CHAPTER FIVE
5. SYMBOL TABLE & TYPE CHECKING

5.1. Introduction to Symbol Table

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program.

A compiler uses a symbol table to keep track of scope and binding information about names. The symbol table is searched every time a name is encountered in the source text. Changes to the table occur if a new name or new information about an existing name is discovered. A symbol-table mechanism must allow us to add new entries and find existing entries efficiently.

A symbol table may serve the following purposes depending upon the language in hand:

- ✓ To store the names of all entities in a structured form at one place.
- ✓ To verify if a variable has been declared.
- ✓ To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- ✓ To determine the scope of a name (scope resolution).

❖ Symbol-Table Entries

Information is entered into the symbol table at various times. Keywords are entered into the table initially, if at all. The lexical analyzer looks up sequences of letters and digits in the symbol table to determine if a reserved keyword or a name has been collected. With this approach, keywords must be in the symbol table before lexical analysis begins. Alternatively, if the lexical analyzer intercepts reserved keywords, then they need not appear in the symbol table. If the language does not reserve keywords, then it is essential that keywords be entered into the symbol table with a warning of their possible use as a keyword.

The symbol-table entry itself can be set up when the role of a name becomes clear. With the attribute values being filled in as the information becomes available. In some cases, the entry can be initiated from the lexical analyzer as soon as a name is seen in the input. More often, one name may denote several different objects, perhaps even in the same block or procedure. For example, the C declarations

```
int x;
struct x {float y,z;};
```

use x as both an integer and as the tag of a structure with two fields, in such cases, the lexical analyzer can only return to the parser the name itself (or a pointer to the lexeme forming that name), rather than a pointer to the symbol-table entry. The record in the symbol table is created when the syntactic role played by this name is discovered. For the above declarations, two symbol-table entries for x would be created; one with x as an integer and one as a structure.

Attributes of a name are entered in response to declarations, which may be implicit. Labels are often identifiers followed by a colon, so one action associated with recognizing such an identifier may be to enter this fact into the symbol table. Similarly, the syntax of procedure declarations specifies that certain identifiers are formal parameters.

❖ Operations

A symbol table should provide the following operations.

- ***insert ( )***

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The *insert*( ) function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

int a;

should be processed by the compiler as: *insert(a, int);*

- ***lookup ( )***

Lookup ( ) operation is used to search a name in the symbol table to determine:

- ✓ if the symbol exists in the table
- ✓ if it is declared before it is being used
- ✓ if the name is used in the scope
- ✓ if the symbol is initialized
- ✓ if the symbol declared multiple times

The format of *lookup*( ) function varies according to the programming language. The basic format should match the following:

lookup(symbol)

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

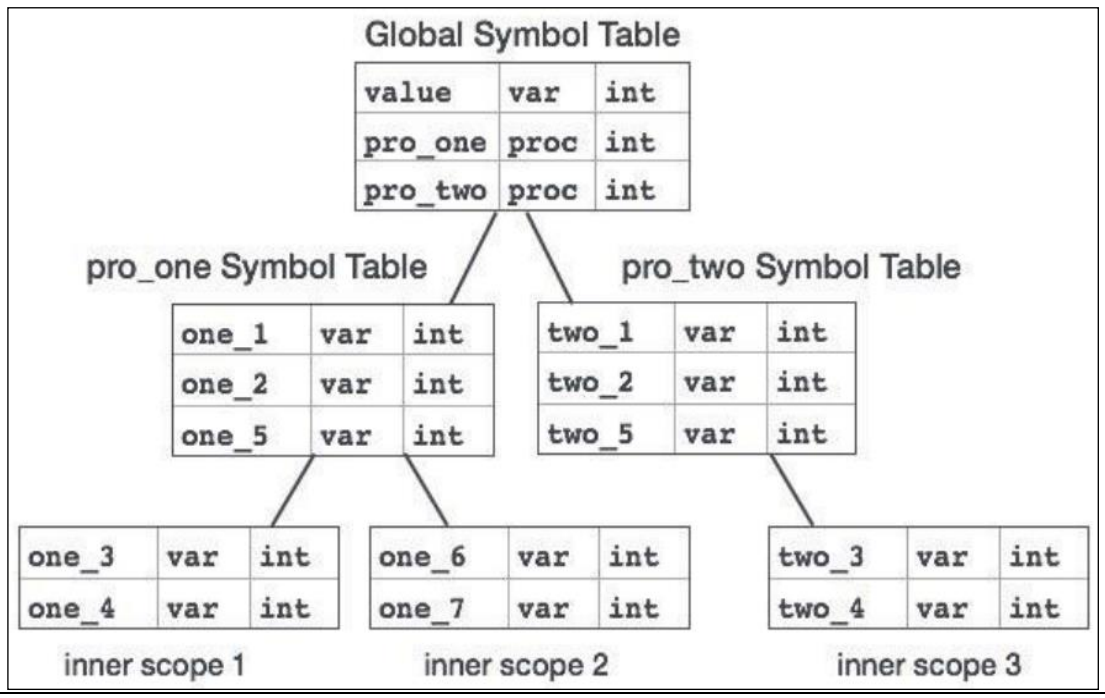
❖ Scope Management

A compiler maintains two types of symbol tables: a global symbol table which can be accessed by all the procedures and scope symbol tables that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . .
int value=10;
int pro_one( )
  {
    int one_1;
    int one_2;
      {
      int one_3;  } inner scope 1
      int one_4;  }
      }
    int one_5;
      {
      int one_6;  } inner scope 2
      int one_7;  }
      }
  }
int pro_two( )
  {
   int two_1;
   int two_2;
     {
     int two_3;  } inner scope 3
     int two_4;  }
     }
   int two_5;
  }
. . .
```

The above program can be represented in a hierarchical structure of symbol tables:



The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the *pro_one* symbol table (and all its child tables) are not available for *pro_two* symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

1. first a symbol will be searched in the current scope, i.e., current symbol table,
2. if a name is found, then search is completed, else it will be searched in the parent symbol table until,
3. either the name is found or the global symbol table has been searched for the name

5.2. Introduction to Type Checking

Type checking is the methodology by which compiler checks whether the source program follows both the syntactic and semantic conventions of the source language. This is called static checking (to distinguish it from dynamic checking executed during execution of the program).

Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic. A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors, or if its type checker does not report any type errors.

Static checking ensures that certain kind of errors are detected and reported. The following are examples of static checks:

1. *Type Checks*. A compiler should report an error if an operator is applied to an incompatible operand; for example, if an array variable and a function variable are added together.
2. *Flow-of-Control Checks*. Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. For example, a break statement in C causes control to leave the smallest enclosing while, for, or switch statement; an error occurs if such an enclosing statement does not exist.
3. *Uniqueness Checks*. There are situations in which an object must be defined exactly once. For example, in Pascal, an identifier must be declared uniquely, labels in a case statement must be distinct, and elements in a scalar type may not be repeated.
4. *Name -Related Checks*. Sometimes, the same name must appear two or more times. For example, in Ada, a loop or block may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

A type checker verifies that the type construct matches that expected by its context. For example, the built-in arithmetic operator *mod* requires integer operands, so a type checker must verify that the operands of *mod* have type integer. A type checker should verify that the type value assigned to a variable is compatible with the type of the variable. Similarly, the type checker must verify that dereferencing is applied only to a pointer, that indexing is done only on an array, that a user-defined function is applied to the correct number and type of arguments, and so forth.

Type information produced by the type checker may be needed when the code is generated. For example, arithmetic operators like + usually apply to either integers or reals, perhaps to other types, and we have to look at the context of + to determine the sense that is intended. A symbol that can represent different operations in different contexts is said to be "overloaded". Overloading may be accompanied by coercion of types, where a compiler supplies an operator to convert an operand into the type expected by the context.

A distinct notation from overloading is that of "polymorphism." The body of a polymorphic function can be executed with arguments of several types.

The following diagram shows the role of type checking in the compilation process of the source program.

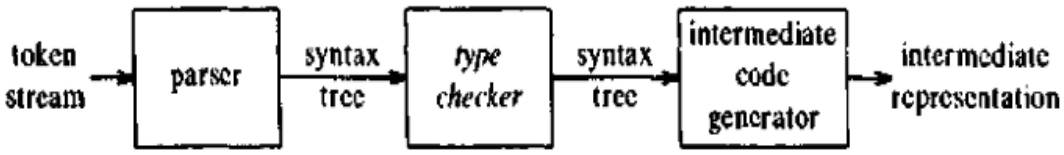


Fig. 5.1. Position of type checker.

5.3. Type Systems

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs. The following excerpts from the Pascal report and the C reference manual, respectively, are examples of information that a compiler writer might have to start with.

- "If both operands of the arithmetic operators of addition, subtraction and multiplication are of type integer, then the result is of type integer".
- "The result of the unary & operator is a pointer to the object referred to by the operand. If the type of the operand is '…', the type of the result is 'pointer to ...' ".

Implicit in the above excerpts is the idea that each expression has a type associated with it. Furthermore, types have structure; the type "pointer to ..." is constructed from the type that ". . ." refers to.

In both Pascal and C, types are either basic or constructed. Basic types are the atomic types with no internal structure as far as the programmer is concerned. In Pascal, basic types are boolean, character, integer and real. Enumerated types like (violet, indigo, blue, green, yellow, orange, red) and subrange types like 1 … 10 can also be considered as basic types. Pascal allows a programmer to construct types from basic types and other constructed types, with arrays, records, and sets being examples. In addition, pointers and functions can also be treated as constructed types.

❖ Type Expressions

The type of a language construct will be denoted by a type expression. It specifies the type of a syntactic construct of any language. A type expression is either a basic type or formed by applying an operator called *type constructor* to other type expressions. The type constructor operator derives constructed type from the basic type. The sets of basic types and constructors depend on the language to be checked.

The following are type expressions:

1. A basic type is a type expression. Among the basic types are boolean, char, integer, and real.
2. Another basic type, 'type-error' is also a type expression which will signal an error during type checking.
3. A basic type "void" is also type expression. It denotes null value (absence of value).
4. Since type expressions may be named, a type name is a type expression. An example of the use of type names appears in 5(c) below;
5. A type constructor applied to a type expression is a type expression. Constructors include:
   a. *Arrays*. If *I* in an index set and *T* is a type expression, then array (*I, T*) is a type expression. If *T* is a type expression, then *array*(*I, T*) is a type expression denoting the type of an array with elements of type *T* and index set *I*. *I* is often a range of integers. For example, the Pascal declaration

      var A: array [1. .10] of integer;

      associates the type expression array (1 . .10, integer) with A.

   b. *Products*. If T1 and T2 are type expressions, the Cartesian product T1 × T2 is a type expression.

   c. *Records*. The difference between products and records is that record fields have names. The *record* type constructor will applied to a tuple formed from field names and field types. The field names should be part of the type constructor, but it is convenient to keep field names together with their associated types.

      For example, in Pascal:

```
type row = record
    address: integer;
    lexeme: array [1 .. 15] of char;
end;
var table : array [1 .. 101] of row;
```

      declares the type name row representing the type expression.

      record ((address × integer) × (lexeme × array (1..15, Char)))

      and the variable table to be an array of records of this type.

   d. *Pointers*. If T is a type expression then *pointer* (*T*) is a type expression. If *T* is a type expression, then *pointer* (*T*) is a type expression denoting the type "pointer to an object of type *T* ". For example, in Pascal, the declaration

      var p: ^ integer

      declares variable p to have type pointer (integer).

   e. *Functions*. Mathematically, a function maps elements of one set, the *domain*, to another set, the *range*. We may treat functions in programming languages as mapping a domain type *D* to a range type *R*.

The type of such a function will be denoted by the type expression $D \rightarrow R$. For example, the built-in function mod of Pascal has domain type *int x int*, i.e., a pair of integers, and range type *int*. Thus, we say mod has the type *int x int* $\rightarrow$ *int*.

The Pascal declaration

function f (a, b : char): ^integer;

says that the domain type of *f* is denoted by *char x char* and range type by pointer (integer).
The type of *f* is thus denoted by the type expression

$char \times char \rightarrow pointer\ (integer)$

6. Type expression may contain variables whose values are type expressions.

❖ Type Systems

A type system is a collection of rules implemented by the type checker for assigning type expressions to the various parts of a program. A type checker implements a type system.

Different type systems may be used by different compilers or processors of the same language. For example, in Pascal, the type of an array includes the index set of the array, so a function with an array argument can only be applied to arrays with that index set. Many Pascal compilers, however, allow the index set to be left unspecified when an array is passed as an argument. Thus these compilers use a different type system than that in the Pascal language definition. In UNIX system, for instance, the **lint**, for instance, examines C programs for possible bugs using a more detailed type system that the C compiler itself uses.

❖ Error Recovery

Since type checking has the potential for catching errors in programs, it is important for a type checker to do something reasonable when an error is discovered. At the very least, the compiler must report the nature and location of the error. It is desirable for the type checker to recover from errors, so it can check the rest of the input. Since error handling affects the type checking rules, it has to be designed into the type system right from the start; the rules must be prepared to cope with errors.

The inclusion of error handling may result in a type system that goes beyond the one needed to specify correct programs. For example, once an error has occurred, we may not know the type of the incorrectly formed program fragment. Coping with missing information requires techniques similar to those needed for languages that do not require identifiers to be declared before they are used.

5.4. Specification of a simple type checker

In this section, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements, and functions.

❖ A Simple Language

The following grammar generates programs, represented by the nonterrninal *P*, consisting of a sequence of declarations *D* followed by a single expression *E*.

P → D ; E
D → D ; D | **id** : T
T → **char | integer | array [num] of T | ^T**
E → **literal** | **num** | **id** | E **mod** E | E [E] | E^

The language itself has two basic types, *char* and *integer*; a third basic type *type_error* is used to signal errors. For simplicity, we assume that all arrays start at 1. For example, *array [256] of char* leads to the type expression *array* (*1..256, char*) consisting of the constructor *array* applied to the subrange 1. .256 and the type *char*. As in Pascal, the prefix operator ^ in declarations builds a pointer type, so

^ integer

leads to the type expression *pointer(integer)*, consisting of the constructor *pointer* applied to the type *integer*.

The following is the part of a translation scheme that saves the type of an identifier.

P → D ; E	
D → D ; D	
D → **id** : T	{addtype(**id**.entry, T.type) }
T → **char**	**{** T.type := char }
T → **integer**	**{** T.type := integer }
T → **^T_1**	**{**T.type := pointer(T_1.type) }
T → **array [num] of T_1**	**{** T.type := array(1.. num.val, T_1.type) }

In the above translation scheme, the action associated with the production *D → id : T* saves a type in a symbol-table entry for an identifier. The action *addtype(**id**.entry, T.type)* is applied to synthesized attribute *entry* pointing to the symbol-table entry for **id** and a type expression represented by synthesized attribute type of nonterrninal *T*.

If *T* generates **char** or **integer**, then *T.type* is defined to be *char* or *integer* respectively. The upper bound of an array is obtained from the attribute *val* of token **num** that gives the integer represented by **num**. Arrays are assumed to start at 1, so the type constructor *array* is applied to the subrange 1.. **num**.*val* and the element type.

Since *D* appears before *E* on the right side of *P → D ; E*, we can be sure that the types of all declared identifiers will be saved before the expression generated by *E* is checked.

❖ Type Checking of Expressions

In the following rules, the synthesized attribute *type* for *E* gives the type expression assigned by the type system to the expression generated by *E*. The following semantic rules say that constants represented by the tokens **literal** and **num** have type *char* and *integer*, respectively:

E → **literal**	{*E.type* := *char*}
E → **num**	{*E.type* := *integer*}

We use a function *lookup*(*e*) to look the type saved in the symbol-table entry pointed to by *e*. When an identifier appears in an expression, its declared type is fetched and assigned to the attribute type:

$E \rightarrow$ **id** {*E.type* := lookup(**id**.entry)}

The expression formed by applying the *mod* operator to two subexpressions of type *integer* has type *integer*; otherwise, its type is *type-error*. The rule is

$E \rightarrow E_1$ **mod** E_2 {*E.type* := **if** E_1*.type = integer* **and** E_2*.type = integer* **then** *integer* **else** *type_error*}

In an array reference $E_1\ [E_2]$, the index expression E_2 must have type integer, in which case the result is the element type *t*, obtained from the type *array(s, t)* of E_1; we make no use of the index set *s* of the array.

$E \rightarrow E_1\ [E_2]$ {*E.type* := **if** E_2*.type = integer* **and** E_1*.type = array (s, t)* **then** *t* **else** *type_error*}

Within expressions, the postfix operator ^ yields the object pointed to by its operand. The type of E^ is the type t of the object pointed to by the pointer *E*:

$E \rightarrow E_1$^ {*E.type* := **if** E_1*.type = pointer (t)* **then** *t* **else** *type_error*}

❖ Type Checking of Statements

Since language constructs like statements typically do not have values, the special basic type *void* can be assigned to them. If an error is detected within a statement, the type assigned to the statement is *type_error*.

The statements we consider are assignment, conditional, and while statements. Sequences of statements are separated by semicolons.

The following is the translation scheme for checking the type of statements.

$S \rightarrow$ **id** := E {*S.type* := **if id**.*type = E.type* **then** *void* **else** *type_error*}

$S \rightarrow$ **if** E **then** S_1 { *S.type* := **if** *E.type = boolean* **then** S_1*.type* **else** *type_error*}

$S \rightarrow$ **while** E **do** S_1 { *S.type* := **if** *E.type = boolean* **then** S_1*.type* **else** *type_error*}

$S \rightarrow S_1 ; S_2$ { *S.type* := **if** S_1*.type = void* **and** S_2*.type = void* **then** *void* **else** *type_error*}

The first rule checks that the left and right sides of an assignment statement have the same type. The second and third rules specify that expressions in conditional and while statements must have type *boolean*. Errors are propagated by the last rule because a sequence of statements has type *void* only if each sub statement has type *void*. In these rules, a mismatch of types produces the type *type_error*; a friendly type checker would, of course, report the nature and location of the type mismatch as well.

❖ Type Checking of Functions

The application of a function to an argument can be captured by the production

$E \rightarrow E\ (E)$

in which an expression is the application of one expression to another. The rules for associating type expressions with nonterminal *T* can be augmented by the following production and action to permit function types in declarations.

$T \rightarrow T_1$ '$\rightarrow$' T_2 {*T.type := T_1.type $\rightarrow$ T_2.type*}

Quotes around the arrow used as a function constructor distinguish it from the arrow used as the meta symbol in a production.

The rule for checking the type of a function application is

E $\rightarrow$ *E_1 (E_2)* {*E.type* := **if** *E_2.type = s* **and** *E_1.type = s* $\rightarrow$ *t* **then** *t* **else** *type_error*}

This rule says that in an expression formed by applying E_1 to E_2, the type of E_1 must be a function $s \rightarrow t$ from the type *s* of E_2 to some range type t; the type of E_1 (E_2) is *t*.

Many issues related to type checking in the presence of functions can be discussed with respect to the simple syntax above. The generalization to functions with more than one argument is done by constructing a product type consisting of the arguments. Note that *n* arguments of type $T_1, \ldots, T_n$, can be viewed as a single argument of type $T_1 \times \ldots \times T_n$ .For example, we might write

root: (real $\rightarrow$ real) $\times$ real $\rightarrow$ real

to declare a function root that takes a function horn reals to reals and a real as arguments and returns a real. Pascal-like syntax for this declaration is

function root (function f (real): real; x: real): real

5.5. Type Conversions

Consider expressions like x + i where x is of type real and i is of type integer. Since the representation of integers and reals is different within a computer, and different machine instructions are used for operations on integers and reals, the compiler may have to first convert one of the operands of + to ensure that both operands are of the same type when the addition takes place.

The language definition specifies what conversions are necessary. When an integer is assigned to a real, or vice versa, the conversion is to the type of the left side of the assignment. In expressions, the usual transformation is to convert the integer into a real number and then perform a real operation on the resulting pair of real operands. The type checker in a compiler can be used to insert these conversion operations into the intermediate representation of the source program.

Conversion from one type to another is said to be implicit if it is to be done automatically by the compiler. Implicit type conversions, also called coercions, are limited in many languages to situations where no information is lost in principle; e.g, an integer may be converted to a real but not vice-versa. In practice, however, loss is possible when a real number must fit into the same number of bits as an integer. Automatic conversion can be done by using the built-in library functions like INT, inttoreal, and others.

Conversion is said to be explicit if the programmer must write something to cause the conversion.

For example, the integer 2 is converted to a real in the code for the expression 2 * 3.14:

t_1 = inttoreal (2)

$t_2 = t_1$ * 3.1 4