SEMANTIC ANALYSIS USING S-ATTRIBUTES

TOPICS COVERED

- This part of the module deals with writing attribute grammars for performing static semantic analysis of various language constructs
- the sequence of topics covered are
  1. Processing of declarations : defining required attributes, symbol table organization for handling nested procedures and scope information, translation scheme for sample declaration grammars
  2. Type analysis and type checking : concept of type expression and type equivalence, an algorithm for structural equivalence, performing type checking in expressions and statements and generating intermediate code for expressions.
  3. Intermediate code forms : three address codes and their syntax
  4. Translation of assignment statement
  5. Translation of Boolean expressions and control flow statements.
- Most of the semantic analysis and translation has been illustrated using synthesized attributes only.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

PROCESSING OF DECLARATIONS

- Declarations are typically part of the syntactic unit of a ***procedure*** or a ***block***. They provide information such as names that are local to the unit, their type, etc.
- Semantic analysis and translation involves,

  collecting the names in a symbol table

  entering their attributes such as type, storage requirement and related information

  checking for uniqueness, etc., as specified by the underlying language

  computing relative addresses for local names which is required for laying out data in the activation record
- We begin with a simple grammar for declarations

PROCESSING OF DECLARATIONS

1.1 ***Type associated with a Single Identifier***

In the grammar given below, type is specified after an identifier.

- We use a synthesized attribute ***T.type*** for storing the type of ***T***.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

PROCESSING OF DECLARATIONS

- ***T.width*** is another synthesized attribute that denotes the number of memory units taken by an object of this type
- `id` . ***name*** is a synthesized attribute for representing the lexeme associated with `id`
- ***offset*** is used to compute the relative address of an object in the data area. It is a global variable.
- Procedure ***enter(name,type,width,offset)*** creates a symbol table entry for `id` . ***name*** and sets the ***type***, ***width*** and ***offset*** fields of this entry.

$P \rightarrow MD$

$M \rightarrow \epsilon$ { ***offset*** := 0 }

$D \rightarrow D ; D$

$D \rightarrow$ `id` : T { ***enter***(`id` . ***name***, ***T.type***, ***T.width***, ***offset***);
offset := ***offset*** + ***T.width*** }

$T \rightarrow$ `integer` { ***T.type*** := `integer`;
T.width := 4 }

$T \rightarrow$ `real` { ***T.type*** := `real`;
T.width := 8 }

SEMANTIC ANALYSIS USING S-ATTRIBUTES

PROCESSING OF DECLARATIONS

$$
\begin{array}{ll}
T \rightarrow \texttt{array [num] of } T_1 & \\
 & \{\ \boldsymbol{T.width} := \texttt{num}.\boldsymbol{val} \times T_1.\boldsymbol{width}; \\
 & \boldsymbol{T.type} := \boldsymbol{array}(\texttt{num}.\boldsymbol{val}, T_1.\boldsymbol{type})\ \} \\
T \rightarrow \uparrow T_1 & \{\ \boldsymbol{T.type} := \boldsymbol{pointer}(T_1.\boldsymbol{type}); \\
 & \boldsymbol{T.width} := 4\ \}
\end{array}
$$

In the code for semantic analysis given above,

- Arrays are assumed to start at 1 and $\texttt{num}.\boldsymbol{val}$ is a synthesized attribute that gives the upper bound (integer represented by token num)
- If a nonterminal occurs more than once in a rule, its references in the rhs are subscripted and then used in the semantic rules , e.g., use of $\boldsymbol{T}$ in the array declaration

1.2 ***Type associated with a List of Identifiers***

If a list of ids is permitted in place of a single id in the above grammar, then

- all the ids must be available when the type information is seen subsequently

SEMANTIC ANALYSIS USING S-ATTRIBUTES

PROCESSING OF DECLARATIONS

- a possible approach is to maintain a list of such `ids`, and carry forward a pointer to the list as a synthesized attribute, ***L.list***
- The procedure ***enter*** given below has different semantics which creates a symbol table entry for each member of a list and also sets the type information
- ***makelist***() and ***append***() are two functions for making a list with one element and inserting an element in the list respectively. These functions return a pointer to the created / updated list.

$P \rightarrow D$

$D \rightarrow D ; D$

$D \rightarrow L : T$ { ***enter***(***L.list***, ***T.type***, ***T.width***) }

$L \rightarrow$ `id`, L_1 { ***L.list*** := ***append***(L_1.***list***, `id`.***name***) }

$L \rightarrow$ `id` { ***L.list*** := ***makelist***(`id`.***name***) }

$T \rightarrow$ `integer` { ***T.type*** := `integer`;
T.width := 4 }

$T \rightarrow$ `real` { ***T.type*** := `real`;
T.width := 8 }

$T \rightarrow$ `array [ num ] of` T_1
{ same as earlier }

$T \rightarrow \uparrow T_1$ { same as earlier }

SEMANTIC ANALYSIS USING S-ATTRIBUTES

PROCESSING OF DECLARATIONS

2. ***Type Specified at the beginning of a list***
   The following grammar provides an example.

$$D \rightarrow D ; D \quad T \; L$$
$$L \rightarrow \texttt{id} \; , \; \mathbf{L} \qquad \texttt{id}$$
$$T \rightarrow \texttt{integer} \quad \texttt{real} \quad \ldots$$

- **The grammar is rewritten to make writing of semantics easier, as illustrated below.**
- **Note the role played by *D.syn* in propagating the type information**

$D \rightarrow D ; D$	
$D \rightarrow D_1$, id	{ ***enter***(id.***name***,D_1.***syn***) ; ***D.syn*** := D_1.***syn*** }
$D \rightarrow T$ id	{ ***enter***(id.***name***,***T.type***) ; ***D.syn*** := ***T.type*** }
$T \rightarrow$ integer real ...	{ same as earlier }

SEMANTIC ANALYSIS USING S-ATTRIBUTES

SCOPE INFORMATION

- **The method described above can be used to process declarations of names local to a procedure.**
- **For a language that permits nested procedures with lexical scoping, the same method can be extended by having multiple symbol tables. Maintain a separate symbol table for each procedure.**
- **We use the earlier grammar for illustration but add another rule to it**

$$
\begin{aligned}
&P \rightarrow D \\
&D \rightarrow D ; D \\
&D \rightarrow \texttt{id} : T \\
&D \rightarrow \texttt{proc id} ; D ; S \\
&T \rightarrow \texttt{integer} \\
&T \rightarrow \texttt{real} \\
&T \rightarrow \texttt{array [num] of } T_1 \\
&T \rightarrow \uparrow T_1
\end{aligned}
$$

- **The rule** $D \rightarrow \texttt{proc id} ; D ; S$
  permits nested procedures, but without parameters.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

SCOPE INFORMATION

- **The design solution is to create a separate table for handling names local to each procedure. The tables are linked in such a manner that lexical scoping rules are honoured.**
- **The following attributes are defined**
  `id.num` T.**type** T.**width**
  The global variable *nest* is used to compute the nesting level for a procedure.
- **Two new nonterminals, *M* and *N*, are introduced so that semantic actions can be written at these points**

$$
\begin{aligned}
P &\rightarrow M\,D \\
D &\rightarrow \texttt{proc}\ \ \texttt{id}\ \ ;\ \ N\ D\ ;\ S \\
M &\rightarrow \epsilon \\
N &\rightarrow \epsilon
\end{aligned}
$$

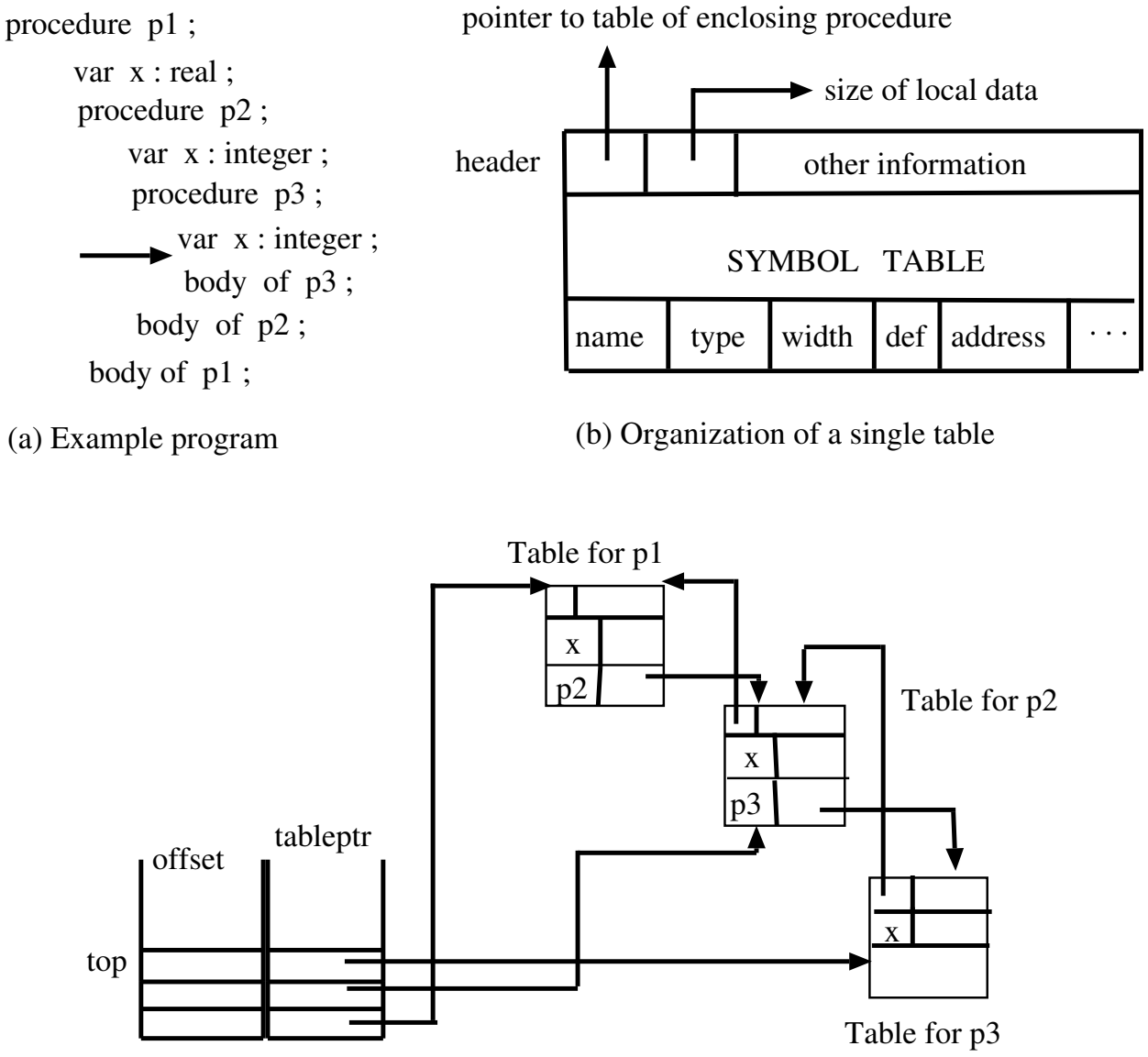
- **The place marked by M is used to initialize all information related to the symbol table associated with this level. Similarly N would signal the start of activity for another table.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

SCOPE INFORMATION

- The various symbol tables have to be linked properly so as to correctly resolve nonlocal references.
- A stack of symbol tables is used for this purpose.
- The symbol table structures and their interconnection are shown in Figure 5.1
- For symbol table management, the following organization and routines are assumed.
    1. Each table is an array of a suitable size which is linked using the fields as shown in Figure 5.1
    2. There are 2 stacks, called *tableptr* and *offset*. The stack *tableptr* points to the currently active procedure's table and the pointers to the tables of the enclosing procedures are kept below in the stack.
    3. The other stack is used to compute the relative addresses for locals within a procedure.
    4. *mktable(previous)* is a procedure that creates a new table and returns a pointer to it. It also gives the pointer to the previous table through the argument, *previous*. This is required in order to link the new table with that of the closest enclosing procedure's.

FIGURE 5.1 : Symbol Table Organization for Nested Structures



(a) Example program

(b) Organization of a single table

(c) Symbol Tables and their interconnection when control reaches the marked point

SEMANTIC ANALYSIS USING S-ATTRIBUTES

SCOPE INFORMATION

5. ***enter(table,name,type,width,offset)*** is a procedure for creating a new entry for the local name, ***name***, along with its attributes.

6. ***addwidth(table,width)*** is used to compute the total space requirement for all the names of a procedure.

7. ***enterproc(table,name,nest_level,newtable)*** creates a new entry for a procedure name. The last argument is a pointer to the corresponding table.

The semantic rules are given now.

$P \rightarrow MD$	{ ***addwidth***(***top***(***tableptr***),***top***(***offset***)) ; }
	pop(***tableptr***) ; ***pop***(***offset***) }
$M \rightarrow \epsilon$	{ ***p*** := ***mktable***(nil) ;
	push(***p***,***tableptr***) ; ***push***(0,***offset***);
	$nest$:= 1; }
$D \rightarrow D; D$	
$D \rightarrow$ `id` : T	{ ***enter***(***top***(***tableptr***),`id.`***name***,***T.type***,***T.width***);
	top(***offset***) := ***top***(***offset***) + ***T.width*** }

SEMANTIC ANALYSIS USING S-ATTRIBUTES

SCOPE INFORMATION

$D \rightarrow$ `proc id ;` $N\ D$ `;` S	{ ***p := top(tableptr); addwidth(p,top(of***
	pop(tableptr) ; pop(offset);
	enterproc(top(tableptr),`id`***.name,nest, p***
	$nest--$; }
$T \rightarrow$ `integer`	{ ***T.type*** := `integer`;
	T.width := 4 }
$T \rightarrow$ `real`	{ ***T.type*** := `real`;
	T.width := 8 }
$T \rightarrow$ `array [ num ] of` T_1	
	{ ***T.width :=*** `num`***.val*** $\times T_1$***.width*** ;
	T.type := array(`num`***.val,***T_1***.type)*** }
$T \rightarrow \uparrow T_1$	{ ***T.type*** := ***pointer***$(T_1$***.type***$)$;
	T.width := 4 }
$N \rightarrow\ \epsilon$	{ ***p := mktable(top(tableptr))*** ;
	push(p,tableptr); ***push(0,offset)*** ;
	$nest++$; }

- **The translation scheme given above is inadequate to handle recursive procedures. The reason being that the name of such a procedure would be entered in its closest enclosing symbol table only after the entire procedure is parsed.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

SCOPE INFORMATION

- **However a recursive procedure would have a call to itself within its body and this call cannot be translated since the name would not be present in the table.**
- **A possible remedy is to enter a procedure's name immediately after it has been found and the corresponding changes are :**

$P \rightarrow MD$ { ***addwidth(top(tableptr),top(offset))*** ; }
pop(tableptr) ; ***pop(offset)*** }

$M \rightarrow \epsilon$ { ***p := mktable***(nil) ;
push(p,tableptr) ; ***push***$(0,$***offset***$)$;
nest $:= 1$; }

$D \rightarrow D; D$

$D \rightarrow$ `id` : T { ***enter(top(tableptr),***`id`***.name,T.type,T.width)***;
top(offset) $:=$ ***top(offset)*** $+$ ***T.width*** }

$D \rightarrow$ `proc` `id` ; N D ; S
{ ***p := top(tableptr)***; ***addwidth(p,top(offset))*** ;
pop(tableptr); ***pop(offset)***;
$nest--$; }

$T \rightarrow$ `integer` { ***T.type*** $:=$ `integer`;
T.width $:= 4$ }

SEMANTIC ANALYSIS USING S-ATTRIBUTES

SCOPE INFORMATION

$T \rightarrow$ `real` { ***T.type*** := `real`;
T.width := 8 }

$T \rightarrow$ `array [ num ] of` T_1
{ ***T.width*** := `num.`***val*** $\times T_1$***.width*** ;
T.type := ***array***(`num.`***val***, T_1***.type***) }

$T \rightarrow \uparrow T_1$ { ***T.type*** := ***pointer***(T_1***.type***) ;
T.width := 4 }

$N \rightarrow \epsilon$ { ***p := mktable(top(tableptr))*** ;
enterproc(top(tableptr),`id.`***name,nest,p)***
push(p,tableptr); ***push***(0,***offset***) ;
$nest++$; }

SYMBOL TABLE ORGANIZATION

- We now consider the issues related to the design and implementation of symbol tables
- Two major factors are the features supported by the programming language and the need to make efficient access of the table
- The scoping rules of the language indicate whether single or multiple tables are convenient

SEMANTIC ANALYSIS USING S-ATTRIBUTES

SYMBOL TABLE ORGANIZATION

- The various types and attributes decide the internal organization for an entry in the table.
- Need for efficient access of the table influence the implementation considerations, such as whether arrays, linked lists, binary trees, hash tables or an appropriate mix should be used. We examine two possible hash table based implementations.

MULTIPLE HASH TABLES

1. Each symbol table is maintained as a separate hash table
2. For nonlocal names, several tables may need to be searched which, in turn, may result in searching several chains in case of colliding entries.
3. An important issue in this design is the access time for globals.
4. Another concern relates to space. The size of each table, if fixed at compile time, could turn out to be a disadvantage, if the estimates are wrong.

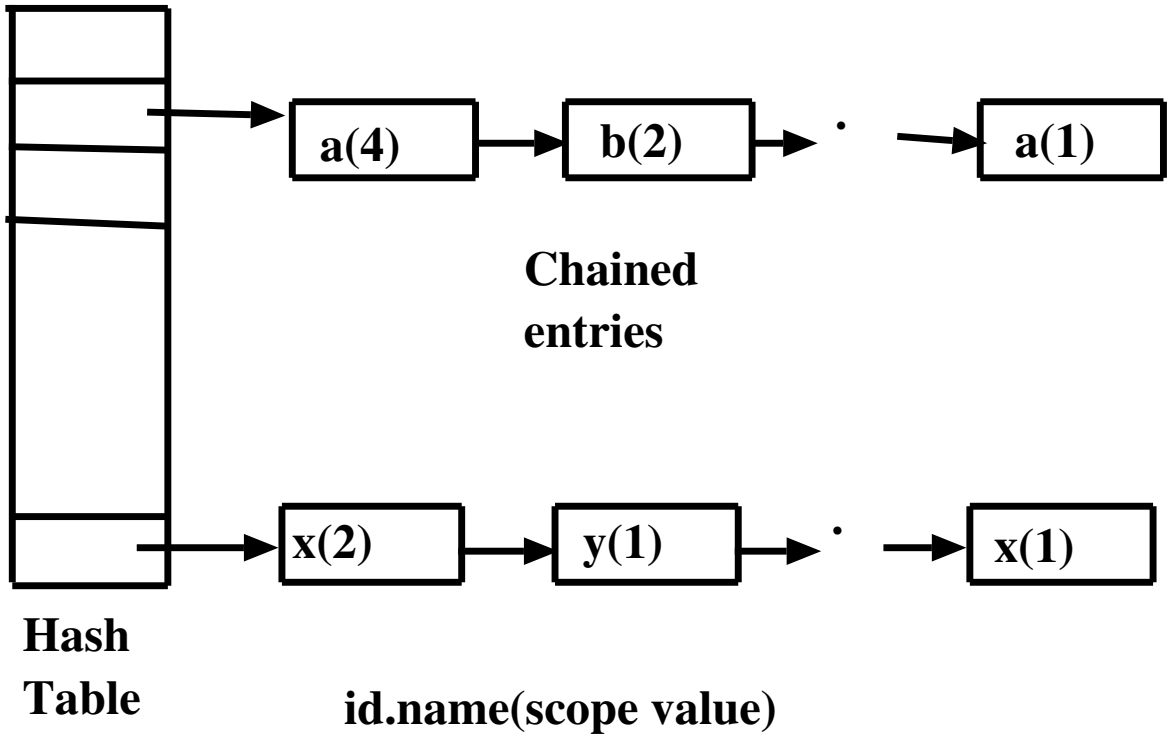
SEMANTIC ANALYSIS USING S-ATTRIBUTES

SINGLE HASH TABLE

- A single hash table for block structured languages is possible, provided one stores the scope number associated with each name. A feasible organization is shown in Figure 5.2

1. new names are entered at the front of the chains, hence search for a name terminates with the first occurence on the chain
2. when a scope is closed, the table has to be updated by deleting all entries with the current scope number. The operation is not too expensive, since the search stops at the first entry where the scope values mismatch
3. basic table operations, enter and search, are efficient because there is a single table. However, storage gains may be compensated due to the inclusion of scope value with each entry
4. all the locals of a scope are not grouped together, hence additional effort in time and space is needed to maintain such information, if so required.

FIGURE 5.2 : Single Table for Handling Nested Scopes



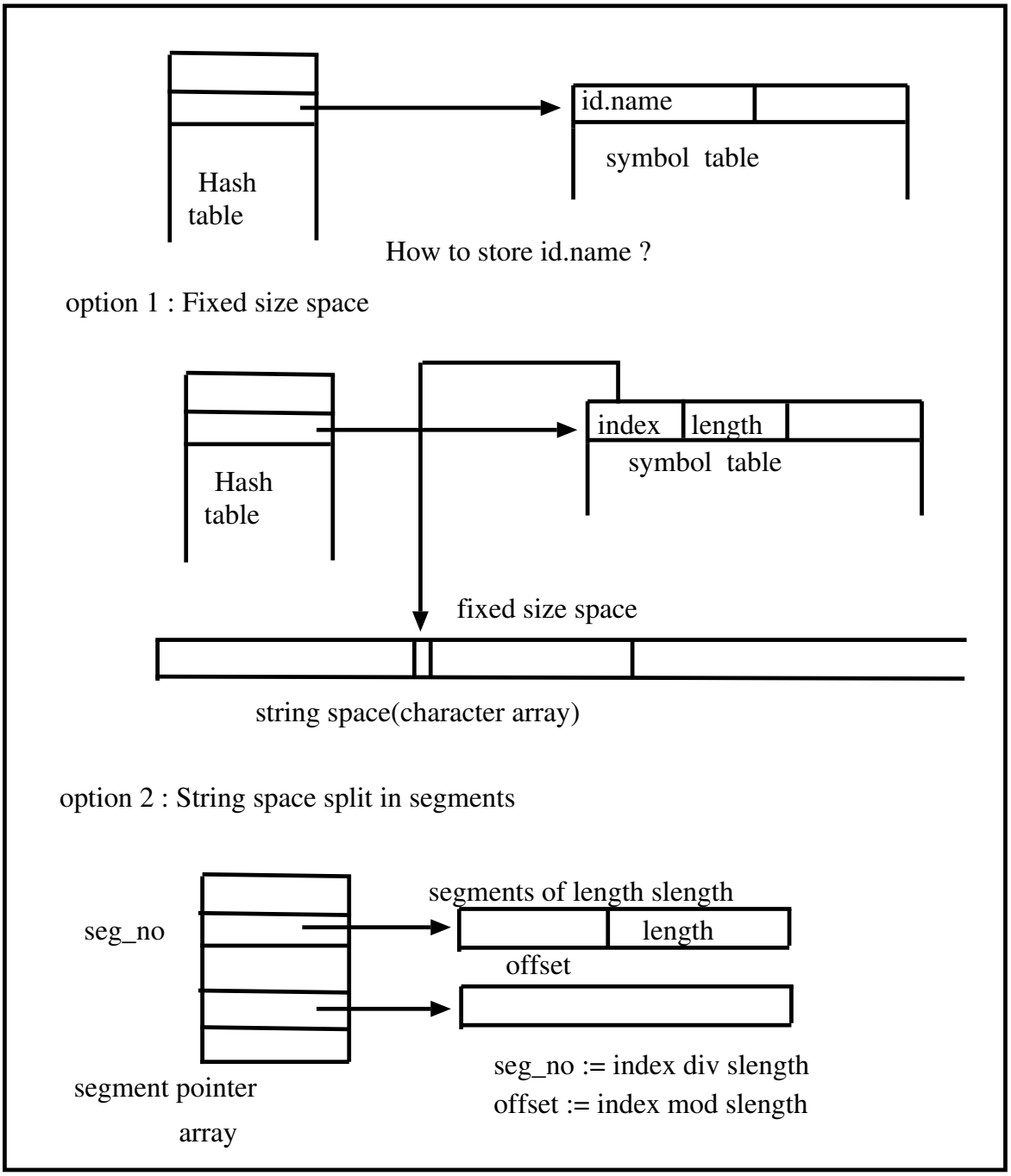
SEMANTIC ANALYSIS USING S-ATTRIBUTES

SYMBOL TABLE ENTRIES

The attributes for each name have to be predecided, depending on the processing they undergo in various phases of the compiler.

- From the viewpoint of semantic analysis, relevant attributes have been identified in the preceding examples.
- The other phases, like error analysis and handling, code generation and optimization , may introduce more attributes for an entry in the table.
- Once the attributes of a name are decided, the next concern is their organization. The use of these attributes in various processing (kind of operations) would dictate their implementation. The concern would be space and/or time. The following example highlights this point.
- ***String Space Representation for a Name*** :
  Consider the string corresponding to a name. Whether this should be explicitly stored in the table or not depends on various factors. A possible organization that does not do so is given in Figure 5.3

FIGURE 5.3 : String Space Managemment



SEMANTIC ANALYSIS USING S-ATTRIBUTES

PROCESSING OF RECORD DECLARATIONS

- The design discussed for handling of nested procedure decalarations can also be used for doing semantic analysis of record declarations
- The earlier grammar is first augmented as given below, ***L*** is a marker nonterminal used for writing semantic code at that point

$$
\begin{aligned}
&D \rightarrow D \texttt{ ; } D \quad \texttt{id : } T \\
&T \rightarrow \texttt{integer} \quad \texttt{real} \\
&T \rightarrow \texttt{array [num] of } T_1 \quad \uparrow T_1 \\
&T \rightarrow \texttt{record } L\ D \texttt{ end} \\
&L \rightarrow \epsilon
\end{aligned}
$$

- Two stacks ***tableptr*** and ***offset*** are used in the same sense as before.
- attribute ***T.width*** gives the width of all the data objects within a record
- attribute ***T.type*** is used to hold the type of a record. The expression used here is explained later during Type Analysis.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

PROCESSING OF RECORD DECLARATIONS

$L \rightarrow \epsilon$

{ ***T.type*** := ***record***(***top***(***tableptr***)) ;
T.width := ***top***(***offset***) ;
pop(tableptr) ; ***pop(offset)*** }

$T \rightarrow$ `record` L D `end`

{ ***p*** := ***mktable***(nil) ;
push(p,tableptr); ***push***(0,***offset***) }

SEMANTIC ERRORS IN DECLARATION PROCESSING

- **The relevant errors in this context are**
  Uniqueness checks, names referenced but not declared, names declared but not referenced, etc.
- **Code for detection and handling of such errors can be incorporated in the semantic actions at the appropriate places.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TYPE ANALYSIS AND TYPE CHECKING

Static type checking is an important part of semantic analysis. It is required to (i) detect errors arising out of application of an operator to an incompatible operand, and (ii) to generate intermediate code for expressions and statements.

- The notion of types and the rules for assigning types to language constructs are defined by the source language. Typically, an expression has a type associated with it and types usually have structure.
- Usually languages support types of two kinds, basic and constructed. Examples of basic types are integer, real, character, boolean and enumerated while array, record, set and pointer are examples of constructed types. The constructed types have structure.
- How to express type of a language construct ?
  For basic types, it is straightforward but for the other type it is nontrivial. A convenient form is a *type expression.*
- A type expression is defined as follows.
  1. A *basic type* is a type expression. Thus *boolean, char, integer, real*, etc. are type expressions.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TYPE ANALYSIS AND TYPE CHECKING

2. For the purposes of type checking, two more basic types are introduced, ***type_error*** and ***void***.

3. A type name is a type expression. In the example given below, the type name 'table' is a type expression.

```
type table=
          array[1 .. 100] of array[1 .. 10] of integer
```

4. A type constructor applied to type expressions is a type expression. The constructors array, product, record, pointer and function are used to create constructed types as described below.

   - Array : If T is a type expression, then $array(I, T)$ is also a type expression; elements are of type T and index set is I (usually a range of integer). Example would be the type expression ***array(1..100, integer)*** for variable `ROW` in the declaration
     `var ROW : array [1 .. 100] of integer;`
   - Product : If T_1 and T_2 are type expressions, then their cartesian product $T_1 \times T_2$ is a type expression; $\times$ is left associative.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TYPE ANALYSIS

- **Records** : The constructor *record* applied to a product of the type of the fields of a record is a type expression. For example,

```
type entry =  record
              token :  array [ 1..  32 ] of char;
              salary :  real
              end;
var employee : array[1 .. 20] of entry;
```

- The type name *entry* has the type expression

record((token$\times$*array(1..32, char))* $\times$ *(salary*$\times$*real))*

Write the type expression for employee.

- **Pointers** : If T is a type expression, then *pointer(T)* is a type expression. For example, if the following declaration `var ptr :` $\uparrow$ entry, is added to that given above, then the type expression for ptr is :

pointer(record((token$\times$*array(1..32, char))* $\times$ *(salary*$\times$*real)))*

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TYPE SYSTEM AND TYPE CHECKER

- **Function : A function in a programming language can be expressed by a mapping $D \rightarrow R$, where D and R are *domain* and *range* type.**

- **The type of a function is given by the type expression $D \rightarrow R$. For the declaration**

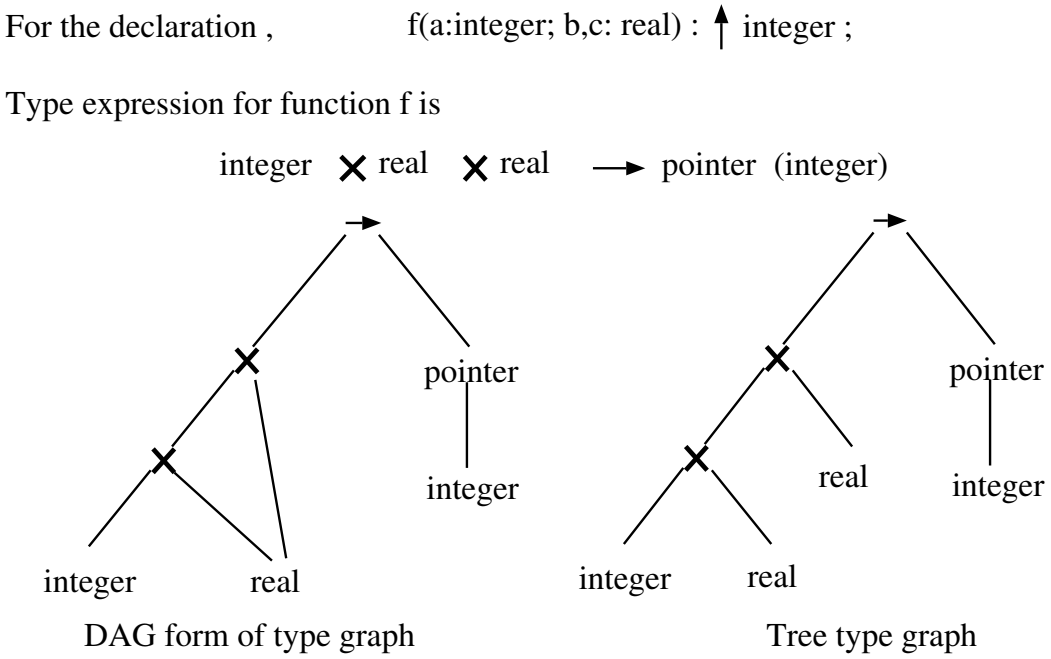
  For the declaration
  `function f(a :real,b:integer) :↑ integer;`
  the type expression for f is

$$\textit{real} \times \textit{integer} \rightarrow \textit{pointer(integer)}$$

- **Programming languages usually restrict the type a function may return, e.g., range R is not allowed to have arrays and functions in several languages.**

- **How to use the type expression ?**
  The type expression is a linear form of representation that can conveniently capture structure of a type. This expression may also be represented in the form of a dag (directed acyclic graph) or tree.

Figure 5.4 : TYPE EXPRESSIONS AND TYPE GRAPHS



- **The tree (or dag) form, as shown in Figure 5.4, is more convenient for type checking.**
- **A *type system* is a collection of rules for assigning type expressions to linguistic constructs of a program.**
- **A *type checker* implements a *type system*.**
- **Checking for type compatibility is essentially finding out when two type expressions are *equivalent*.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TYPE CHECKING

- **For type checking, the general approach is :**
  if two type expressions are equivalent then return an appropriate type else return type_error.
  This, in turn, requires a definition of the notion of equivalence of type expressions.

Example 1 :

```
type link = ↑ node;
var first, last : link;
p : ↑ node;
q , r : ↑ node;
```

Example 2 :

```
typeperson =  record
              id :  integer;
              weight :  real
              end;
    car = record
          id :  integer;
          weight :  real
          end;
var x :  person; var y :  car;
```

SEMANTIC ANALYSIS USING S-ATTRIBUTES

EQUIVALENCE OF TYPE EXPRESSIONS

- The type expressions in Example 1 are :

Variable	Type Expression
next	link
last	link
p	pointer(cell)
q	pointer(cell)
r	pointer(cell)

- The issue is to decide whether all the type expressions given above are type equivalent. Presence of names in type expressions give rise to two distinct notions of equivalence.
- ***Name Equivalence*** treats each type name as a distinct type, two expressions are name equivalent if and only if they are identical
- Under name equivalence, variables next and last are equivalent. Also variables p, q and r are name equivalent, but next and p are not. In Example 2, x and y are not name equivalent.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

NAME AND STRUCTURAL EQUIVALENCE

- If type names are replaced by the expressions defined by them, then we get the notion of *structural equivalence*.

  Two expressions are structurally equivalent if they represent structurally equivalent type expressions after the type names have been substituted in.

- All the variables in Example 1 and also x and y in Example 2 are structurally equivalent

- Example 2 indicates the problem with structural equivalence. If user declares two different types, they probably represent different objects; making them type equivalent, merely because they have the same structure, may not be desirable.

- Name equivalence is safer but more restrictive. It is also easy to implement. Compiler only needs to compare the strings representing names of the types. Language Ada uses name equivalence.

- We consider structural equivalence in detail.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

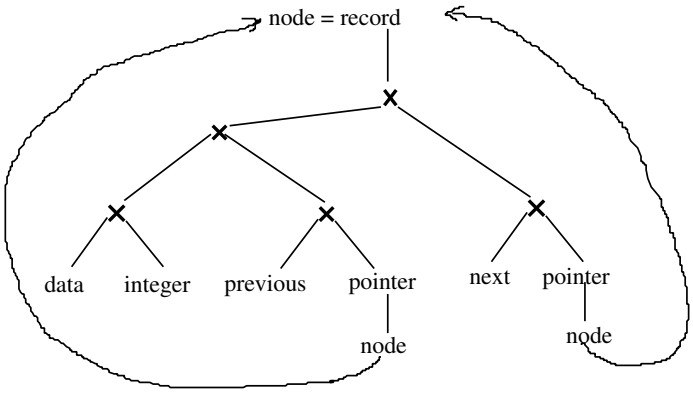
IMPLEMENTATION OF STRUCTURAL EQUIVALENCE

- Two expressions are structurally equivalent if
  i) the expressions are the same basic type, or
  ii) are formed by applying the same constructor to structurally equivalent types
  Languages Pascal and C use structural equivalence
- The objective is to develop an algorithm to be used by a compiler for detecting structural equivalence.
- The complexity of the task is clearly dependent on the structure of the type graphs involved. When the type graph is a tree or a dag, a simpler algorithm is possible. For type graphs containing cycles, such an algorithm is nontrivial.
- Cycle in a type graph typically arises due to recursively defined type names. A common situation is depicted by pointers to records as shown below.

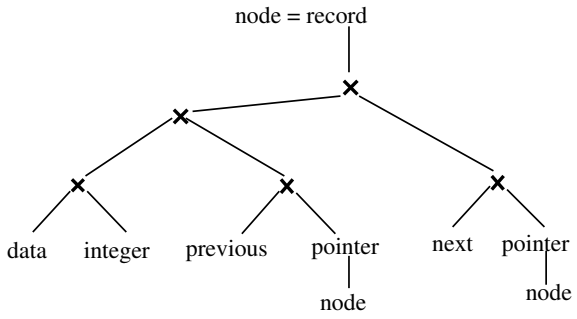
```
type link = ↑ node ;
     node = record
              data : integer ;
              previous : link;
              next : link
              end;
```

- Type graph for node is shown in Figure 5.5.
- Figure 5.5(a) is a natural way of representing the ***type expression*** for node, it contains two cycles.
- Figure 5.5(b) is an acyclic equivalent of Figure 5.5(a). Here the recursive references to node are not substituted out.

FIGURE 5.5 : TYPE GRAPH WITH CYCLE



(a) Type graph containing a cycle



(b) Equivalent acyclic graph

SEMANTIC ANALYSIS USING S-ATTRIBUTES

ALGORITHM FOR STRUCTURAL EQUIVALENCE

The following function sequiv can be applied for checking structural equivalence of two type expressions (or type graphs).

- **s and t are the input type expressions (or roots of the associated type graphs, in which case the graphs must be acyclic)**

```
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 × s2 and t = t1 × 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
```

- **Rewrite function sequiv, so that it can be used for detecting structural equivalence of general type graphs.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

WRITING A TYPE CHECKER

We use the material on types studied so far to implement a type checker for a sample language given below. The purpose is to demonstrate the working of a type checker only (and hence no code is generated).

$$
\begin{aligned}
P &\rightarrow D \ ; S \\
D &\rightarrow D \ ; \ D \quad \texttt{id} \ : \ T \\
T &\rightarrow \texttt{char} \quad \texttt{integer} \quad \texttt{boolean} \\
&\qquad \texttt{array [num] of} \ T \quad \uparrow T \\
S &\rightarrow \texttt{id} := \quad E \quad \texttt{if} \ E \ \texttt{then} \ S \\
&\qquad \texttt{while} \ E \ \texttt{do} \ S \quad S \ ; S \\
E &\rightarrow \texttt{literal} \quad \texttt{num} \quad \texttt{id} \quad E \ \texttt{mod} \ E \\
&\qquad E \ [E] \quad E \ \texttt{binop} \ E \quad E \uparrow
\end{aligned}
$$

- **The first rule indicates that the type of each identifier must be declared before being referenced**
- **The first task is to construct type expressions for each name and code which does that is given against the relevant rules. In case type graphs are used, these actions need to be re-coded appropriately.**
- **`id`.*entry* is a sythesized attribute that gives the symbol table entry for `id`.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

WRITING A TYPE CHECKER

- The procedure enter has the same meaning as given earlier. ***T.type*** is a synthesized attribute that gives the type expression associated with T.

$D \rightarrow$ `id` : T	{ ***enter(***`id`.***entry, T.type)*** }
$T \rightarrow$ `char`	{ ***T.type*** := char }
$T \rightarrow$ `integer`	{ ***T.type*** := integer }
$T \rightarrow$ `boolean`	{ ***T.type*** := boolean }
$T \rightarrow \uparrow T_1$	{ ***T.type*** := ***pointer*** (T_1.***type***) }
$T \rightarrow$ `array [num] of` T_1	{ ***T.type*** := ***array*** (1..`num`.***val***,T_1.***type***) }

TYPE CHECKING FOR EXPRESSIONS

Once the type expression for each declared data item has been constructed, semantic actions for type checking of expressions can be written, as explained below.

- ***E.type*** is a synthesized attribute that holds the type expression assigned to the expression generated by E. The function ***lookup***(e) returns the type expression saved in the symbol table for e.
- The semantic code given below assigns either the correct type to E or the special basic type, ***type_error***.