;
```

`A` and `B` are of the same type and `C` and `D` are of the same type, but the two types are defined by distinct type definitions and thus are incompatible. As a result, assignment of the value of `C` to `A` would be illegal. This rule is easily enforced by a compiler. Since every type definition generates a distinct type descriptor, the test for type equivalence requires only a comparison of pointers.

Other languages, most notably C, C++ and Java, use different rules to define type equivalence, though Java does use name equivalence for classes. The most common alternative is to use *structural* type equivalence. As the name suggests, two types are equivalent under this rule if they have the same definitional structure. Thus Type1 and Type2 from the previous example would be considered equivalent. At first glance, this rule seems like a much more useful choice because it is apparently more convenient for programmers using the language. However, counterbalancing this convenience is the fact the structural type equivalence rule makes it impossible for a programmer to get full benefit from the concept of type checking. That is, even if a programmer wants the compiler to distinguish between Type1 and Type2 because they represent different concepts in the program despite their identical implementations, the compiler is unable to do so.

Structural equivalence is much harder to implement, too. Rather than being determined by a single pointer comparison, a parallel traversal of two type descriptor structures is required. The code to do such a traversal requires special cases for each of the type descriptor alternatives. Alternatively, as a type definition is processed by the semantic checking pass through the tree, the type being defined can be compared against previously defined types so that equivalent types are represented by the same data structure, even though they are defined separately. This technique allows the type equivalence test to be implemented by a pointer comparison, but it requires an indexing mechanism that makes it possible to tell during declaration processing whether each newly defined type is equivalent to *any* previously defined type.

Further, the recursion possible with pointer types poses subtle difficulties to the implementation of a structural type equivalence test. Consider the problem of writing a routine that can determine whether the following two Ada types are structurally equivalent, given the following example. (Access types in Ada roughly correspond to pointers in other languages.)

```
type A is access B;
type B is access A;
```

Even though such a definition is meaningless semantically, it is syntactically legal. Thus a compiler for a language with structural type equivalence rules must be able to make the appropriate determination – that `A` and `B` are equivalent. If parallel traversals are used to implement the equivalence test, the traversal routines must "remember" which type descriptors they have visited during the comparison process in order to avoid an infinite loop. Suffice it to say that comparing pointers to type descriptors is much simpler!

9.2.3 Variable Declarations Revisited

The variable declaration can be more complex than the simple form shown in Section 9.2.1. Many languages allow various modifiers, such as constant or static, to be part of a variable declaration. Visibility modes like public, private and protected may also be part of a declaration, particularly in object-oriented languages. Finally, an initial value might be specified. The abstract syntax tree node for a variable declaration obviously must be more complex than the one in Section 9.4 to encompass all of these possibilities. Thus we need a structure like the one we see in Figure 9.9.

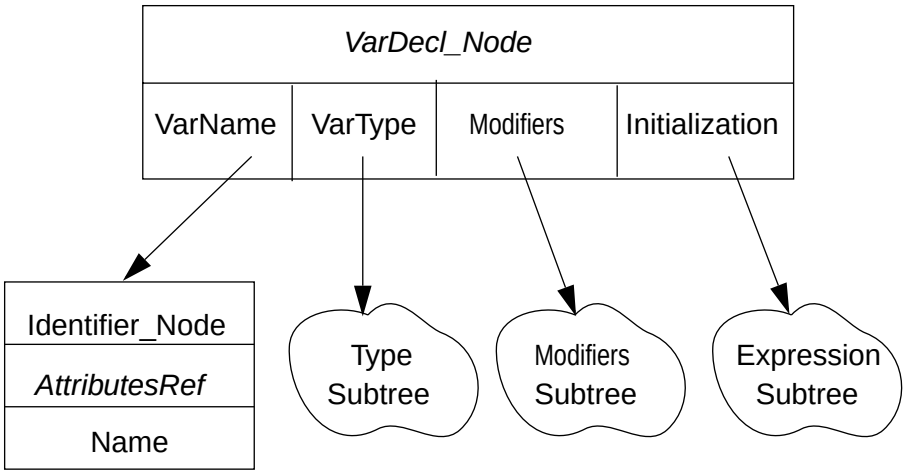

Figure 9.9 AST for a Complete Variable Declaration

Both the modifiers and initialization parts are optional; in the case that neither are present, we are left with the same information as in the simple version of this node seen

inSection 9.2.1. When present, the expression subtree referenced by Initialization may be a literal constant whose value can be computed at compile time or its value may be determined by an expression that cannot be evaluated until run-time. If it has a compile-time value and constant is one of the Modifiers, the "variable" being declared is actually a constant that may not even require run-time space allocation. Those that do require space at run-time act much like variables, though the Attributes descriptor of such a constant must include an indication that it may not be modified. We will leave the structure of the Modifiers subtree unspecified, vary according to the specification of the particular language being compiled. For a language that allows only a single modifier (constant, perhaps), there need be no subtree since a single boolean component in the VarDecl_Node will suffice. At the other extreme, Java allows several modifiers to be present along with a name being declared, necessitating use of a list or some other structure with sufficient flexibility to handle all of the possible combinations of modifiers.

The Semantics action for this extended VarDecl_Node (in Figure 9.10) handles the specified type and the list of variable or constant names much as they were handled for the simpler VarDecl_Node in Section 9.2.1.

9.2.4 Enumeration Types

An enumeration type is defined by a list of distinct identifiers denoting its values; each identifier is a constant of the enumeration type. Enumeration constants are ordered by their position in the type definition and are represented internally by integer values. Typically, the value used to represent the first identifier is zero, and the value for each subsequent identifier is one more than that for its predecessor in the list (though some languages do allow a programmer to specify the values used to represent enumeration literals).

The abstract syntax tree representation for an enumeration type definition is shown in Figure 9.11. The TypeSemantics action for processing an EnumType_Node, like those for processing records and arrays, will build a TypeDescriptor that describes the enumeration type. In addition, each of the identifiers used to define the enumeration type is entered into the current symbol table. Its attributes will include the fact that it is an enumeration constant, the value used to represent it, and a reference to the TypeDescriptor for the enumeration type itself. The type will be represented by a list of the Attributes records for its constants. The required specializations of Attributes and TypeDescriptor are illustrated in Figure 9.12 and the psuedocode for the enumeration type semantics action is in Figure 9.13.

Figure 9.14 shows the TypeDescriptor that would be constructed to represent the enumeration type defined by

```
(red, yellow, blue, green)
```

The Size of the enumeration type is set during the CodeGen pass through the tree to the size of an integer (or perhaps just a byte), which is all that is needed to store an enumeration value.

```
VARDECL_NODE.SEMANTICS( )
1.  Local Variables: DeclType, InitType
2.  Set DeclType To VarType.TypeSemantics()
3.  if Initialization is not a null pointer
4.      then  Set Inittype to Initialization.ExprSemantics ()
5.            if not Assignable (InitType, DeclType)
6.                then  Generate an appropriate error message
7.                      Set DeclType to ErrorType
        else  • Initialization Is Null
8.            Check that Constant is not among the modifiers
9.  Enter VarName.Name in the current symbol table
10. if it is already there
11.     then  Produce an error message indicating a duplicate declaration
12. elsif Constant is among the modifiers
13.     then  Associate a ConstAttributes descriptor with it indicating:
14.             - Its type is DeclType

                - Its value is defined by Initialization (a pointer to an expression tree)
15.     else  • It is a variable
16.           Associate a VariableAttributes descriptor with it indicating:
17.             - Its type is DeclType
18.             - it modifiers are those specified by Modifiers
19.           • Any initialization will be handled by the code generation pass
```

Figure 9.10 VARDECL_NODE.SEMANTICS (VERSION 2)

9.3 Semantic Processing for Simple Names and Expressions: An Introduction to Type Checking

The semantic processing requirements for names and expressions can be easily understood in the context of an assignment statement. As illustrated in the AST representations shown in Figure 9.15, an assignment is defined by a name that specifies the target of the assignment and an expression that must be evaluated in order to provide the value to be assigned to the target. The name may be either a simple identifier or a qualified identifier, the latter in the case of a reference to a component of a data structure or object.

In this chapter, we will only consider the case where the target is a simple identifier, which can be represented by the same Identifier_Node that has been used n several other contexts in this chapter. The simplest form of an expression is also just a name (as seen in the assignment statement `A = B;`) or a literal constant (as in `A = 5;`). The section that follows will present the Semantics actions necessary to handle these simple

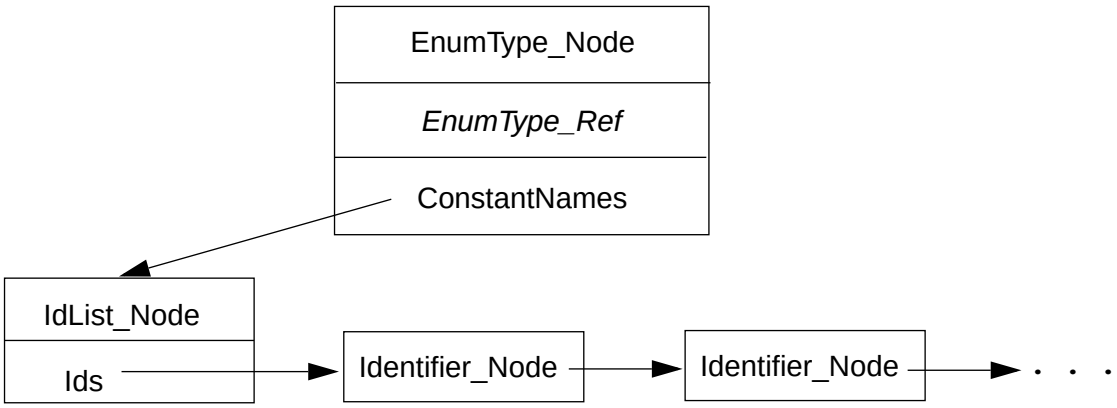

Figure 9.11 AST for an Enumeration Type Definition

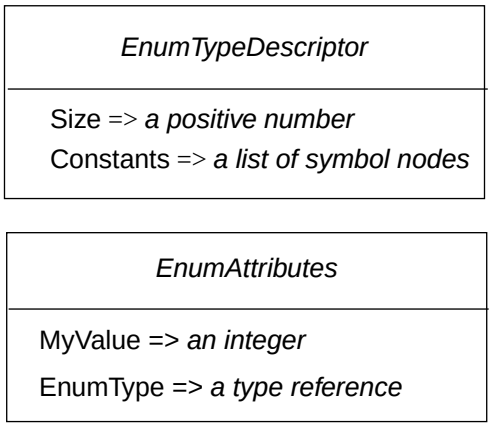

Figure 9.12 Type and Attribute Descriptor Objects for Enumerations

kinds of expressions and assignments. Section 9.3.2 will deal with expressions involving unary and binary operators and in Chapter 14 the techniques needed to compile simple record and array references will be presented.

9.3.1 Handling Simple Identifiers and Literal Constants

The syntax trees for these assignments are the simple ones seen in Figure 9.9. The Semantics actions needed to handle these assignment statements are equally simple. However, thoughtful examination of the tree for `A = B` leads to the observation that the two identifiers are being used in rather different ways. The `A` stands for the address of `A`, the location that receives the assigned value. The `B` stands for the value at the address associated with `B`. We say that the `A` is providing an L-value (an address) because this the

```
ENUMTYPE_NODE.TYPESEMANTICS( ) )
1.    [Local Variable: Nextvalue, Enumtype_ref]
2.    Create An Enumtypedescriptor For The New Enumeration
        Type With:
3.          Constants Pointing To An Empty List Of Symbols
4.    Set Enumtype_ref To Point To This Enumtypedescriptor
5.    Set Nextvalue To 0
6.    for     Each Of The Identifiers In The Constantnames List
7.          do  Enter The Identifier In The Current Symbol Table
8.              Associate An Enumattributes Descriptor With It
                  Indicating:
9.                        Its Type Is The Typedescriptor Pointed To
                            By Enumtype_ref
10.                       Its Value Is Nextvalue
11.             Increment Nextvalue
12.   return  Enumtype_ref
```

Figure 9.13 ENUMTYPE_NODE.TYPESEMANTICS

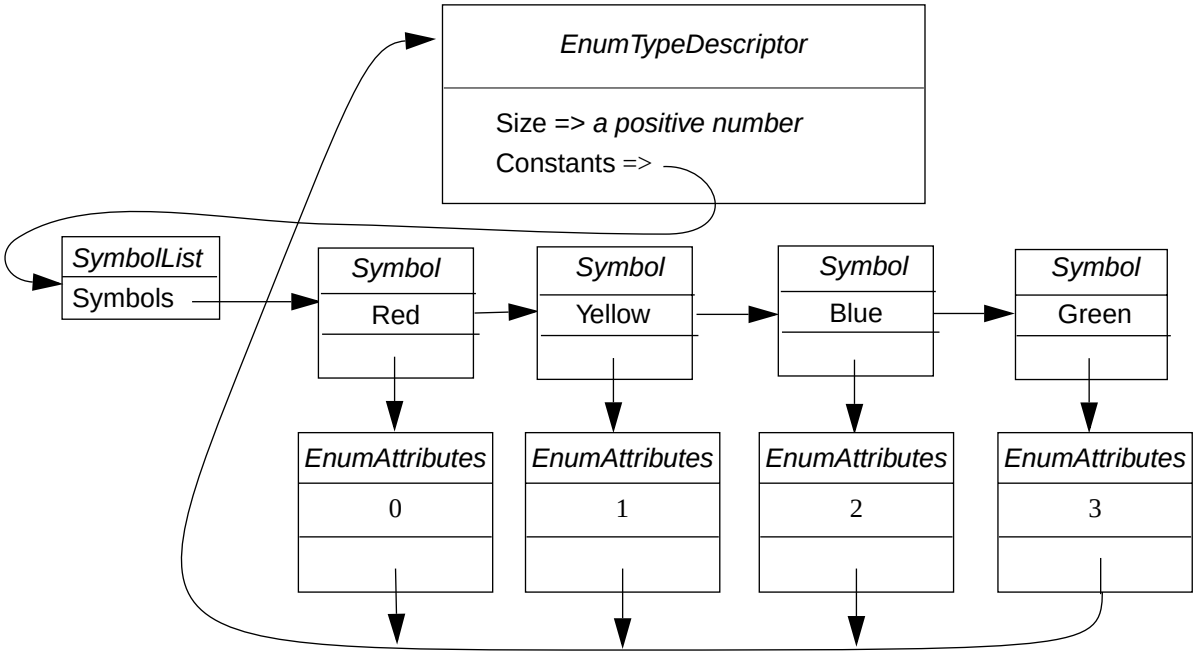

Figure 9.14 Representation of an Enumeration Type

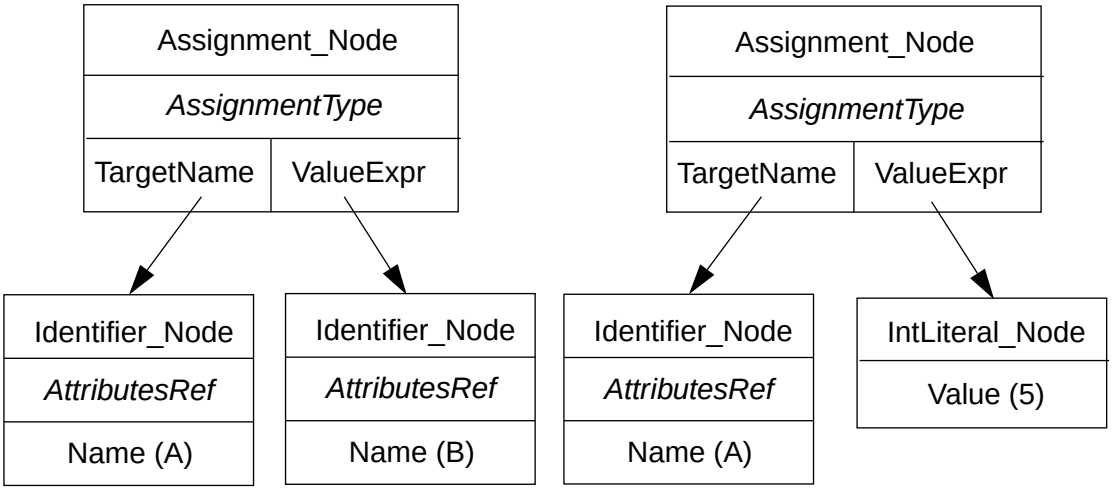

Figure 9.15 AST Representation of an Assignment Statement

meaning of an identifier on the left-hand-side of an assignment statement. Similarly, we say that `B` denotes an R-value (a value or the contents of an address) because that is how we interpret an identifier on the right-hand-side of an assignment. From these concepts, we can infer that when we consider more general assignment statements, the TargetName subtree must supply an L-value, while the ValueExpr subtree must supply an R-value.

Since we would like to be able to treat all occurrences of Identifier_Node in an expression tree consistently, we will say that they always provide an L-value. When an identifier is used in a context where an R-value is required, the necessary conversion must be implicitly performed. This requirement actually has no implication on type checking in most languages, but will require that appropriate code be generated based on the context in which a name is used.

The major task of the Assignment_Node Semantics action, seen in Figure 9.20, involves obtaining the types of the components of the assignment statement and checking whether they are compatible according to the rules of the language. First, the target name is interpreted using a specialized version of Semantics called LvalueSemantics. It insures that the target designates an assignable entity, that is, one that produces an L-value. Named constants and constant parameters look just like variables and can be used interchangeably when an R-value is required, but they are not acceptable target names. A semantic attribute *AssignmentType* is used to record the type to be assigned after this determination has been made so that the type is easily available to the CodeGen actions that will be invoked during the next pass.

A second specialized version of Semantics, called ExprSemantics, is used to evaluate the type of an expression AST subtree. Figure 9.17 includes two versions of ExprSemantics (and more will be seen shortly, for processing binary and unary expressions). The one for Identifier_Node makes use of the general Identifier_Node.Seman-

```
ASSIGNMENT_NODE.SEMANTICS( )
1.  [Local variables: LeftType, RightType]
2.  Set LeftType to TargetName.LvalueSemantics ()
3.  Set RightType to ValueExpr.ExprSemantics ()
4.  if RightType is assignable to LeftType
5.      then  Set AssignmentType to LeftType
6.      else  Generate an appropriate error message
7.            Set AssignmentType to ErrorType

IDENTIFIER_NODE.LVALUESEMANTICS ( )
1.  Call this_node.Semantics ()
2.  if AttributesRef indicates that Name denotes a L-value
3.      then  return the type associated with Name
4.      else  Produce an error message indicating that Name cannot be interpreted as a variable
5.            return ErrorType
```

Figure 9.16

tics action defined earlier in this chapter (in Section 9.2). It simply looks up Name in the symbol table and sets the semantic attribute *AttributesRef* to references Name's attributes. Note that both LvalueSemantics and ExprSemantics are much like TypeSemantics, which was defined along with Identifier_Node.Semantics in Section 9.2.2. They differ only in the properties they require of the name they are examining. The ExprSemantics action for an integer literal node is quite simple. No checking is necessary, since a literal is known to be an R-value and its type is immediately available from the literal node itself.

```
IDENTIFIER_NODE.EXPRSEMANTICS ( )
1.  Call this_node.Semantics ()
2.  if AttributesRef indicates that Name denotes an R-value (or an
      L-value which can be used to provide one)
3.      then  return the type associated with Name
4.      else  Produce an error message indicating that Name cannot be interpreted as a variable
5.            return ErrorType

INTLITERAL_NODE.EXPRSEMANTICS( )
1.  return  IntegerType
```

Figure 9.17 EXPRSEMANTICS for Identifiers and Literals

We will see the type checking alternatives for more complex names, such as record field and array element references, in a later chapter. They will be invoked by the same calls from the Semantics actions for an Assignment_Node and they will return types just as in the simple cases we have just seen. Thus the processing done for an assignment can completely ignore the complexity of the subtree that specifies the target of the assignment, just as it need not consider the complexity of the computation that specifies the value to be assigned.

9.3.2 Processing Expressions

In the previous section, we saw two examples of the general concept of an expression, namely, a simple identifier and a literal constant. More complex expressions in any programming language are constructed using unary and binary operators. The abstract syntax tree representations for expressions follow naturally from their common syntactic structure, as illustrated in Figure 9.18. The nodes referenced by LeftExpr, RightExpr and SubExpr in the figure can be additional expression nodes or they can be from among the group that specifies operands (that is, literals, identifiers, etc.). As discussed in the previous section, the semantic processing and code generation actions for all of these nodes will behave in a consistent fashion, which will enable the actions for expression processing to be independent of concerns about the details of the expression's operands.

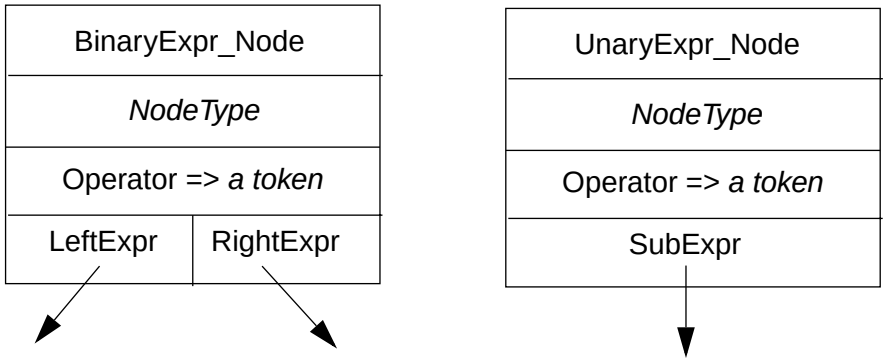

Figure 9.18 AST Representations for Expressions

The type checking aspects of handling expression nodes depend primarily on the types of the operands. Thus, both of the ExprSemantics actions below first invoke ExprSemantics on the operands of the expression. The next step is to find whether the operator of the expression is meaningful given the types of the operands. We illustrate this test as being done by the routines BinaryResultType and UnaryResultType in the psuedocode in Figure 9.19. These routines must answer this question based on the definition of the language being compiled. If the specified operation is meaningful, the type of the result of the operation is returned; otherwise, the result of the call must be ErrorType. To consider a few examples: adding integers is defined in just about all programming languages, with the result being an integer. Addition of an integer and a real will

produce a real in most languages, but in a language like Ada that does not allow implicit type conversions, such an expression is erroneous. Finally, comparison of two arithmetic values typically yields a boolean result.

```
BINARYEXPR_NODE.EXPRSEMANTICS ( )
1.  [Local variables: LeftType, RightType]
2.  Set LeftType to LeftExpr.ExprSemantics ()
3.  Set RightType to RightExpr.ExprSemantics ()
4.  Set NodeType to BinaryResultType (Operator, LeftType,
      RightType)
5.  return NodeType

UNARYEXPR_NODE.EXPRSEMANTICS ( )
1.  [Local variable: SubType]
2.  Set SubType to SubExpr.ExprSemantics ()
3.  Set NodeType to UnaryResultType (Operator, SubType)
4.  return NodeType
```

Figure 9.19 EXPRSEMANTICS for Binary and Unary Expressions

9.4 Key Idea Summary

Section 9.1 presented three key ideas: a symbol table is used to associate names with a variety of information, generally referred to as attributes; types are a common, distinguished kind of attribute; and types and other attributes are represented by records (structures) with variants (unions) designed as needed to store the appropriate descriptive information. Section 9.1 also introduced the mechanism of recursive traversal by which Semantics and CodeGen passes over an abstract syntax tree can be used to type check and generate intermediate or target code for a program.

Section 9.2 described the Semantics actions for handling the abstract syntax tree nodes found in simple declarations during the declaration processing and type checking pass done by the Semantics traversal. The concept of using specialized semantics actions, such as TypeSemantics and ExprSemantics was introduced in this section as was the notion of using an ErrorType to deal with type errors in a way so as to minimize the number of resulting error messages.

Finally, Section 9.3 introduced type checking using abstract syntax trees for simple assignments as an example framework. The concept of interpreting names as L-values or R-values was described, along with its implications for AST processing.