CHAPTER FIVE

Contextual Analysis

Given a parsed program, the purpose of contextual analysis is to check that the program conforms to the source language's contextual constraints. For a typical programming language (statically typed and with static bindings), contextual constraints consist of:

- *Scope rules:* These are rules governing declarations and applied occurrences of identifiers.
- *Type rules:* These are rules that allow us to infer the types of expressions, and to decide whether each expression has a valid type.

It follows that contextual analysis consists of two subphases:

- *Identification:* applying the source language's scope rules to relate each applied occurrence of an identifier to its declaration (if any).
- *Type checking:* applying the source language's type rules to infer the type of each expression, and compare that type with the expected type.

In Section 5.1 we study identification, and in Section 5.2 we study type checking. In Section 5.3 we develop a particular contextual analysis algorithm, combining identification and type checking in a single pass, and show how the results of contextual analysis may be recorded. Throughout, we assume that the source language exhibits static bindings and is statically typed.

5.1 Identification

The first task of the contextual analyzer is to relate each applied occurrence of an identifier in the source program to the corresponding declaration. If there is no corresponding declaration, the source program is ill-formed, and the contextual analyzer must generate an error report. This task is called ***identification***. Once an applied occurrence of an identifier has been identified, the contextual analyzer will check that the identifier is used in a way consistent with its declaration: that is type checking, to be considered in Section 5.2.

Identification can have a disproportionate effect on the efficiency of the whole compiler. Longer source programs contain more applied occurrences of identifiers, and

hence require more identifications to be performed. But also, longer source programs contain more declarations, so each identification is likely to take more time – especially if identification is implemented naively. Some compilers (and assemblers) are indeed very slow, for this reason.

If the source program is represented by an AST, a naive identification algorithm would be to search the AST: starting from a leaf node representing an applied occurrence of an identifier, find the subtree representing the corresponding declaration of that identifier. But such an algorithm would be very cumbersome. (See Exercise 5.5.)

A better method is to employ an ***identification table*** that associates identifiers with their attributes. The basic operations on the identification table are as follows:

- Make the identification table empty.
- Add an entry associating a given identifier with a given attribute.
- Retrieve the attribute (if any) associated with a given identifier.

An identifier's ***attribute*** consists of information relevant to contextual analysis, and is obtained from the identifier's declaration. The attribute could be information distilled from the declaration, or just a pointer to the declaration itself. For the moment we need not be specific, since the attributes do not influence the structure of the identification table. We shall return to attributes in Section 5.1.4.

Each declaration in a program has a definite ***scope***, which is the portion of the program over which the declaration takes effect. A ***block*** is any program phrase that delimits the scope of declarations within it. For example, Triangle has a block command, of the form '`let` *D* `in` *C*', in which the scope of each declaration in *D* extends over the subcommand *C*. A Triangle procedure declaration, of the form '`proc` *I* (*FPS*) ~ *C*', is also a block, in which the scope of each formal parameter in *FPS* is the procedure body *C*.

The organization of the identification table depends on the source language's ***block structure***, which is the textual relationship of blocks in programs. There are three possibilities:

- Monolithic block structure (exemplified by Basic and Cobol).
- Flat block structure (exemplified by Fortran).
- Nested block structure (exemplified by Pascal, Ada, C, and Java).

These block structures are covered in the following subsections.

5.1.1 Monolithic block structure

A programming language exhibits ***monolithic block structure*** if the only block is the entire program. All declarations are global in scope.

A language with monolithic block structure has very simple scope rules, typically:

(1a) No identifier may be declared more than once.

(1b) For every applied occurrence of an identifier *I*, there must be a corresponding declaration of *I*. (In other words, no identifier may be used unless declared.)

In the case of monolithic block structure, the identification table should contain entries for all declarations in the source program. There will be at most one entry for each identifier. Each entry in the table consists of an identifier *I* and the attribute *A* associated with it.

Example 5.1 Monolithic block structure

Consider a hypothetical programming language in which a program takes the form:

```
program
    D
begin
    C
end
```

D is a sequence of declarations (the only ones in the program). *C* is a command sequence, the executable part of the program. In this example it is not important what kinds of declaration and command are provided. What is important is that the only block is the whole program.

Figure 5.1 shows a program outline, together with a picture of the identification table after all declarations have been processed. The table contains one entry for each declared identifier. The declarations are numbered for cross-referencing, and in the table each identifier's attribute is shown as a cross-reference to the identifier's declaration.

□

Attributes and identification tables can be defined by the Java classes outlined here:

```
public class Attribute {
    ... // Attribute details.
}
public class IdentificationTable {
    ... // Variables representing the identification table.
    public IdentificationTable ()
    // Make an empty identification table.
    { ... }
    public void enter (String id, Attribute attr)
    // Add an entry to the identification table, associating identifier id
    //  with attribute attr.
    { ... }
```

```
    public Attribute retrieve (String id)
    // Return the attribute associated with identifier id in the identification
    // table. If there is no entry for id, return null.
    { ... }
}
```

The contextual analyzer will use these operations as follows:

- To create a new table, the `IdentificationTable` constructor will be called.
- At a declaration of identifier *I*, the method `enter` will be called to add an entry for *I*.
- At an applied occurrence of identifier *I*, the method `retrieve` will be called to find the entry for *I*. If there is no such entry, an error report will be generated.

The identification table should be organized for efficient retrieval. A good implementation would be a standard data structure such as a binary search tree or a hash table. (See Exercise 5.1.)

```
program
(1) integer b = 10
(2) integer n
(3) char c
begin
   ...
   n = n * b
   ...
   write c
   ...
end
```

Ident.	Attr.
b	(1)
n	(2)
c	(3)

Figure 5.1 Identification table: monolithic block structure.

5.1.2 Flat block structure

A programming language exhibits ***flat block structure*** if a program can be partitioned into several disjoint blocks. There are two scope levels:

- Some declarations are *local* in scope. Applied occurrences of locally declared identifiers are restricted to a particular block.
- Other declarations are *global* in scope. Applied occurrences of locally declared identifiers are allowed anywhere in the program. In effect, the program as a whole is a block, enclosing all the other blocks.

The scope rules for a language with flat block structure might be:

(2a) No globally declared identifier may be redeclared globally. (But the same identifier may also be declared locally.)

(2b) No locally declared identifier may be redeclared in the same block. (But the same identifier may be declared locally in several different blocks.)

(2c) For every applied occurrence of an identifier I in a block B, there must be a corresponding declaration of I. This must be either a global declaration of I or a declaration of I local to B.

In the case of flat block structure, the identification table should contain entries for both global and local declarations. The contents of the table will vary during contextual analysis. During analysis of block B, the table should contain entries both for global declarations and for declarations local to B. Once analysis of B is completed, the entries for local declarations should be discarded. It follows that the entries for local and global declarations must be distinguished in some way.

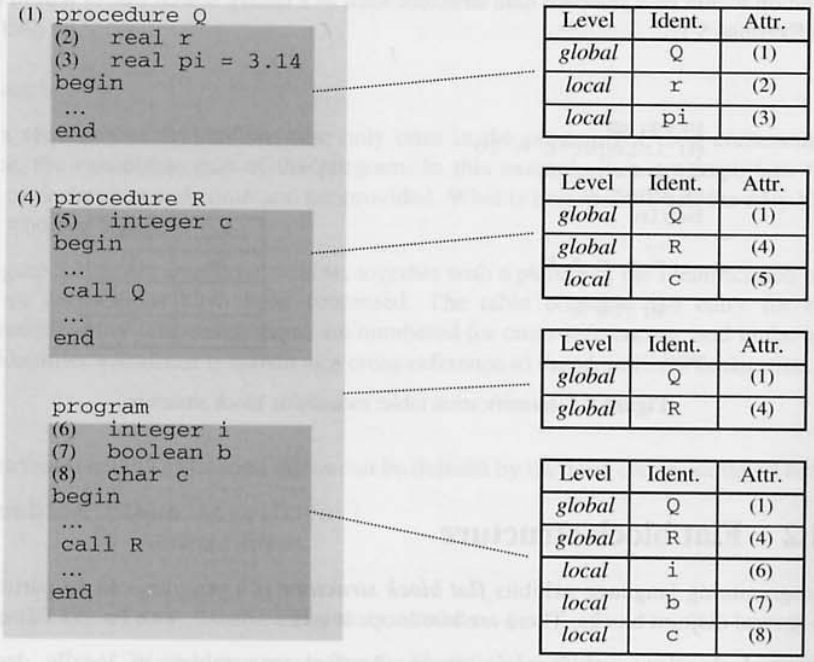


Figure 5.2 Identification table: flat block structure.

Example 5.2 Flat block structure

Consider a hypothetical programming language in which a main program takes the form:

```
program
    D
begin
    C
end
```

The main program's body is a block, and the declarations *D* are local to it. The main program may be preceded by a number of procedure declarations, which take the form:

```
procedure I
    D
begin
    C
end
```

The procedure body is a block, and the declarations *D* are local to it. The procedure declaration itself is global in scope.

Figure 5.2 shows a program outline, with the blocks shaded to distinguish between global and local scopes. It also shows a picture of the identification table as it stands during contextual analysis of each block.

During analysis of procedure `Q`, the table contains a *global* entry for `Q` itself, and *local* entries for `r` and `pi`. The body of `Q` may contain applied occurrences of these identifiers only. If the contextual analyzer encounters any other identifier, that identifier will not be found in the table, and the contextual analyzer will generate an error report. After analysis of `Q`, all the *local* entries are removed from the table. Similar points can be made about the other two blocks.

Note that the program contains two local declarations of identifier `c`. This causes no confusion, because the two declarations are local to different blocks. Their entries never appear in the identification table at the same time.

□

We still need the `IdentificationTable` constructor and the `enter` and `retrieve` methods specified in Section 5.1.1, but the latter method now has a slightly more complicated specification:

```
public Attribute retrieve (String id)
// Return the attribute associated with identifier id in the identification
// table. If there are both global and local entries for id, return the attribute
// from the local entry. If there is no entry for id, return null.
{ ... }
```

In addition, we need the following new methods:

```
public void openScope ()
// Add a local scope level to the identification table, with no entries yet
// belonging to it.
{ ... }

public void closeScope ()
// Remove the local scope level from the identification table, and all
// entries belonging to it.
{ ... }
```

The contextual analyzer will use the operations as follows:

- To create a new table, the `IdentificationTable` constructor will be called.
- At the start of a block, `openScope` will be called.
- At the end of a block, `closeScope` will be called.
- At a declaration of identifier *I*, `enter` will be called to add an entry for *I*. If `openScope` has been called but not canceled by `closeScope`, the new entry will be marked as *local*; otherwise it will be marked as *global*.
- At an applied occurrence of identifier *I*, `retrieve` will be called to find the entry for *I*. If there is no such entry, an error report will be generated.

It is still easy to implement the identification table. The only minor complication is to distinguish the global and local declaration entries. (See Exercise 5.2.)

5.1.3 Nested block structure

A programming language exhibits ***nested block structure*** if blocks may be nested one within another. Thus there may be many scope levels:

- Declarations in the outermost block are global in scope. We say that the outermost block is at *scope level 1*.
- Declarations inside an inner block are local to that block. Every inner block is completely enclosed by another block. If enclosed by the outermost block, we say that the inner block is at *scope level 2*; if enclosed by a level-2 block, we say that the inner block is at *scope level 3*; and so on.

The scope rules for a language with nested block structure are typically as follows:

(3a) No identifier may be declared more than once in the same block. (But the same identifier may be declared in different blocks, even if they are nested.)

(3b) For every applied occurrence of an identifier *I* in a block *B*, there must be a corresponding declaration of *I*. This declaration must be in *B* itself, or (failing that) in the block *B´* that immediately encloses *B*, or (failing that) in the block *B´´* that immediately encloses *B´*, etc. (In other words, the corresponding declaration is in the smallest enclosing block that contains any declaration of *I*.)

In the case of nested block structure, the identification table should contain entries for declarations at all scope levels. Again, the contents of the table will vary during contextual analysis. During analysis of block B, the table should contain entries for declarations in B, entries for declarations in the block B' that encloses B, entries for declarations in the block B'' that encloses B', etc. Once analysis of B is completed, the entries for the declarations in B should be discarded. To make this possible, each entry should contain a scope level number.

Example 5.3 Nested block structure

The language Mini-Triangle introduced in Example 1.3 has block commands of the form '`let` *D* `in` *C*'. These may be nested. Mini-Triangle's scope rules are (3a) and (3b) above.

Figure 5.3 shows a program outline, with the blocks shaded to indicate their scope levels. It also shows a picture of the identification table as it stands during contextual analysis of each block.

During analysis of the outermost block, the table contains only entries for identifiers `a` (declaration (1)) and `b` (declaration (2)). These entries are marked as level 1.

During analysis of the innermost block, the table contains entries for all the declarations in this block (marked as level 3), the enclosing block (level 2), and the outermost block (level 1). Notice that there are two entries for `b` (declarations (2) and (3)), but this is legitimate since they are in different blocks, and so their scope levels are different. If the innermost block contains an applied occurrence of `b`, the table must be searched in such a way as to retrieve attribute (3) – in accordance with scope rule (3b).

□

We still need the `IdentificationTable` constructor, and the `enter`, `retrieve`, `openScope`, and `closeScope` methods, but some of these now have modified specifications:

```
public Attribute retrieve (String id)
// Return the attribute associated with identifier id in the
// identification table. If there are several entries for id,
// return the attribute from the entry at the highest scope level.
// If there is no entry for id, return null.
{ ... }

public void openScope ()
// Add a new highest scope level to the identification table.
{ ... }

public void closeScope ()
// Remove the highest scope level from the identification table,
// and all entries belonging to it.
{ ... }
```

These are generalizations of the operations specified in Section 5.1.2.

The contextual analyzer will use the operations as follows:

- To create a new, empty table, the `IdentificationTable` constructor will be called.
- At the start of a block, `openScope` will be called.
- At the end of a block, `closeScope` will be called.
- At a declaration of identifier *I*, `enter` will be called to add an entry for *I*. (This entry will contain a scope level number determined by the number of calls of `openScope` not yet canceled by calls of `closeScope`.)
- At an applied occurrence of identifier *I*, `retrieve` will be called to find the correct entry for *I*. If there is more than one entry for *I*, the one with the highest scope level number will be retrieved. If there is no entry for *I*, an error report will be generated.

Nested block structure makes implementation of the identification table a more challenging problem. There may be several entries for each identifier, although there is at most one entry for each (scope level, identifier) combination. The table must be searched in such a way that the highest-level entry is retrieved when there are several entries for the same identifier. And, as usual, retrieval efficiency is important. Some possible implementations are outlined in Section 5.4 and in Exercises 5.4 and 5.5.

5.1.4 Attributes

So far we have been deliberately unspecific about the nature of the attributes associated with identifiers in the identification table. These attributes are stored in the table, and later retrieved, but they have no influence on the *structure* of the table.

Let us now look at these attributes in more detail. At an applied occurrence of an identifier *I*, the attribute associated with *I* is retrieved for use in type checking. If *I* occurs as an operand in an expression, the type checker will need to ensure that *I* has been declared as a constant or variable, and will need to know its type. If *I* occurs as the left-hand side of an assignment command, the type checker will need to ensure that *I* has been declared as a variable (not a constant), and again will need to know its type. If *I* occurs as the first symbol in a procedure call, the type checker will need to ensure that *I* has indeed been declared as a procedure, and will need to know the types of its formal parameters (for comparison with the types of the actual parameters). These examples illustrate the kind of information that must be included in attributes.

One possibility is for the contextual analyzer to extract type information from declarations, and store that information in the identification table. Later that information can be retrieved whenever required.

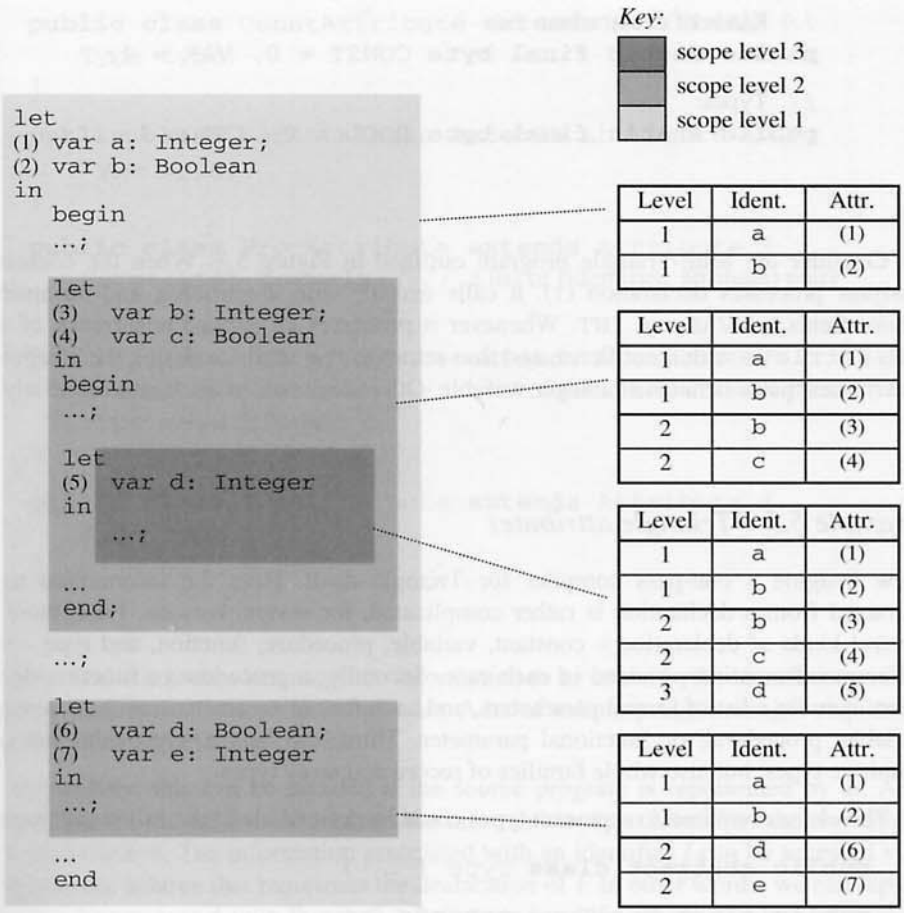


Figure 5.3 Identification table: nested block structure.

Example 5.4 Mini-Triangle attributes

Consider a Mini-Triangle contextual analyzer that extracts type information from a declaration, and uses that information to construct an attribute.

For Mini-Triangle, the relevant information is just whether the declaration is of a constant or a variable, and whether its type is *bool* or *int*. (Other information, such as the actual value of a constant, is irrelevant in contextual analysis of Mini-Triangle.)

Thus the type `Attribute` could be defined as follows:

```
public class Attribute {

    byte kind;          // either CONST or VAR
    byte type;          // either BOOL or INT
```

```
    // Kinds of declaration:
    public static final byte CONST = 0, VAR = 1;

    // Types:
    public static final byte BOOL = 0, INT = 1;

    ...
}
```

Consider the Mini-Triangle program outlined in Figure 5.3. When the contextual analyzer processes declaration (1), it calls `enter` with identifier `a` and an attribute whose fields are `VAR` and `INT`. Whenever it processes an applied occurrence of `a`, it calls `retrieve` with identifier `a`, and thus retrieves that attribute. Using the attribute, it determines that `a` denotes an integer variable. Other declarations are treated similarly.

□

Example 5.5 Triangle attributes

Now imagine a one-pass compiler for Triangle itself. Here the information to be extracted from a declaration is rather complicated, for several reasons. First, there are several kinds of declaration – constant, variable, procedure, function, and type – with different information provided in each case. Secondly, a procedure or function declaration includes a list of formal parameters, and each formal parameter may be a constant, variable, procedural, or functional parameter. Third, the language provides not only primitive types, but also whole families of record and array types.

The classes required to represent types could be defined along the following lines:

```
public abstract class Type { ... }

public class BoolType extends Type { ... }

public class CharType extends Type { ... }

public class IntType extends Type { ... }

public class RecordType extends Type {
    FieldList fields;      // a list of (identifier, type) pairs
}

public class ArrayType extends Type {
    int elementCount;
    Type elementType;
}
```

And the classes required to represent attributes could be defined as follows:

```
public abstract class Attribute { ... }
```

```
public class ConstAttribute extends Attribute {
    Type type;
}

public class VarAttribute extends Attribute {
    Type type;
}

public class ProcAttribute extends Attribute {
    FormalList formals;   // a list of (identifier, attribute) pairs
}

public class FuncAttribute extends Attribute {
    FormalList formals;   // a list of (identifier, attribute) pairs
    Type resultType;
}

public class TypeAttribute extends Attribute {
    Type type;
}
```

□

For a realistic source language, the information to be stored in the identification table is quite complex, as Example 5.5 illustrates. A lot of tedious programming is required to declare and construct the attributes.

Fortunately, this can be avoided if the source program is represented by an AST. This is because the AST itself contains the information about identifiers that we need to store and retrieve. The information associated with an identifier *I* can be accessed via a pointer to the subtree that represents the declaration of *I*. In other words, we can replace the class `Attribute` with the class `Declaration` throughout the definition of the `IdentificationTable` class (assuming the AST representation described in Section 4.4.1).

Example 5.6 Mini-Triangle attributes represented by declaration ASTs

Consider once more the Mini-Triangle program outlined in Figure 5.3. Figure 5.4 shows part of the AST representing this program, including one of the inner blocks, with the subtree representing each block shaded to indicate its scope level. Figure 5.4 also shows a picture of the identification table as it stands during contextual analysis of each block.

When the contextual analyzer visits the declaration at subtree (1), it calls `enter` with identifier `a` and a pointer to subtree (1). Whenever it visits an applied occurrence of `a`, it calls `retrieve` with identifier `a`, and thus retrieves the pointer to subtree (1). By inspecting this subtree, it determines that `a` denotes an integer variable. The other declarations are treated similarly.

□

```
public class ConstAttribute extends Attribute {
    Type type;
}

public class VarAttribute extends Attribute {
    Type type;
}

public class ProcAttribute extends Attribute {
    FormalList formals;   // a list of (identifier, attribute) pairs
}

public class FuncAttribute extends Attribute {
    FormalList formals;   // a list of (identifier, attribute) pairs
    Type resultType;
}

public class TypeAttribute extends Attribute {
    Type type;
}
```

□

For a realistic source language, the information to be stored in the identification table is quite complex, as Example 5.5 illustrates. A lot of tedious programming is required to declare and construct the attributes.

Fortunately, this can be avoided if the source program is represented by an AST. This is because the AST itself contains the information about identifiers that we need to store and retrieve. The information associated with an identifier *I* can be accessed via a pointer to the subtree that represents the declaration of *I*. In other words, we can replace the class `Attribute` with the class `Declaration` throughout the definition of the `IdentificationTable` class (assuming the AST representation described in Section 4.4.1).

Example 5.6 Mini-Triangle attributes represented by declaration ASTs

Consider once more the Mini-Triangle program outlined in Figure 5.3. Figure 5.4 shows part of the AST representing this program, including one of the inner blocks, with the subtree representing each block shaded to indicate its scope level. Figure 5.4 also shows a picture of the identification table as it stands during contextual analysis of each block.

When the contextual analyzer visits the declaration at subtree (1), it calls `enter` with identifier `a` and a pointer to subtree (1). Whenever it visits an applied occurrence of `a`, it calls `retrieve` with identifier `a`, and thus retrieves the pointer to subtree (1). By inspecting this subtree, it determines that `a` denotes an integer variable. The other declarations are treated similarly.

□

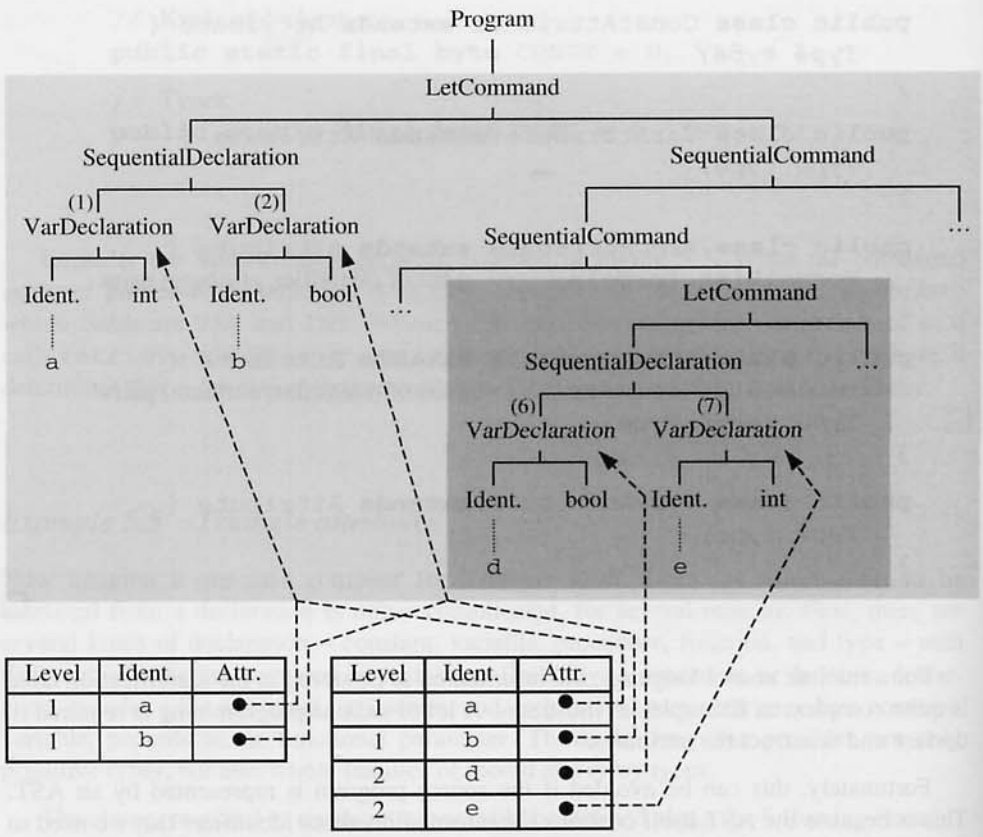


Figure 5.4 Identification table: relationship to AST.

5.1.5 Standard environment

Most programming languages contain a standard collection of predefined constants, variables, types, procedures and functions that the programmer can use without having to introduce them explicitly. For example, there is the package `java.lang` in Java, the standard prelude in Haskell, and the package `Standard` in Ada. This collection we call the ***standard environment*** of the language.

At the start of identification, therefore, the identification table is not empty, but already contains entries corresponding to the declarations of the standard environment. This presents us with a slight problem, as we must construct the corresponding attributes for these identifiers. In the example languages above, this can be achieved by processing the text of the appropriate package before starting on the source program. In other cases, however, the contextual analyzer must contain code that explicitly constructs the corresponding attribute values and enters them into the identification table.

A programming language must also specify the appropriate scope rule for the standard environment. Most programming languages consider the standard environment to be a scope enclosing the whole program, so that the source program may contain a declaration of an identifier present in the standard environment without causing a scope error. Some other programming languages (such as C) introduce the standard environment at the same scope level as the global declarations of the source program.

If the standard environment is to be at a scope enclosing the whole program, the declarations of the standard environment should be entered at scope level 0 in the identification table.

Example 5.7 Standard environment in Mini-Triangle

The standard environment of Mini-Triangle contains the following constant, type, procedure, and operator declarations:

```
type Boolean ~ ...;
const false ~ ...;
const true ~ ...;
func \ (b: Boolean) : Boolean ~ ...;
type Integer ~ ...;
const maxint ~ ...;
func + (i1: Integer, i2: Integer) : Integer ~ ...;
func - (i1: Integer, i2: Integer) : Integer ~ ...;
func * (i1: Integer, i2: Integer) : Integer ~ ...;
func / (i1: Integer, i2: Integer) : Integer ~ ...;
func < (i1: Integer, i2: Integer) : Boolean ~ ...;
func > (i1: Integer, i2: Integer) : Boolean ~ ...;
proc putint (i: Integer) ~ ...;
```

In addition, the following operator is available for every type *T* (i.e., both `Integer` and `Boolean`):

```
func = (val1: T, val2: T) : Boolean ~ ...;
```

Here, a unary operator declaration is treated like a function declaration with one argument, and a binary operator declaration is treated like a function declaration with two arguments. The operator symbol is treated like a normal identifier. The contextual analyzer only requires information about the types of the arguments and result of an operator, and so these declarations have no corresponding expressions.

- *Binary operator application*: Consider the expression 'E_1 O E_2', where O is a binary operator of type $T_1 \times T_2 \rightarrow T_3$. The type checker ensures that E_1's type is equivalent to T_1, and that E_2's type is equivalent to T_2, and thus infers that the type of 'E_1 O E_2' is T_3. Otherwise there is a type error.

In general, the type of a nontrivial expression is inferred from the types of its subexpressions, using the appropriate type rule.

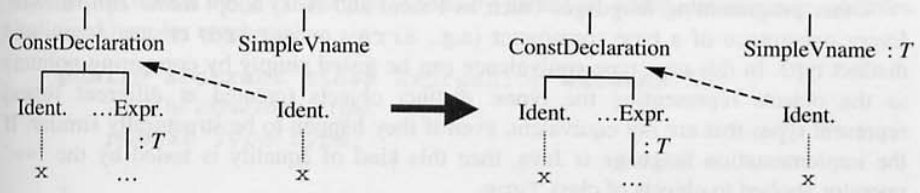
In some phrases the type checker must test whether an inferred type is equivalent to an expected type, or test whether two inferred types are equivalent to each other. In a typical language, the type of the expression in an if- or while-command must be equivalent to the type *bool*; and the type of an actual parameter must be equivalent to the type of the corresponding formal parameter. So the type checker must be able to test whether two given types T and T' are ***equivalent***.

Example 5.8 Mini-Triangle type checking

Mini-Triangle has only two types (denoted by `Boolean` and `Integer`), so they can easily be represented as follows:

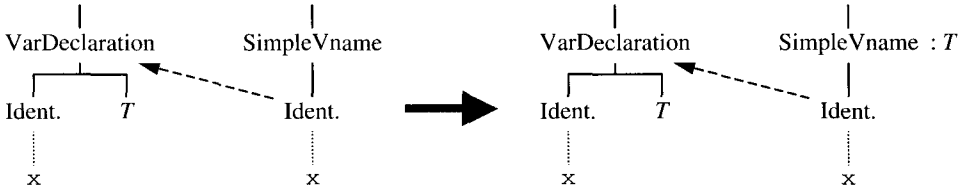
```
public class Type {
   private byte kind;   // either BOOL or INT

   public static final byte
      BOOL = 0, INT = 1;

   public static Type (byte sort) {
      this.sort = sort;
   }

   public boolean equals (Object other) {
      // Test whether this type is equivalent to other.
      Type otherType = (Type) other;
      return (this.kind == otherType.kind);
   }

}
```

It is a simple matter to infer the type of an applied occurrence of a constant or variable identifier I, provided that a link has already been established to the declaration of I. If I is declared in a constant declaration, whose right-side expression's type has been inferred to be T, then the type of the applied occurrence of I is T:

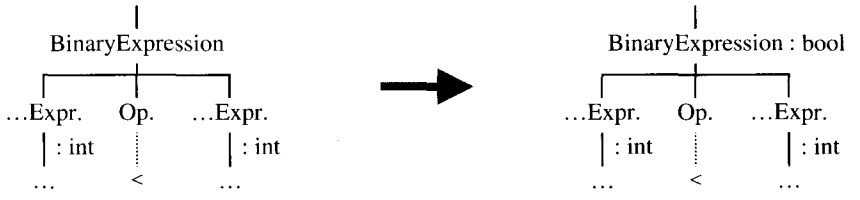


(We show the inferred type T by annotating the AST node with ': T'.)

If I is declared in a variable declaration, whose right side is type T, then the type of the applied occurrence of I is T:



An application of a binary operator such as '<' would be handled as follows:



The operator '<' is of type $int \times int \rightarrow bool$. Having checked that the type of E_1 is equivalent to *int*, and that the type of E_2 is equivalent to *int*, the type checker infers that the type of '$E_1 < E_2$' is *bool*. Other operators would be handled similarly.

□

Of course, Mini-Triangle type checking is exceptionally simple: the representation of types is trivial, and testing for type equivalence is also trivial. Type checking is more complicated if the source language has composite types. For example, Triangle array and record types have component types, which are unrestricted. Thus we need to represent types by trees.

Furthermore, there are two possible definitions of type equivalence.

Some programming languages (such as Triangle) adopt ***structural equivalence***, whereby two types are equivalent if and only if their structures are the same. If types are represented by trees, structural equivalence can be tested by comparing the structures of these trees. If the implementation language is Java, then this kind of equality is conventionally tested by an `equals` method in the `Type` class.

Other programming languages (such as Pascal and Ada) adopt ***name equivalence***. Every occurrence of a type constructor (e.g., `array` or `record`) creates a new and distinct type. In this case type equivalence can be tested simply by comparing pointers to the objects representing the types: distinct objects (created at different times) represent types that are not equivalent, even if they happen to be structurally similar. If the implementation language is Java, then this kind of equality is tested by the '`==`' operator applied to objects of class `Type`.

In Section 5.4 we shall return to look at a more realistic example of type checking, in the context of the Triangle compiler.

5.3 A contextual analysis algorithm

Contextual analysis consists of identification and type checking. Each applied occurrence of an identifier must be identified before type checking can proceed. Identification could, in principle, be completed before type checking is started, but there would be little advantage in that. Instead, identification and type checking are usually interleaved in a single pass over the source program (or its representation). If the source program is represented by an AST, contextual analysis can be done in a single depth-first traversal.

Throughout this section, we shall assume that the AST is represented as described in Section 4.4.1.

5.3.1 Decoration

The results of contextual analysis can be recorded by ***decorating*** the AST, as explained in Section 3.1.2. The following decorations prove to be useful:

- The results of *identification* can be recorded by making an explicit link from each applied occurrence of an identifier *I* to the corresponding declaration of *I*. This has the advantage of making the decorated AST a self-contained representation of the source program; the identification table may be discarded once identification is complete. In the compiler, we represent this link by a pointer field in each identifier node. In diagrams, we show this link as a dashed arrow.
- The results of *type checking* can be recorded by storing each expression *E*'s inferred type *T* at the root node of *E*. In the compiler, we represent this by a type field in each expression node. In diagrams, we show this inferred type by an annotation ': *T*' to the right of the expression node. The same point applies to the other typed phrases such as value-or-variable-name.

Example 5.9 Representation of decorated ASTs

A class `AST` suitable for representing Mini-Triangle undecorated ASTs was defined in Example 4.19. Here we extend the definition of `AST` and its subclasses to make them suitable for Mini-Triangle decorated ASTs, by means of additional instance variables (italicized here for emphasis):

```
public abstract class Expression extends AST {
    // The expression's type:
    public Type type;
    ...
}
```

```
public abstract class Vname extends AST {
    // The value-or-variable-name's type, and an indication of whether
    // it is a variable or a value:
    public Type type;
    public boolean variable;
    ...
}

public class Identifier extends Terminal {
    // A pointer to the identifier's declaration (applied occurrences only):
    public Declaration decl;
    ...
}

public class Operator extends Terminal {
    // A pointer to the operator's declaration:
    public OperatorDeclaration decl;
    ...
}
```

(The decoration of an operator node will be explained later.)

□

5.3.2 Visitor classes and objects

The work to be done by the contextual analyzer depends on the class of phrase to be checked. Checking of a command *C* will determine simply whether *C* is well-formed or not. Checking of an expression *E* will determine whether *E* is well-formed, but also infer the type of *E*. Checking of a declaration *D* will determine whether *D* is well-formed, but also make entries in the identification table for the identifiers declared in *D*.

In more detail, the checking of a command depends on the particular form of that command. For example, checking an assignment-command '*V* `:=` *E*' entails (i) checking *V* to determine its type and ensure that it is a variable, (ii) checking *E* to determine its type, and (iii) testing whether the two types are equivalent. Checking a block-command '`let` *D* `in` *C*' entails (i) opening an inner scope, (ii) checking *D*, (iii) checking *C*, and (iv) closing the inner scope. Similarly, the checking of an expression or declaration depends on the particular form of expression or declaration.

In all cases, checking a particular phrase entails checking its subphrases (if any). If the source program is represented by an AST, contextual analysis therefore entails traversing the AST, visiting the nodes in some suitable order. We shall see in Chapter 7 that code generation also entails traversing the AST, visiting the nodes in some (possibly different) order. Therefore the compiler should be designed such that AST traversals are organized systematically.

The work of the contextual analyzer will be done by a set of ***visitor methods***. The will be exactly one visitor method, `visitA`, for each concrete `AST` subclass *A*. The visitor methods will cooperate to traverse the AST in the desired order.

Example 5.10 Mini-Triangle visitor methods

The visitor methods for Mini-Triangle are summarized by the following Java interface

```
public interface Visitor {
    // Programs:
    public Object visitProgram
              (Program prog, Object arg);

    // Commands:
    public Object visitAssignCommand
              (AssignCommand com, Object arg);
    public Object visitCallCommand
              (CallCommand com, Object arg);
    public Object visitSequentialCommand
              (SequentialCommand com, Object arg);
    public Object visitIfCommand
              (IfCommand com, Object arg);
    public Object visitWhileCommand
              (WhileCommand com, Object arg);
    public Object visitLetCommand
              (LetCommand com, Object arg);

    // Expressions:
    public Object visitIntegerExpression
              (IntegerExpression expr, Object arg);
    public Object visitVnameExpression
              (VnameExpression expr, Object arg);
    public Object visitUnaryExpression
              (UnaryExpression expr, Object arg);
    public Object visitBinaryExpression
              (BinaryExpression expr, Object arg);

    // Value-or-variable-names:
    public Object visitSimpleVname
              (SimpleVname vname, Object arg);

    // Declarations:
    public Object visitConstDeclaration
              (ConstDeclaration decl, Object arg);
    public Object visitVarDeclaration
              (VarDeclaration decl, Object arg);
    public Object visitSequentialDeclaration
              (SequentialDeclaration decl, Object arg);
```

```
    // Type-denoters:
    public Object visitSimpleTypeDenoter
                  (SimpleTypeDenoter type, Object arg);

    // Identifiers:
    public Object visitIdentifier
                  (Identifier id, Object arg);

    // Operators:
    public Object visitOperator
                  (Operator op, Object arg);
}
```

□

Each visitor method has an argument that is the subtree (phrase) to be visited. It also has an `Object` argument to allow additional data to be passed into the method, where required. Finally, it has an `Object` result to allow data to be passed out of the method, where required.

For example, the result of each `visit...Expression` method in the contextual analyzer will be the expression's type, whereas the result of each `visit...Command` method will be **`null`**. (Later we shall see why it is worthwhile to provide every visitor method with both an `Object` argument and an `Object` result.)

We will organize AST traversals using the object-oriented ***visitor design pattern***. Given a set of node classes, a ***visitor class*** is one that implements the corresponding set of visitor methods. For example, a Mini-Triangle AST visitor class is one that implements the `Visitor` interface defined in Example 5.10. A ***visitor object*** (an object of a visitor class) therefore contains a particular set of visitor methods.

Any AST traversal can be implemented as a visitor object. In fact, both the contextual analyzer and the code generator can be implemented as visitor objects. We enhance the `AST` class with an instance method `visit`, which can be used to visit any AST node:

```
public abstract class AST {
    ...
    public abstract Object visit
                  (Visitor v, Object arg);
}
```

Like the visitor methods, `visit` has an `Object` argument and an `Object` result. It also has a `Visitor` argument; this tells it which particular visitor object (i.e., which particular set of visitor methods) to apply to the AST node and its descendants.

Each concrete subclass of `AST` must implement the `visit` method, simply by calling the appropriate visitor method. For example, the `AssignCommand` and `IfCommand` classes will implement `visit` as follows:

```
public class AssignCommand extends Command {
    ...
    public Object visit (Visitor v, Object arg) {
        return v.visitAssignCommand(this, arg);
    }
}

public class IfCommand extends Command {
    ...
    public Object visit (Visitor v, Object arg) {
        return v.visitIfCommand(this, arg);
    }
}
```

Each `visit` method 'knows' which particular visitor method to call. For example, `IfCommand`'s `visit` method knows that **this** is an `IfCommand` object, so it calls the `v.visitIfCommand` method, passing **this** (and `arg`) as arguments. In general, the `visit` method in the concrete `AST` subclass *A* simply calls `v.visit`*A*:

```
public class A extends ... {
    ...
    public Object visit (Visitor v, Object arg) {
        return v.visitA(this, arg);
    }
}
```

When `visit`*A* visits an AST node, it may visit any child of that node by calling that child's own `visit` method.

5.3.3 Contextual analysis as a visitor object

The contextual analyzer is a visitor object that performs identification and type checking. Each visitor method `visit`*A* in the contextual analyzer will check a node of class *A*, generating an error report if it determines that the phrase represented by the node is ill-formed. Visitor methods in the contextual analyzer can conveniently be called ***checking methods***.

Example 5.11 Mini-Triangle contextual analysis

The Mini-Triangle contextual analyzer is an implementation of the `Visitor` interface given in Example 5.10.

We shall assume the following representation of Mini-Triangle types, adapted from Example 5.8:

```
public class Type {
    private byte kind;    // either BOOL, INT or ERROR
```

```
    public static final byte
       BOOL = 0, INT = 1, ERROR = -1;

    public static Type (byte kind) {
       this.kind = kind;
    }

    public boolean equals (Object other) {
       Type otherType = (Type) other;
       return (this.kind == otherType.kind
                  || this.kind == ERROR
                  || otherType.kind == ERROR);
    }

    public static Type bool = new Type(BOOL);
    public static Type int = new Type(INT);
    public static Type error = new Type(ERROR);
}
```

The Mini-Triangle visitor/checking methods are summarized in Table 5.1. Now we outline how they are implemented.

Table 5.1 Summary of visitor/checking methods for the Mini-Triangle contextual analyzer.

Phrase class	Visitor/checking method(s)	Behavior of visitor/checking method(s)
Program	`visitProgram`	Check that the program is well-formed, and return **`null`**.
Command	`visit...Command`	Check that the command is well-formed and return **`null`**.
Expression	`visit...Expression`	Check that the expression is well-formed, decorate it with its type, and return that type.
V-name	`visitSimpleVname`	Check that the value-or-variable-name is well-formed, decorate it with its type and a flag indicating if it is a variable, and return that type.
Declaration	`visit...Declaration`	Check that the declaration is well-formed, enter all declared identifiers into the identification table, and return **`null`**.
Type-Denoter	`visit...TypeDenoter`	Check that the type-denoter is well-formed, decorate it with its type, and return that type.
Identifier	`visitIdentifier`	Check that the identifier is declared, link the applied occurrence of the identifier to its declaration, and return that declaration.
Operator	`visitOperator`	Check that the operator is declared, link the applied occurrence of the operator to its declaration, and return that declaration.

Each of the `visit...Command` visitor/checking methods checks that the given command is well-formed. (The method's result is **`null`**, and the `arg` object is ignored.) The following are typical:

```
public Object visitAssignCommand
                (AssignCommand com, Object arg) {
   Type vType = (Type) com.V.visit(this, null);
   Type eType = (Type) com.E.visit(this, null);
   if (! com.V.variable)
      ... // Report an error – the left side is not a variable.
   if (! eType.equals(vType))
      ... // Report an error – the left and right sides are not of
          // equivalent type.
   return null;
}

public Object visitSequentialCommand
                (SequentialCommand com, Object arg) {
   com.C1.visit(this, null);
   com.C2.visit(this, null);
   return null;
}

public Object visitIfCommand
                (IfCommand com, Object arg) {
   Type eType = (Type) com.E.visit(this, null);
   if (! eType.equals(Type.bool))
      ... // Report an error – the expression is not boolean.
   com.C1.visit(this, null);
   com.C2.visit(this, null);
   return null;
}

public Object visitLetCommand
                (LetCommand com, Object arg) {
   idTable.openScope();
   com.D.visit(this, null);
   com.C.visit(this, null);
   idTable.closeScope();
   return null;
}
```

These visitor/checking methods are fairly self-explanatory. In the case of the assignment command 'V `:=` E', `visitAssignCommand` calls V`.visit` to check V and E`.visit` to check E, then ensures that V is indeed a variable and that V's type is equivalent to E's type. In the case of the if-command '`if` E `then` C_1 `else` C_2', `visitIfCommand` calls E`.visit`, C_1`.visit`, and C_2`.visit`, and ensures that E's type is *bool*.

Here `idTable` is the identification table used during contextual analysis. It is an object of class `IdentificationTable`, as given in Section 5.1.3.

Each of the `visit...Expression` visitor/checking methods checks that the expression is well-formed, and decorates the expression node with its inferred type. The method's result is that type. (The `arg` object is ignored.)

```
public Object visitIntegerExpression
                (IntegerExpression expr, Object arg) {
   expr.type = Type.int;                    // decoration
   return expr.type;
}

public Object visitVnameExpression
                (VnameExpression expr, Object arg) {
   Type vType = (TypeDenoter) expr.V.visit(this, null);
   expr.type = vType;                       // decoration
   return expr.type;
}

public Object visitBinaryExpression
                (BinaryExpression expr, Object arg) {

   Type e1Type = (Type) expr.E1.visit(this, null);
   Type e2Type = (Type) expr.E2.visit(this, null);
   OperatorDeclaration opdecl =
      (OperatorDeclaration) expr.O.visit(this, null);
   if (opdecl == null) {
      ... // Report an error – no such operator.
      expr.type = Type.error;               // decoration
   } else if (opdecl instanceof
            BinaryOperatorDeclaration) {
      BinaryOperatorDeclaration bopdecl =
            (BinaryOperatorDeclaration) opdecl;
      if (! e1Type.equals(bopdecl.operand1Type))
         ... // Report an error – the left subexpression has the wrong type.
      if (! e2Type.equals(bopdecl.operand2Type))
         ... // Report an error – the right subexpression has the
             // wrong type.
      expr.type = bopdecl.resultType; // decoration
   } else {
      ... // Report an error – the operator is not a binary operator.
      expr.type = Type.error;               // decoration
   }
   return expr.type;
}
```

The visitor/checking methods `visitIntegerExpression` and `visitVnameExpression` are self-explanatory. In the case of a binary operator application 'E_1 O E_2', `visitBinaryExpression` assumes that `O.visit` returns a pointer to a 'declaration' of operator O, where the operator's operand and result types may be found. (This declaration will be the attribute value returned by searching the identification table for the operator.) Method `visitUnaryExpression` is similar to `visitBinaryExpression`.

The `visitSimpleVname` visitor/checking method checks that the value-or-variable-name is well-formed, and decorates it with its inferred type together with an indication of whether it is a variable or not. The method's result is the inferred type. (The `arg` object is ignored.) The following are typical:

```
public Object visitSimpleVname
                (SimpleVname vname, Object arg) {
   Declaration decl =
         (Declaration) vname.I.visit (this, null);
   if (decl == null) {
      ... // Report an error – this identifier is not declared.
      vname.type = Type.error;
      vname.variable = true;            // decoration
   } else if (decl instanceof ConstDeclaration) {
      vname.type = ((ConstDeclaration) decl).E.type;
      vname.variable = false;           // decoration
   } else if (decl instanceof VarDeclaration) {
      vname.type = ((VarDeclaration) decl).T.type;
      vname.variable = true;            // decoration
   }
   return vname.type;
}
```

Each of the `visit...Declaration` visitor/checking methods checks that the declaration is well-formed, and enters all declared identifiers into the identification table. (The method's result is **null**, and the `arg` object is ignored.)

```
public Object visitConstDeclaration
                (ConstDeclaration decl, Object arg) {
   decl.E.visit(this, null);            // result is ignored
   idTable.enter(decl.I.spelling, decl);
   return null;
}

public Object visitVarDeclaration
                (VarDeclaration decl, Object arg) {
   decl.T.visit(this, null);            // result is ignored
   idTable.enter(decl.I.spelling, decl);
   return null;
```

```
public Object visitSequentialDeclaration
             (SequentialDeclaration decl, Object arg) {
    decl.D1.visit(this, null);
    decl.D2.visit(this, null);
    return null;
}
```

Each of the `visitConstDeclaration` and `visitVarDeclaration` visitor/checking methods makes an entry in the identification table for the declared identifier. This entry consists of the identifier's spelling and a pointer to the declaration itself. Also, in both of these visitor/checking methods, a subtree is visited only for the side-effect of decorating that subtree with type information; the type information returned by the visitor/checking method is ignored.

The visitor/checking method `visitSimpleTypeDenoter` (not shown) checks whether the given type-denoter is the identifier '`Boolean`' or '`Integer`', decorates the node with the corresponding type (`Type` object), and returns that type.

The `visitIdentifier` visitor/checking method links an applied occurrence of an identifier to the corresponding declaration (if any). Its result is a pointer to that declaration. (The `arg` object is ignored.).

```
public Object visitIdentifier
              (Identifier id, Object arg) {
    id.decl = idTable.retrieve          // decoration,
                 (id.spelling);         // possibly null
    return id.decl;
}
```

Finally, the contextual analyzer must define a method that checks an entire program. The completed contextual analyzer becomes:

```
public final class Checker implements Visitor {

    private IdentificationTable idTable;

    ... // Visitor/checking methods, as above.

    public void check (Program prog) {
        idTable = new IdentificationTable();
        idTable.enter("false", ...);
        idTable.enter("true", ...);
        ...
        idTable.enter("putint", ...);
        prog.visit(this, null);
    }
}
```

Method `check` illustrates how the source language's standard environment can be handled. It initializes the identification table with entries for all standard constants,

types, procedures, and so on. These entries will then be retrieved in the usual way at applied occurrences of the corresponding identifiers.

□

Making the contextual analyzer a visitor object has important advantages:

- It brings all of the contextual analyzer code together in a single class, `Checker`. This makes the contextual analyzer easier to study and modify. (The principal alternative, whereby each `AST` subclass contains its own visitor/checking method, would result in the contextual analyzer code being spread over a large number of class definitions, and thus harder to understand and harder to modify.)
- It ensures that the AST traversal is complete, i.e., that the contextual analyzer includes the code to visit every class of node in the AST. (The alternative, whereby the contextual analyzer traverses the AST in an *ad hoc* manner, would risk accidental omission of code for some classes of node.)
- The same structure can later be used for the code generator, and indeed for any other process that needs to traverse the AST.

5.4 Case study: contextual analysis in the Triangle compiler

The Triangle contextual analyzer consists of a package `Triangle.ContextualAnalyzer` that contains the `Checker` and `IdentificationTable` classes. The `Checker` class depends on the package `Triangle.AbstractSyntaxTrees`, which contains all of the class definitions for ASTs.

Triangle has static bindings and is statically-typed. The Triangle contextual analyzer works in much the same way as described in Section 5.3, interleaving identification and type checking in a single traversal of the AST. It is structured as a visitor object.

5.4.1 Identification

Triangle exhibits nested block structure, so identification is performed with the aid of a multilevel identification table as described in Section 5.1.3. The attribute stored in each table entry is a pointer to a declaration in the AST. At each applied occurrence of an identifier *I*, the table is used to find the corresponding declaration of *I*, and the applied occurrence is linked to this declaration. Once contextual analysis of the source program is completed, the identification table is no longer required and is discarded.

The current implementation of the identification table is naive. It is a stack in which each entry contains a scope level, an identifier, and a declaration. Method `enter` pushes a new entry on to the stack. Method `retrieve` searches the stack from the top

down, i.e., it uses linear search. Method `openScope` simply increments the current scope level. Method `closeScope` pops entries belonging to the current scope level, which it then decrements.

Two rather better implementations of the identification table are suggested by Exercises 5.4 and 5.5.

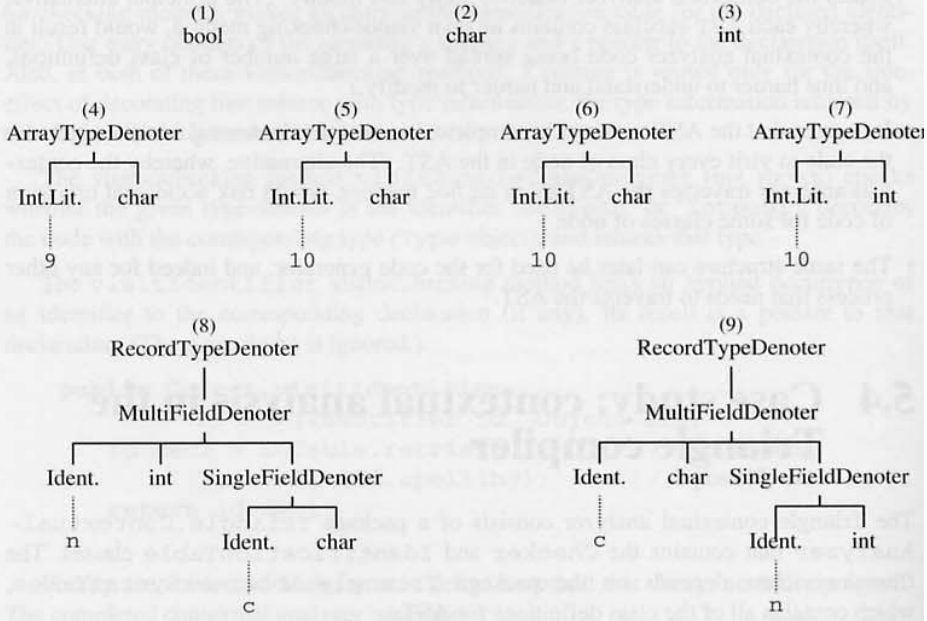


Figure 5.5 Representation of Triangle types by small ASTs.

5.4.2 Type checking

Triangle has not only primitive types (denoted by `Boolean`, `Char`, and `Integer`) but also array types and record types. An array type is characterized by the number and type of the elements. A record type is characterized by the identifiers, types, and order of its fields. These types are conveniently represented by small ASTs. So there is no need for the class `Type` used in Section 5.3; it is replaced by the class `TypeDenoter`. The ASTs used to represent the following Triangle types are shown in Figure 5.5:

(1) `Boolean`
(2) `Char`
(3) `Integer`
(4) `array 9 of Char`

```
(5) array 10 of Char
(6) array 10 of Char
(7) array 10 of Integer
(8) record n: Integer, c: Char end
(9) record c: Char, n: Integer end
```

In Triangle type equivalence is structural. Of the types shown in Figure 5.5, only (5) and (6) are equivalent to each other. To test whether two types are equivalent, the type checker just compares their ASTs structurally. This test is performed by defining an `equals` method in each subclass of `TypeDenoter`. Class `TypeDenoter` itself is enhanced as follows:

```
public abstract class TypeDenoter extends AST {
   public abstract boolean equals (Object other);
   ...
}
```

Type *identifiers* in the AST would complicate the type equivalence test. To remove this complication, the visitor/checking methods for type-denoters are made to eliminate all type identifiers. This is achieved by replacing each type identifier by the type it denotes.

Figure 5.6 shows the ASTs representing the following Triangle declarations:

```
type Word ~ array 8 of Char;
var w1: Word;
var w2: array 8 of Char
```

Initially the type subtrees (1) and (2) in the two variable declarations are different. After these subtrees have been checked, however, the type identifiers 'Char' and 'Word' have been eliminated. The resulting subtrees (3) and (4) are structurally similar. The elimination of type identifiers makes it clear that the types of variables `w1` and `w2` are equivalent.

A consequence of this transformation is to make each type 'subtree' (and hence the whole AST) into a *directed acyclic graph*. Fortunately, this causes no serious complication in the Triangle compiler. (But recursive types – as found in Pascal, Ada, and ML – would cause a complication: see Exercise 5.9.)

The Triangle type checker infers and checks the types of expressions and value-or-variable-names in much the same way as in Example 5.8. Types are tested for structural equivalence by using the `equals` method of the `TypeDenoter` class. (Instead, comparing types by means of '==' would implement name equivalence.)

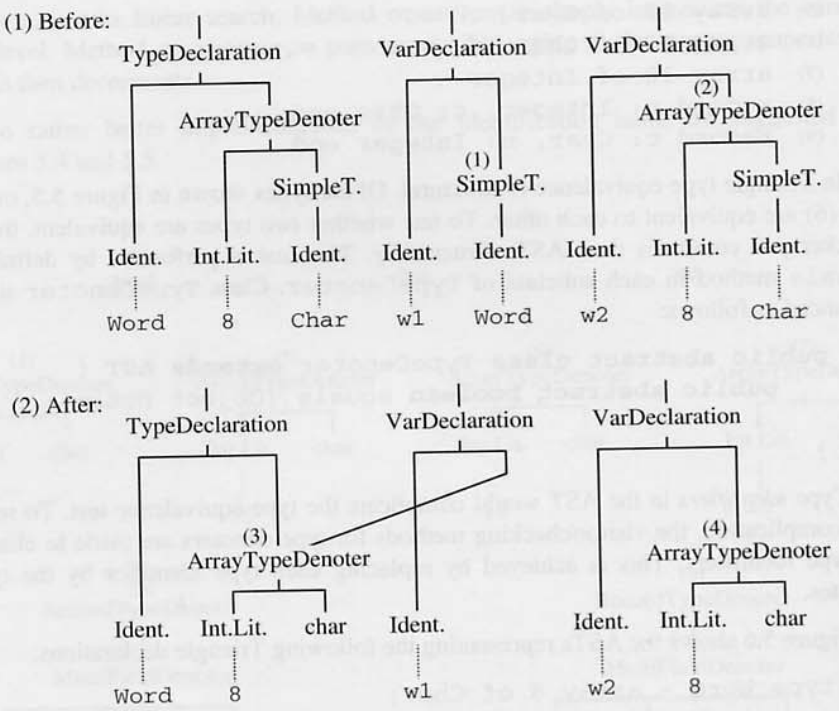


Figure 5.6 Triangle ASTs before and after elimination of type identifiers.

5.4.3 Standard environment

Like all programming languages, Triangle has a standard environment (described in Section B.9). This is represented by a collection of small ASTs, representing the 'declarations' of the standard identifiers. Some of these 'declarations' are:

```
(1) type Boolean ~ ...;
(2) const false ~ ...;
(3) const true ~ ...;
(4) func eof () : Boolean  ...;
(5) proc get (var c: Char ...;
(6) proc put (c: Char) ~ ...;
(7) func \   (b: Boolean) : Boolean ~ ...;
(8) func <   (i1: Integer, i2: Integer) : Boolean ~ ...;
```

The ASTs corresponding to these 'declarations' are shown in Figure 5.7. There are 'type declarations' for standard types, such as `Boolean` (1); 'constant declarations' for standard constants, such as `false` (2) and `true` (3); 'function declarations' for standard functions, such as `eof` (4); and 'procedure declarations' for standard procedures, such as `put` (5) and `get` (6).

Before analyzing a source program, the contextual analyzer initializes the identification table with entries for the standard identifiers, at scope level 0, as shown in Figure 5.8. The attribute stored in each of these entries is a pointer to the appropriate 'declaration'. Thus standard identifiers are treated in exactly the same way as identifiers declared in the source program.

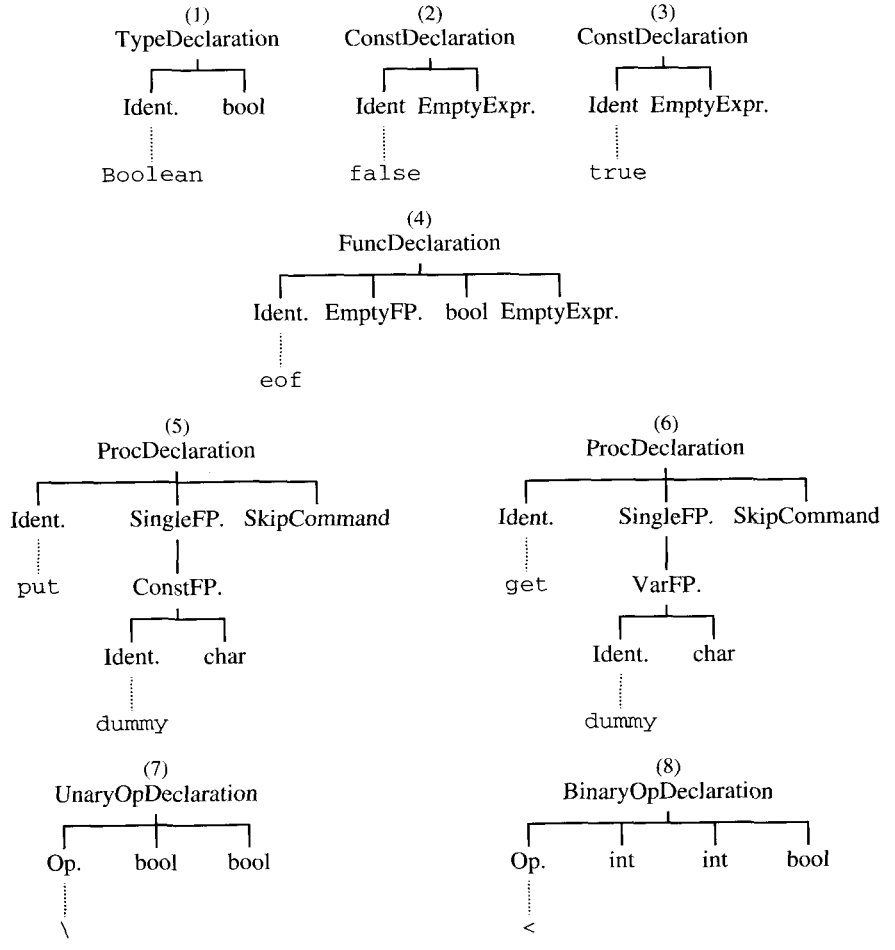


Figure 5.7 Small ASTs representing the Triangle standard environment (abridged).

Level	Ident.	Attr.
0	Boolean	(1)
0	false	(2)
0	true	(3)
0	eof	(4)
0	put	(5)
0	get	(6)
0	\	(7)
0	<	(8)
...	...	...

Figure 5.8 Identification table for the Triangle standard environment (abridged).

The Triangle standard environment also includes a collection of unary and binary operators. It is convenient to treat operators in much the same way as identifiers, as shown in Figures 5.7 and 5.8.[1]

The representation of the Triangle standard environment therefore includes small ASTs representing 'operator declarations', such as one for the unary operator '\' (7), and one for the binary operator '<' (8). (See Figure 5.7.) An 'operator declaration' merely defines the types of the operator's operand(s) and result. Entries are also made for operators in the identification table. (See Figure 5.8.) At an application of operator *O*, the identification table is used to retrieve the 'operator declaration' of *O*, and thus to find the operand and result types for type checking.

5.5 Further reading

For a more detailed discussion of declarations, scope, and block structure, see Chapter 4 of the companion textbook by Watt (1990). Section 2.5 of the same textbook discusses simple type systems (of the kind found in Triangle, Pascal, and indeed most programming languages). Chapter 7 goes on to explore more advanced type systems. *Coercions* (found in most languages) are implicit conversions from one type to another. *Overloading* (found in Ada and Java) allows several functions/procedures/methods with different bodies and different types to have a common identifier, even in the same scope. In a function/procedure/method call with this common identifier, a technique called *overload resolution* is needed to identify which of several functions/procedures/methods is being called. *Parametric polymorphism* (found in ML) allows a single function to operate

[1] Indeed, some programming languages, such as ML and Ada, actually allow operators to be declared like functions in the source program. This emphasizes the analogy between operators and function identifiers.

uniformly on arguments of a family of types (e.g., the list types). Moreover, the types of functions, parameters, etc., need not be declared explicitly. *Polymorphic type inference* is a technique that allows the types in a source program to be inferred in the context of a polymorphic type system.

For a comprehensive account of type checking, see Chapter 6 of Aho *et al.* (1985). As well as elementary techniques, the authors discuss techniques required by the more advanced type systems: type checking of coercions, overload resolution, and polymorphic type inference. For some reason, however, Aho *et al.* defer discussion of identification to Chapter 7 (run-time organization).

A classic paper on polymorphic type inference by Milner (1978) was the genesis of the type system that was adopted by ML, and borrowed by later functional languages.

For a good short account of contextual analysis in a one-pass compiler for a Pascal subset, see Chapter 2 of Welsh and McKeag (1980). The authors clearly explain ways of representing the identification table, attributes, and types. They also present a simple error recovery technique that enables the contextual analyzer to generate sensible error reports when an identifier is declared twice in the same scope, or not declared at all.

The visitor pattern used to structure the Triangle compiler is not the only possible object-oriented design. One alternative design, explained in Appel (1997), is to associate the checking methods (and the encoding methods in the code generator) for a particular AST object with the AST object itself. This design is initially easier to understand than the visitor design pattern, but it has the disadvantage that the checking methods (and encoding methods) are spread all over the `AST` subclass definitions instead of being grouped together in one place.

You should be aware of a lack of standard terminology in the area of contextual analysis. Identification tables are often called 'symbol tables' or 'declaration tables'. Contextual analysis itself is often misnamed 'semantic analysis'.

Exercises

Section 5.1

5.1 Consider a source language with monolithic block structure, as in Section 5.1.1, and consider the following ways of implementing the identification table:

(a) an ordered list;
(b) a binary search tree;
(c) a hash table.

In each case implement the `IdentificationTable` class, including the methods `enter` and `retrieve`.

In efficiency terms, how do these implementations compare with one another?

5.2 Consider a source language with flat block structure, as in Section 5.1.2. Devise an efficient way of implementing the identification table. Implement the `IdentificationTable` class, including the methods `enter`, `retrieve`, `openScope`, and `closeScope`.

5.3* For a source language with nested block structure, as in Section 5.1.3, we could implement the identification table by a stack of binary search trees (BSTs). Each BST would contain entries for declarations at one scope level. Consider the innermost block of Figure 5.3, for example. At the stack top there would be a BST containing the level-3 entries; below that there would be a BST containing the level-2 entries; and at the stack bottom there would be a BST containing the level-1 entries.

Implement the `IdentificationTable` class, including the methods `enter`, `retrieve`, `openScope`, and `closeScope`.

In efficiency terms, how does this implementation compare with that used in the Triangle compiler (Section 5.4.1)?

5.4* For a source language with nested block structure, we can alternatively implement the identification table by a sparse matrix, with columns indexed by scope levels and rows indexed by identifiers. Each column links the entries at a particular scope level. Each row links the entries for a particular identifier, in order from innermost scope to outermost scope. In the innermost block of Figure 5.3, for example, the table would look like Figure 5.9.

Implement the `IdentificationTable` class, including the methods `enter`, `retrieve`, `openScope`, and `closeScope`.

In efficiency terms, how does this implementation compare with that used in the Triangle compiler (Section 5.4.1), and with a stack of binary search trees (Exercise 5.3)?

5.5* Outline an identification algorithm that does not use an identification table, but instead searches the AST. For simplicity, assume monolithic block structure.

In efficiency terms, how does this algorithm compare with one based on an identification table?

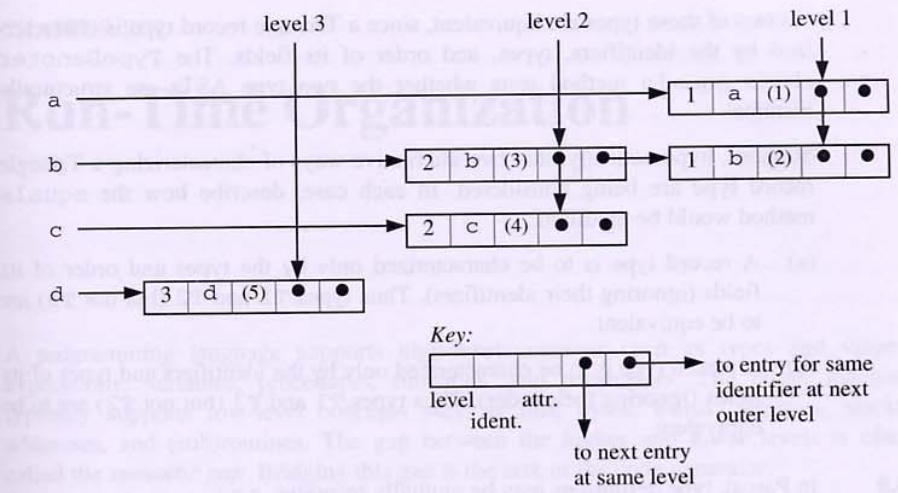


Figure 5.9 Representation of the identification table by a sparse matrix.

Section 5.2

5.6 Draw (undecorated) ASTs representing the following type declarations.

(a) Triangle type declarations:

```
type Age      = Integer;
type Letter   = Char;
type Alphanum = Char;
type Name     = array 20 of Letter;
type Address  = array 20 of Char;
type City     = array 10 of Letter;
type ZipCode  = array 10 of Alphanum
```

Eliminate the type identifiers (as in Figure 5.6). Which of the types are structurally equivalent to one another?

(b) Repeat with the corresponding type definitions in Pascal. (*Note:* Pascal adopts name equivalence of types, rather than structural equivalence.)

5.7 Suppose that name equivalence were to be adopted in Triangle. How would the Triangle contextual analyzer be modified?

5.8* Consider the following Triangle record types:

```
type T1 = record i: Integer; c: Char end;
type T2 = record j: Integer; h: Char end;
type T3 = record c: Char; i: Integer end
```

No two of these types are equivalent, since a Triangle record type is characterized by the identifiers, types, and order of its fields. The `TypeDenoter` class's `equals` method tests whether the two type ASTs are structurally identical.

Suppose, hypothetically, that two alternative ways of characterizing a Triangle record type are being considered. In each case, describe how the `equals` method would be modified.

(a) A record type is to be characterized only by the types and order of its fields (ignoring their identifiers). Thus types `T1` and `T2` (but not `T3`) are to be equivalent.

(b) A record type is to be characterized only by the identifiers and types of its fields (ignoring their order). Thus types `T1` and `T3` (but not `T2`) are to be equivalent.

5.9 In Pascal, type definitions may be mutually recursive, e.g.:

```
type IntList  = ^ IntNode;
     IntNode  = record
                  head: Integer;
                  tail: IntList
                end
```

These definitions introduce two new types: `IntList` is a pointer to an `IntNode` record, and `IntNode` is a record that contains an integer and an `IntList` pointer.

Draw (undecorated) ASTs representing these type definitions. Then eliminate the type identifiers (as in Figure 5.6). What do you observe about the transformed ASTs? Why would this complicate type checking if Pascal adopted structural equivalence for types?

5.10 Suppose that Mini-Triangle were to be extended with single-parameter function declarations:

```
single-Declaration ::= ...
                     | func Identifier ( Identifier : Type-denoter )
                       : Type-denoter ~
                       Expression
```

and function calls:

```
primary-Expression ::= ...
                     | Identifier ( Expression )
```

Describe how function calls would be type-checked.