# DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

**Declarations in a Procedure:**

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say offset, can keep track of the next available relative address.

In the translation scheme shown below:

* Non terminal P generates a sequence of declarations of the form id :T.
* Before the first declaration is considered, offset is set to 0. As each new name is seen, that name is entered in the symbol table with offset equal to the current value of offset, and offset is incremented by the width of the data object denoted by that name.
* The procedure enter( name, type, offset ) creates a symbol-table entry for name, gives its type type and relative address offset in its dataarea.
* Attribute type represents a type expression constructed from the basic types integer and real by applying the type constructors pointer andarray. If type expressions are represented by graphs, then attribute type might be a pointer to the node representing a type expression.
* The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be4.

Computing the types and relative addresses of declared names

```
P→D             { offset : = 0}
D→D ; D
D→id: T         { enter(id.name, T.type, offset);
                  offset : = offset + T.width }
T→integer       { T.type : =integer;
                  T.width : = 4  }
```

T→real { T.type : = real;
T.width : = 8 }

T→array [ num ]ofT1 { T.type : = array(num.val,T1.type);
T.width : = num.val X T1.width }

T→↑T1 { T.type : = pointer ( T1.type);
T.width : = 4}

**Keeping Track of Scope Information:**

When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language:

P:D

D:D; D |id: T |proc id; D ; S

One possible implementation of a symbol table is a linked list of entries for names. A new symbol table is created when a procedure declaration D:proc id D1;S is seen, and entries for the declarations in D1 are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations is that the procedure enter is told which symbol table to make an entryin.

For example, consider the symbol tables for procedures read array, exchange, and quicksort pointing back to that for the containing procedure sort, consisting of the entire program. Since partition is declared within quicksort, its table points to that of quicksort.

**The semantic rules are defined in terms of the following operations:**

1. Mktable (previous) creates a new symbol table and returns a pointer to the new table. The argument previous points to a previously created symbol table, presumably that for the enclosing procedure.
2. enter(table, name, type, offset) creates a new entry for name name in the symbol table pointed to by table. Again, enter places type type and relative address offset in fields within the entry.
3. Add width(table, width) records the cumulative width of all the entries in table in the header associated with this symbol table.

4. enterproc(table, name, newtable) creates a new entry for procedure name in the symbol table pointed to by table. The argument newtable points to the symbol table for thisprocedure name.

**Syntax directed translation scheme for nested procedures**

| | |
|---|---|
| P→MD | { addwidth ( top( tblptr) , top (offset)); pop (tblptr); pop (offset) } |
| M→ε | { t : = mktable(nil); push (t,tblptr); push (0,offset) } |
| D→D1 ; D2 | |
| D→proc id ; ND1; S | { t : = top(tblptr); addwidth ( t, top (offset)); pop (tblptr); pop (offset); enterproc (top (tblptr), id.name, t) } |
| D→id: T | { enter (top (tblptr), id.name, T.type, top (offset)); top (offset) := top (offset) + T.width} |
| N→ε | { t := mktable (top (tblptr)); push (t, tblptr); push (0,offset)} |

* The stack tblptr is used to contain pointers to the tables for sort, quicksort, and partition when the declarations in partition are considered.
* The top element of stack offset is the next available relative address for a local of the current procedure.
* All semantic actions in the sub trees for B and Cin

A→BC{actionA}

are done before action A at the end of the production occurs. Hence, the action associated with the marker M is the first to be done.

The action for non terminal M initializes stack tblptr with a outermost scope, created by operation mktable(nil). The action also pushes relative address 0 onto stack offset. Similarly, the non terminal N uses the operation mktable(top(tblptr)) to create a new symbol table. The argument top(tblptr) gives the enclosing scope for the new table. For each variable declaration id: T, an entry is

created for id in the current symbol table.

The top of stack offset is incremented by T.width. When the action on the right side of D → proc id; ND1; S occurs, the width of all declarations generated by D1 is on the top of stack offset; it is recorded using add width. Stacks tbl ptr and offset are then popped. At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

**ASSIGNMENT STATEMENTS**

Suppose that the context in which an assignment appears is given by the following grammar.

P→M D

M→ε

D→D ; D | id: T | proc id; N D ; S N→ε

Non terminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Translation scheme to produce three-address code for assignments

S→id :=E { p : = lookup (id.name);
if p ≠ nil then
emit( p ‘ : =’ E.place) else error }

E→E1+E2 { E.place : =newtemp;
emit( E.place ‘: =’ E1.place ‘ + ‘ E2.place ) }

E→E1*E2 { E.place : =newtemp;
emit( E.place ‘: =’ E1.place ‘ * ‘ E2.place ) }

E→-E1 { E.place : =newtemp;
emit ( E.place ‘: =’ ‘uminus’ E1.place ) }

E→ (E1) { E.place : = E1.place}

E→id { p : = lookup ( id.name); if p
≠ nil then
E.place : = p else error }