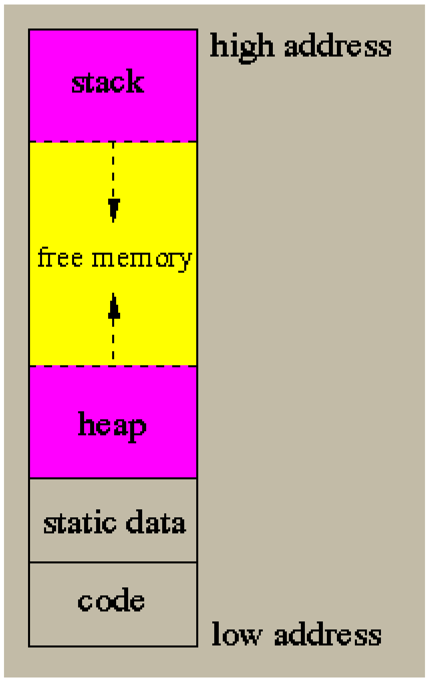
COMPILER DESIGN

Module-4 6th Semester CSE

Runtime Environment: Storage organization



Runtime environment manages runtime memory requirements for the following entities:

- **Code** : It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
- **Procedures** : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- **Variables** : Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Static Allocation

- 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.

Dynamic Allocation: Two types

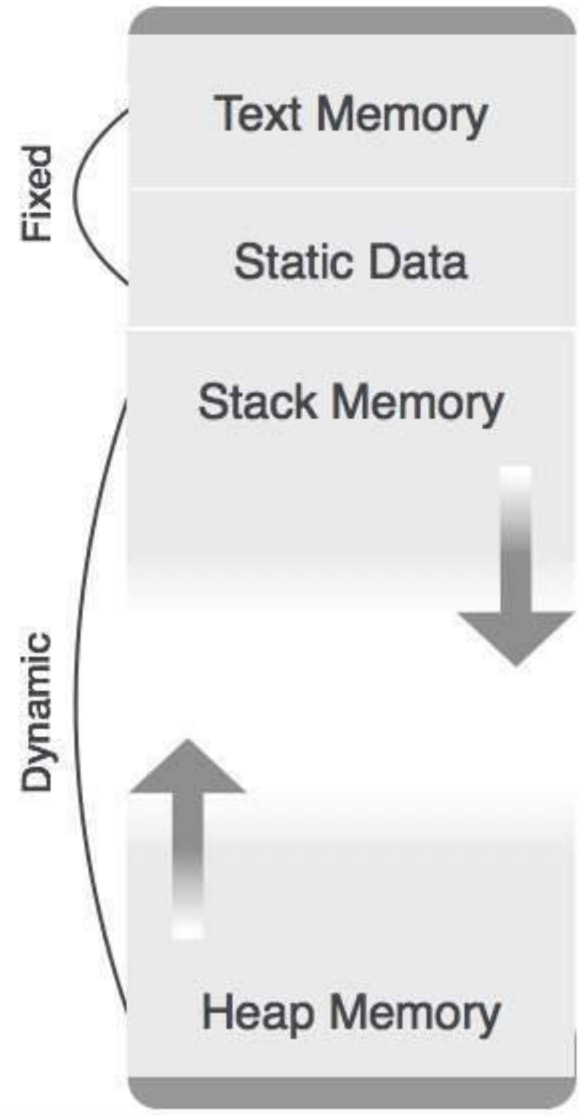
Stack Allocation

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.

Heap Allocation

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.

Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.



Activation record:

Activation record: data about an execution of a procedure.

• Parameters: .

Formal parameters: the declaration of parameters. .

Actual parameters: the values of parameters for this activation.

• Links: .

Access (or static) link: a pointer to places of non-local data, .

Control (or dynamic) link: a pointer to the activation record of the caller.

returned value
actual parameters
optional access link
optional control link
saved machine status
local data
temporaries

Syntax Directed Translation:

Syntax Directed Translation are augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes. Syntax directed translation rules use 1) lexical values of nodes, 2) constants & 3) attributes associated to the non-terminals in their definitions.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

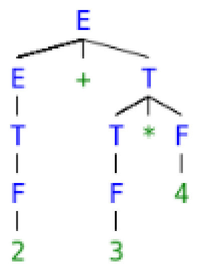
Example

```
E -> E+T | T
T -> T*F | F
F -> INTLIT
```

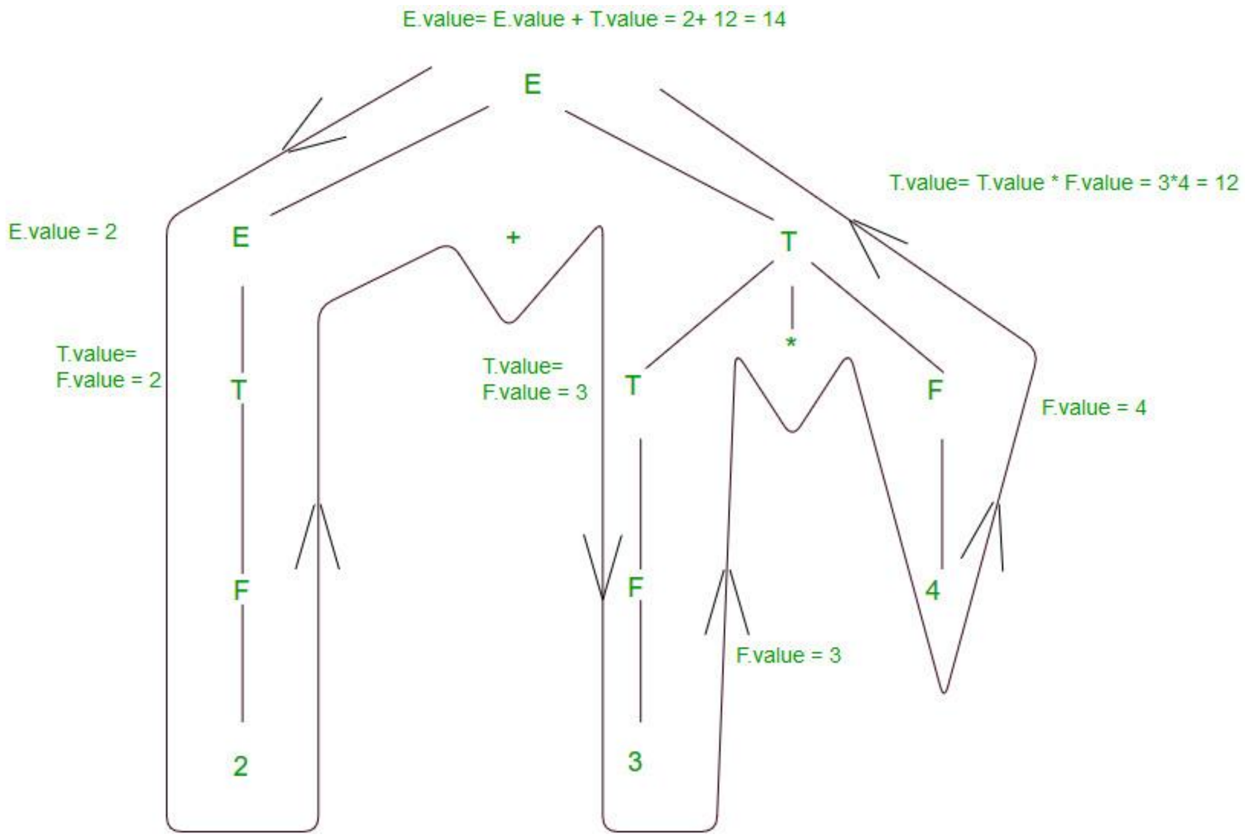
This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any. In this example we will focus on evaluation of the given expression, as we don't have any semantic assertions to check in this very basic example.

```
E -> E+T     { E.val = E.val + T.val }   PR#1
E -> T       { E.val = T.val }           PR#2
T -> T*F     { T.val = T.val * F.val }   PR#3
T -> F       { T.val = F.val }           PR#4
F -> INTLIT  { F.val = INTLIT.lexval }   PR#5
```

Let's take a string to see how semantic analysis happens – S = 2+3*4. Parse tree corresponding to S would be



To evaluate translation rules, we can employ one depth first search traversal on the parse tree. This is possible only because SDT rules don't impose any specific order on evaluation until children attributes are computed before parents for a grammar having all synthesized attributes. Otherwise, we would have to figure out the best suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals. For better understanding, we will move bottom up in left to right fashion for computing translation rules of our example.



Above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children attributes are computed before parents, as discussed above. Right hand side nodes are sometimes annotated with subscript 1 to distinguish between children and parent.
Additional Information.

Synthesized Attributes are such attributes that depend only on the attribute values of children nodes.
Thus [E -> E+T { E.val = E.val + T.val }] has a synthesized attribute val corresponding to node E. If all the semantic attributes in an augmented grammar are synthesized, one depth first search traversal in any order is sufficient for semantic analysis phase.
Inherited Attributes are such attributes that depend on parent and/or siblings attributes.
Thus [Ep -> E+T { Ep.val = E.val + T.val, T.val = Ep.val }], where E & Ep are same production symbols annotated to differentiate between parent and child, has an inherited attribute val corresponding to node T.

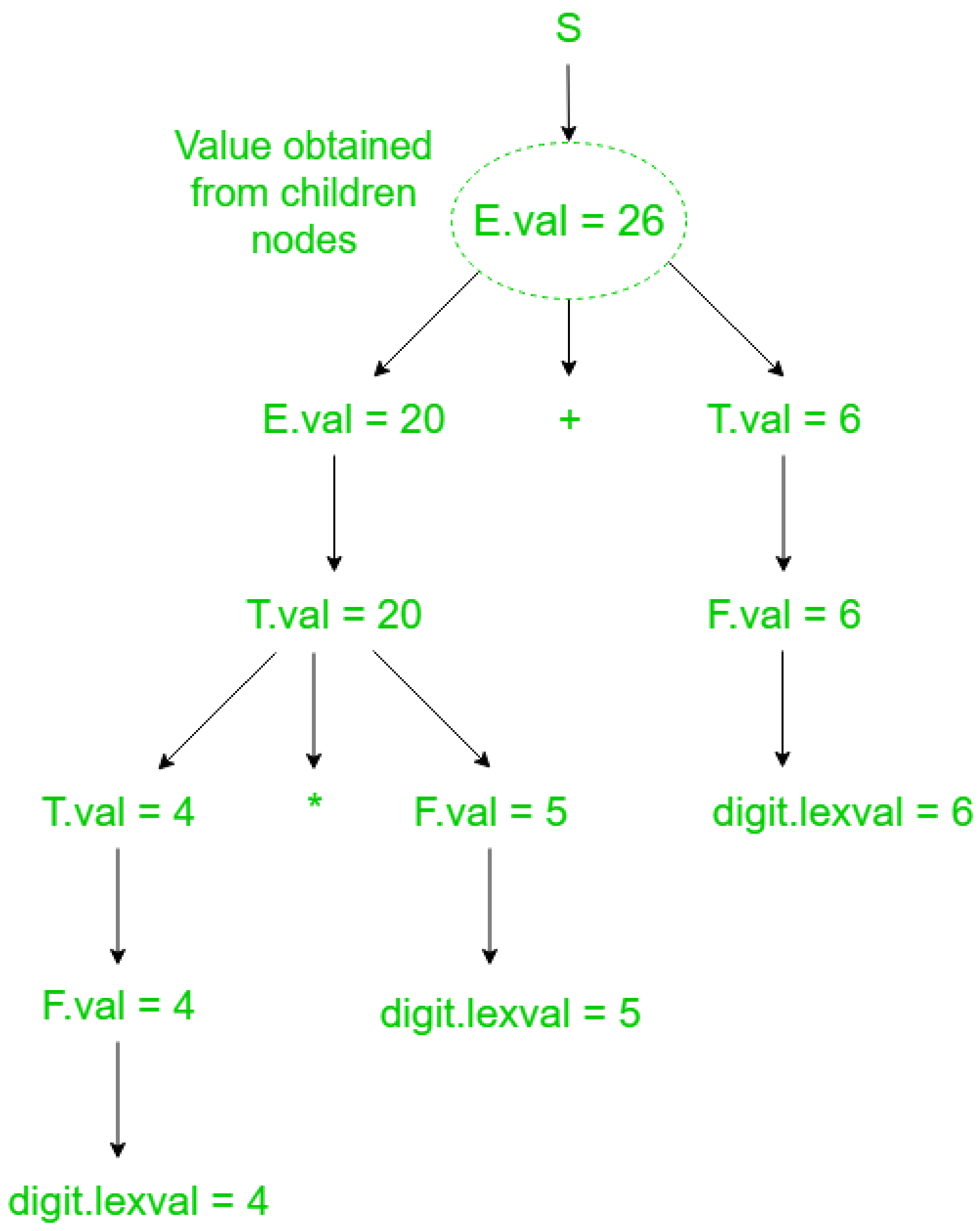
Dependency graphs

The basic idea behind **dependency graphs** is for **compiler** to look for various kinds if dependence among statements to prevent their execution in wrong order i.e. the order that changes the meaning of the program.

Dependency Graph

Input: 5+3*4

$$\begin{tikzcd}
 & & \text{L} & & \\
 & \text{E.val=17} \arrow[ur] & & & \\
\text{E.val=5} \arrow[ur] & & & \text{T.val=12} \arrow[ull] \arrow[dl, dash] \arrow[dr, dash] & \\
\text{T.val=5} \arrow[u] & & \text{T.val=3} & & \text{F.val=4} \\
\text{F.val=5} \arrow[u] & & \text{F.val=3} \arrow[u] & & \text{digit.lexval=4} \arrow[u] \\
\text{digit.lexval=5} \arrow[u] & & \text{digit.lexval=3} \arrow[u] & &
\end{tikzcd}$$



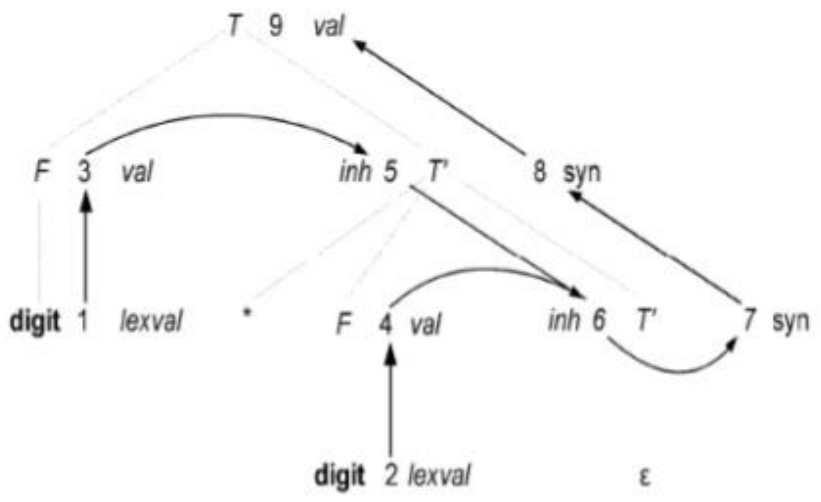
Annotated Parse Tree

Evaluation orders for SDD's

- A dependency graph is used to determine the order of computation of attributes
- Dependency graph
  - For each parse tree node, the parse tree has a node for each attribute associated with that node
  - 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.c to B.c
- Example!

Evaluation Orders for SDD's

Dependency graph for the annotated parse tree for 3*5



Application of Syntax Directed Translation

- SDT is used for Executing Arithmetic Expression.
- In the conversion from infix to postfix expression.
- In the conversion from infix to prefix expression.
- It is also used for Binary to decimal conversion.
- In counting number of Reduction.
- In creating a Syntax tree.
- SDT is used to generate intermediate code.
- In storing information into symbol table.
- SDT is commonly used for type checking also.

Symbol table:Structure and features of Symbol Table

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

- 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).

- A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

```
<symbol name,  type,  attribute>
```

- For example, if a symbol table has to store information about the following variable declaration:

```
static int interest;
```

- then it should store the entry such as:

```
<interest, int, static>
```

- The attribute clause contains the entries related to the name.

Scope Management

A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . .
int value=10;

void pro_one()
   {
   int one_1;
   int one_2;

      {              \
      int one_3;      |_  inner scope 1
      int one_4;      |
      }              /

   int one_5;

      {              \
      int one_6;      |_  inner scope 2
      int one_7;      |
      }              /
   }

void pro_two()
   {
   int two_1;
   int two_2;

      {              \
      int two_3;      |_  inner scope 3
      int two_4;      |
      }              /

   int two_5;
   }
. . .
```

The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first a symbol will be searched in the current scope, i.e. current symbol table.
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or global symbol table has been searched for the name.