UNIT – IV
Semantic Analysis

Objectives:
To gain knowledge on syntax directed translation, symbol table organization for languages and various storage allocation strategies

Syllabus:
SDT, evaluation of semantic rules, Symbol tables- use of symbol tables, contents of symbol-table, operations on symbol tables, symbol table organization for block and non-block structured languages. Runtime Environment- storage organization- static, stack allocation, heap management.

Learning Outcomes:
Students will be able to
- understand syntax-directed translation.
- understand symbol table organization for the block structured and non-block structured languages.
- understand various storage allocation strategies.

Learning Material

4. Semantic Analysis
- Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.
- A compiler must check that the source program follows both the syntactic and semantic conventions of the source language.
- This checking called static checking ensures that certain kinds of programming errors will be detected and reported.

4.1 Syntax-Directed Translation
When we associate semantic rules with productions, we use two notations:
- Syntax-Directed Definition
- Translation Scheme

4.1.1 Syntax-Directed Definition(SDD):

A syntax-directed definition is a generalization of a CFG in which:

- Each grammar symbol is associated with a set of attributes.
- Each production rule is associated with a set of semantic rules.
- Syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.
- This set of attributes for a grammar symbol is partitioned into two subsets called synthesized and inherited attributes of that grammar symbol.
- The value of a synthesized attribute at a node is computed from the values of attributes at the children in that node of the parse tree.
- The value of an inherited attribute at a node is computed from the values of attributes at the siblings and parent of that node of the parse tree.
- An attribute may hold almost anything i.e. a string, a number, a memory location, a complex record.

Example 4.1

Write SDD for a simple desk calculator.

Production	**Semantic Rules**
$L \rightarrow E\ n$	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ digit	F.val = digit.lexval

❖ Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an annotated parse tree.
- Values of Attributes in nodes of annotated parse-tree are either
  - initialized to constant values by the lexical analyzer or
  - determined by the semantic-rules.
- The process of computing the attributes values at the nodes is called annotating (or decorating) of the parse tree.
- The order of these computations depends on the dependency graph induced by the semantic rules.

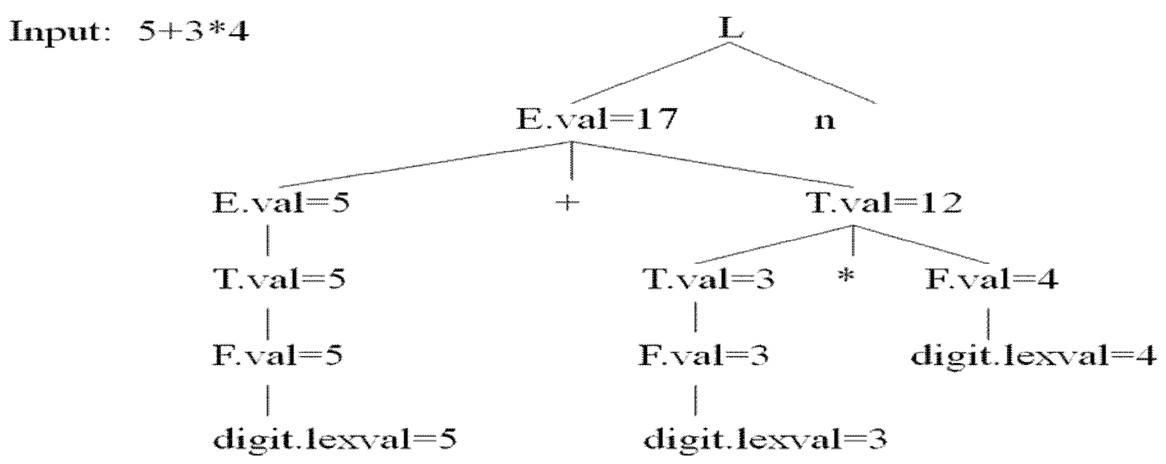


Fig. 4.1 Annotated parse tree

4.1.2 Translation Scheme

- A **translation scheme** is a context-free grammar in which:
  – attributes are associated with the grammar symbols and
  – semantic actions enclosed between braces { } are inserted within the right sides of productions.
- Each semantic rule can only use the information compute by already executed semantic rules.
- Semantic actions indicate the order of evaluation of semantic actions associated with a production rule.

Example 4.2

Translation scheme that converts infix expressions to the corresponding postfix expressions.

E → T R
R → + T { print("+") } R1
R → ε
T → **id** { print(**id**.name) }
a+b+c ↑   ab+c+ ↑
infix expression   postfix expression

The depth first traversal of parse tree executing the semantic actions in that order will produce the postfix representation of infix expression.

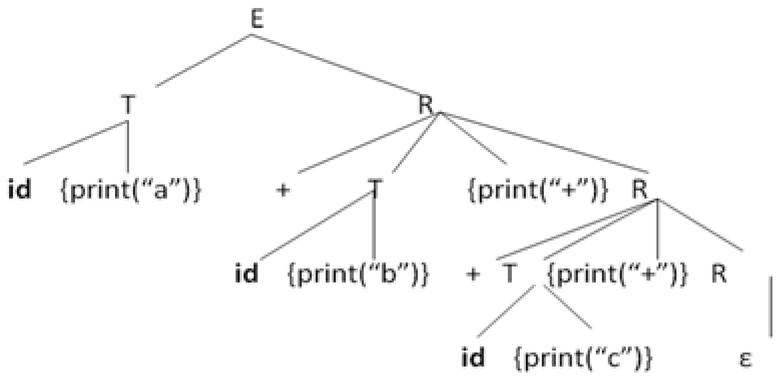


Fig 4.2 Infix to postfix conversion

4.2 Evaluation of semantic rules

A dependency graph is used to determine the order of computation of attributes

❖ Dependency Graph

- It is a directed graph that shows interdependencies between attributes.
- Put each semantic rule into the form $b=f(c_1,\ldots,c_k)$ by introducing dummy synthesized attribute b for every semantic rule that consists of a procedure call.
- The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c
- If a semantic rule defines the value of synthesized attribute A.b in terms of the value of X.c then the dependency graph has an edge from X.c to A.b
- If a semantic rule defines the value of inherited attribute B.c in terms of the value of X.a then the dependency graph has an edge from X.a to B.c
- If an attribute b at a node depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c.

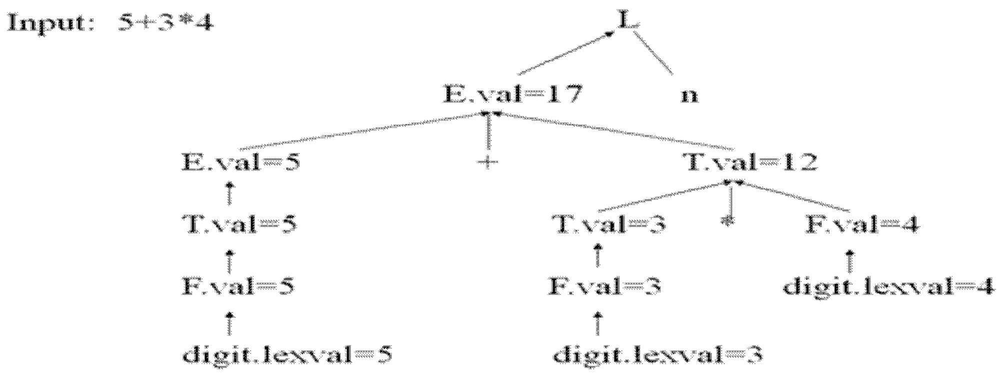


Fig. 4.3 Dependency graph

❖ **S-attributed definition**

- Syntax directed translation that uses synthesized attributes exclusively is said to be a S-attributed definition.
- An SDD is S-attributed if every attribute is synthesized.
- A parse tree for a S-attributed definition can be annotated by evaluating the semantic rules for the attributes at each node, bottom up from leaves to the root.
- We can have a post-order traversal of parse-tree to evaluate attributes in S-attributed definitions

```
postorder(N)
{
        for (each child C of N, from the left) postorder(C);
        evaluate the attributes associated with node N;
}
```

❖ **L-attributed definition**

- A SDD is L-attributed if the edges in dependency graph go from Left to Right but not from Right to Left.
- More precisely, each attribute must be either synthesized or inherited.

Example 4.3

The inherited attribute distributes type information to the various identifiers in a declaration.

Production	**Semantic Rules**
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 **id**	L_1.in = L.in, add type (**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)

Annotated parse tree

Input: real p,q,r

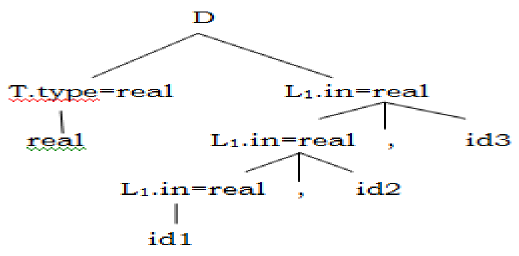


Fig 4.4 Annotated parse tree

Dependency Graph

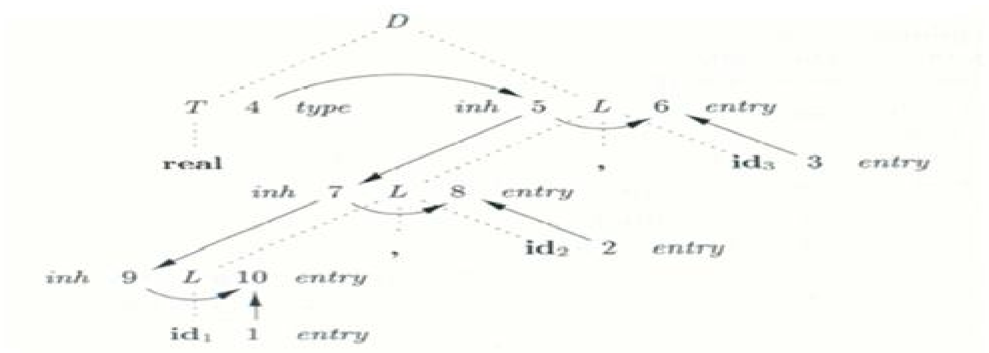


Fig 4.5 Dependency graph for a declaration float id1, id2, id3

4.3 Symbol Tables

- A symbol table is a series of rows, each row containing a list of attribute values that are associated with a particular variable.
- The kinds of attributes appearing in a symbol table are dependent to some degree on the nature of the programming language for which the compiler is written.
- The organization of the symbol table will vary depending on memory and access-time constraints.
- The primary measure which is used to determine the complexity of a symbol-table operation is the average length of search.
- This measure is the average number of comparisons required to retrieve a symbol-table record in a particular table organization.
- The collection of attributes stored in the table for a given variable will be called a symbol-table record.
- The name of the variable for which an insertion or lookup operation is to be performed will be referred to as the search argument.

4.3.1 Use of symbol tables

A symbol table may serve the following purposes depending upon the language

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

4.3.2 Contents of symbol-table

The following list of attributes is not necessary for all compilers.

1. Variable name
2. Object-code address
3. Type
4. Dimension or number of parameters for a procedure
5. Source line number at which the variable is declared
6. Source line numbers at which the variable is referenced
7. Link field for listing in alphabetical order

Table 4.1 Typical view of a symbol table

	Variable Name	**Address**	**Type**	**Dimension**	**Line Declared**	**Lines Referenced**	**Pointer**
1	COMPANY#	0	2	» 1	2	9, 14,25	7
2	X3	4	1	0	3	12,14	0
3	FORM 1	8	3	2	4	36,37,38	6
4	B	48	1	0	5	10, 11,13, 23	1
5	ANS	52	1	0	5	11,23,25	4
6	M	56	6	0	6	17,21	2
7	FIRST	64	1	0	7	28, 29, 30, 38	3

- A variable's name must always reside in the symbol table, since it is the means by which a particular variable is identified for semantic analysis and code generation.
- A major problem in symbol-table organization can be the variability in the length of identifier names.
- While there are many ways of handling the storage of variable names, two popular approaches will be outlined—one which facilitates quick table access and another which supports the efficient storage of variable names.

- To provide quick access, it is best to insist on a predefined, yet sufficiently large, maximum variable name length. A length of sixteen or greater is very likely adequate.
- The complete identifier can then be stored, left-justified, in a fixed-length field in the symbol table. With this straightforward approach, table access is fast but the storage of short variable names is inefficient.
- A second approach is to place a string descriptor in the variable-name field of the table.
- The descriptor contains position and length subfields.
- The pointer subfield indicates the position of the first character of the variable name in a general string area, and the length subfield describes the number of characters in the variable name.

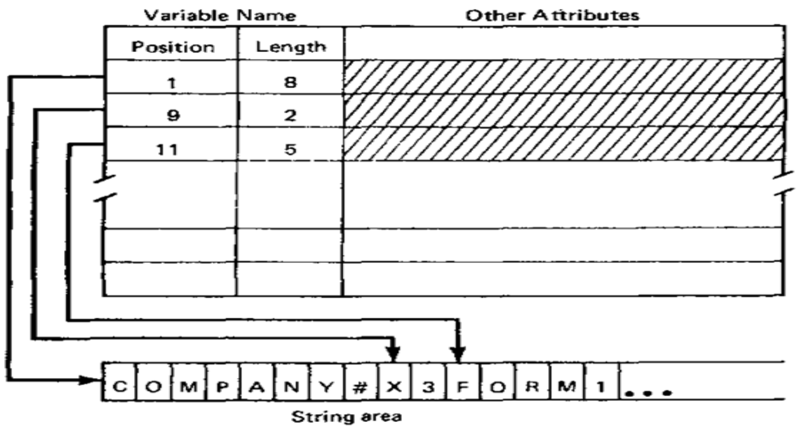


Fig 4.6 using a string descriptor to represent a variable name

4.3.3 Operations on symbol sable

- The two operations that are most commonly performed on symbol tables are insertion and lookup (also referred to as retrieval).
- For block-structured languages, two additional operations, set and reset, are required for symbol-table interaction.
- The set operation is invoked when the beginning of a block is recognized during compilation.
- The complementary operation, the reset operation, is applied when the end of a block is encountered.

❖ Symbol-table organization for non-block structured language

- Non-block-structured language is a language in which each separately compiled unit is a single module that has no sub modules.
- All variables declared in a module are known throughout the module.

- The symbol table organizations for non block structured languages are:
  - ✓ Unordered Symbol Tables
  - ✓ Ordered Symbol Tables
  - ✓ Tree Structured Symbol Table
  - ✓ Hash Symbol Tables

Unordered Symbol Table

- The simplest method of organizing a symbol table is to add the attribute entries to the table in the order in which the variables are declared.
- In an insertion operation no comparisons are required (other than checking for symbol-table overflow), since the attributes for a newly defined variable are simply added at the current end of the table.
- The lookup operation requires, on the average, a search length of (n + l)/2, assuming that there are n records in the table.
- An unordered table organization should be used only if the expected size of the symbol table is small, since the average time for lookup and insertion is directly proportional to the table size.

Ordered Symbol Table

- An ordered symbol table is lexically ordered on the variable names.
- A number of table-lookup strategies can be applied to a table organization; we will examine the two most popular search techniques: the linear search and the binary search.
- One major advantage is that the ordered symbol table can be used in the direct generation of a cross-reference listing, whereas an unordered table would have to be sorted before printing such a listing.

Variable Name	**Type**	**Dimension**	**Other Attributes**
ANS	1	0	▨
B	1	0	▨
COMPANY*	2	1	▨
FIRST	1	0	▨

FORM1	3	2	▨
M	6	0	▨
X3	1	0	▨

Fig 4.7 an ordered symbol table.

Tree-Structured Symbol Table

- The time to perform a table-insertion operation can be reduced from that of an ordered table by using a tree-structured type of storage organization.
- In this type of organization an attempt is made to reflect the search structure of the binary search in the structural links of a binary-search tree.
- A binary tree-structured symbol table that contains the two new fields are present in a record structure.
- These two link fields will be called the left pointer and right pointer fields. Access to the tree is gained through the root node (i.e., the top node of the tree).
- A search proceeds down the structural links of the tree until the desired node is found or a NULL link field (denoted by /) is encountered. The search is unsuccessful, since a match between the search argument and the names in the table is not achieved.

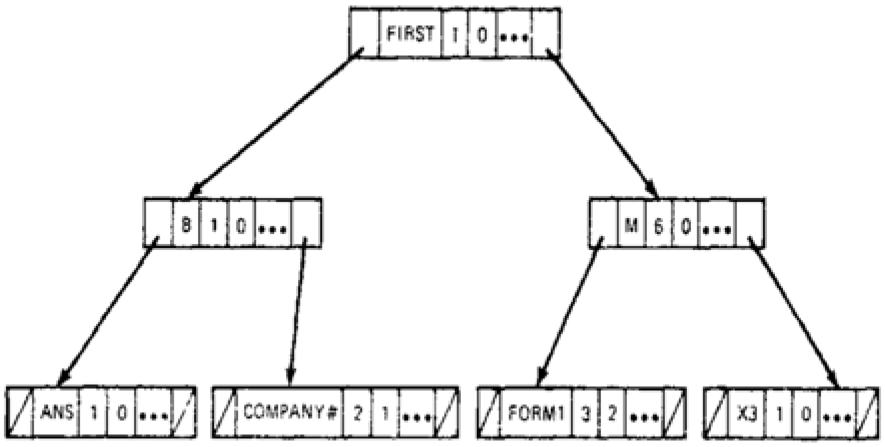


Fig 4.8 Binary-tree organization of a symbol table showing the logical relationships

Table Position	Name Field	Type	Dimension	Other Attributes	Left Pointer	Right Pointer
1	FIRST	1	0	▨	.... !' 2	5
2	B	1	0	▨	3	4
3	ANS	1	0	▨	0	0
4	COMPANY #	2	1	▨	0	0
5	M	6	0	▨	6	7
6	F0RM1	3	2	▨	0	0
7	X3	1	0	▨	0	0
			(6)			

Fig 4.9 Binary-tree organization of a symbol table showing a physical representation

Hash Symbol Table

- The best search methods for the symbol-table organizations had an average length of search requiring on the order of $\log_2 n$ comparisons.
- An organization in which the search time is essentially independent of the number of records in the table (i.e., the search time is constant for any n).
- The name space (also called identifier space or key space) is the set K of unique variable names that can appear in a program.
- A hashing function (or key-to-address transformation) is defined as a mapping $H: K \rightarrow A$. That is, a hashing function H takes as its argument a variable name and produces a table address (i.e., location) at which the set of attributes for that variable are stored.
- The function generates this address by performing some simple arithmetic or logical operations on the name or some part of the name.

- The most widely accepted hashing function is the division method, which is defined as H(x) =(x mod m)+1 for divisor m.
- Two records cannot occupy the same location, and therefore some method must be used to resolve the collisions that can result. There are basically two collision-resolution techniques: open addressing and separate chaining.

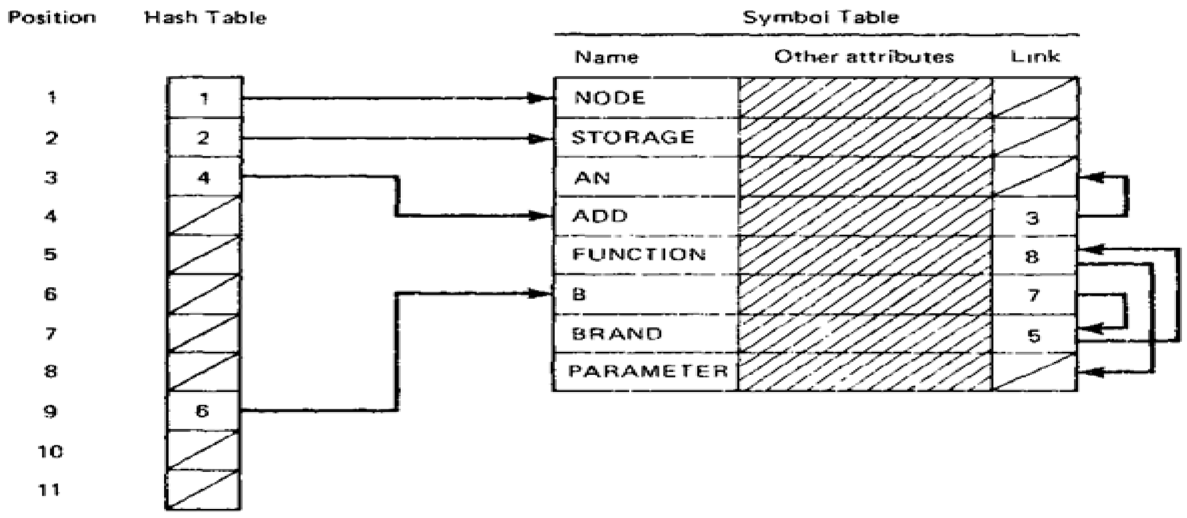
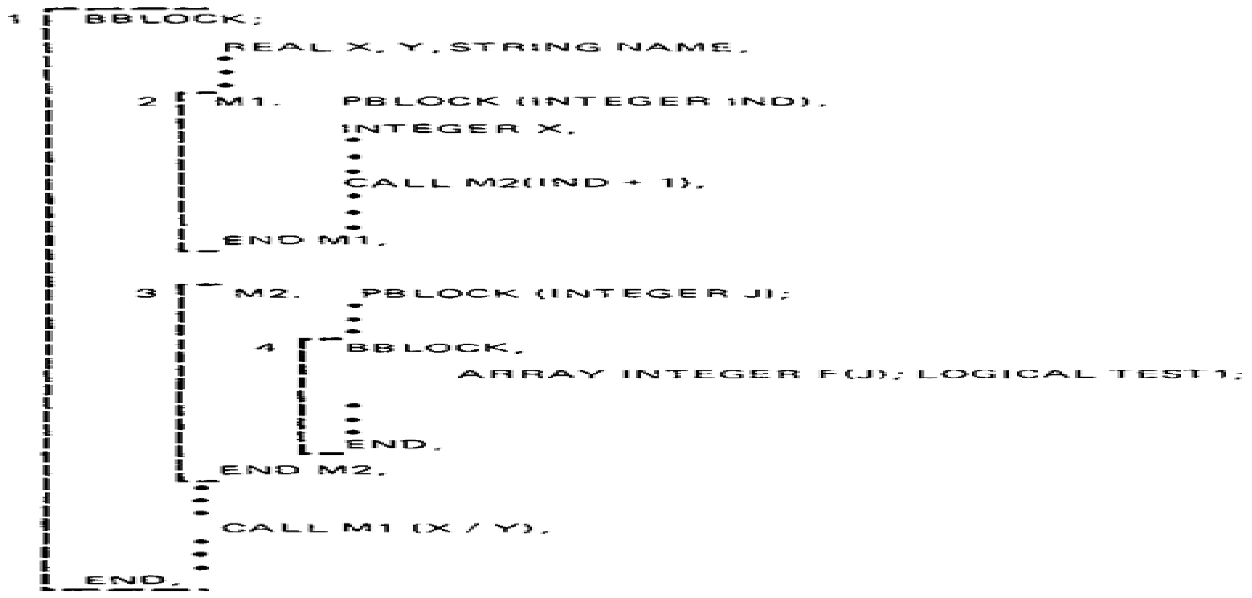


Fig 4.10 Hashing using a hash table

❖ Symbol-Table Organizations For Block-Structured Languages

- By a block-structured language we mean a language in which a module can contain nested sub modules and each sub module can have its own set of locally declared variables.
- A variable declared within a module A is accessible throughout that module unless the same variable name is redefined within a sub module of A.
- The redefinition of a variable holds throughout the scope (i.e., physical limits) of the sub module.
- The two additional symbol-table operations, set and reset, were introduced for block-structured languages.
- The symbol table organizations for block structured languages are:
  - ✓ Stack Symbol Table
  - ✓ Stack-Implemented Tree-Structured Symbol Table
  - ✓ Stack-Implemented Hash-Structured Symbol Table

Example 4.4 A program segment from a nested language



```
1  BBLOCK;
            REAL X, Y, STRING NAME.
            .
            .
   2  M1.      PBLOCK (INTEGER IND).
               INTEGER X.
               .
               .
               CALL M2(IND + 1).
               .
               .
      END M1.

   3  M2.      PBLOCK (INTEGER J);
               .
               .
         4  BBLOCK.
                  ARRAY INTEGER F(J); LOGICAL TEST1;
               .
               .
            END.
      END M2.
       .
       .
       CALL M1 (X / Y).
       .
       .
   END.
```

Operation	Symbol-table contents (variable name only)	
	Active	**Inactive**
Set BLK.1	empty	empty
Set BLK2	MI, NAME, Y, X	empty
Reset BLK2	X, IND, MI, NAME, Y, X	empty
Set BLK3	M2, MI, NAME, Y, X	X, IND
Set BLK4	J, M2, MI, NAME, Y, X	X, IND
Reset BLK4	TEST1, F, J, M2, MI, NAME, Y, X	X, IND
Reset BLK3	J, M2, MI, NAME, Y, X	TEST1, F, X, IND
Reset BLK1	M2, MI, NAME, Y, X	J, TEST1, F, X, IND
End of compilation	empty	M2, MI, NAME. Y. X. J.
		TEST1, F, X, IND

Fig 4.11 a trace showing the effects of the set and reset operations

Stack Symbol Table

- The conceptually simplest symbol-table organization for a block-structured language is the stack symbol table.
- In this organization the records containing the variables' attributes are simply stacked as the declarations are encountered.
- Upon reaching the end of a block, all records for variables declared in the block are removed since these variables cannot be referenced outside the block.

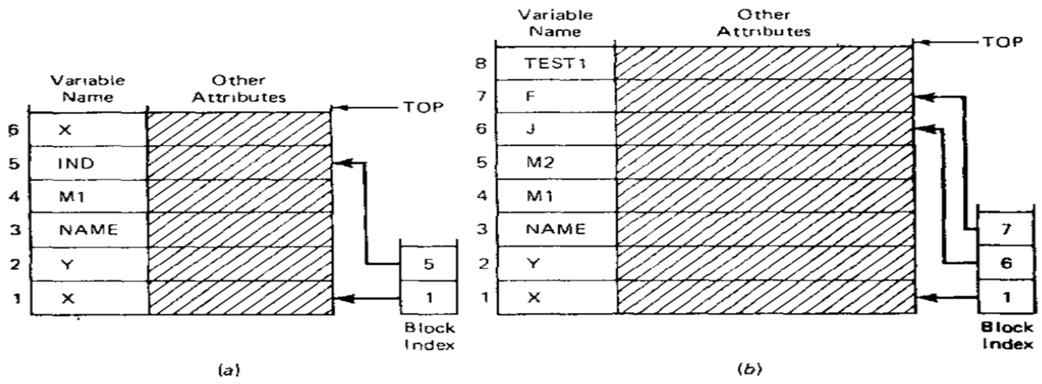


Fig 4.12(a) and 4.12(b) Example of a stack symbol table

- The insertion operation is very simple in a stack symbol table. New records are added at the top location in the stack.
- The lookup operation involves a linear search of the table from the top to the bottom. The search must be conducted in this order to guarantee that the latest occurrence of a variable with a particular name is located first.
- The set operation for a stack symbol table generates a new block index entry at the top of the block index structure. This entry is assigned the value of the current top of the symbol stack and therefore establishes a pointer to the first record location for a record in the block just entered.
- The reset operation effectively removes all the records in the stack symbol table for the block just compiled. This is accomplished by setting TOP, the index pointing at the first open record location at the top of the symbol table, to the value of the top element in the block index.
- The stack symbol table also suffers from the same poor performance characteristics

Stack-Implemented Tree-Structured Symbol Table

- The two possible approaches for organizing a symbol table using a tree-structured organization:
- The first approach involves a single tree organization. The major difference that arises when the compilation of block-structured languages is considered is the fact that records for a block must be removed from the table when the compilation of the block is completed. As a result, the problem of deleting table records must be addressed. Since the records for all blocks are merged in one tree, a number of steps are necessary in order to delete a record:
  1. Locate the position of the record in the tree.
  2. Remove the record from the tree by altering the structural links so as to bypass the record.
  3. Rebalance the tree if the deletion of the record has left the tree unbalanced.
- A second organization which we will call a forest of trees is amenable to the problem of deleting records. In this organization each block is allocated its own tree-structured table. When the compilation of a block is finished, the entire table associated with that block is released.

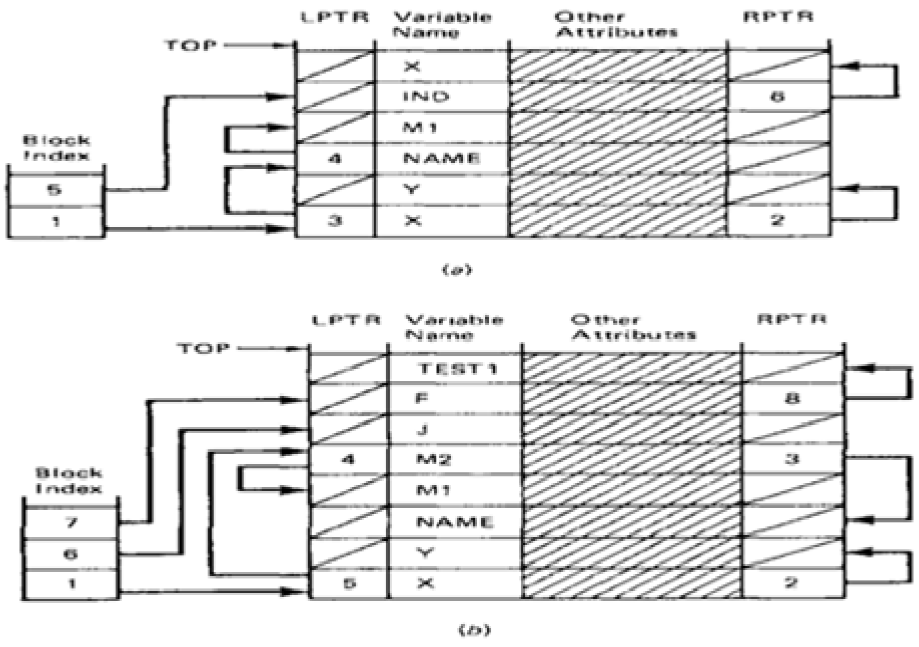


Fig 4.13(a) and 4.13(b) Example of a stack-implemented tree-structured symbol table

Stack-Implemented Hash-Structured Symbol Table

- Hashing is, in general, a non-order-preserving transformation and because block-structured languages require that the variables in a block be grouped in some manner, it would seem that a hashing methodology and a symbol-table organization for block-structured languages are incompatible.
- Figure 4.13 depicts a symbol-table organization for the program segment in example 4.4, assuming a hashing function which performs the following transformations:
  the names X and M1 are mapped to 1
  the name NAME is mapped to 3
  the names IND and J are mapped to 5
  the name TEST1 is mapped to 6
  the names F and Y are mapped to 8
  the name M2 is mapped to 11
- An intermediate hash table of size 11 is used in the example.
- The insert and lookup operations for stack-implemented hash symbol tables are essentially the same
- The expected length of search may be slightly less than for the hash symbol table for non-block-structured languages because local variables are deleted as blocks are compiled in a block-structured language.

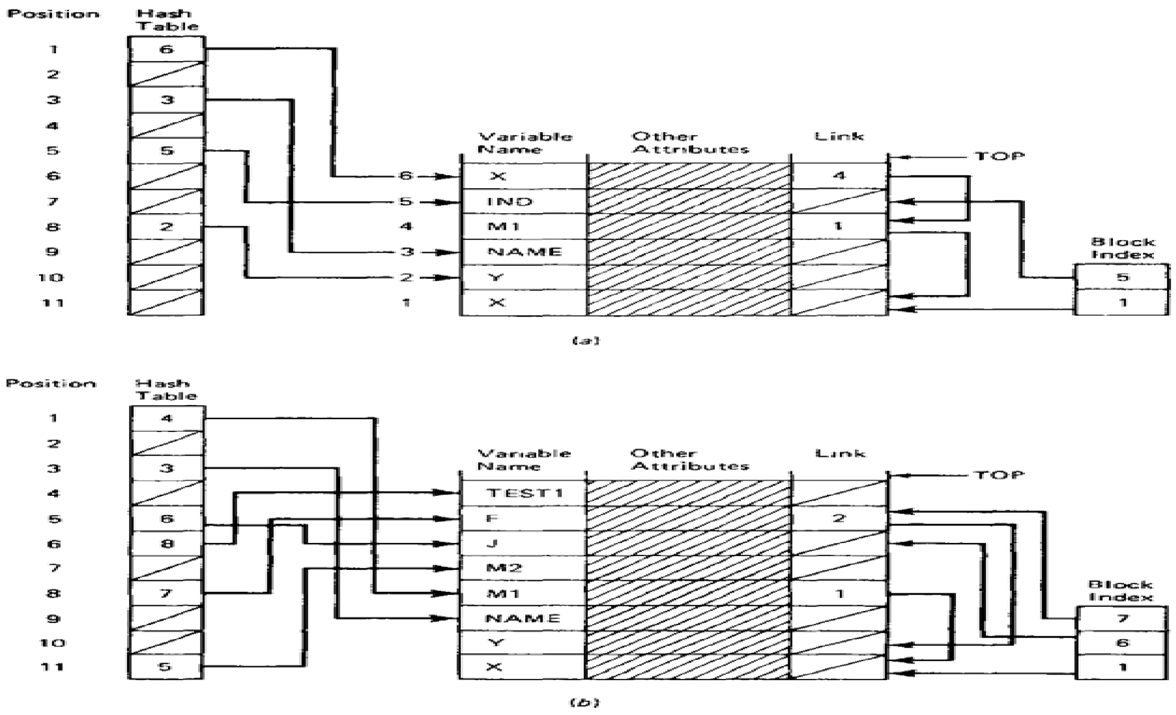


Fig 4.14(a) and 4.14(b) Example of a stack-implemented hash symbol table

4.4 Runtime Environment

- The compiler creates and manages a run-time environment in which it assumes its target programs are being executed.
- This environment deals with a variety of issues such as the layout and allocation of storage locations for the objects named in the source program, the mechanisms used by the target program to access variables, the linkages between procedures, the mechanisms for passing parameters, and the interfaces to the operating system, input/output devices, and other programs.

4.4.1 Storage Organization

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system, and target machine.
- The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory.

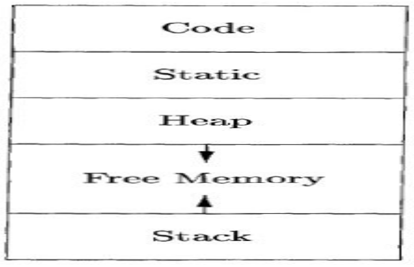


Fig 4.15 Typical subdivision of run-time memory into code and data areas

- The different ways to allocate memory are:
  - ✓ Static storage allocation
  - ✓ Stack storage allocation
  - ✓ Heap storage allocation

Static Storage Allocation

- A storage-allocation decision is static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.
- In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes.

- As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

❖ Limitations of Static Allocation

- The size of a data object and constraints on its position in memory must be known at compile time.
- Recursive procedures are restricted, because all activation of a procedure use the same bindings for local names.
- Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.

Stack storage allocation:

- Stack allocation is based on the idea of a control stack.
- Activation records are pushed and popped as activation begin and end.
- Procedure calls and their activations are managed by means of stack memory allocation.
- It works in last-in-first-out LIFO method and this allocation strategy is very useful for recursive procedure calls.
- Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

❖ Activation Records

- Procedure calls and returns are usually managed by a run-time stack called the control stack.
- Each live activation has an activation record (sometimes called a frame) on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.
- The latter activation has its record at the top of the stack.
- The contents of activation records vary with the language being implemented

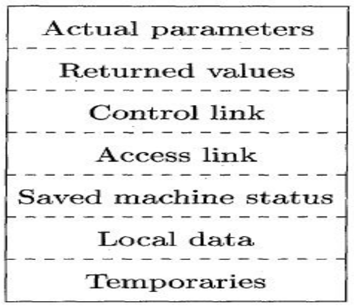


Fig 4.16 A general activation record

1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.
2. Local data belonging to the procedure whose activation record this is.
3. A **saved machine status**, with information about the state of the machine just before the call to the procedure. This information typically includes the return address (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.
4. An "**access link**" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.
5. A **control link**, pointing to the activation record of the caller.
6. Space for the **return value** of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
7. The **actual parameters** used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency.

Example 4.5 Sketch of a quick sort program

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p − 1] are less than v, a[p] = v, and a[p + 1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

❖ **Activation Tree**

- We can represent the activations of procedures during the running of an entire program by a tree, called an activation tree.
- Each node corresponds to one activation, and the root is the activation of the "main" procedure that initiates execution of the program.

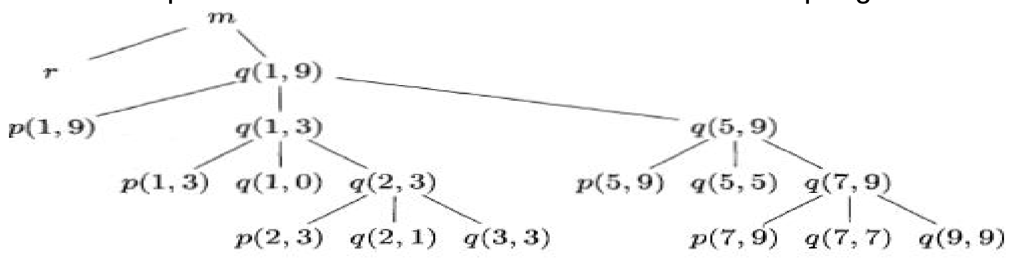


Fig 4.17 Activation tree representing calls during an execution of quick sort

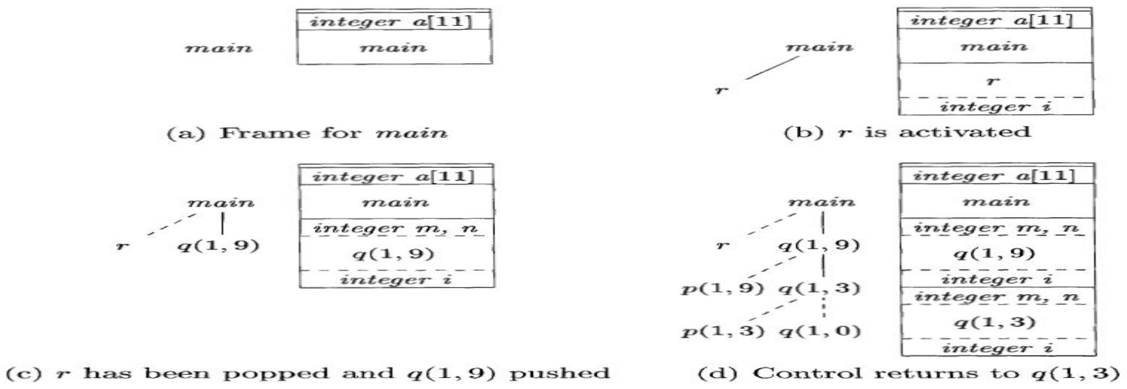


Fig 4.18 Downward-growing stack of activation records

Heap management

- Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage.
- The heap is an area of virtual memory that allows objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.
- To support heap management, "garbage collection" enables the run-time system to detect useless data elements and reuse their storage, even if the programmer does not return their space explicitly.
- Variables local to a procedure are allocated and de-allocated only at runtime.
- Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

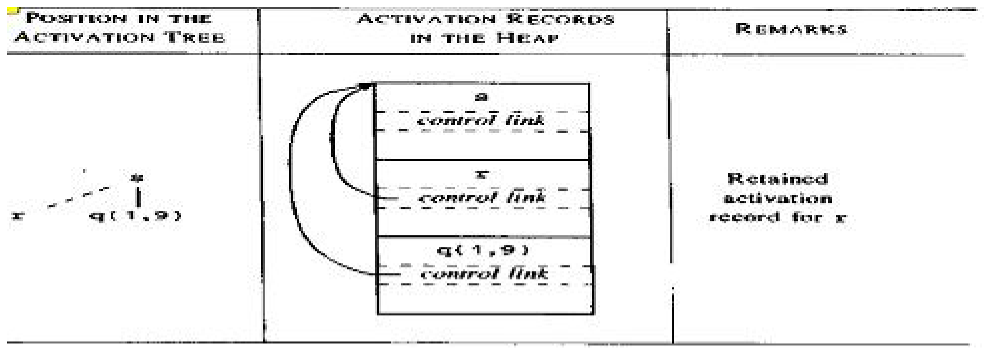

Position in the Activation Tree
Activation Records in the Heap
Remarks
s
r
q(1,9)
control link
Retained activation record for r

Fig 4.19 Records for live activations need not be adjacent in a heap

UNIT – IV
Assignment-Cum-Tutorial Questions

A. Objective Questions

1. If a parent node takes a value from its children is called ____________ attributes.
2. Every S-attributed definition is L-attributed definition. [True | False]
3. The interdependencies among attributes are shown by _______________ graph.
4. Symbol Table can be used for: [ ]
   a) Checking type compatibility c) Storage allocation
   b) Suppressing duplication of error message d) All of these
5. A programming language which allows recursion can be implemented with static storage allocation. [True | False]
6. The two basic operations that are often performed in the symbol table are
   a) Set and reset c) Insert and lookup [ ]
   b) Set and insert d) Reset and lookup
7. In ordered symbol tables, the entries in the table are lexically ordered on the [ ]
   a) variable name c) variable size
   b) variable type d) All of the above
8. Information needed by a single execution of a procedure is managed using a contiguous block of storage called________________.
9. The static binding occurs during: [ ]
   a) Compile time c) Run time
   b) Linking time d) Pre-processing time
10. The dynamic binding occurs during ___________ [ ]
    a) Compile time c) Run time
    b) Linking time d) Pre-processing time
11. Type checking is normally done during [ ]
    a) Lexical analysis c) Syntax analysis
    b) Syntax directed translation d) Code generation
12. Consider the following Syntax Directed Translation Scheme (SDTS),with nonterminals {S, A} and terminals {a, b}

    $S \rightarrow aA$ {print 1}
    $S \rightarrow a$ {print 2}
    $A \rightarrow Sb$ {print 3}

    Using the above SDTS, the output printed by a bottom-up parser for the input aab is [ ]
    a) 1 3 2 b) 2 2 3 c) 2 3 1 d) Syntax error
13. $A \rightarrow BC$ {B.s=A.s} [ ]
    a) S-attributed b) L-attributed c) Both d) None

14. Consider the following translation scheme.

    S → ER
    R → *E {print("*");}R | ε
    E → F + E {print("+");} | F
    F → (S) | id {print(id.value);}

    Here id is a token that represents an integer and id.value represents the corresponding integer value. For an input '2 * 3 + 4', this translation scheme prints [ ]

    a) 2 * 3 + 4 b) 2 * +3 4 c) 2 3 * 4 + d) 2 3 4+*

15. Synthesized attribute can be easily simulated by a [ ]

    a) LL grammar c) Ambiguous grammar
    b) LR grammar d) None of the above

16. A parse tree is an annotated parse tree if: [ ]

    a) it shows attribute values at each node.
    b) there are no inherited attributes.
    c) it has synthesized nodes as terminal nodes.
    d) every non-terminal nodes is an inherited attribute.

B. Descriptive Questions

1. Define synthesized and inherited attributes with an example.
2. Construct a annotated parse tree for the expression 7*4+5/n?
3. Write an SDD for construction of syntax trees.
4. Write short notes on Syntax directed translation with an example.
5. Explain the concept of syntax directed definition and its usage.
6. What is an S-attributed definition and L-attributed definition? Explain with an example.
7. Explain any one syntax-directed translation schemes.
8. Describe the general structure of an activation record. Explain the purpose of each item in the activation record.
9. Explain Operations on symbol tables for block structured and non-block structured languages. What is an unordered symbol table?
10. Write short notes on Symbol table organization for block structured languages.
11. What are the various attributes of a symbol table?
12. Explain tree structured symbol tables.
13. Explain stack implemented tree-structured symbol tables.
14. Explain stack implemented Hash-structured symbol tables.
15. Explain static storage allocation strategy.
16. Explain in detail, the strategy for reducing fragmentation in heap memory.

17. Differentiate between Static and dynamic storage allocation strategies.

C. GATE Questions

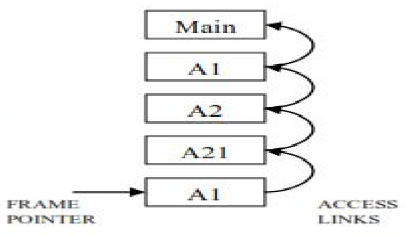
1. Consider the program given below, in a block-structured pseudo-language with lexical scoping and nesting of procedures permitted.

```
Program main;
    Var ...
    Procedure A1;
        Var ...
        Call A2;
    End A1
    Procedure A2;
        Var ...
        Procedure A21;
            Var ...
            Call A1;
        End A21
        Call A21;
    End A2
    Call A1;
End main.
```

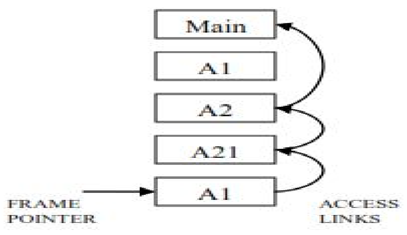
Consider the calling chain: Main → A1 → A2 → A21 → A1

The correct set of activation records along with their access links is given by [ ] **[GATE 2012]**

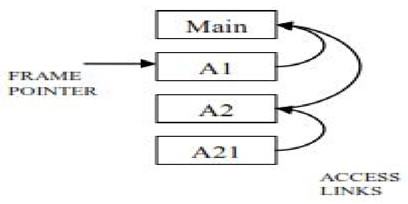
○ (A)



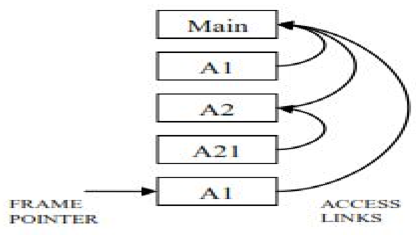
○ (B)



○ (C)



○ (D)



2. Which languages necessarily need heap allocation in the runtime environment? [ ] **[GATE 2010]**
   a) Those that support recursion
   b) Those that use dynamic scoping
   c) Those that allow dynamic data structures
   d) Those that use global variables

3. Consider the grammar with the following translation rules and E as the start symbol.

   E → E1 #T {E.value = E1.value * T.value}

   | T {E.value = T.value}

   T → T1 & F {T.value = T1.value + F.value}

   |F {T.value= F.value}

   F → num {F.value = num.value}

   Compute E.value for the root of the parse tree for the expression:
   2 # 3 & 5 # 6 & 4. [ ] **[GATE 2004]**

   a) 200 b) 180 c) 160 d) 40

4. Consider the following Syntax Directed Translation Scheme (SDTS),with non terminals { E,T,F} and terminals {2,4}

E->E*T	{E.VAL=E.VAL*T.VAL;}
E->T	{E.VAL=T.VAL;}
T->F-T	{T.VAL=F.VAL-T.VAL;}
T->F	{T.VAL=F.VAL;}
F->2	{F.VAL=2;}
F->4	{F.VAL=4;}

   Using the above SDTS, the total number of reductions done by a bottom-up parser for the input 4-2-4*2 is [ ] **[GATE 2000]**

   a) 10 b) 9 c) 11 d) 13

5. A shift reduce parser carries out the actions specified within braces immediately after reducing with the corresponding rule of grammar

   S→ xxW (PRINT "1")

   S→ y { print " 2 " }

   S→ Sz { print " 3 ")

   What is the translation of xxxxyzz using the syntax directed translation scheme described by the above rules? [ ] **[GATE 1995]**

   a) 23131 b) 11233 c) 11231 d) 33211