Type Checking of Statements

- Since the 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 productions in Fig.3 can be combined with those of Fig.1 if we change the production for a complete program to $P \rightarrow D ; S$

lookup Symbol Table?

$S \rightarrow id = E$ { 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 ; }

Fig 3 Translation scheme for checking the type of statements

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 distinguishes it from the arrow used as the metasymbol in a production.

- The rule for checking the type of a function application is

$E \rightarrow E_1 (E_2)$ { $E.type$ = if $E_1.type = s \rightarrow t$ and $E_2.type = s$ then t else type-error ; }

- The generalisation to functions with more than one argument is done by constructing a product type consisting of the arguments. Note that n arguments of type $T_1, T_2, \ldots, T_n$ can be viewed as a single ~~assignment~~ argument of type $T_1 \times T_2 \times \ldots \times T_n$.

Equivalence of Type Expressions

- The type checking rules have the form
  "If two type expressions are equal
  then return a certain type else return type-error".
- It is therefore important to have a precise definition of when two type expressions are equivalent.
- Potential ambiguities arise when names are given to type expressions and the names are then used in subsequent type expressions.
- The key issue is whether a name in a type expression stands for itself or whether it is an abbreviation for another type expression.
- The notion of type equivalence implemented by a specific compiler can often be explained using the concepts of 'structural' and 'name' equivalence.
- The discussion is in terms of a graph representation of type expressions with leaves for basic types and type names, and interior nodes for type constructors.
- Recursively defined types lead to cycles in the type graph if a name is treated as an abbreviation for a type expression

Structural Equivalence of Type Expressions

- As long as type expressions are built from basic types and constructors, a natural notion of equivalence between two type expressions is "structural equivalence"; i.e. two expressions are either the same basic type, or are formed by applying the same constructor to structurally equivalent types.
- Two type expressions are structurally equivalent iff they are identical.
- The algorithm for testing structural equivalence in the figure below assumes that the only type constructors are for arrays, products, pointers, and functions. The algorithm recursively compares the structure of type expressions without checking for cycles so it can be applied to a tree or a dag representation.

```
function sequiv (s, t) : boolean ;
  begin
      if s and t are the same basic type
          then return true
      else if s = array (s1, s2) and t = array (t1, t2)
          then return sequiv (s1, t1) and sequiv (s2, t2)
      else if s = s1 x s2 and t = t1 x t2
          then return sequiv (s1, t1) and sequiv (s2, t2)
      else if s = pointer (s1) and t = pointer (t1)
          then return sequiv (s1, t1)
      else if s = s1 → s2 and t = t1 → t2
          then return sequiv (s1, t1) and sequiv (s2, t2)
      else return false ;
  end
```

Name Equivalence of Type Expressions

- In some languages, types can be given names. For example, in Pascal

  type link = ^cell ;

  declares identifier 'link' to be a name for the type '^cell'.
- When names are allowed in type expressions, two notions of equivalence of type expressions arise, depending on the treatment of names.
- Name equivalence views each type name as a distinct type, so two type expressions are name equivalent iff they are identical.
- Under structural equivalence, names are replaced by the type expressions they define, so two type expressions are structurally equivalent if they represent two structurally equivalent expressions when all names have been substituted out.

Symbol Table

- Symbol Attributes
  - Each symbol in a program has associated with it a series of attributes that are derived both from the syntax and semantics of the source language and the symbol's declaration and use in the particular program.
  - The typical attributes include type, scope, and size.
- Symbol Table Structure
  - The scope rules of the source language dictate the structure of the global symbol table.
  - For many languages, such as ALGOL-60, PASCAL, and PL/I the scoping rules have the effect of structuring the entire global symbol table as a tree of local symbol tables with the table for the global scope as its root and the local tables for nested scopes as the children of the table for the scope they are nested in.
  - A simpler data structure (stack) can be used for such languages since at any point during compilation, we are processing a particular node of the tree and only need access to the symbol tables on the path from that node to the root of the tree.