STATIC SEMANTICS
&
INTERMEDIATE CODE GENERATION

TEACHING MATERIAL

BASIC CONCEPTS AND ISSUES

WHAT IS STATIC SEMANTIC ANALYSIS?

Static semantic analysis ensures that a program is correct, in that it conforms to certain extra-syntactic rules.

The analysis is called

Static: because it can be done by examining the program text, and

Semantic: because the properties analysed are beyond syntax, i.e. they cannot be captured by context free grammars.

Examples of such analyses are:

a. type analysis,

b. name and scope analysis,

c. declaration processing,

and the techniques used for static semantic analysis can also be used for

d. intermediate code generation.

BASIC CONCEPTS AND ISSUES

TYPE ANALYSIS

The type of an object should match that expected by its context.

Examples of issues involved in type analysis are:

1. Is the variable on the left hand side of an assignment statement 'compatible' with the expression on the right hand side?

2. Is the number of actual parameters and their types in a procedure call compatible with the formal parameters of the procedure declaration?

3. Are the operands of a built-in operator of the right type? If not, can they be forced (coerced) to be so? What is the resultant operator, then, to be interpreted as?

   For example, the expression $3 + 4.5$ can be interpreted as:

   a. Integer addition of 3 and 4 (4.5 truncated).

   b. Real addition of 3.0 (3 coerced to real) and 4.5.

Note: The notion of 'type compatibility' is non-trivial. For example, in the declaration

```
type ptr = ^ object;
var a :  ^ object; b :  ptr;
```

are `a` and `b` type compatible?

BASIC CONCEPTS AND ISSUES

NAME AND SCOPE ANALYSIS

The issues here are:

1. What names are visible at a program point? Which declarations are associated with these names?
2. Conversely, from which regions of a program is a name in a declaration visible?

EXAMPLES

1. In Pascal, the following is not allowed:

```
begin
   ...
   go to 10;
   ...
   while E do
      begin
      ...
      10:  ...
      end
   ...
end;
```

Here the label 10 is the name. It is only visible in the block in which it is defined.

2.

```
program S;
  var a,x:  _
    procedure R;
      var i:  _
        procedure T;
          var m,n:  _
          body of T;
      body of R;
    procedure E;
      body of E;
    procedure Q;
      var k,v:  _
        function P(y,z:_):_
          var i,j:  _
          body of P;
      body of Q;
  body of S.
```

a. **The names visible from the body of** `P` **are** `y, z, i, j, k, v, a, x, P, Q, E, R and S.`

b. **The procedure** `P` **is visible from** `P` **itself and** `Q`**.**

BASIC CONCEPTS AND ISSUES

DECLARATION PROCESSING

Declaration processing involves:

1. ***Uniqueness checks:*** An identifier must be declared uniquely in an declaration.

2. ***Symbol Table updation:*** As declarations provide most of the information regarding the attributes of a name, this information must be recorded in a data structure called the symbol table. The main issues here are:

   a. What information must be entered into the symbol table, and how should such information be represented?

   b. What should be the organization of the symbol table itself.

3. ***Address Resolution:*** As declarations are processed, the addresses of variables may also be computed. This address is in the form of an offset relative to the beginning of the storage allocated for a block (refer to the module on runtime environments for more details).

BASIC CONCEPTS AND ISSUES

INTERMEDIATE CODE GENERATION

The front end translates the program into an intermediate representation, from which the back end generates the target code. There is a choice of intermediate representations such as abstract syntax trees, postfix code and three address code.

EXAMPLE

The three address code generated for the assignment statement `x := A[y,z] < b`, where `A` is a 10×20 array of integers, and `b` and `x` are reals is:

```
1.  t1 := y * 20
2.  t1 := t1 + z
3.  t2 := addr(A) - 21
4.  t3 := t2[t1]
5.  t4 := intoreal(t3)
6.  if t4 < b goto 9
7.  t5 := 0
8.  goto 10
9.  t5 := 1
10.  x:= t5
```

SYNTAX DIRECTED ANALYSIS

Can we always represent the properties stated above through a context free grammar? There are theoretical results to show that the answer is in the negative.

1. **The language** $\{wcw|\ w\ is\ a\ string\ from\ some\ alphabet\ \Sigma\}$ **can be thought of as an abstraction of languages with a single identifier declaration (the first occurrence of** w**), which must be declared before use (the second occurrence of** w**). This language is not context free.**

2. **The language** $\{a^n b^m c^n d^m | n \geq 1\ and\ m \geq 1\}$ **is an abstraction of languages with two procedures and their corresponding calls, which require that the number of formal parameters (represented by the** a**s and** b**s) be the same as the number of actual parameters (**c**s and** d**s). This language is not context free.**

However, we would like to do static analysis in a ***syntax directed*** **fashion, i.e. along with the process of parsing.**

SYNTAX DIRECTED ANALYSIS

SYNTAX DIRECTED DEFINITION

Associate with each terminal and non-terminal a set of values called *attributes*. The evaluation of attributes takes place during parsing., i.e. when we use a production of the form $A \to XYZ$ for derivation or reduction, the attributes of X, Y and Z could be used to calculate the attributes of A.

EXAMPLE: For the grammar shown below, suppose we wanted to generate intermediate code:

$$S \to id := E$$
$$E \to E_1 + E_2$$
$$E \to E_1 * E_2$$
$$E \to -E_1$$
$$E \to id$$

It is convenient to have the following attributes:

$S.code$ – **The intermediate code associated with S.**
$E.code$ – **The intermediate code associated with E.**
$E.place$ – **The temporary variable which holds the value of E.**
$id.place$ – **The lexeme corresponding to the token id.**

SYNTAX DIRECTED ANALYSIS

SYNTAX DIRECTED DEFINITION: EXAMPLE

The grammar is augmented with the following semantic rules:

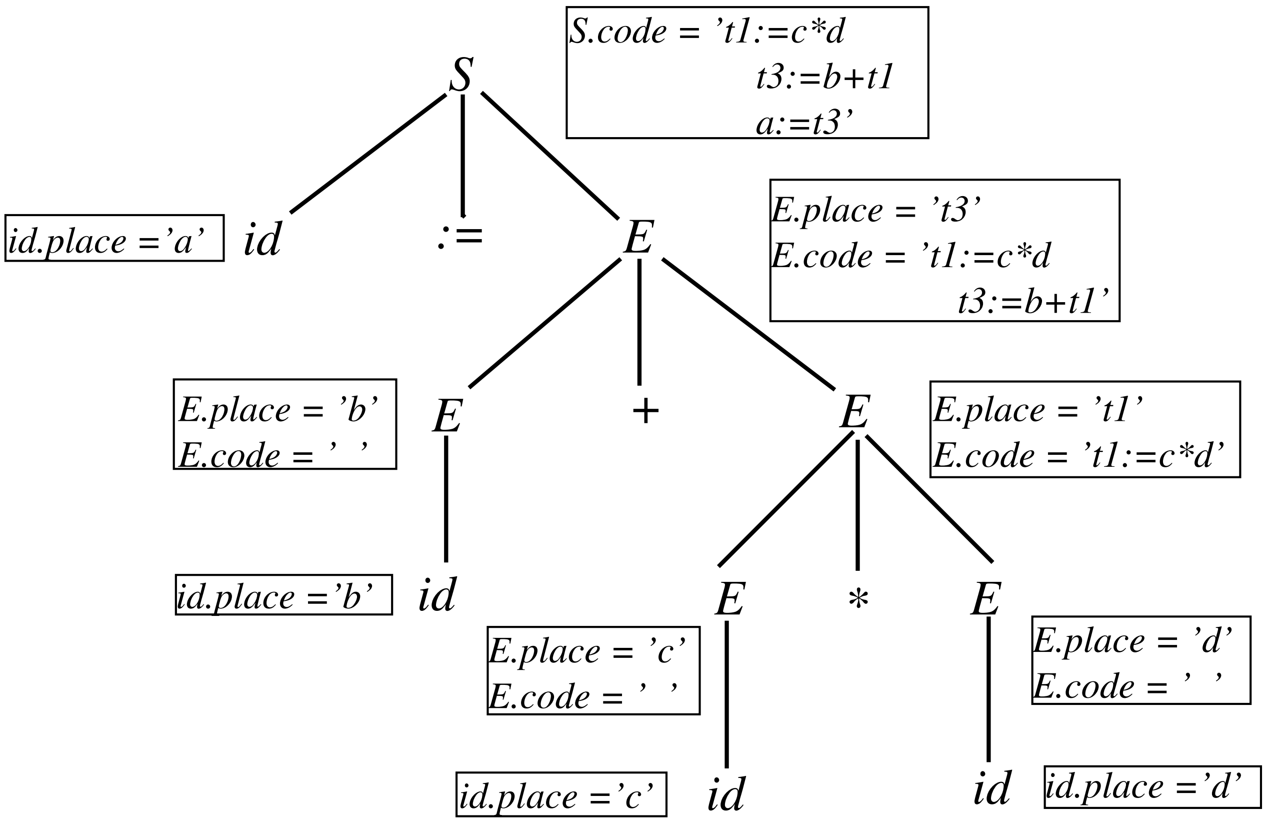
$$
\begin{array}{ll}
S \rightarrow id := E & S.code := E.code \parallel gen(id.place\ ':='\ E.place); \\
E \rightarrow E_1 + E_2 & E.place := newtemp; \\
 & E.code\ := E_1.code \parallel E_2.code \parallel gen(E.place \\
 & \qquad\quad ':='\ \ E_1.place\ '+'\ \ E_2.place); \\
E \rightarrow E_1 * E_2 & E.place := newtemp; \\
 & E.code\ := E_1.code \parallel E_2.code \parallel gen(E.place \\
 & \qquad\quad ':='\ \ E_1.place\ '*'\ \ E_2.place); \\
E \rightarrow -E_1 & E.place := newtemp; \\
 & E.code\ := E_1.code \parallel gen(E.place\ ':='\ \ 'uminus' \\
 & \qquad\quad E_1.place); \\
E \rightarrow id & E.place := id.place; \\
 & E.code\ :=\ '\ '; \\
\end{array}
$$

where

1. *newtemp* **is a function which generates a new temporary variable name, each time it is invoked.**
2. **'$\parallel$' is the concatenation operator.**
3. *gen* **evaluates its non-quoted arguments, concatenates the arguments in order, and returns the resulting string.**

SYNTAX DIRECTED ANALYSIS

The attributes of the various nonterminals for the statement `a := b + c * d` **is**



NOTE:

1. **Unlike the example, the attribute evaluation should proceed *along with the construction of the parse tree.* This would introduce restrictions on the possible relations between attribute values.**

2. **Nothing has been said about the implementational details of attribute evaluation.**

3. **The attributes of terminals are usually supplied by the lexical analyser.**

SYNTAX DIRECTED ANALYSIS

SYNTAX DIRECTED DEFINITION: FORMALIZATION

A syntax directed definition is an augmented context free grammar, where each production $A \to \alpha$, is associated with a set of semantic rules of the form $b := f(c_1, c_2, \ldots, c_k)$, where f is a function, and

1. b is a ***synthesized attribute*** of A and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of α, or

2. b is an ***inherited attribute*** of one of the grammar symbols α, and $c_1, c_2, \ldots, c_k$ are attributes belonging to A or α.

In either case, we say that the attribute b ***depends*** on the attributes $c_1, c_2, \ldots, c_k$.

In the previous example, $S.code$, $E.code$, and $E.place$ were all synthesized attributes.

SYNTAX DIRECTED ANALYSIS

WHY INHERITED ATTRIBUTES?

1. **For a variable declaration**

$$
\begin{aligned}
&var\_dec \rightarrow T : L \\
&L \rightarrow L_1, id \\
&L \rightarrow id \\
&T \rightarrow integer \\
&T \rightarrow real
\end{aligned}
$$

if we wanted to record the type of each variable, the semantic rules would be:

$$
\begin{array}{ll}
var\_dec \rightarrow T : L & \{\ L.type := T.type\} \\
L \rightarrow L_1, id & \{L_1.type := L.type, \\
 & id.type := L.type\} \\
L \rightarrow id & \{id.type := L.type\} \\
T \rightarrow integer & \{T.type := integer\} \\
T \rightarrow real & \{T.type := real\}
\end{array}
$$

Here $L.type$ **and** $id.type$ **are inherited attributes.**

2. Consider a (simplified) procedure declaration

$$p\_decl \rightarrow p\_name\ var\_decl\ body$$

Suppose we wanted to find the list of variables accessible (environment) inside the body of a procedure. This will be the list of variables accessible just outside the procedure declaration, augmented by the variables declared locally in the procedure itself. In such a case, we can have the following attributes:

$p\_decl.env$ –	**environment just outside the procedure declaration.**
$body.env$ –	**environment in the body.**
$var\_decl.list$ –	**list of local variables declared**

Clearly, the required syntax directed definition is:

$$p\_decl \rightarrow p\_name\ var\_decl\ body \qquad \{body.env := p\_decl.env \cup var\_decl.list\}$$

Again, $body.env$ **is an inherited attribute.**

SYNTAX DIRECTED ANALYSIS

TRANSLATION SCHEMES

A *translation scheme* is a refinement of a syntax directed definition in which:

(i) Semantic rules are replaced by *actions*.

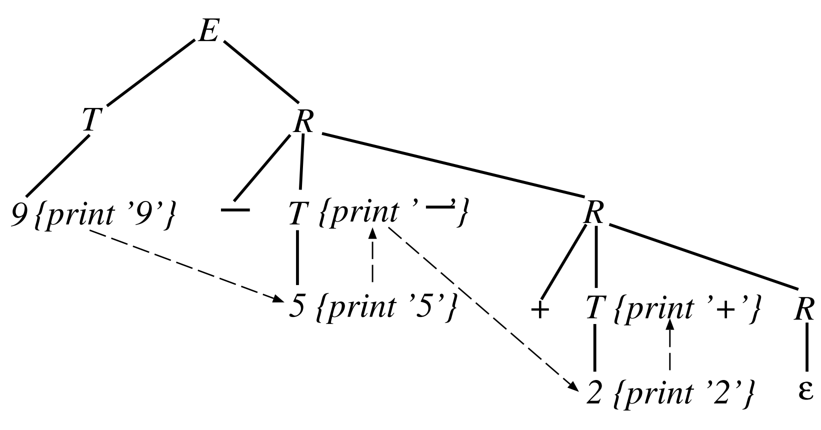
(ii) The exact point in time when the actions are to be executed is specified. This is done by embedding the actions within the right hand side of productions.

In the translation scheme $A \to X_1 \ldots X_i, \{action\} X_{i+1} \ldots X_n$, *action* **is executed after completion of the parse for** X_i**.**

EXAMPLE A translation scheme to map infix expressions into postfix expressions:

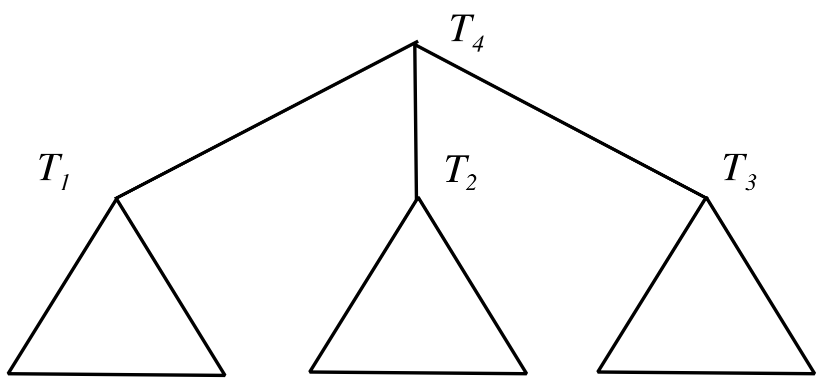
$$E \to TR$$
$$R \to addop\ T\ \{print(addop.lexeme)\}\ R_1 \mid \epsilon$$
$$T \to num\ \{print(num.val)\}$$

The order of execution of actions for the string `9-5+2` **is shown by dotted arrows:**



RESTRICTIONS ON ATTRIBUTE DEPENDENCES

Certain restrictions must be placed on the dependences between attributes, so that they can be evaluated along with parsing.
It is interesting to note that top-down parsers or bottom-up parsers would construct the parse tree shown in the figure in the order T_1, T_2, T_3, T_4.



Clearly an attribute of T_2 cannot depend on any attribute of T_3.

The extent of restriction gives rise to two classes of syntax directed definitions,

a. S-attributed definition.

b. L-attributed definition.

S-ATTRIBUTED DEFINITIONS

A translation scheme is *S-attributed* if:

(i) Every non-terminal has synthesized attributes only.

(ii) All actions occur on the right hand side of productions.

The syntax directed definition shown below is an example of an S-attributed definition:

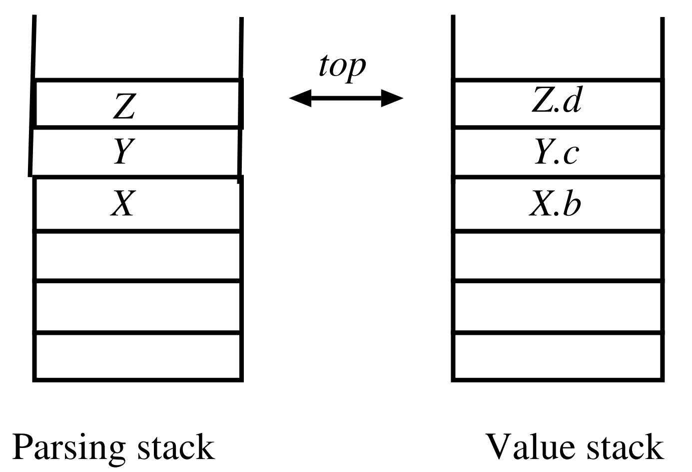
$$
\begin{array}{ll}
S \to id := E & S.code := E.code \parallel gen(id.place\ ' :=' \ E.place); \\
E \to E_1 + E_2 & E.place := newtemp; \\
 & E.code \ := E_1.code \parallel E_2.code \parallel gen(E.place \\
 & \qquad\qquad ' :=' \ E_1.place\ ' +' \ E_2.place); \\
E \to E_1 * E_2 & E.place := newtemp; \\
 & E.code \ := E_1.code \parallel E_2.code \parallel gen(E.place \\
 & \qquad\qquad ' :=' \ E_1.place\ ' *' \ E_2.place); \\
E \to -E_1 & E.place := newtemp; \\
 & E.code \ := E_1.code \parallel gen(E.place\ ' :=' \ 'uminus' \\
 & \qquad\qquad E_1.place); \\
E \to id & E.place := id.place; \\
 & E.code \ := \ ' \ ';
\end{array}
$$

S-ATTRIBUTED DEFINITIONS

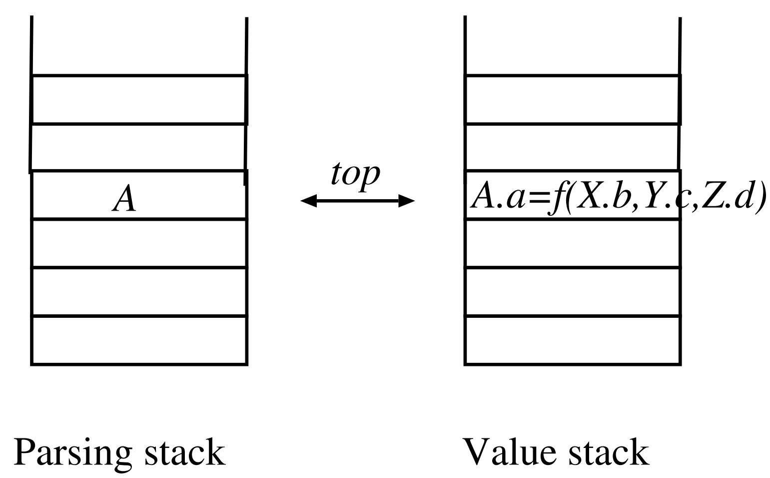
IMPLEMENTATION OF S-ATTRIBUTED DEFINITION

Assume a bottom up parsing strategy. We augment the parsing stack with a value stack (val). If location i in the parsing stack contains a grammar symbol A. then $val[i]$ will contain all the attributes of A.

Prior to a reduction using a production $A \to XYZ$:



If an attribute a of A is defined as $f(X.b, Y.c, Z.d)$, then, after the reduction:



S-ATTRIBUTED DEFINITIONS

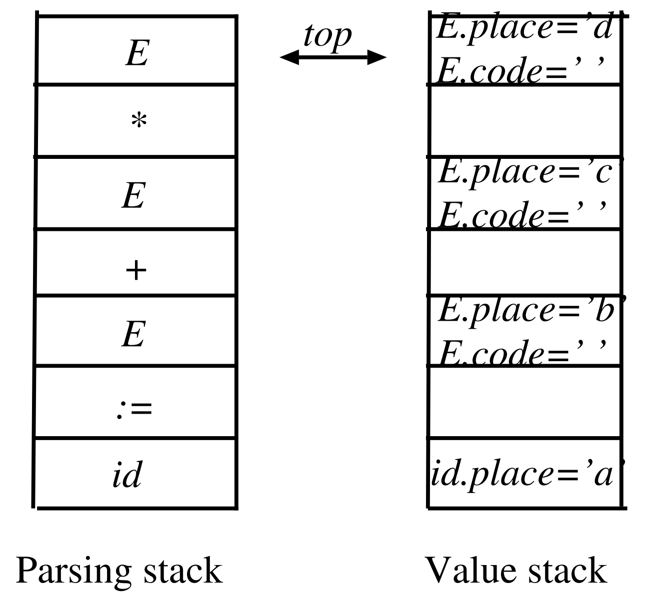
We can now replace the semantic rules by actions which refer to the actual data structures used to hold and manipulate the attributes.

Let each element of val be a record with two fields $val[i].code$ and $val[i].place$. Then the syntax directed definition is;

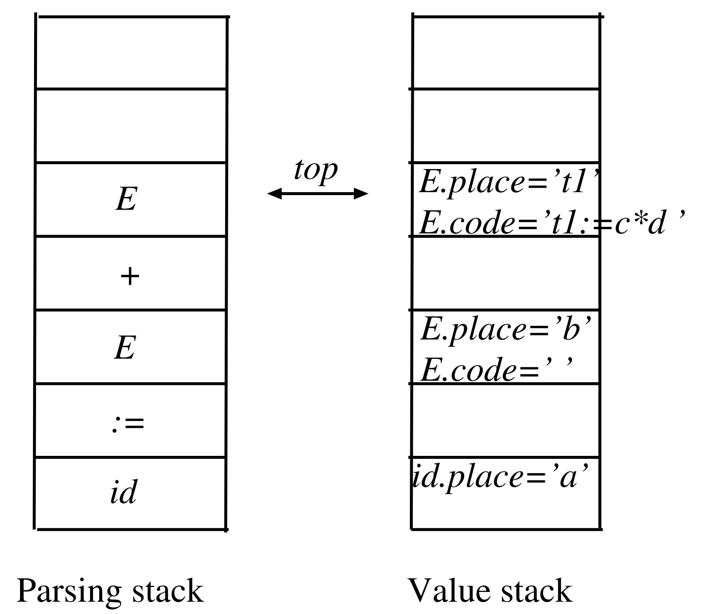
$$
\begin{array}{ll}
S \rightarrow id := E & val[top-2].code := val[top].code \parallel gen(val[top-2].place \\
 & \qquad\qquad ' :=' \; val[top].place); \\
E \rightarrow E_1 + E_2 & T := newtemp; \\
 & val[top-2].code \;:=\; val[top-2].code \parallel val[top].code \parallel \\
 & \qquad gen(T \; ' :=' val[top-2].place \; '+' \\
 & \qquad\qquad val[top].place); \\
 & val[top-2].place := T; \\
E \rightarrow E_1 * E_2 & T := newtemp; \\
 & val[top-2].code \;:=\; val[top-2].code \parallel val[top].code \parallel \\
 & \qquad gen(T \; ' :=' val[top-2].place \; '*' \\
 & \qquad\qquad val[top].place); \\
 & val[top-2].place := T; \\
E \rightarrow -E_1 & T := newtemp; \\
 & val[top-1].code \;:=\; val[top].code \parallel \\
 & \qquad gen(T \; ' :=' \;\; 'uminus' \; val[top].place); \\
 & val[top-2].place := T; \\
E \rightarrow id & val[top].place := val[top].place; \\
 & val[top].code \;:=\; ' \; '; \\
\end{array}
$$

S-ATTRIBUTED DEFINITIONS

EXAMPLE Before reduction using the production $E \rightarrow E_1 * E_2$.



After reduction using $E \rightarrow E_1 * E_2$.



S-ATTRIBUTED DEFINITIONS

Carefully distinguish between

1. ***Syntax directed definition*** **A context free grammar augmented with semantic rules. No assumptions are made regarding when the semantic rules are to be evaluated.**

2. ***Attribute grammar*** **A syntax directed definition in which the functions used in the semantic rules are free of side effects.**

3. ***Translation scheme*** **A syntax directed definition with actions instead of semantic rules. Actions are embedded within the right hand side of productions, their positions indicating the time of their evaluation during a parse.**

4. ***Implementation of a syntax directed definition*** **A translation scheme in which the actions are described in terms of actual data structures (e.g. value stack) instead of attribute symbols.**

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.

$$
\begin{array}{ll}
P \rightarrow M D & \\
M \rightarrow \epsilon & \{\ \boldsymbol{offset} := 0\ \} \\
D \rightarrow D ; D & \\
D \rightarrow \texttt{id} : T & \{\ \boldsymbol{enter}(\texttt{id.}\boldsymbol{name}, \boldsymbol{T.type}, \boldsymbol{T.width}, \boldsymbol{offset}); \\
 & \quad \boldsymbol{offset} := \boldsymbol{offset} + \boldsymbol{T.width}\ \} \\
T \rightarrow \texttt{integer} & \{\ \boldsymbol{T.type} := \texttt{integer}; \\
 & \quad \boldsymbol{T.width} := 4\ \} \\
T \rightarrow \texttt{real} & \{\ \boldsymbol{T.type} := \texttt{real}; \\
 & \quad \boldsymbol{T.width} := 8\ \}
\end{array}
$$

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 `num.`*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 *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 `id`s, 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.

$$
\begin{array}{ll}
P \rightarrow D & \\
D \rightarrow D ; D & \\
D \rightarrow L : T & \{\ \boldsymbol{enter}(\boldsymbol{L.list}, \boldsymbol{T.type}, \boldsymbol{T.width})\ \} \\
L \rightarrow \texttt{id},\ L_1 & \{\ \boldsymbol{L.list} := \boldsymbol{append}(L_1.\boldsymbol{list}, \texttt{id}.\boldsymbol{name})\ \} \\
L \rightarrow \texttt{id} & \{\ \boldsymbol{L.list} := \boldsymbol{makelist}(\texttt{id}.\boldsymbol{name})\ \} \\
T \rightarrow \texttt{integer} & \{\ \boldsymbol{T.type} := \texttt{integer}; \\
 & \boldsymbol{T.width} := 4\ \} \\
T \rightarrow \texttt{real} & \{\ \boldsymbol{T.type} := \texttt{real}; \\
 & \boldsymbol{T.width} := 8\ \} \\
T \rightarrow \texttt{array [num] of}\ T_1 & \\
 & \{\ \text{same as earlier}\ \} \\
T \rightarrow \uparrow T_1 & \{\ \text{same as earlier}\ \}
\end{array}
$$

SEMANTIC ANALYSIS USING S-ATTRIBUTES

PROCESSING OF DECLARATIONS

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

$$
\begin{aligned}
D &\rightarrow D; D \mid T\ L \\
L &\rightarrow \texttt{id}\ ,\ \mathbf{L}\ \text{—}\ \texttt{id} \\
T &\rightarrow \texttt{integer} \mid \texttt{real} \mid \ldots
\end{aligned}
$$

- **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**

$$
\begin{array}{ll}
D \rightarrow D; D & \\
D \rightarrow D_1\ ,\ \texttt{id} & \{\ \textit{enter}(\texttt{id}.\textit{name}, D_1.\textit{syn})\ ; \\
 & D.\textit{syn} := D_1.\textit{syn}\ \} \\
D \rightarrow T\ \texttt{id} & \{\ \textit{enter}(\texttt{id}.\textit{name}, T.\textit{type})\ ; \\
 & D.\textit{syn} := T.\textit{type}\ \} \\
T \rightarrow \texttt{integer} \mid \texttt{real} \mid \ldots & \\
 & \{\ \text{same as earlier}\ \}
\end{array}
$$

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

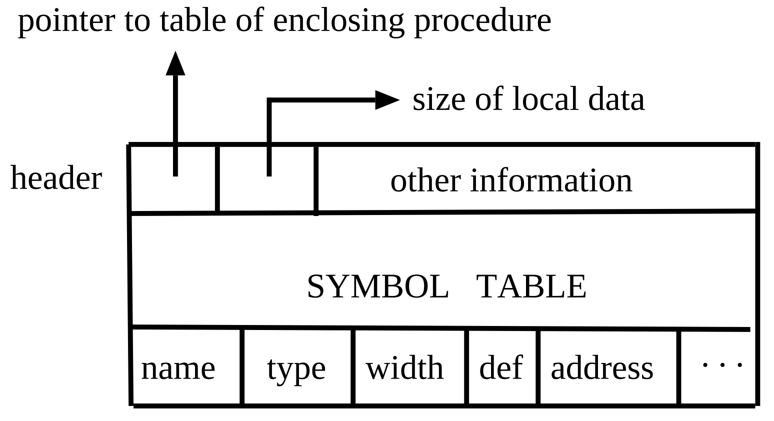
SCOPE INFORMATION

- 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.

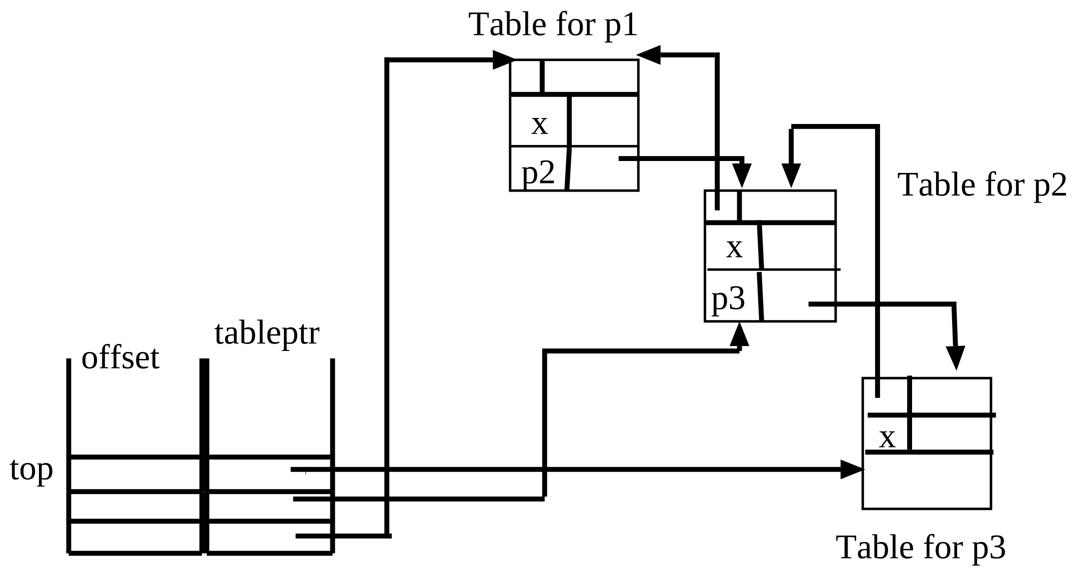
FIGURE 5.1 : Symbol Table Organization for Nested Structures

```
procedure  p1 ;
      var  x : real ;
      procedure  p2 ;
          var  x : integer ;
          procedure  p3 ;
  ------>     var  x : integer ;
              body  of  p3 ;
          body  of  p2 ;
    body of  p1 ;
```

(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.

$$
\begin{array}{ll}
P \rightarrow M D & \{\ \boldsymbol{addwidth}(\boldsymbol{top}(\boldsymbol{tableptr}),\boldsymbol{top}(\boldsymbol{offset}))\ ;\ \} \\
 & \boldsymbol{pop}(\boldsymbol{tableptr})\ ;\ \boldsymbol{pop}(\boldsymbol{offset})\ \} \\
M \rightarrow \epsilon & \{\ \boldsymbol{p} := \boldsymbol{mktable}(nil)\ ; \\
 & \boldsymbol{push}(\boldsymbol{p},\boldsymbol{tableptr})\ ;\ \boldsymbol{push}(0,\boldsymbol{offset}); \\
 & nest := 1;\ \} \\
D \rightarrow D ; D & \\
D \rightarrow \texttt{id} : T & \{\ \boldsymbol{enter}(\boldsymbol{top}(\boldsymbol{tableptr}),\texttt{id}.\boldsymbol{name},\boldsymbol{T.type},\boldsymbol{T.width}); \\
 & \boldsymbol{top}(\boldsymbol{offset}) := \boldsymbol{top}(\boldsymbol{offset}) + \boldsymbol{T.width}\ \}
\end{array}
$$

SEMANTIC ANALYSIS USING S-ATTRIBUTES

SCOPE INFORMATION

$D \rightarrow$ `proc` `id` ; N D ; S	
	{ *p := top(tableptr); addwidth(p,top(offset))* ;
	***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` $\{$ $\boldsymbol{T.type} :=$ `real`;
$\boldsymbol{T.width} := 8$ $\}$

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

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

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

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.

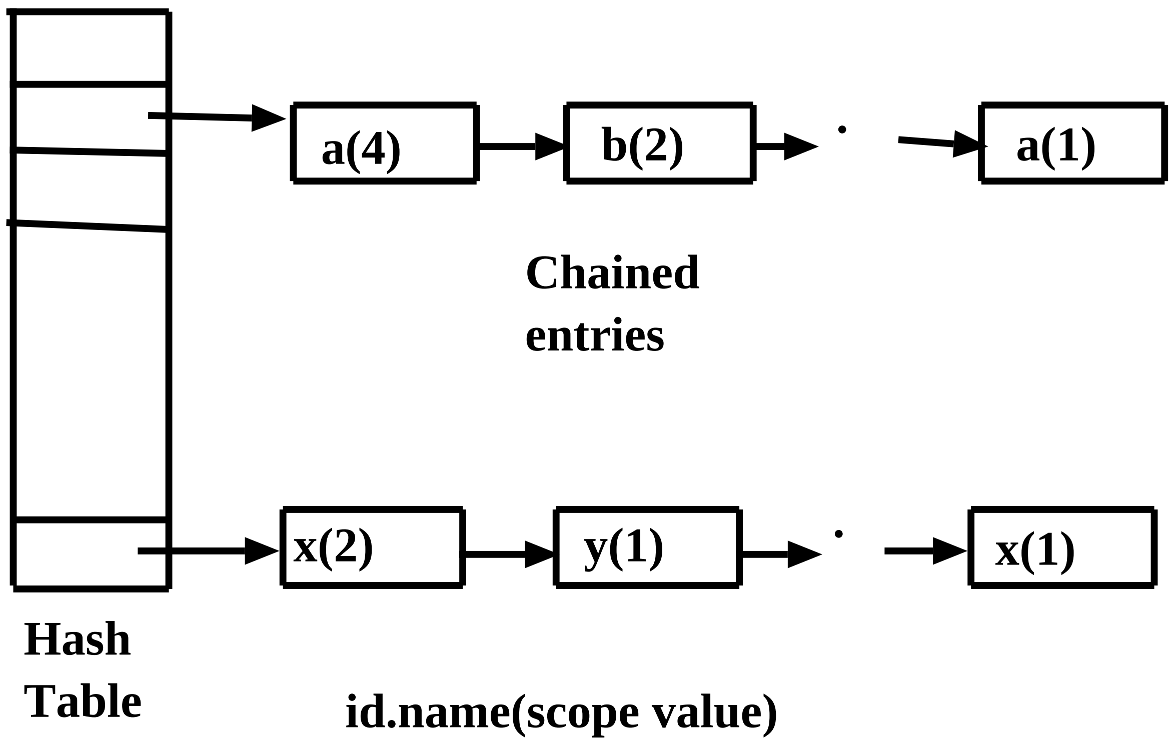
SEMANTIC ANALYSIS USING S-ATTRIBUTES

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



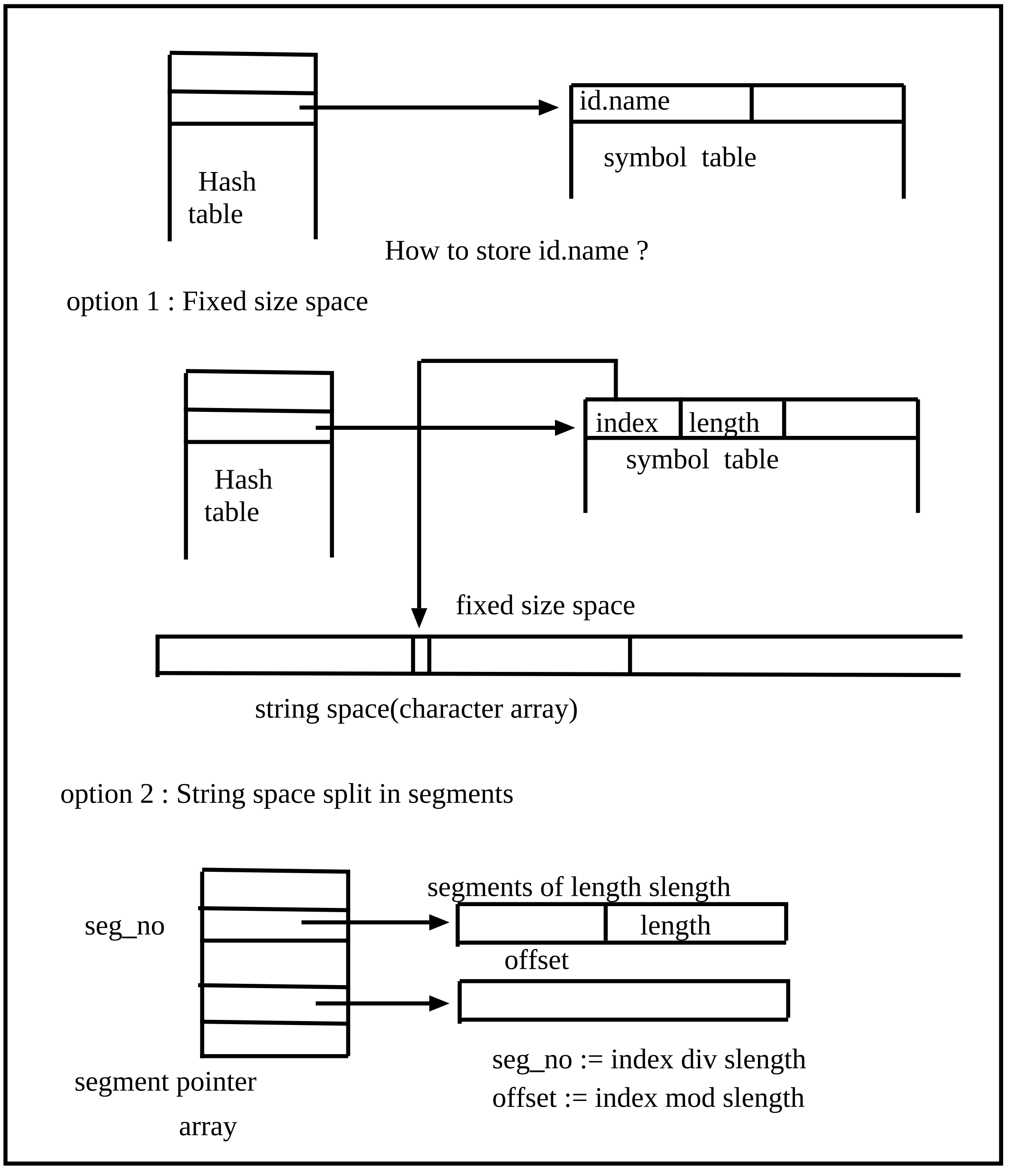
SEMANTIC ANALYSIS USING S-ATTRIBUTES

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**

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

- **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×array(1..32, char)) × (salary×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 : ↑ entry, is added to that given above, then the type expression for ptr is :**

pointer(record((token×array(1..32, char)) × (salary×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.**

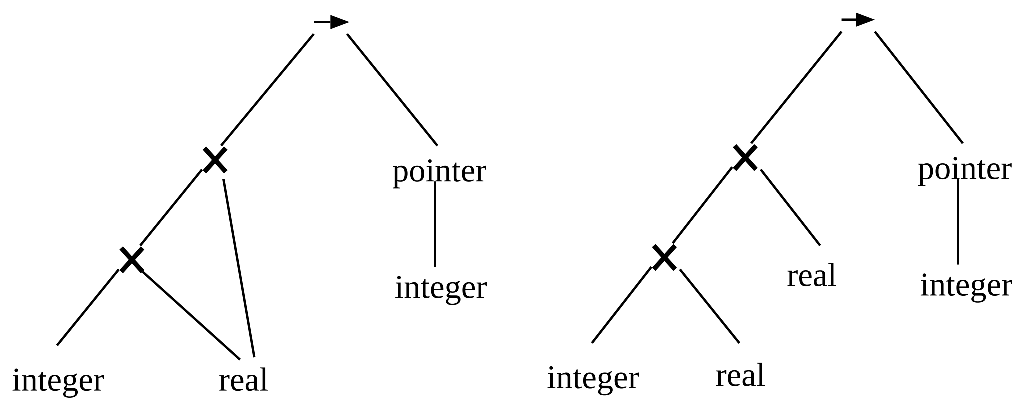
- **How to use the type expression ?**
  The type expression is a linear form of representation that can conveniently capture structure of a type. This expression may also be represented in the form of a dag (directed acyclic graph) or tree.

Figure 5.4 : TYPE EXPRESSIONS AND TYPE GRAPHS

For the declaration , $f(a{:}integer;\ b,c{:}\ real) : \uparrow integer ;$

Type expression for function f is

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



DAG form of type graph

Tree type graph

- **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.

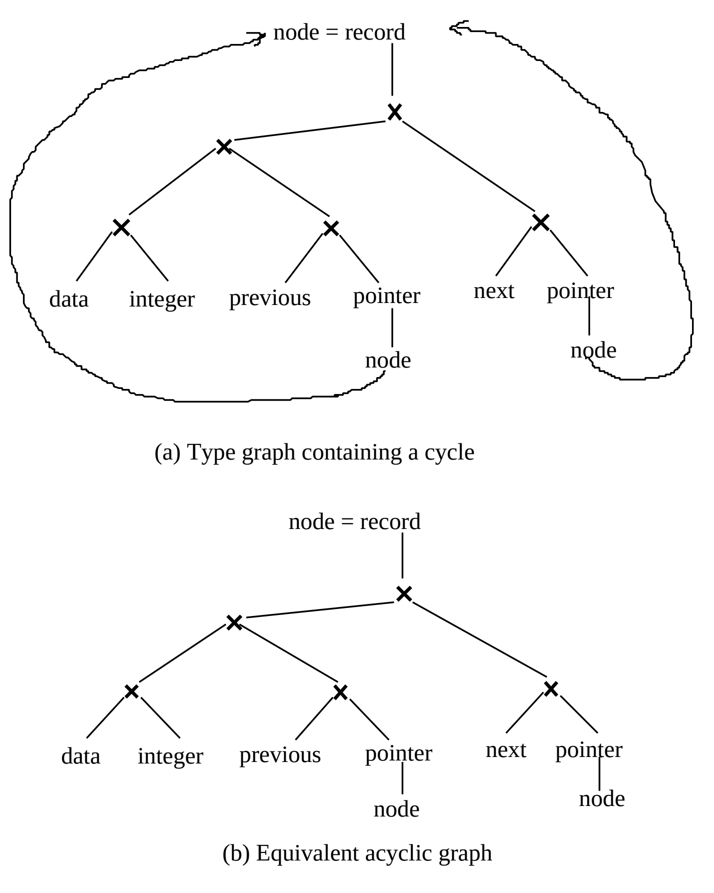
- The complexity of the task is clearly dependent on the structure of the type graphs involved. When the type graph is a tree or a dag, a simpler algorithm is possible. For type graphs containing cycles, such an algorithm is nontrivial.

- 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



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 $s = \textbf{\textit{array}}(s_1, s_2)$ **and** $t = \textbf{\textit{array}}(t_1, t_2)$ then
        return $\textbf{\textit{sequiv}}(s_1, t_1)$ and $\textbf{\textit{sequiv}}(s_2, t_2)$
    else if $s = s_1 \times s_2$ **and** $t = t_1 \times t_2$ then
        return $\textbf{\textit{sequiv}}(s_1, t_1)$ and $\textbf{\textit{sequiv}}(s_2, t_2)$
    else if $s = \textbf{\textit{pointer}}(s_1)$ and $t = \textbf{\textit{pointer}}(t_1)$ then
        return $\textbf{\textit{sequiv}}(s_1, t_1)$
    else if $s = s_1 \rightarrow s_2$ and $t = t_1 \rightarrow t_2$ then
        return $\textbf{\textit{sequiv}}(s_1, t_1)$ and $\textbf{\textit{sequiv}}(s_2, t_2)$
    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).

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

- **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.

$$
\begin{array}{ll}
D \rightarrow \texttt{id : } T & \{ \ \textbf{\textit{enter}}(\texttt{id}.\textbf{\textit{entry}},\ \textbf{\textit{T.type}}) \ \} \\
T \rightarrow \texttt{char} & \{ \ \textbf{\textit{T.type}} := \textbf{char} \ \} \\
T \rightarrow \texttt{integer} & \{ \ \textbf{\textit{T.type}} := \textbf{integer} \ \} \\
T \rightarrow \texttt{boolean} & \{ \ \textbf{\textit{T.type}} := \textbf{boolean} \ \} \\
T \rightarrow \uparrow T_1 & \{ \ \textbf{\textit{T.type}} := \textbf{\textit{pointer}}\ (T_1.\textbf{\textit{type}}) \ \} \\
T \rightarrow \texttt{array [num] of } T_1 & \\
 & \{ \ \textbf{\textit{T.type}} := \textbf{\textit{array}}\ (1..\texttt{num}.\textbf{\textit{val}}, T_1.\textbf{\textit{type}}) \ \}
\end{array}
$$

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***.

WRITING A TYPE CHECKER

- **The type checker can be enhanced to provide details of the semantic error and its location.**

$E \rightarrow \texttt{literal}$ $\quad \{ \ \boldsymbol{E.type} := \textbf{char} \ \}$

$E \rightarrow \texttt{num}$ $\quad \{ \ \boldsymbol{E.type} := \textbf{integer} \ \}$

$E \rightarrow \texttt{id}$ $\quad \{ \ \boldsymbol{E.type} := \boldsymbol{lookup} \ (\texttt{id}.\boldsymbol{entry}) \ \}$

$E \rightarrow E_1 \ \texttt{mod} \ E_2$

$\quad \{ \ \boldsymbol{E.type} := \texttt{if } E_1.\ \boldsymbol{type} = \textbf{integer} \texttt{ and}$

$\quad\quad \boldsymbol{sequiv} \ (E_1.type, E_2.type)$

$\quad \texttt{then } \textbf{integer} \texttt{ else } \boldsymbol{type\_error} \ \}$

$E \rightarrow E_1 \ \texttt{binop} \ E_2$

$\quad \{ \ \boldsymbol{E.type} := \texttt{if } \boldsymbol{sequiv} \ (E_1.type, E_2.type)$

$\quad \texttt{then } E_1.type \texttt{ else } \boldsymbol{type\_error} \ \}$

$E \rightarrow E_1[E_2]$

$\quad \{ \ \boldsymbol{E.type} := \texttt{if } E_2.\ \boldsymbol{type} = \textbf{integer} \texttt{ and}$

$\quad\quad E_1.type = \boldsymbol{array(s,t)}$

$\quad \texttt{then } t \texttt{ else } \boldsymbol{type\_error} \ \}$

$E \rightarrow E_1 \uparrow$ $\quad \{ \ \texttt{if } E_1.type = \boldsymbol{pointer(t)}$

$\quad \texttt{then } t \texttt{ else } \boldsymbol{type\_error} \ \}$

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TYPE CHECKING FOR STATEMENTS

Semantic code for type checking of the statement related rules of the sample grammar are given in the following.

- ***S.type*** **is a synthesized attribute that is assigned either type** ***void*** **or** ***type_error*****, depending upon whether there was a type error detected within it or not**
- **The semantic action in the last rule propagates type errors in a sequence of statements**

$S \rightarrow \texttt{id} := E$ { $\boldsymbol{S.type} :=$ `if` $\boldsymbol{sequiv}$ ($\texttt{id}.\boldsymbol{type}$,$\boldsymbol{E.type}$)
`then` $\boldsymbol{void}$ `else` $\boldsymbol{type\_error}$ }

$S \rightarrow \texttt{if } E \texttt{ then } S_1$
{ $\boldsymbol{S.type} :=$ `if` $\boldsymbol{E.type} =$ **boolean**
`then` $S_1.\boldsymbol{type}$ `else` $\boldsymbol{type\_error}$ }

$S \rightarrow \texttt{while } E \texttt{ do } S_1$
{ $\boldsymbol{S.type} :=$ `if` $\boldsymbol{E.type} =$ **boolean**
`then` $S_1.\boldsymbol{type}$ `else` $\boldsymbol{type\_error}$ }

$S \rightarrow S_1; S_2$ { $\boldsymbol{S.type} :=$ `if` $S_1.\boldsymbol{type} = \boldsymbol{void}$ `and`
$S_2.\boldsymbol{type} = \boldsymbol{void}$
`then` $\boldsymbol{void}$ `else` $\boldsymbol{type\_error}$ }

- **Combining all the grammar rules and the semantic actions listed against them gives a type checker for the sample language.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

WRITING A TYPE CHECKER

- In the type checker given above, it is assumed that for all operators, the types of the operands are compatible.
- This assumption is too restrictive and not followed in practice. For example, a ***op*** i , where a is a real and i is an integer, is a legal expression in most languages.
- The compiler is supposed to first convert one of the operands to that of the other and then translate the ***op***. Language definition specifies, for a given ***op***, the kind of conversions that are permitted. Conversion of integer to real is usually common.
- Semantic rules for performing type conversion for expressions and statements are included in the discussion on generation of intermediate code for these constructs.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

INTERMEDIATE CODE FORM

- **The front end typically outputs an explicit form of the source program for subsequent analysis by the back end. The semantic actions incorporated along with the grammar rules are responsible for emitting them**
- **Among the several existing intermediate code forms, we shall consider one that is known as *Three-Address Code*.**
- **In three address code, each statement contains 3 addresses, two for the operands and one for the result. The form is similar to assembly code and such statements may have a symbolic label.**
- **The commonly used three address statements are given below.**
  1. **Assignment statements of the form :**
     x := y *op* z, where *op* is binary, or
     x := *op* y, when *op* is unary
  2. **Copy statements of the form x := y**
  3. **Unconditional jumps, `goto L`. The three address code with label `L` is the next instruction where control flows.**

THREE ADDRESS CODE

4. Conditional jumps, such as

    `if x` ***relop*** `y goto` **L**
    `if x goto` **L**

- The first instruction applies a relational operator, ***relop***, to x and y and executes statement with label L, if the relation is true. Else the statement following `if x` ***relop*** `y goto` L is executed.
- The semantics of the second one is similar.

5. For procedure calls, this language provides the following :

    `param` x and `call` p,n

   `n` indicates the number of parameters in the call of p.

6. For handling arrays and indexed statements, it supports statements of the form :

   - x := y[i] and x[i] := y
   - The first is used to assign to x the value in the location that is i units beyond location y, where `x, y and i` refer to data objects.

THREE ADDRESS CODE

7. For pointer and address assignments, it has the statements
    - x := & y, x := * y and * x := y .
    - In the first form, y should be an object (perhaps an expression) that admits a l-value.
    - In the second one, y is a pointer whose r-value is a location.
    - The last one sets r-value of the object pointed to by x to the r-value of y.

- A three address code is an abstract form of intermediate code. It can be realised in several ways, one of them is called ***Quadruples***.
- A quadruple is a record structure with 4 fields, usually denoted by op, arg1, arg2 and result. The contents of these fields are pointers to the symbol table entries for the associated field names. For example, the quadruples for the expression `a + b * c` is

op	arg1	arg2	result
*	b	c	t_1
+	a	t_1	t_2

  where t_1 and t_2 are compiler generated temporaries.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ASSIGNMENT STATEMENTS

The semantic analysis and translation of assignment statements involve the following activities

(i) generating intermediate code for expressions (which are assumed to be free from type error) and the statement

(ii) generation of temporary names for holding the values of subexpressions during translation

(iii) handling of array references ; performing address calculations for array elements and associating them with the grammar rules

(iv) performing ***type conversion*** (or ***coercion***) operations while translating ***mixed mode*** expressions as specified by the underlying language.

Translation of Assignment Statement

- We start with a simple grammar and incrementally add features to it.

1. The first grammar, given below, excludes boolean expressions and array references.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ASSIGNMENT STATEMENTS

$$
\begin{array}{l}
P \rightarrow M\ D \\
M \rightarrow \epsilon \\
D \rightarrow D\ ;\ D \mid \texttt{id}\ :\ T \mid \texttt{proc id}\ ;\ N\ D\ ;\ S \\
N \rightarrow \epsilon \\
T \rightarrow \texttt{integer} \mid \texttt{real} \\
S \rightarrow \texttt{id := } E \\
E \rightarrow E + E \mid E * E \mid\ -E \mid (E) \mid \texttt{id}
\end{array}
$$

- **The rules for P, D and T give the declaration context in which an assignment statement is to be translated.**
- **Expressions of type real and integer only are considered. Ambiguous grammar for expressions have been chosen intentionally, in order to work with a small grammar. The semantic rules can be easily extended to an unambiguous version.**
- **Three address code is generated during semantic analysis. The basic scheme for evaluation of expressions is as follows For an expression of the form E_1 *op* E_2**

  i) code to evaluate E_1 into a temporary t_1

  ii) code to evaluate E_2 into a temporary t_2

  iii) $t_3 := t_1$ *op* t_2

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ASSIGNMENT STATEMENTS

- For compiler generated temporaries, we assume that a new temporary is created whenever required. This is achieved by means of a function *newtemp*() which returns a distinct name on succesive invocations.
- It is also assumed that temporaries are stored in the symbol table like any other user defined name.
- The following synthesized attributes and routines are defined.
  - `id`.*name* holds the lexeme for the name represented by `id`.
  - ***E.place*** holds the value of E.
  - Function *lookup*(`id`.*name*) gives an entry for `id`.*name* if it exists in the symbol table, otherwise returns `nil`.
  - Procedure *emit*(*statement*) appends the 3-address code *statement* to an output file.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ASSIGNMENT STATEMENTS

$S \rightarrow$ `id :=` E	**{** p := ***lookup*** (`id.`***name***); `if` p $\neq$ `nil then` ***emit*** (p ':=' ***E.place***) `else` **error /* undeclared** `id` ***/ }**
$E \rightarrow E_1 + E_2$	**{** ***E.place := newtemp***(); ***emit*** (***E.place*** ':=' E_1.***place*** '+' E_2.***place***) **}**
$E \rightarrow E_1 * E_2$	**{** ***E.place := newtemp***(); ***emit*** (***E.place*** ':=' E_1.***place*** '*' E_2.***place***) **}**
$E \rightarrow -E_1$	**{** ***E.place := newtemp***(); ***emit*** (***E.place*** ':=' '`uminus`' E_1.***place***) **}**
$E \rightarrow (E_1)$	**{** ***E.place*** := E_1.***place*** **}**
$E \rightarrow$ `id`	**{** p := ***lookup*** (`id.`***name***); `if` p $\neq$ `nil then` ***E.place*** := p `else` **error /* undeclared** `id` ***/ }**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ASSIGNMENT STATEMENTS

Example : The 3-address code generated for the expression

$$\mathbf{d} := -\ (\ \mathbf{a} + \mathbf{b}\ ) * \mathbf{c}$$

is given below.

We assume that all variables are of same type and bottom-up parsing is used.

Rule used	Attribute Evaluation	Output File
$E \rightarrow \texttt{id}$	$\boldsymbol{E.place} := \mathbf{a}$	——
$E \rightarrow \texttt{id}$	$\boldsymbol{E.place} := \mathbf{b}$	——
$E \rightarrow E_1 + E_2$	$\boldsymbol{E.place} := t_1$	$t_1 := \mathbf{a} + \mathbf{b}$
$E \rightarrow -E_1$	$\boldsymbol{E.place} := t_2$	$t_2 := \texttt{uminus}\ t_1$
$E \rightarrow E_1 * E_2$	$\boldsymbol{E.place} := t_3$	$t_3 := t_2 * \mathbf{c}$
$S \rightarrow \texttt{id} := E$	——	$\mathbf{d} := t_3$

TRANSLATION OF ASSIGNMENT STATEMENTS

2. ***Type Conversion***: In the translation given above, it is assumed that all the operands are of the same type. However that is not always the case.

- Languages usually support many types of variables and constants and permit certain operations on operands of mixed types. Most languages allow an expression of the kind a ***op*** b, where a or b may be integer or real and ***op*** is an arithmetic operation, such as $+$, $*$.
- For mixed mode expressions, a language specifies whether the operation is legal or not. If illegal, then the compiler has to indicate semantic error.
- For legal mixed mode operations, the compiler has to perform type conversion (or type coercion) and then generate appropriate code. ***Coercion is an implicit context dependent type conversion performed by a compiler.***
- To illustrate this concept, we use the earlier grammar where only real and integer types are permitted and convert integers to reals, wherever necessary.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ASSIGNMENT STATEMENTS

- **In addition to the attributes mentioned earlier, we use**
  - ***E.type*** **that holds the type of E, and**
  - **assume the existence of an unary operator** `int_to_real` **to perform the desired type conversion.**
- **The function *newtemp()* is now replaced by two functions, *newtemp_int()* and *newtemp_real()* which generate distinct temporaries of type integer and real respectively.**
- **The semantic actions for the rule $E \rightarrow E + E$ is given in the following. The semantics for the other rule $E \rightarrow E * E$ can be written analogously.**

  The operator + has been further qualified as `real +` **or** `integer +` **depending upon the type of operands it acts on.**
- **The semantic actions for the rule $S \rightarrow$ `id :=` E uses a function *type_of(p)* whose purpose is to access the symbol table for the entry p and return its type.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ASSIGNMENT STATEMENTS

$E \rightarrow E_1 + E_2$

{ if E_1.***type*** = integer and ***sequiv***(E_1.***type***,E_2.***type***)
then begin
    E.place := ***newtemp_int***();
    emit(***E.place*** ':=' E_1.***place*** 'int +' E_2.***place***);
    E.type := integer
end
else if E_1.***type*** = real and ***sequiv***(E_1.***type***,E_2.***type***)
then begin
    E.place := ***newtemp_real***();
    emit(***E.place*** ':=' E_1.***place*** 'real +' E_2.***place***);
    E.type := real
end
else if E_1.***type*** = real and E_2.***type*** = integer
then begin
    E.place := ***newtemp_real***();
    t := ***newtemp_real***();
    emit(t ':=' 'int_to_real ' E_2.***place***);
    emit(***E.place*** ':=' E_1.***place*** 'real +' t);
    E.type := real
end

(Continued)

TRANSLATION OF ASSIGNMENT STATEMENTS

```
        else if E1.type = integer and E2.type = real
        then begin
              E.place := newtemp_real ();
              t := newtemp_real ();
              emit( t ':=' 'int_to_real ' E1.place);
              emit(E.place ':=' t 'real +' E2.place);
              E.type := real
        end
        else E.type := type_error; }
S → id:= E
      { p := lookup (id.name) ;
      if p = nil then error /* undeclared id */
      else if sequiv (type_of (p),E.type)
           then if E.type = integer
                then  begin
                      emit ( p 'int:=' E.place) ;
                      S.type := void
                      end
                else  begin
                      emit ( p 'real:=' E.place) ;
                      S.type := void
                      end
```

(Continued)

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ASSIGNMENT STATEMENTS

```
        else if type_of (p) = real and E.type = integer
        then begin
              t := newtemp_real ();
              emit ( t ':=' 'int_to_real' E.place);
              emit ( p 'real:= ' t ) ;
              S.type := void;
              end
  else S.type := type_error; /* type mismatch */ }
```

Example : A program fragment, alongwith the 3-address code generated for it, is given below.

Program Fragment	Code Generated
var a, b, c : integer ;	t_1 := b int * c
var d : real ;	t_2 := int_to_real b
a := b * c + b * d ;	t_3 := t_2 real * d
	t_4 := int_to_real t_1
	t_5 := t_4 real + t_3
	type_error

- The ***type_error*** is indicated because a is of type integer while the type of rhs , i.e., t_5 is real.
- The temporaries used above are of different types. Only t_1 is of type integer and the rest are real.

TRANSLATION OF ARRAY REFERENCES

3. ***Compiling Array References*** **:**

We now augment the grammar to include array references as well. The basic issues are explained through an example.

Consider the following declaration and assignment statement.

```
a : array [ 1..10, 1..20] of integer
i, j, k : integer
i :=   ; j :=    ;
k := a[i+2, j-5] ;
```

- **In order to translate the array reference given above, we must know the address of this array element. For statically declared arrays, it is possible to compute the relative address of each element.**
  - **(i) Array is usually stored in contiguous locations; there are two possible layouts known as, row-major representation (used in Pascal) and column-major representation (used in Fortran).**

TRANSLATION OF ARRAY REFERENCES

The row-major form is shown in Figure 5.6.

1	2	3	4	5	6	7	8	9
A(1,1)	A(1,2)	A(1,3)	A(2,1)	A(2,2)	A(2,3)	A(3,1)	A(3,2)	A(3,3)

(ii) **Let *base* be the relative address of a[1,1]. Then the relative address of a[i_1, i_2] is given by**
base **+ (($i_1 - 1$) * 20 + i_2 - 1) * w , where w is the width of an array element.**
The above expression may be rewritten as

$$\mathbf{c + (20 * } i_1 + i_2 \mathbf{) * w , \text{ where } c =} \textbf{\textit{base}} - \mathbf{21 * w}$$

(iii) **The term c given above is a compilation time constant and can be precomputed and stored in the symbol table, while processing array declarations.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ARRAY REFERENCES

- Assuming c is available, the 3-address code generated for the statement k := a[i+2, j-5] ; is the following :

$$t_1 := \text{i} + 2$$
$$t_2 := t_1 * 20$$
$$t_3 := \text{j} - 5$$
$$t_4 := t_2 + t_3$$
$$t_5 := 4 * t_4 \quad \{ \text{ w} = 4 \text{ } \}$$
$$t_6 := \text{c}$$
$$t_7 := t_6[t_5]$$
$$\text{k} := t_7$$

- In the following we write a suitable grammar for array references and attach semantic actions to it.
- As mentioned above, the address computation has two parts, one is a compile-time constant and the other is dependent on the index expressions.
- Generalisation of the address computation formula for n-dimensional arrays is done as follows.
  Let the declaration be of the form : a$[u_1,\ u_2,\ \ldots,\ u_n]$, the lower bounds for each dimension is assumed to be 1.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ARRAY REFERENCES

Relative address of a[e_1,e_2, …,e_n] = **c+v, where**
c = ***base***-((…((u_2+1)*u_3 +1)* …)*u_n+1)***w; and**
v = ((…(($e_1 * u_2$+e_2)*$u_3 + e_3$)* …)*u_n+e_n)***w;**

- **In order to calculate v incrementally, the following observation is useful. Assume an array reference a**[e_1,e_2, …,e_n].

Expression seen	Code Generated
a[e_1	e_1 is evaluated in t_1
e_2	e_2 is evaluated in t_2
	$t_3 := t_1 * u_2$
	$t_3 := t_3 + t_2$
…	…
…	…
e_n	e_n is evaluated in t_{2n-2}
	$t_{2n-1} := t_{2n-3} * u_2$
	$t_{2n-1} := t_{2n-1} + t_{2n-2}$
	, n $\geq$ 2
]	t_{2n} := c

- **At this stage the array reference may be replaced by** $t_{2n}[t_{2n+1}]$. **Note that this scheme requires a large number of temporaries.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ARRAY REFERENCES

In view of the translation scheme outlined above, we write a grammar that exposes the index expressions one by one. The skeleton grammar is given below.

$$S \rightarrow L := E$$
$$E \rightarrow L \mid \text{other rules for expression}$$
$$L \rightarrow \textbf{\textit{elist}}\ ] \mid \texttt{id}$$
$$\textbf{\textit{elist}} \rightarrow \textbf{\textit{elist}}\ ,\ E \mid \texttt{id}\ [\ E$$

- The rules for L take care of array references in both the lhs and rhs of an assignment statement. The rules for the nonterminal ***elist*** are written in such a manner so as to match the intended translation scheme.
- The semantic actions that need to be taken against the grammar rules given above are explained below.
  1. The rule ***elist*** $\rightarrow$ `id` $[\ E$, indicates the array name and the first index expression. The location which holds the value of E must be noted in order to compute v.
  2. The rule $elist \rightarrow elist, E$ exposes the next index expression. The part of v that has been computed so far can be kept as an attribute of ***elist***.

TRANSLATION OF ARRAY REFERENCES

3. **Using the current value of E , the term corresponding to the dimension under consideration can now be added to v.**

   Since each term of the address computation formula requires the related upper bound, it is necessary to keep track of the dimension that corresponds to this occurence of E.

4. **The rule $L \rightarrow elist]$ indicates that an array reference has been completely seen. The two parts c and v of the address computation are available at this point and can be kept as attributes of L.**

5. **The rule $E \rightarrow L$ indicates that instance of an array reference (r-value) has been found. The appropriate code can be generated and location where this value is held can be kept as an attribute of E.**

6. **The rule $S \rightarrow L := E$ indicates an array reference whose l-value is required. Appropriate code may be generated since all the required information is available.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ARRAY REFERENCES

The semantic rules are given below.

$S \rightarrow L := E$

  { if ***L.offset*** = null
  then ***emit***(***L.place*** ':=' ***E.place***)
  else ***emit***(***L.place***'['***L.offset***']' ':=' ***E.place***) }

$E \rightarrow L$

  { if ***L.offset*** = null
  then ***E.place := L.place***
  else begin
    E.place := newtemp() ;
    emit(***E.place*** ':=' ***L.place***'['***L.offset***']')
   end }

$L \rightarrow$ ***elist***]

  { ***L.place := newtemp***() ;
  L.offset := newtemp() ;
  emit(***L.place*** ':=' ***c(elist.array)***);
  emit(***L.offset*** ':=' ***elist.place*** '*' ***width(elist.array)***) }

$L \rightarrow$ id

  {***L.place*** := id.***place***;
  L.offset := null }

TRANSLATION OF ARRAY REFERENCES

$elist \rightarrow elist_1,\ E$

$\{$ t := ***newtemp***() ;

m := $elist_1$.***dim*** + 1 ;

emit(t ':=' $elist_1$.***place*** '*' ***limit***($elist_1$.***array***));

emit(t ':=' t '+' ***E.place***);

$elist$.***array*** := $elist_1$.***array***;

$elist$.***place*** := t ;

$elist$.***dim*** := m $\}$

$elist \rightarrow$ `id` [E

$\{$ $elist$.***array*** := `id`.***place***;

$elist$.***place*** := ***E.place***;

$elist$.***dim*** := 1 $\}$

- **The attributes and functions used in above are :**
  - **(i) For L, *L.place* and *L.offset* are the attributes for holding c and v in case of arrays. In case of scalars, the second attribute has `null` value.**
  - **(ii) For E, the attribute *E.place* is used in the same sense as earlier.**

TRANSLATION OF ARRAY REFERENCES

(iii) For *elist*, the attributes used are *elist.array*, *elist.place* and *elist.dim*.

- Attribute *elist.array* is a pointer to the symbol table entry for the array name.
- Attribute *elist.place* is a placeholder for the v part of the address calculation performed so far.
- *elist.dim* is used to keep track of the dimension information.

(iv) The function *c(array_name)* returns the c part of the address formula from the symbol table entry for *array_name*.

(v) The function *width(array_name)* gives the width information from the symbol table.

(vi) The function *limit*(*array_name*,m) returns the upper bound for the dimension m from the symbol table.

TRANSLATION OF BOOLEAN EXPRESSIONS

4. ***Translation of Boolean Expressions*** :

   We wish to enrich our grammar by permitting boolean expressions in an assignment statement. The following rules are representative for our purpose.

$$
\begin{aligned}
B \rightarrow\ & B\ \texttt{or}\ B \mid B\ \texttt{and}\ B \\
& \mid \texttt{not}\ B \mid (\ B\ ) \\
& \mid \texttt{true} \mid \texttt{false} \\
& \mid E\ \texttt{relop}\ E \mid E \\
E \rightarrow\ & E + E \mid \text{other rules for arithmetic expressions}
\end{aligned}
$$

- The rules for B give the syntax of boolean expressions. For arithmetic expressions, the same rules for E are assumed.
- `relop` stands for the class of relational operators such as $<, \leq, =, \neq, >, \geq$.
- The common approach to translate boolean expressions (in the context of an assignment) is to encode `true` and `false` numerically and use the general translation scheme for arithmetic expressions.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF BOOLEAN EXPRESSIONS

We shall demonstrate the translation based on numerical encoding by using 1 for `true` and 0 for `false`.

- Semantic actions for the rule $B \rightarrow E_1$ `relop` E_2 is the only one which needs explaining.

  We generate a temporary, say t, for holding the value of the expression E_1 `relop` E_2. The value assigned to t is either 1 or 0 depending on whether the expression is true or false.

  In either case, t is carried forward in the translation of the rest of the expression.

- We assume that all three address statements are labeled numerically and that *nextstat* gives the label of the next 3-address statement.

  By using the unconditional and conditional branching 3-address statements and nextstat, the semantic actions for this rule can be written as shown below.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF BOOLEAN EXPRESSIONS

$B \rightarrow B_1$ `or`B_2

{ ***B.place := newtemp()***;

emit(B.***place*** ':=' B_1.***place*** '`or`' B_2.***place***) }

$B \rightarrow B_1$ `and` B_2

{ ***B.place := newtemp()***;

emit(B.***place*** ':=' B_1.***place*** '`and`' B_2.***place***) }

$B \rightarrow$ `not` B_1

{ ***B.place := newtemp()***;

emit(B.***place*** ':=' '`not`' B_1.***place***) }

$B \rightarrow (B_1)$

{ B.***place*** := B_1.***place*** }

$B \rightarrow$ `true`

{ ***B.place := newtemp()***;

emit(B.***place*** ':=' '1') }

$B \rightarrow$ `false`

{ ***B.place := newtemp()***;

emit(B.***place*** ':=' '0') }

$B \rightarrow E$

{ ***B.place := E.place*** }

(Continued)

TRANSLATION OF BOOLEAN EXPRESSIONS

$B \rightarrow E_1$ **relop** E_2

{ ***B.place := newtemp()***;

emit('if' E_1.***place*** 'relop.***op***' E_2.***place*** 'goto' ***nextstat***+3) ;

emit(B.***place*** ':=' '0') ;

emit('goto' ***nextstat***+2) ;

emit(B.***place*** ':=' '1') }

- Attribute ***op*** is used to distinguish the relational operator.
- The sequence of ***emit*** statements in the semantic actions for the last rule gives the rationale behind the usage of ***nextstat***+2 (or ***nextstat***+3).
- The method evaluates a boolean expression completely and is a safe implementation.
- The translation ignores issues of type checking and type coercion.

Translation of Assignment Statement : A Summary

- We have covered a wide range of issues dealing with translation of an assignment statement. However the treatment is not exhaustive and several details have been skipped for sake of clarity and brevity.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ASSIGNMENT STATEMENTS

A Summary (Continued)

- To be able to write a general grammar that includes expressions of all permissible types, such as boolean and arithmetic.
- Though the salient issues of translation have been exhibited with ambiguous grammars, it is desirable that unambiguous grammars that take care of precedence of operators be used.
- The type ckecking and type coercion part of semantic analysis need to be integrated and extended over all permissible types
- The translation of array references made assumptions about lower bounds, availability of c and type checking/coercion. The scheme needs to be elaborated by appropriate generalisations, and inclusion of the missing analyses.
- Once the design for the basic grammar is complete, other features may be added along the same lines. For instance, record accesses and function calls are common in expressions.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF ASSIGNMENT STATEMENTS

A Summary (Continued)

- **The approach to incorporate more features comprises of the following activities.**

  i) Add relevant rules to the existing grammar.

  ii) Decide on the 3-address code skeleton to be generated.

  iii) Identify type checking (and type coercion , if necessary) and insert semantic code to that effect. Function calls may require elaborate type checking of all arguments.

  iv) Integrate the grammar and semantic actions with the existing one.

- **Detection of semantic errors at various places have been indicated. More elaborate detection of such errors, issuing of appropriate error messages and possible strategies for recovery need to be worked out and incorporated.**

- **It should be fairly obvious that semantic analysis for expressions is a nontrivial task and comprises the bulk of semantic analysis for the entire language.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Translation and generation of intermediate code for typical control flow constructs is considered here. We use the following grammar.

$$
\begin{aligned}
S &\rightarrow \texttt{if}\ B\ \texttt{then}\ S \\
&\quad \mid \texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ S \\
&\quad \mid \texttt{while}\ B\ \texttt{do}\ S \\
&\quad \mid \texttt{begin}\ Sl\ \texttt{end} \\
&\quad \mid L := E \\
Sl &\rightarrow Sl\ ;\ S \\
&\quad \mid S \\
B &\rightarrow B_1\ \texttt{or}\ B_2 \mid B_1\ \texttt{and}\ B_2 \\
&\quad \mid \texttt{not}\ B_1 \mid (\ B_1\ ) \\
&\quad \mid \texttt{true} \mid \texttt{false} \\
&\quad \mid E_1\ \texttt{relop}\ E_2 \mid E \\
E &\rightarrow E_1 + E_2 \mid \textbf{other rules} \\
L &\rightarrow \textbf{rules for assignment statement}
\end{aligned}
$$

- **A comment about the grammar. It is capable of generating nested control flow statements (e.g.,nested if-then-else).**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF CONTROL FLOW CONSTRUCTS

- The rules for assignment statement are assumed without elaboration. Rule for a compound statement enclosed in `begin` and `end` is also included.

  The translation issues are introduced through an example.

- Example : Consider the boolean expression given below.

  a+b > c `and` flag

  where a,b and c are integer and flag is boolean.

  Equivalent 3-address code sequence is given below.

  10 : t_1 := a + b
  11 : `if` t_1 > c `goto` 14
  12 : t_2 := 0
  13 : `goto` 15
  14 : t_2 := 1
  15 : t_3 := t_2 `and` flag

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Assume that the expression a+b > c `and` flag is part of an if-then-else, as shown below.

```
ifa+b > c and flag then
  begin
     a := b + c ;
     flag := false
  end
else a := b - c ;
```

- **In order to generate code for above statement, it is required to append the following two statements after the code for expression evaluation.**

  The two labels enable the control to flow to the then and else parts respectively.

  16 : `if` t_3 `goto` ***label1***
  17 : `goto` ***label2***

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF CONTROL FLOW CONSTRUCTS

- **The statements that need to be generated for the then and else parts are given below. Note the inclusion of a jump statement after the then part, in order to prevent control flow to the else part.**

Code for Then part	Code for Else part
18 : t_4 **:= b + c**	**23 :** t_6 **:= b - c**
19 : a := t_4	**24 : a :=** t_6
20 : t_5 **:= 0**	**25 :**
21 : flag := t_5	
22 : `goto` ***label3***	

- **It is clear that *label1* should be 18 (or *nextstat* + 2) and may be filled in during generation of the statement 16.**

- ***label2* enables a jump to the begining of the else part, i.e., 22 in our example. Howvever when the statement 17 is being generated, it is not possible to know the value of this label. It clearly depends upon the number of statements in the then part.**

TRANSLATION OF CONTROL FLOW CONSTRUCTS

- Similarly, the value of *label3* should be 25, where the code for the statement immediately following the if-then-else, would be placed. Again its value depends on the number of statements in the else part and cannot be resolved while generating statement 22.

- The issues for translation of the if-then-else statement are

  1. to decide the code sequences for each component, the inclusion of a unconditional jump after the then part, for instance.
  2. find a scheme to resolve the forward labels of the jump statements, such as *label2* and *label3*.

Control Flow Translation of Boolean Expressions

- Instead of using the method of complete evaluation of a boolean expression, it is possible to use another method which is known as *partial evaluation*.

- In partial evaluation of boolean expressions, the value of the expression is not explicitly stored and it may not even be evaluated completely.

FLOW TRANSLATION OF BOOLEAN EXPRESSIONS

- **The essence of partial evaluation can be explained by considering the two 3-address code sequences given below.**

Complete Evaluation	Partial Evaluation
10 : t_1 := a + b	50 : t_1 := a + b
11 : if t_1 > c goto 14	51 : if t_1 > c goto 53
12 : t_2 := 0	52 : goto 60
13 : goto 15	53 : if flag goto 55
14 : t_2 := 1	54 : goto 60
15 : t_3 := t_2 and flag	55 : t_2 := b + c
16 : if t_3 goto 18	56 : a := t_2
17 : goto 23	57 : t_3 := 0
18 : t_4 := b + c	58 : flag := t_3
19 : a := t_4	59 : goto 62
20 : t_5 := 0	60 : t_4 := b − c
21 : flag := t_5	61 : a := t_4
22 : goto 25	62 :
23 : t_6 := b − c	
24 : a := t_6	
25 :	

FLOW TRANSLATION OF BOOLEAN EXPRESSIONS

- The code generated for partial evaluation does not use the temporaries t_2 and t_3 to hold the values of subexpressions as they have been used in the first column.
- Using the fact that the expression is of the form b_1 `and` b_2, it evaluates b_2 only when b_1 is true.
- The code in the second column shows that it needs less number of temporaries and is also shorter in length as compared to that in the first column.
- There are more jump statements in the code for partial evaluation. The number of forward resolutions of jump labels are also correspondingly higher in the second column.

  It may be noted that in case of complete evaluation the generated code for expression has exactly two unresolved jump labels.
- In order to translate using this method, the issues are

  i) to select code sequences to be generated for the subexpressions of the form e_1 `relop` e_2 ; b_1 `and` b_2, etc.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

FLOW TRANSLATION OF BOOLEAN EXPRESSIONS

ii) To resolve the target labels of jump statements. The context in which the expression occurs , e.g., if-then-else , while, etc. also play a role in this part.

- It may be noted that target labels of some jump statements may be resolved at generation time itself. Statement 51 in the second column is an example. The fact is that jump statements whose targets lie locally (within the code for the expression) can be resolved easily.

- Jump statements whose targets are beyond the expression, such as the begining of then or else part, cannot be resolved in this manner.

  - For example, after processing the expression a+b > c `and` flag, the statements numbered 50 to 54 are generated. However the target labels of statements 52 and 54 are not known at this stage.

  - Also the target of 53 is known at generation time only because of the context of the if-then-else statement, and may not be known in a different situation.

- The unresolved labels occuring in statements 52 and 54 have a common characteristic, the expression evaluates to false at these points in the code.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

FLOW TRANSLATION OF BOOLEAN EXPRESSIONS

- **A general method to resolve such labels would be to remember all the statement numbers where the expression evaluates to false. When the target becomes known later it can be inserted in all these statements.**
- **Why did we not need to remember all the statement numbers where the expression evaluates to true ?**

  This problem did not surface in our example. However, it can be easily seen that in general, we need to keep this information as well (try the same example, replacing `and` by `or`).

Semantic rules for Boolean Expressions

- **The grammar alongwith semantic rules for partial evaluation of Boolean expressions is given below.**
- **The synthesized attributes that have been used are described in the following.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

FLOW TRANSLATION OF BOOLEAN EXPRESSIONS

1. ***B.truelist*** and ***B.falselist*** for nonterminal B. These attributes are pointers to lists.

   In the code generated for B, the label field of certain statements are left incomplete. These statements contain jumps to the true and false exits of B.

   The list of such incomplete statements are held by ***B.truelist*** and ***B.falselist*** respectively.

2. ***M.stat*** for nonterminal M. This attribute is used to hold the index of the next 3-address statement at strategic points in the code. We use the term ***marker nonterminal*** for nonterminals which are used in this manner.

- The following functions and variables have also been used

  1. ***makelist***(i) - this function creates a list containing a single element, { i } ; returns a pointer to the list created. The element i is an index of a 3-address statement.

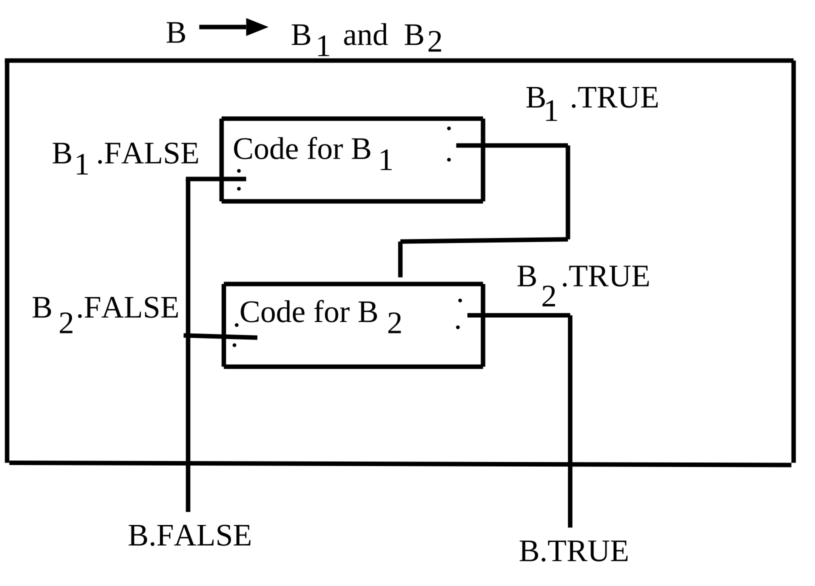
  2. ***merge***(p_1, p_2) - this function concatenates two lists pointed by p_1 and p_2 and returns a pointer to the merged list.

FLOW TRANSLATION OF BOOLEAN EXPRESSIONS

3. ***bachpatch***(p,i) - this function inserts i in the place for the missing target label in each element of the list (of 3-address statements) pointed to by p.

4. ***nextstat*** - a global variable that holds the index of the next 3-address statement.

- The scheme for linking the code sequences for the rule $B \rightarrow B_1$ `and` $M\ B_2$ is illlustrated in the figure.



SEMANTIC ANALYSIS USING S-ATTRIBUTES

FLOW TRANSLATION OF BOOLEAN EXPRESSIONS

The semantic rules are given below.

$B \rightarrow B_1$ `or` $M\ B_2$

$\{$ ***backpatch***$(B_1.falselist, M.stat);$
$B.truelist := merge(B_1.truelist, B_2.truelist)$;
$B.falselist := B_2.falselist$ $\}$

$B \rightarrow B_1$ `and` $M\ B_2$

$\{$ ***backpatch***$(B_1.truelist, M.stat);$
$B.truelist := B_2.truelist$
$B.falselist := merge(B_1.falselist, B_2.falselist)$ $\}$

$B \rightarrow$ `not` B_1

$\{$ $B.truelist := B_1.falselist$;
$B.falselist := B_1.truelist$ $\}$

$B \rightarrow (\ B_1\ )$

$\{$ $B.truelist := B_1.truelist$;
$B.falselist := B_1.falselist$ $\}$

$B \rightarrow$ `true`

$\{$ $B.truelist$:= ***makelist***(***nextstat***);
emit('`goto__`') $\}$

$B \rightarrow$ `false`

$\{$ $B.falselist$:= ***makelist***(***nextstat***);
emit('`goto___`') $\}$

SEMANTIC ANALYSIS USING S-ATTRIBUTES

FLOW TRANSLATION OF BOOLEAN EXPRESSIONS

$B \rightarrow E_1$ `relop` E_2

$\{$ $B.truelist$:= ***makelist***(***nextstat***)

$\{$ $B.falselist$:= ***makelist***(***nextstat*** + 1)

emit('if' E_1.***place*** '`relop`.***op***' E_2.***place*** '`goto___`') ;

emit('`goto___`') $\}$

$B \rightarrow E$

$\{$ $B.place := E.place$;

$\{$ $B.truelist$:= ***makelist***(***nextstat***)

$\{$ $B.falselist$:= ***makelist***(***nextstat*** + 1)

emit('if' $B.place$ '`goto___`') ;

emit('`goto___`') $\}$

$M \rightarrow \epsilon$ $\{$ ***M.stat*** := ***nextstat*** $\}$

$E \rightarrow E_1 + E_2$ | other rules

$\{$ same as given earlier $\}$

- **The method for partial evaluation manages to generate code in a single pass.**
- **It holds the list of incomplete statements and carries them around, along with parsing, till such time that till their target labels become available.**
- **The code generated is assuredly correct but may not be efficient.**

SEMANTIC ANALYSIS USING S-ATTRIBUTES

COMPLETE VS PARTIAL EVALUATION

- Boolean expressions may be evaluated by either of the two methods. However, the choice of translation does not rest with the implementor, it is a part of the language specifications.
- Partial evaluation is in some sense an efficient form of complete evaluation. However the two translations may not give identical results, particularly in the presence of side effects.
- Partial evaluation is typically suited for translation of boolean expressions which are part of control flow constructs. Complete evaluation is used in the context of assignment statements.

TRANSLATION OF CONTROL FLOW CONSTRUCTS

We now take up translation of flow of control statements, such as `if-then-else` and `while-do`.

- The boolean expression component of these statements are partially evaluated as explained above.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF CONTROL FLOW CONSTRUCTS

- For each control flow construct, the following need to be done.

  1. Determine from the context, how to resolve the true and false lists of the boolean expression.
  2. How to link the different components of the construct so that the translation is semantically equivalent ?

     For example, `if-then-else` statement has three components, boolean expression and two statement parts. These have to be connected so as to ensure the correct flow of control.
  3. 3-address statements, which have jumps to outside the body of the construct, cannot be resolved at generation time.
     These statements must be carried forward till the required target becomes known. Incomplete 3-address code may occur within the code generated for

     (i) boolean expression ,
     (ii) other components of the construct and
     (iii) 3-address jump statements inserted to link the various components.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF CONTROL FLOW CONSTRUCTS

We assume, for now, that the only kinds of jump allowed from within a control flow construct is to the statement following the construct (i.e., unconditional `goto` is absent in the source program).

- In addition to the attributes ***B.truelist, B.falselist, M.stat***; functions ***makelist(), merge() and backpatch()*** and variable ***nextstat*** the following are required.

(i) Nonterminal N , whose purpose is to prevent control flow from then part into the else part of an if-then-else statement.
It causes the generation of an incomplete unconditional 3-address jump statement.
The index of this statement is held in an attribute ***N.nextlist***.
The target jump is resolved by backpatching it with the index of the 3-address code following the if-then-else.

(ii) Indices of the incomplete 3-address statements generated for S are held in an attribute ***S.nextlist***.

(iii) Similarly ***Sl.nextlist*** holds a list of indices of all the incomplete 3-address code corresponding to the list of statements denoted by ***Sl***.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF CONTROL FLOW CONSTRUCTS

$S \rightarrow$ `if` B `then` $M\ S_1$

{ ***backpatch***($B.truelist, M.stat$) ;

S.nextlist := ***merge***($B.falselist, S_1.nextlist$) }

$S \rightarrow$ `if` B `then` $M_1\ S_1\ N$ `else` $M_2\ S_2$

{ ***backpatch***($B.truelist, M_1.stat$) ;

backpatch($B.falselist, M_2.stat$) ;

$S.next := merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist))$ }

$S \rightarrow$ `while` $M_1\ B$ `do` $M_2\ S_1$

{ ***backpatch***($B.truelist, M_2.stat$);

backpatch($S_1.nextlist, M_1.stat$) ;

$S.nextlist := B.falselist$;

emit('`goto`' $M_1.stat$) }

$S \rightarrow$ `begin` Sl `end`

{ $S.nextlist := Sl.nextlist$ }

$Sl \rightarrow Sl_1$; $M\ S$

{ ***backpatch***($Sl_1.nextlist, M.stat$) ;

$Sl.nextlist := S.nextlist$ }

$Sl \rightarrow S$ { $Sl.nextlist := S.nextlist$ }

$S \rightarrow L := E$ { **semantic rules given earlier** ;

S.nextlist := `nil` }

$M \rightarrow \epsilon$ { $M.stat$:= ***nextstat*** }

$N \rightarrow \epsilon$ { $N.nextlist$:= ***makelist(nextstat)***;

emit('`goto___`') }

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF CONTROL FLOW CONSTRUCTS

- **EXAMPLE : We illustrate the discussion with an example given below.**
- **The symbols $\downarrow_i$ are used to mark certain points of the program fragment (to illustrate generation of code) and are not part of the source program.**

```
begin
        if a > b  ↓1 then
            if b > c  ↓2 then
                begin
                    d := b + c ;
                    b := b / 2
                end  ↓3
            else b := 2 * b  ↓4
        else
            while a > 0  ↓5 do
            begin
                d := b * b ;
                a := a / 2
            end ↓6 ;
        b := b / 2  ↓7
end  ↓8
```

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF CONTROL FLOW CONSTRUCTS

MarkedPoint	3-address Code Generated
$\downarrow_1$	**10 :** `if` **a** $>$ **b** `goto___`
	11 : `goto___`
$\downarrow_2$	**12 :** `if` **b** $>$ **c** `goto___`
	13 : `goto___`
$\downarrow_3$	**14 :** t_1 **:= b + c**
	15 : d := t_1
	16 : t_2 **:= b / 2**
	17 : b := t_2
	18 : `goto___`
$\downarrow_4$	**19 :** t_3 **:= 2 * b**
	20 : b := t_3
	{ backpatch({12}, 14); backpatch({13},19) due to the inner if-then-else }
	21 : `goto___`
$\downarrow_5$	**22 :** `if` **a** $>$ **0** `goto___`
	23 : `goto___`

TRANSLATION OF CONTROL FLOW CONSTRUCTS

MarkedPoint	3-address Code Generated
$\downarrow_6$	**24 :** t_4 **:= b * b**
	25 : d := t_4
	26 : t_5 **:= a / 2**
	27 : a := t_5
	28 : `goto 22`
	{ backpatch({22},24) due to while }
	{ backpatch({10},12); backpatch({11},22)
	due to outer if-then-else }
$\downarrow_7$	**29 :** t_6 **:= b / 2**
	30 : b := t_6
	{ backpatch({18,21,23}, 29) due to sequencing }
$\downarrow_8$	**S.nextlist has the value `nil`, no unresolved**
	jump statements }

Translation of Other Constructs

The basic issues in semantic analysis and translation have been covered. However, a compiler has to deal with more details, some of which are listed below.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

TRANSLATION OF OTHER CONSTRUCTS

- Translation of `goto` statement. It is handled by storing labels in the symbol table, associating index of 3-address code with it and using backpatching to resolve forward references.
- Translation of procedure call statement. Let parameter passing be call by reference and storage be statically allocated.

  (a) Semantic analysis would involve type checking of the formal and actual arguments.
  (b) Translation would require generating 3-address code for evaluating each argument(in case it is an expression)i. Then generate a list of `param` 3-address statements, one for each argument, followed by the 3-address statement `call`.
- Translation of other constructs such as `case` statement, `for` statement and `repeat` - `until` statement. The method discussed earlier can be extended, in principle, for these constructs as well.
- We have ignored semantic analysis for checking structural integrity of single-entry control structures. Code for testing such violations may be included.

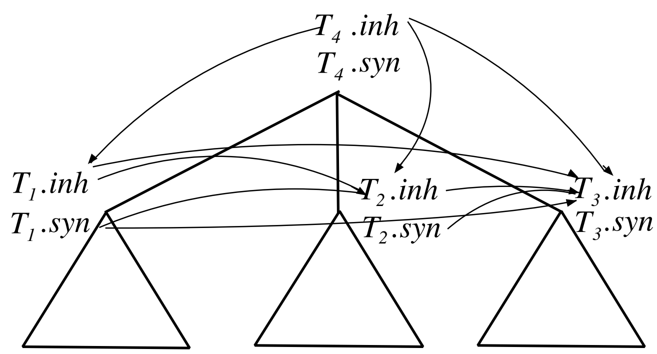
L-ATTRIBUTED DEFINITIONS

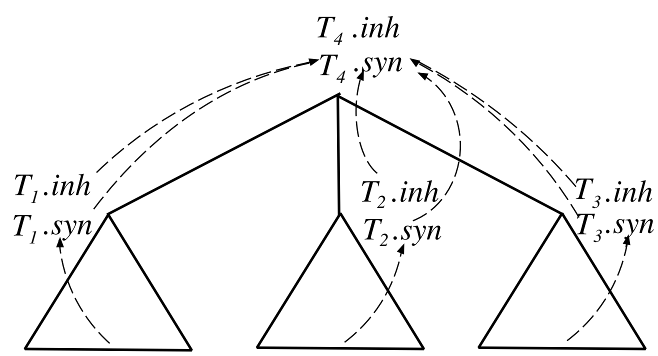
Since S-attributed definitions do not permit inherited attributes, they are too restrictive. Therefore L-attributed definitions are used.

DEFINITION: A syntax directed definition is L-attributed, if each inherited attribute of X_j, $1 \leq j \leq n$, on the right hand side of $A \to X_1 X_2 \ldots X_n$, depends only on

1. **The attributes of the symbols $X_1, \ldots, X_{j-1}$.**
2. **The inherited attributes of A.**

The flow of attribute information in a L-attributed definition is shown below:





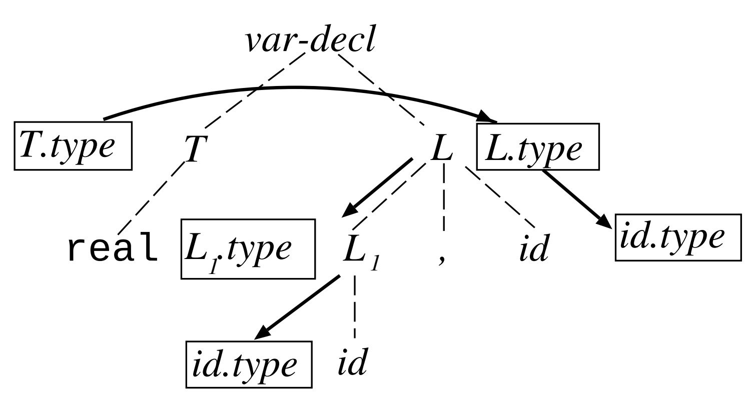
NOTE: Every S-attributed definition is also a L-attributed definition.

L-ATTRIBUTED DEFINITIONS

DEPENDENCE GRAPH

The dependences among the attributes (inherited and synthesized) of the nodes of a given parse tree is represented by a graph called the *dependence graph*.
This graph has a node for each attribute of each node of the parse tree. In addition, there is an edge from the node for c to b, if the attribute b depends on c.
The dependency graph for the syntax directed definition and the input string `a,b: real` is shown below:

$var\_dec \rightarrow T : L$	$\{\ L.type := T.type\}$
$L \rightarrow L_1, id$	$\{L_1.type := L.type$
	$id.type := L.type\}$
$L \rightarrow id$	$\{id.type := L.type\}$
$T \rightarrow integer$	$\{T.type := integer\}$
$T \rightarrow real$	$\{T.type := real\}$



In the diagram above, the dotted lines represent edges of the parse tree, whereas the continuous lines denote dependence edges.

L-ATTRIBUTED DEFINITIONS

RECURSIVE DESCENT WITH L-ATTR. EVALUATION

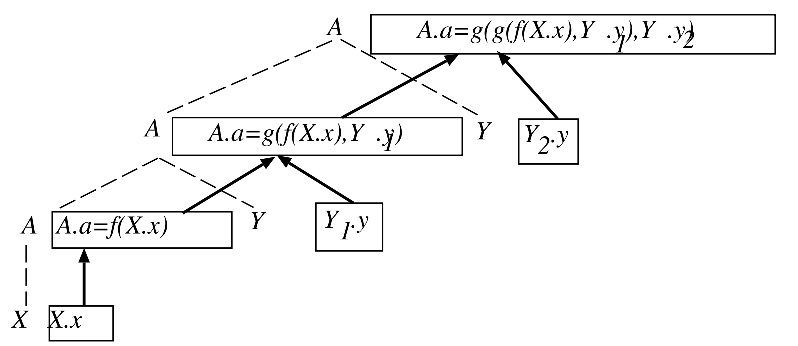
If we have a translation scheme whose underlying grammar involves left recursion, what happens to the actions when we transform the grammar to remove left recursion?

There is a procedure to transform the actions. But the procedure works provided the original translation scheme has synthesized attributes only.

A generic translation scheme involving a grammar with left recursion:

$$
\begin{array}{ll}
A \rightarrow A_1 Y & \{\ A.a := g(A_1.a, Y.y)\} \\
A \rightarrow X & \{A.a := f(X.x)\}
\end{array}
$$

The dependency graph for the string XYY would be:



L-ATTRIBUTED DEFINITIONS

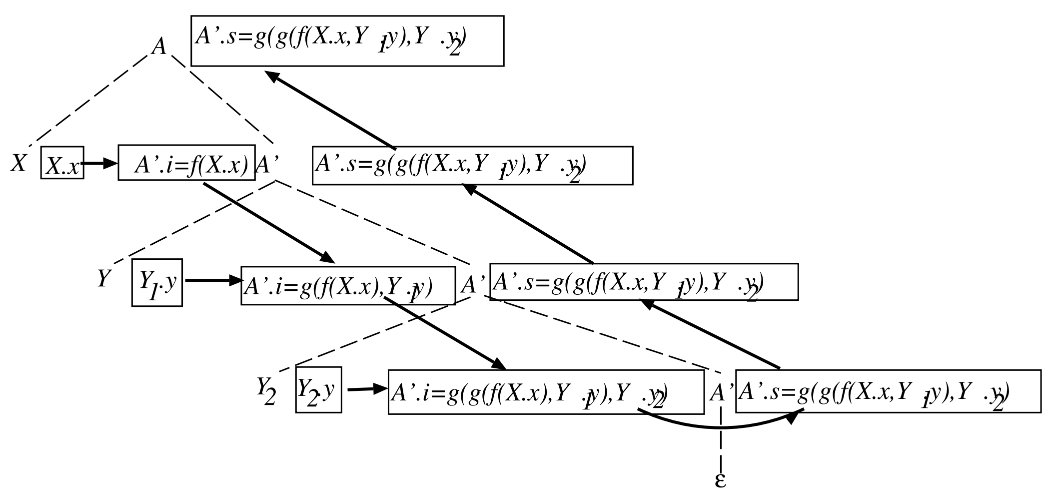
After removing left recursion, the grammar becomes:

$$A \rightarrow XA'$$
$$A' \rightarrow YA' \mid \epsilon$$

Transform the actions in the manner shown:

$$A \rightarrow X\ \{A'.i := f(X.x)\}\ A'\ \{A.a := A'.s\}$$
$$A' \rightarrow Y\ \{A'_1.i := g(A'.i, Y.y)\}\ A'_1\ \{A'.s := A'_1.s\}$$
$$A' \rightarrow \epsilon\ \{A'.s := A'.i\}$$

The new dependency graph, which calculates the same attribute at the root is:



L-ATTRIBUTED DEFINITIONS

EXAMPLE

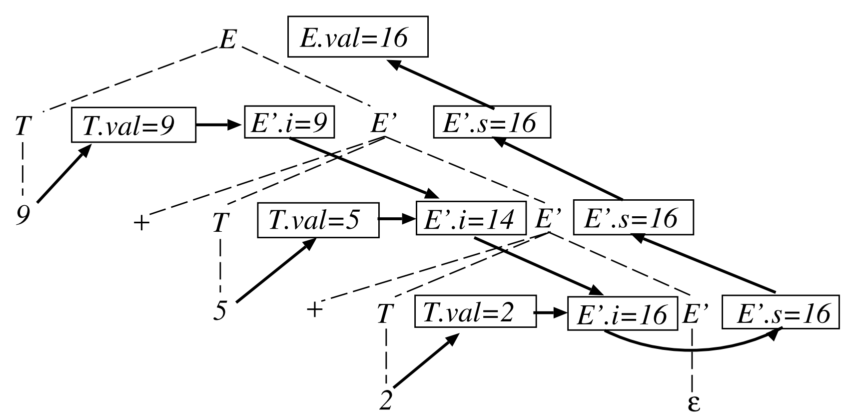
The translation scheme

$$
\begin{array}{ll}
E \to E_1 + T & \{\ E.val := E_1.val + T.val\} \\
E \to T & \{E.val := T.val\} \\
T \to num & \{T.val := num.val\}
\end{array}
$$

is transformed into

$$
\begin{array}{l}
E \to T\ \{E'.i := T.val\}\ E'\ \{E.val := E'.s\} \\
E' \to +\ T\ \{E'_1.i := E'.i + T.val\}\ E'_1\ \{E'.s := E'_1.s\} \\
E' \to \epsilon\ \{E'.s := E'.i\} \\
T \to num\ \{T.val := num.val\}
\end{array}
$$

For the input string $9 + 5 + 2$, the dependency graph will be



L-ATTRIBUTED DEFINITIONS

RECURSIVE DESCENT WITH L-ATTR EVALUATION

ALGORITHM This algorithm takes as input a translation scheme, whose underlying grammar is suitable for predictive parsing, and produces code for a predictive parser.

1. For each nonterminal A, construct a function that has

   a. A formal parameter for each inherited attribute of A.

   b. Returns the synthesized attributes of A as a record.

   c. Has a local variable for each attribute of each grammar symbol occurring in any production of A.

2. The body of the function will depend on the productions with A on the left hand side. Examine the production $A \rightarrow X_1 \ldots X_n$. For each X_i do the following:

   a. If X_i is a token with synthesized attribute x, save the value of x in the variable for $X_i.x$. Then generate a call to match X_i and advance the input pointer.

   b. If X_i is a non-terminal, generate an assignment statement $c := X_i(b_1, \ldots, b_k)$, where $b_1, \ldots, b_k$ are the variables for the inherited attributes of X_i and c is the variable for the synthesized attribute of X_i.

   c. For an action, copy the code into the parser, replacing each reference to an attribute by the variable for the attribute.

L-ATTRIBUTED DEFINITIONS

EXAMPLE

The code for

$$E' \rightarrow +T \ \{E'_1.i := E'.i + T.val\} \ E'_1 \ \{E'.s := E'_1.s\}$$
$$E' \rightarrow \epsilon \ \{E'.s := E'.i\}$$

is

```
function E_PRIME(e_prime_i: integer): integer;
    var t_val, e1_prime_i, e1_prime_s: integer;
    begin
        if nextsymbol = '+' then
          begin
              t_val := T;
              e1_prime_i := eprime_i + t_val;
              e1_prime_s := E1_PRIME(e1_prime_i);
              E_PRIME := e1_prime_s
          end
          else E_PRIME := e_prime_i
    end;
```

L-ATTRIBUTED DEFINITIONS

BOTTOM-UP PARSING WITH L-ATTR. EVALUATION

Given a translation scheme, we shall transform it so that all actions appear to the right of productions. This is because we want the attribute evaluation to take place *just before a reduction.*
This is done by the use of marker non-terminals. For example

$$\begin{aligned}
&E \rightarrow TR \\
&R \rightarrow addop\ T\ \{print(addop.lexeme)\}\ R_1 \mid \epsilon \\
&T \rightarrow num\ \{print(num.val)\}
\end{aligned}$$

is changed into

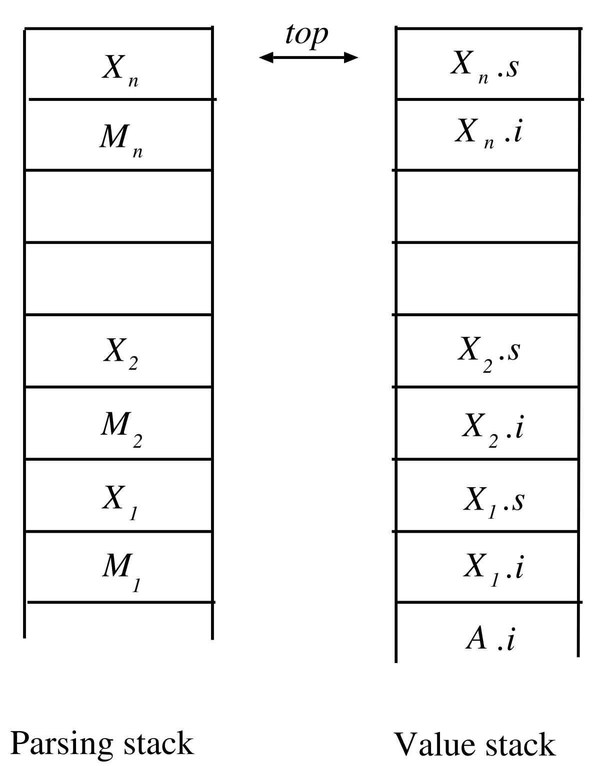
$$\begin{aligned}
&E \rightarrow TR \\
&R \rightarrow addop\ T\ M\ \ R_1 \mid \epsilon \\
&T \rightarrow num\ \{print(num.val)\} \\
&M \rightarrow \epsilon\ \{print(addop.lexeme)\}
\end{aligned}$$

Note that there is a reference to an attribute of a grammar symbol $addop.lexeme$ **in the production** $M \rightarrow \epsilon$**, which does not have an occurrence of** $addop$**. This is different from the examples considered earlier.**
However, the method of implementation ensures that the positions of such attributes, which are non-local to a production, can be determined in the value stack.

L-ATTRIBUTED DEFINITIONS

The key idea is that a production $A \to X_1, \ldots, X_n$ is transformed into $A \to M_1X_1, \ldots, M_nX_n$ by the introduction of marker non-terminals M_i.

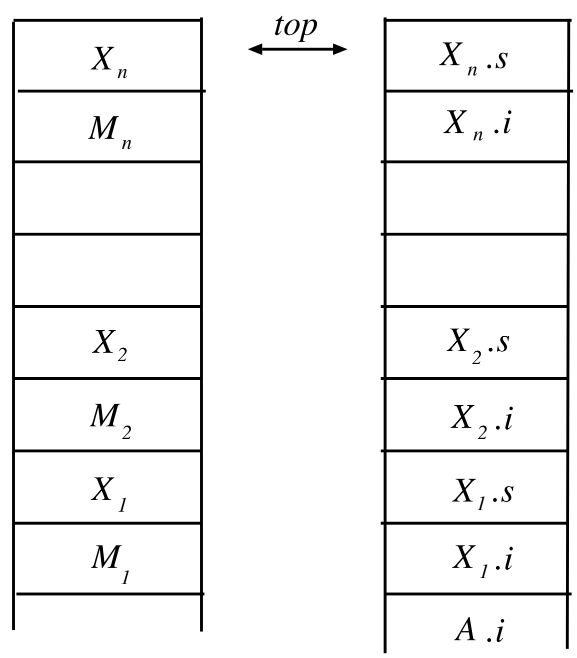
1. **The synthesized attribute $X_j.s$ will go on the value stack (*val array*) in the position associated with the entry X_j.**
2. **The inherited attribute $X_j.i$, if there is one, appears in the same array, associated with the marker non-terminal M_j.**



Where do we find $A.i$? It is found in the position immediately below M_1 in the value stack. This is always true for the inherited attribute of the nonterminal of the left hand side of the production.

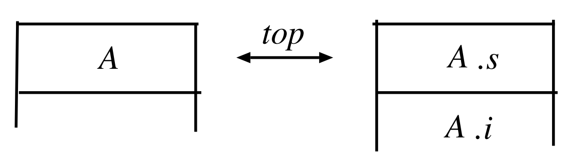
L-ATTRIBUTED DEFINITIONS

Just before a reduction using $A \to M_1X_1, \ldots, M_nX_n$, the parsing stack and the value stack will be



Parsing stack Value stack

After the reduction the configuration is

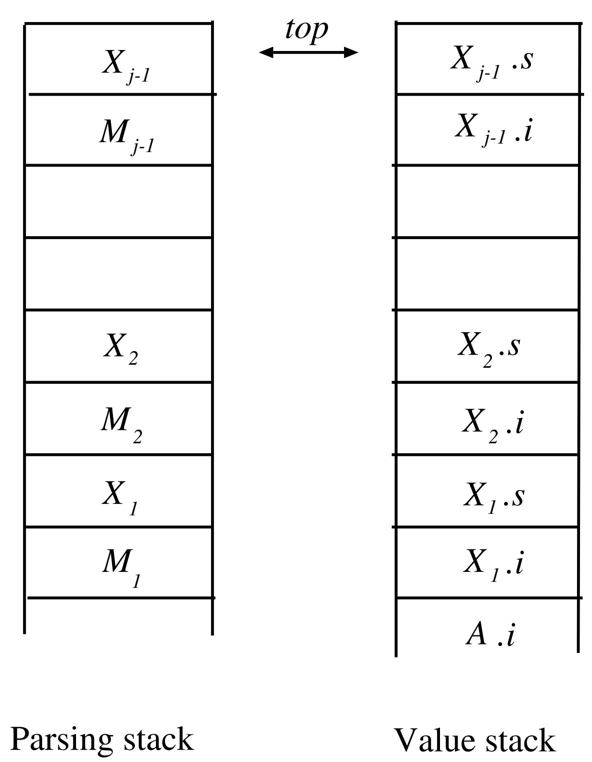


Parsing stack Value stack

If we are to evaluate the attributes just prior to a reduction, can these attributes be located in the value stack?

L-ATTRIBUTED DEFINITIONS

1. **Consider a reduction using $M_j \to \epsilon$ along with an action to be performed to evaluate an inherited attribute of X_j. The stack configurations are**



If the action corresponding to $M_j \to \epsilon$ refers to

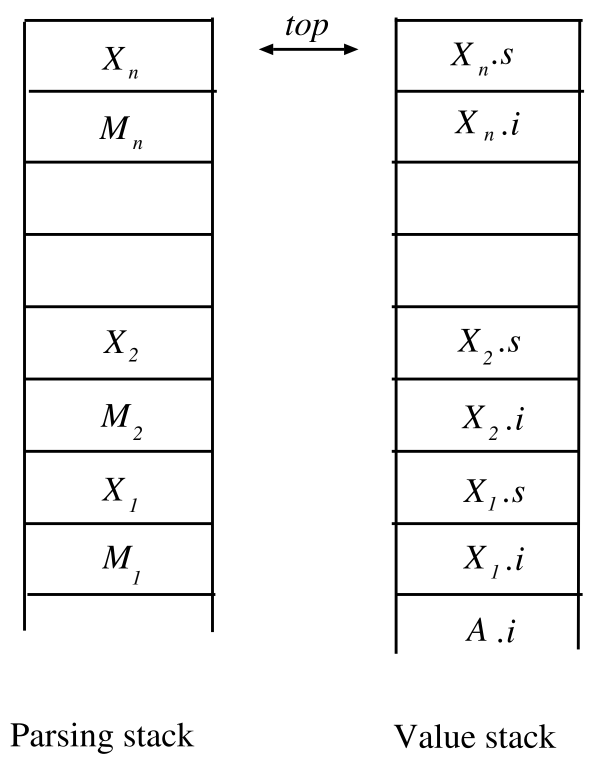
a. $A.i$, **it would be found in the location** $top - 2j - 2$.

b. $X_1.i$, **it would be found in the location** $top - 2j - 1$.

c. $X_1.s$, **it would be found in the location** $top - 2j$.

So the reduction can be done, and the value of $X_j.i$ **put in** $top + 1$.

2. Now consider a reduction using $A \to M_1X_1, \ldots, M_nX_n$ along with an action which computes a synthesized attribute of A. The stack configurations in this case are



If the action corresponding to $M_j \to M_1X_1, \ldots, M_nX_n$ refers to

a. $A.i$, it would be found in location $top - 2n - 2$.

b. $X_1.i$, it would be found in location $top - 2n - 1$.

Once again the reduction can be done, and the value of $A.s$ put in $top - 2n - 1$.

L-ATTRIBUTED DEFINITIONS

Consider the example shown below

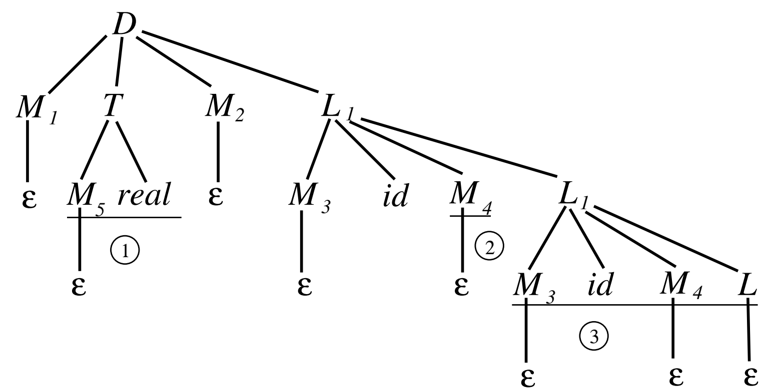
$D \to T$	$\{\ L.in := T.type\}$
$\quad L$	
$L \to$	$\{id.type := L.in\}$
$\quad id$	$\{L_1.in := L.in\}$
$\quad L_1$	
$L \to \epsilon$	
$T \to real$	$\{T.type := real\}$

The grammar will be transformed into

$D \to M_1\ T\ M_2\ L$	
$L \to M_3\ id\ M_4\ L_1$	
$L \to \epsilon$	
$T \to M_5\ real$	$\{T.type := real\}$
$M_1 \to \epsilon$	
$M_2 \to \epsilon$	$\{L.in := T.type\}$
$M_3 \to \epsilon$	$\{id.type := L.in\}$
$M_4 \to \epsilon$	$\{L_1.in := L.in\}$
$M_5 \to \epsilon$	

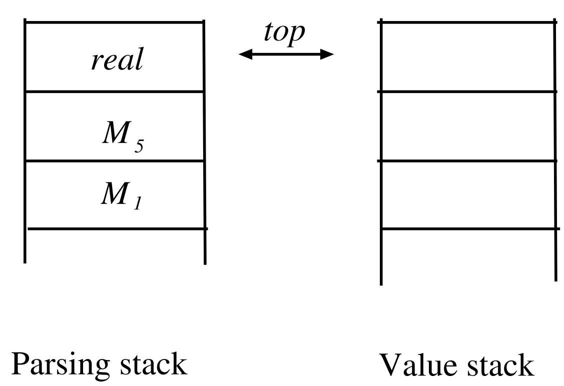
L-ATTRIBUTED DEFINITIONS

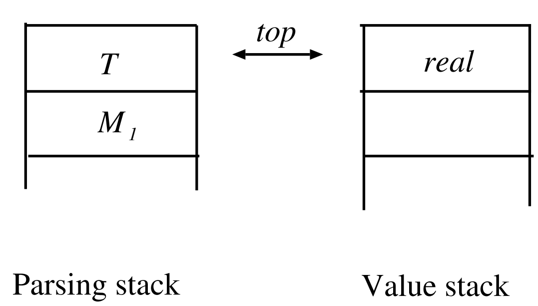
The parse tree for the string `real a b` is



The parse and value stack configurations just before and after the reductions of the handles 1, 2 and 3, indicated in the figure are:

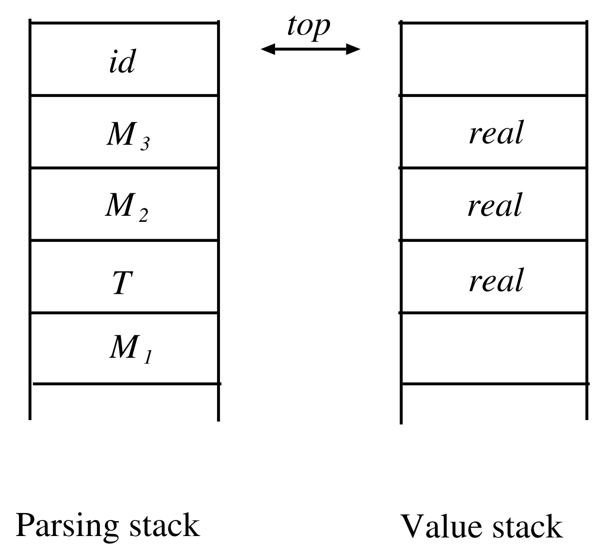
handle 1

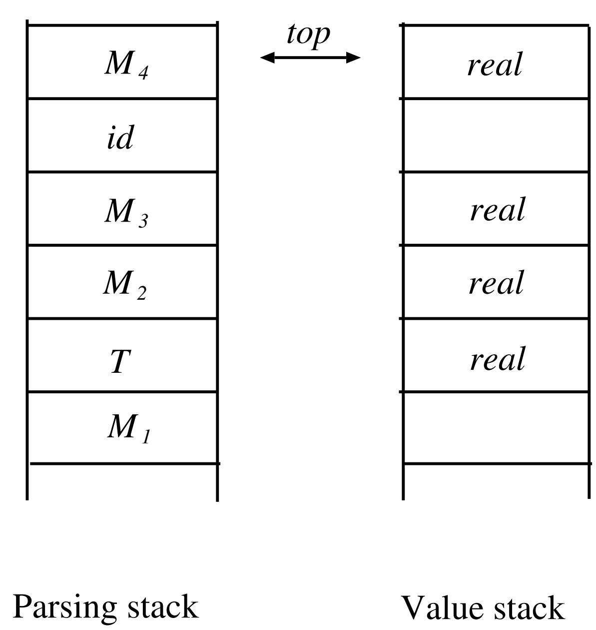




L-ATTRIBUTED DEFINITIONS

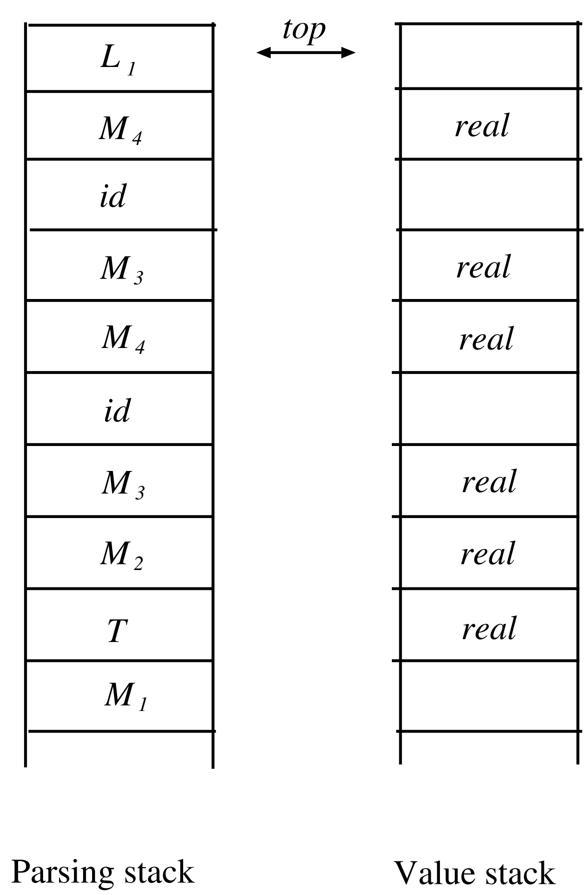
handle 2



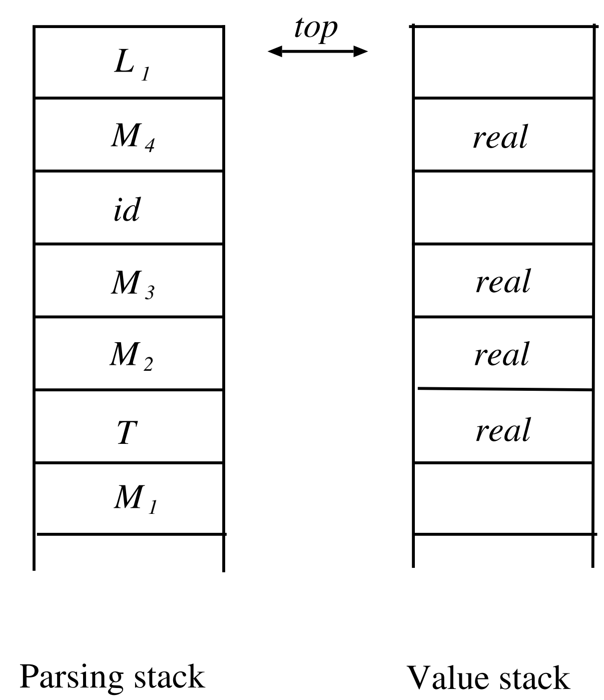


L-ATTRIBUTED DEFINITIONS

handle 3



L-ATTRIBUTED DEFINITIONS



GRADED EXERCISES

5.1 **The following grammar generates expressions formed by applying an arithmetic operator + to integer and real constants. When two integers are added, the resulting type is integer, else it is real.**

$$E \rightarrow E + T \mid T$$
$$T \rightarrow \texttt{num.num} \mid \texttt{num}$$

- **a. Give a syntax-directed definition to determine the type of each subexpression.**
- **b. Extend the syntax-directed definition of a. to translate expressions into postfix notation as well as determining types. Use the unary operator `intoreal` to convert an integer value into an equivalent real value, so that both operators of + in the postfix form have the same type.**
- **c. Implement the syntax directed definition above assuming a bottom-up parsing strategy, i.e. rewrite the definition so that it refers to positions in the value stack instead of attributes.**
- **d. Eliminate left recursion from the definitions in b.**
- **e. Write a recursive descent parser for the resulting definition in d.**

STATIC SEMANTICS ANALYSIS

GRADED EXERCISES

5.2 Consider the grammar

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \texttt{num}$$

a. **Write a syntax directed definition to calculate the value of a given expression.**

b. **Modify the definition to eliminate left recursion in the underlying grammar.**

c. **Implement the modified definition assuming a bottom up parsing strategy.**

d. **Show the parsing and the value stacks at different points of time during the parse of `(2 + 3) * 5`.**

5.3 Enhance the grammar for declarations to include type definitions. Suggest a design for installing these names in the symbol table (should these names be distinguished from variable declarations ?). Write semantic code for constructing the associated type expressions and entering them into the table.

GRADED EXERCISES

5.4 Write a grammar for function declarations (Pascal-like) and add semantic rules to it for constructing type expressions. The type expressions also need to be stored in the symbol table.

5.5 Consider the grammar for array declarations. Write semantic rules for this grammar so that the constant part of address calculation can be performed during parsing and stored in the symbol table. Explain your organization of the symbol table and attributes/functions employed.

5.6 Modify the structural equivalence algorithm to test for type compatibility of type graphs containing cycles (or an equivalent acyclic form given in Figure 5.5).

5.7 Consider the grammar for declarations and assignment statement. The intent is to use type graphs for the purpose of type checking.

a. Write semantic code for the declarations part to construct type graphs.

b. Suggest a design to store these graphs in the symbol table and ensure that this is implemented through your attribute grammar.

GRADED EXERCISES

c. Write semantic code to test for structural equivalence using type graphs for the assignment statement part of the grammar.

5.8 Consider the following three rules for the while-do statement which have marker nonterminals at different places. Which of these would enable correct translation of this construct and why ?

(a) $S \rightarrow \texttt{while}\ M_1 B\ \texttt{do}\ M_2 S_1$

(b) $S \rightarrow M_1\ \texttt{while}\ B M_2\ \texttt{do}\ S_1$

(c) $S \rightarrow \texttt{while}\ M_1 B M_2\ \texttt{do}\ S_1$

5.9 Write a grammar for translation of the following features and add semantic rules as stated against each of these.

a. repeat until control flow construct of Pascal. Explain your scheme for 3-address code generation and write semantic rules for implementing them.

GRADED EXERCISES

b. for statement of Pascal. Explain your scheme of intermediate code generation and the possible semantic errors that need to be detected. Write an attribute grammar for effecting the desired translation.

c. Write a grammar for the case statement of Pascal. Repeat the same actions as mentioned in step(b) above.

d. Write a grammar for the break statement. Its semantics is to exit from the loop within which it is placed to the statement lexically following the loop statement. Write semantic rules to achieve the desired translation.

e. Write a grammar that includes all the control flow statements discussed in the lectures , those mentioned above and the unconditional jump statement, viz., goto label. Write semantic rules for testing the single-entry character of the relevant constructs such as while-do, for statement, case, etc. Your translation should check for jump into the body of these statements from outside (including nested loops) and detect such errors.

5.10 Write a grammar for assignment statement that permits function calls along with array references. Explain your design to implement function calls (assuming only call by value) and write attribute grammar for implementing the same.