UNIT-III

Syllabus: Syntax-Directed Translation: Syntax-Directed Definitions, Evaluation Orders for SDD's Applications of Syntax-Directed Translation.
Symbol Table: Structure, Operations, Implementation and Management.
Objective:
- Describe semantic analyses using an attribute grammar
- To learn how to build symbol tables

Outcome: Construct an abstract syntax tree for language constructs.

SYNTAX-DIRECTED TRANSLATION:

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar.

Concepts related to syntax-directed translation are:

1. **Attributes**	**(Attributes are used in SDD)**
2. **Translation Schemes**	**(Written in SDT)**

Attributes: An attribute is any quantity associated with a programming construct.
- Some of the attributes: Data types of expression, number of instructions, location of the instruction, nonterminals, and terminal symbols.

Translation schemes: A translation scheme is a notation for attaching program fragments to the productions of a grammar

If **X** is a grammar symbol and **'a'** is one of its attributes, then we write **X.a** to denote the value of **'a'** at a particular parse-tree node labelled as **X**.

Example:

Production	**Semantic Rule**
$E \rightarrow E_1 + T$	$E.code=E_1.code + T.code$ (or) $E.code=E_1.code \| \| T.code \| \| '+'$

- The above production has two non-terminals, E_1 and T. E_1 and T have a string-valued attribute **code**. The semantic rule species that the string E:code is formed by concatenating E1:code, T:code, and the character '+'.

Alternatively, we can also insert the **semantic actions/translation scheme** inside the grammar.

E -> E1 + T { print '+' }

- By convention, semantic actions are enclosed within curly braces.

The position of a semantic action in a production body determines the order in which the action is executed. (i.e., semantic actions can be place anywhere in the RHS part of the production rule)

Out of these two notations, syntax-directed definitions (SDD) are more readable, and hence more useful for specifications. However, translation schemes (SDT) can be more efficient, and hence more useful for implementations.

SYNTAX-DIRECTED DEFINITIONS:

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write X:a to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X.

- Attributes may be of any kind: numbers, types, table references, or strings, for instance.
- Rules describe how the attributes are computed at those nodes of the parse tree.

For each **nonterminal** of a production rule, there are two types of attributes.

1. Synthesized Attributes
2. Inherited Attributes

Synthesized Attribute:

- A synthesized attribute for a nonterminal **A** at a parse-tree node **N** is defined by a semantic rule associated with the production at **N**.
- Note that the production must have **A** as its head.
- A synthesized attribute at node **N** is defined only in terms of attribute values of children of **N** and at **N** itself.

Inherited Attribute:

- An inherited attribute for a nonterminal **B** at a parse-tree node **N** is defined by a semantic rule associated with the production at the parent of **N**.
- Note that the production must have **B** as a symbol in its body.
- An inherited attribute at node **N** is defined only in terms of attribute values at **N'**s parent, **N** itself and **N'**s siblings.

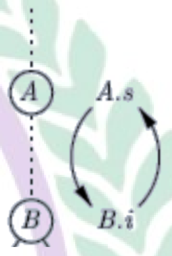
Terminals can have synthesized attributes, but not inherited attributes.

- Attributes for terminals have lexical values that are supplied by the lexical analyzer.
- There are no semantic rules in the SDD itself for computing the value of an attribute for a terminal.

Note:
An inherited attribute at node N cannot be defined in terms of attribute values at the children of node N.
But a synthesized attribute at node N can be defined in terms of inherited attribute values at node N itself.

Example:

PRODUCTION	SEMANTIC RULES
A -> B	**A.s := B.i;**
	B.i := A.s + 1



A and B, have 's' and 'i' as attributes.
Here 's' is synthesized and 'i' is inherited, because 's' is evaluated from its children and 'i' is evaluated from its parent.

EVALUATION ORDERS FOR SDD'S

S-attributed SDD
An SDD that involves only synthesized attributes is called S-attributed SDD.

In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

An S-attributed SDD can be implemented naturally in conjunction with an LR parser. An S-attributed definition is suitable for use during bottom-up parsing.

An SDD without side effects is sometimes called an **attribute grammar**. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

To visualize the translation specified by an SDD, annotated parse trees are used.
A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree.**

Example of S-Attributed SDD

Production	Semantic Rules
L -> E n	L.val = E.val
E -> E_1 + T	E.val = E_1.val + T.val
E -> T	E.val = T.val
T -> T_1 * F	T.val = T_1.val * F.val
T -> F	T.val = F.val
F -> (E)	F.val = E.val
F -> digit	F.val = digit.lexval

In the above SDD, each of the non-terminals has a single synthesized attribute, called **val**.

- Terminal digit has a synthesized attribute **lexval**, which is an integer value returned by the lexical analyzer.
- For production 1: L -> E n, sets L.val to E.val, which is a numerical value of the entire expression (n indicates that).
- For production 2: E -> E_1 + T, also has one rule, which computes the **val** attribute for the head E as the sum of the values at E_1 and T. At any parse tree node N labeled E, the value of **val** for E is the sum of the values of **val** at the children of node N labeled E_1 and T.
- For production 3: E -> T, has a single rule that defines the value of **val** for E
- to be the same as the value of **val** at the child for T.
- For production 4: T -> T_1 * F, also has one rule, which computes the **val** attribute for the head T as the product of the values at T_1 and F. At any parse tree node N labeled T, the value of **val** for T is the product of the values of **val** at the children of node N labeled T_1 and F.
- For productions 5 and 6 copy values at a child, like that for the third production.
- For production 7: It gives F.val the value of a digit, that is, the numerical value of the token digit that the lexical analyzer returned.

Evaluating an SDD at the Nodes of a Parse Tree

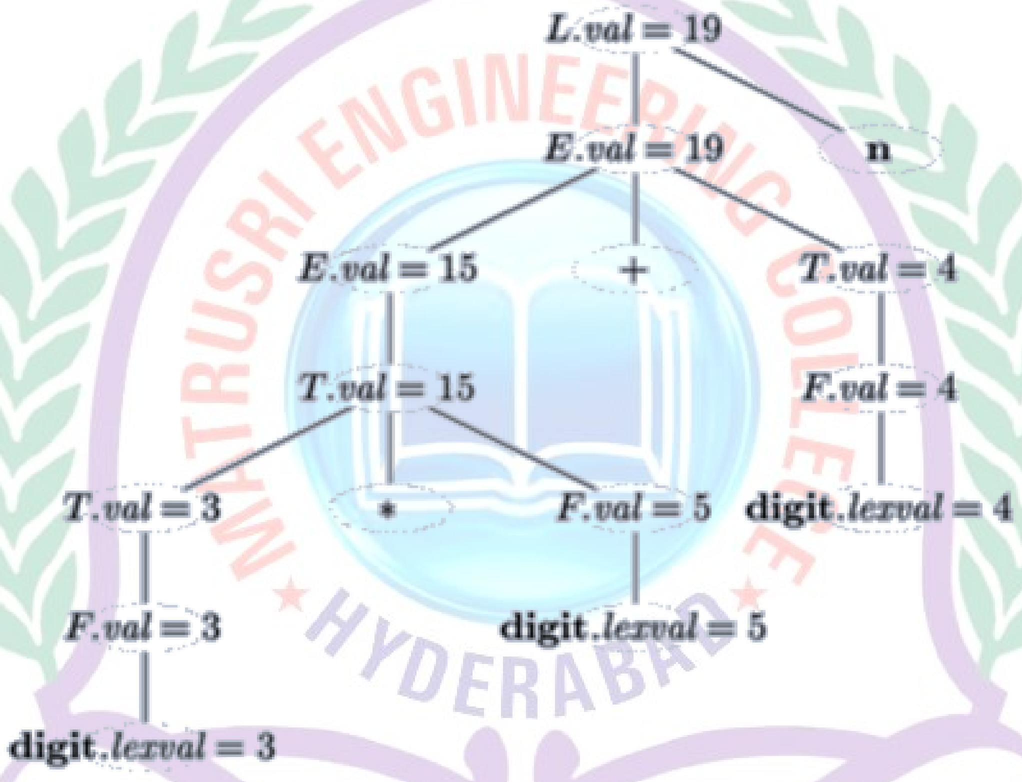
First constructing a parse tree for the grammar and then by using the rules evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.

Before evaluating an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends.

For example,
If all attributes are synthesized, then we must evaluate the **val** attributes at all of the children of a node before we can evaluate the **val** attribute at the node itself.

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

Annotated Parse tree for the above SDD for the expression **3 * 5 + 4 n** is



L-attributed SDD
If an SDD uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

Semantic actions are placed anywhere in RHS.

L-attributed, is suitable for use during top-down parsing.
Example:
A -> XYZ {Y.S = A.S, Y.S = X.S}

is an L-attributed grammar since Y.S = A.S and Y.S = X.S

Note – If a definition is S-attributed, then it is also L-attributed but NOT vice-versa.

Example:

L-Attributed definitions

Production	Semantic Rules
T —> FT’ T’ —> *FT$_1$’	T’.inh = F.val T’$_1$.inh =T’.inh * F.val

In production 1, the inherited attribute T’ is computed from the value of F which is to its left.

In production 2, the inherited attributed T$_l$’ is computed from T’. inh associated with its head and the value of F which appears to its left in the production. i.e., for computing inherited attribute, it must either use from the above or from the left information of SDD.

Syntax-directed definition with inherited attributes

Production	Semantic Rules
D —>TL	L.inh = T.type
T —> int	T.type =integer
T —> float	T.type = float
L —> L$_1$, id	L$_1$.inh = L.inh addType (id.entry, Linh)
L —> id	addType (id.entry, L.inh)

Evaluation Orders for L-Attributed SDD's

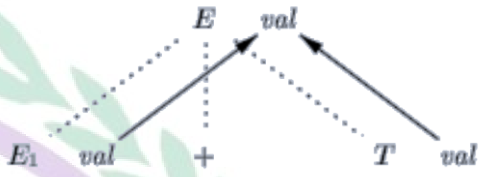
"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

Dependency Graph: A dependency graph depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.

- For each parse tree node X, the dependency graph has a node for each attribute associated with the node X.

- For a production p, 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
- For a production p, 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

PRODUCTION	SEMANTIC RULE
$E \to E_1 + T$	$E.val = E_1.val + T.val$



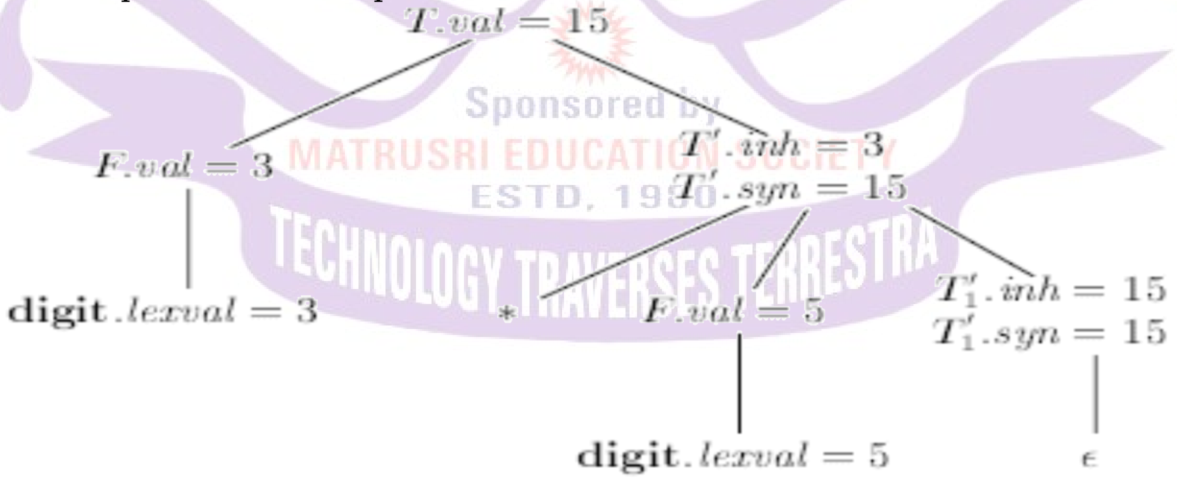
Ordering the evaluation of attributes

- If dependency graph has an edge from M to N then M must be evaluated before the attribute of N
- Thus, the only allowable orders of evaluation are those sequence of nodes $N_1, N_2, \ldots, N_k$ such that if there is an edge from N_i to N_j then i<j
- Such an ordering is called a topological sort of a graph.
- If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree.

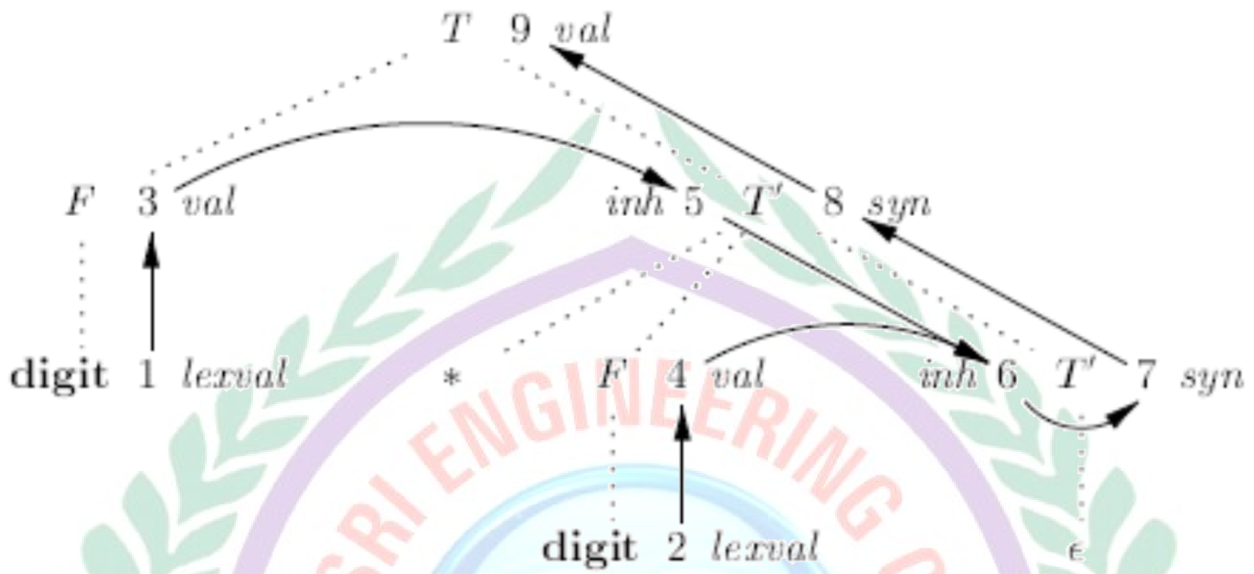
Example: L-Attributed SDD Example

Production	Semantic Rules
T -> FT'	T'.inh = F.val
	T.val = T'.syn
T' -> *FT'$_1$	T'1.inh = T'.inh*F.val
	T'.syn = T'$_1$.syn
T' -> ε	T'.syn = T'.inh
F -> digit	F.val = F.val = digit.lexval

Annotated parse tree for expression: 3 * 5



Dependency Graph for the expression: 3 * 5



- Nodes 1 and 2 represent the attribute lexval associated with the two leaves labeled digit.
- Nodes 3 and 4 represent the attribute val associated with the two nodes labeled F. The edges to node 3 from 1 and to node 4 from 2 result from the semantic rule that defines F.val in terms of digit.lexval. In fact, F.val equals digit.lexval, but the edge represents dependence, not equality.
- Nodes 5 and 6 represent the inherited attribute T'.inh associated with each of the occurrences of nonterminal T'.
- Nodes 7 and 8 represent the synthesized attribute syn associated with the occurrences of T'.

APPLICATIONS OF SYNTAX-DIRECTED TRANSLATION.

Syntax Directed Translation (SDT)

An SDT is a Context Free grammar with program fragments embedded within production bodies. These program fragments are called semantic actions.

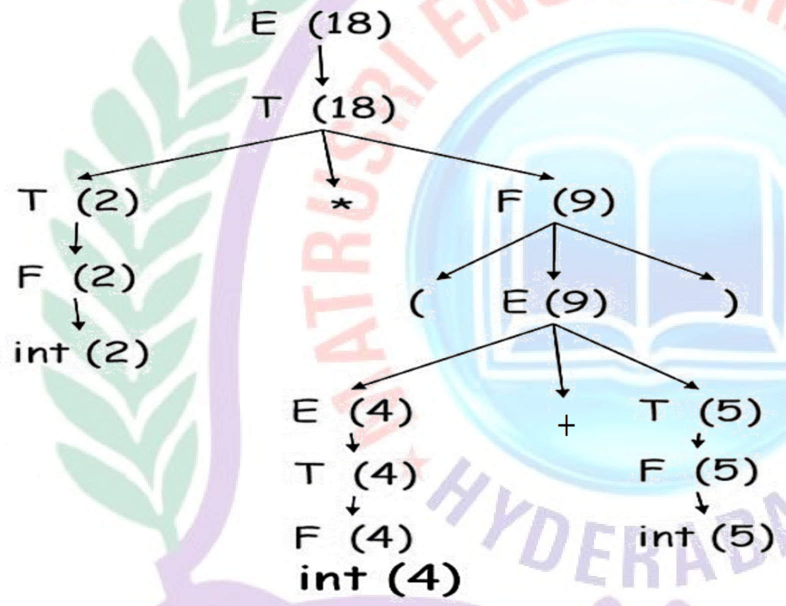
- They can appear at any position within production body, must be placed in { }
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order
- Typically, SDT's are implemented during parsing without building a parse tree

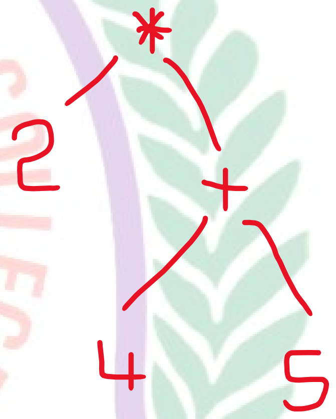
Example of SDT

A common way of syntax-directed translation is translating a string into a sequence of actions by attaching one such action to each grammar rule.

Production	**Semantic Rules**
S → E $	{ print E.VAL }
E → E + E	{ E.VAL := E.VAL + E.VAL }/{ print('+'); }
E → E * E	{ E.VAL := E.VAL * E.VAL }/{ print('*'); }
E → (E)	{ E.VAL := E.VAL }
E → I	{ E.VAL := I.VAL }
I → I digit	{ I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL }

For arithmetic expression 2*(4+5).





Applications of SDT

1. Primary application of SDT is construction of Syntax Trees
   - Since some compilers use the syntax trees as an intermediate representation, a common form of SDD(Syntax Directed Definition) turns its input string into a tree.
2. SDT is used for Executing Arithmetic Expression.
3. In the conversion from infix to postfix expression.
4. In the conversion from infix to prefix expression.
5. It is used for Binary to decimal conversion.
6. In counting number of Reduction.
7. SDT is used to generate intermediate code.
8. In storing information into symbol table.
9. SDT is used for type checking also.

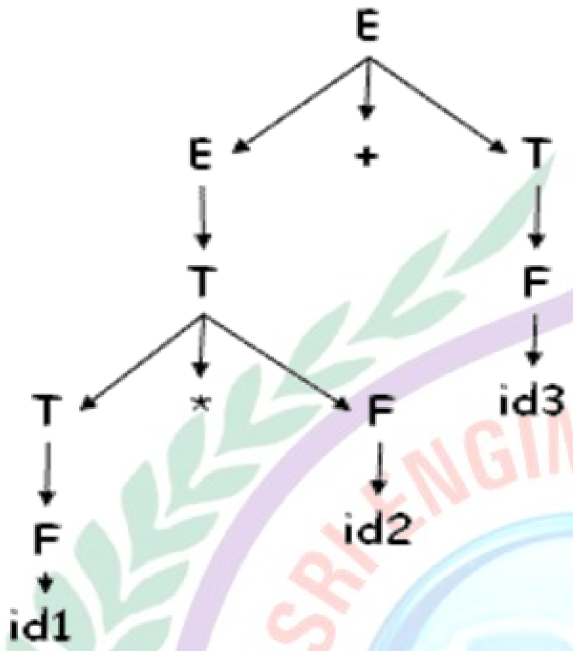
Differences between SDD and SDT

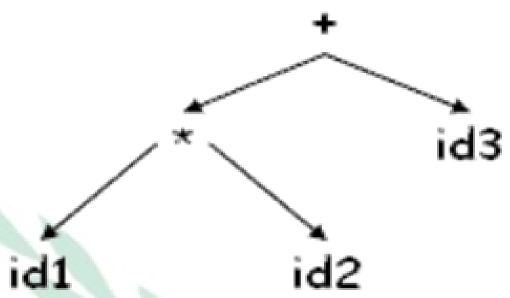
	SDD (Syntax Directed Definition)	**SDT (Syntax Directed Translation)**
1	It is a context-free grammar where attributes and rules are combined and associated with grammar symbols and productions, respectively.	It refers to the translation of a string into an array of actions. This is done by adding an action to a rule of context-free grammar. It is a type of compiler interpretation.
2	Attribute Grammar	Translation Schemes
3	SDD: Specifies the values of attributes by associating semantic rules with the productions	SDT: embeds program fragments (also called semantic actions) within production bodies
4	E -> E + T { E.val := E1.val + T.val }	E -> E + T { print('+'); }
5	Always written at the end of body of production	The position of the action defines the order in which the action is executed (in the middle of production or end)
6	More Readable	More Efficient
7	Used to specify the values of nonterminals	Used to implement S-Attributed SDD and L-Attributed SDD
8	Specifies what calculation is to be done at each production	Specifies what calculation is to be done at each production and at what time they must be done
9	Left to right evaluation	Left to right evaluation
10	Used to know the value of nonterminals	Used to generate Intermediate Code

Abstract Syntax Tree:

Abstract Syntax Tree: It is a condensed form of parse trees. Normally operators and keywords appear as leaves in parse tree, but in an abstract syntax tree they are associated with the interior nodes that would be the parent of those leaves in the parse tree.

- Useful for representing language constructs





Parse Tree/Syntax Tree **Abstract Syntax Tree**

Chain of single production of parse tree are collapsed into one node with the operators moving up to become the node in abstract syntax tree.

Constructing Abstract Syntax tree for expression

Each node in an abstract syntax tree can be implemented as a record with several fields.

operators: one field for operator, remaining fields ptrs to operands **mknode(op, left, right)**

identifier: one field with label id and another ptr to symbol table **mkleaf(id, entry)**

number: one field with label num and another to keep the value of the number. **mkleaf(num, val)**

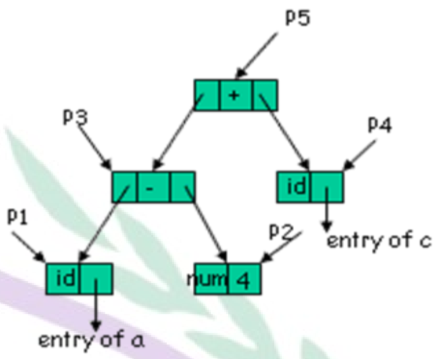
Example of Abstract Syntax Tree

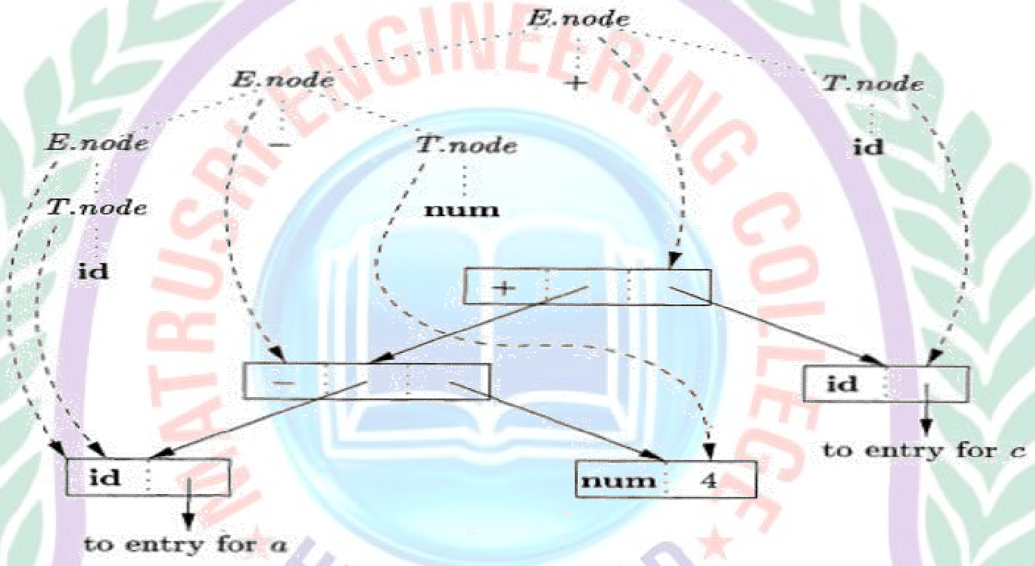
A syntax directed definition for constructing syntax tree

Production	**Semantic Rules**
E -> E1 + T	E.node=new mknode('+', E1.node,T.node)
E -> E1 - T	E.node=new mknode('-', E1.node,T.node)
E -> T	E.node = T.node
T -> (E)	T.node = E.node
T -> id	T.node = new mkleaf(id,id.entry)
T -> num	T.node = new mkleaf(num,num.val)

the following Sequence of function calls create a syntax tree for **a – 4 + c**

P 1 = mkleaf(id, entry.a)
P 2 = mkleaf(num, 4)
P 3 = mknode(-, P 1 , P 2)
P 4 = mkleaf(id, entry.c)
P 5 = mknode(+, P 3 , P 4)





Syntax tree for $a - 4 + c$

1) p_1 = **new** *Leaf*(**id**, *entry-a*);
2) p_2 = **new** *Leaf*(**num**, 4);
3) p_3 = **new** *Node*('−', p_1, p_2);
4) p_4 = **new** *Leaf*(**id**, *entry-c*);
5) p_5 = **new** *Node*('+', p_3, p_4);

SYMBOL TABLE

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program.

A symbol table is a major data structure used in a compiler:

- It associates attributes with identifiers used in a program. For instance, a type attribute is usually associated with each identifier.

A symbol table is a necessary component. Definition (declaration) of identifiers appears once in a program. Use of identifiers may appear in many places of the program text. Identifiers and attributes are entered by the analysis phases when processing a definition (declaration) of an identifier.

In simple languages with only global variables and implicit declarations:

- The scanner can enter an identifier into a symbol table if it is not already there.

In block-structured languages with scopes and explicit declarations:

- The parser and/or semantic analyzer enter identifiers and corresponding attributes

Symbol table information is used by the analysis and synthesis phases:

- To verify that used identifiers have been defined (declared)
- To verify that expressions and assignments are semantically correct – type checking
- To generate intermediate or target code

Symbol Table Structure:

Assign variables to storage classes that prescribe scope, visibility, and lifetime

  - scope rules prescribe the symbol table structure

- scope: unit of static program structure with one or more variable declarations
  - scope may be nested
- Pascal: procedures are scoping units
- C: blocks, functions, files are scoping units

- Visibility, lifetimes, global variables
    - Common (in Fortran)
- Automatic or stack storage
- Static variables

storage class : A storage class is an extra keyword at the beginning of a declaration which modifies the declaration in some way. Generally, the storage class (if any) is the first word in the declaration, preceding the type name. Ex. static, extern etc.

Scope: The scope of a variable is simply the part of the program where it may be accessed or written. It is the part of the program where the variable's name may be used. If a variable is declared within a function, it is local to that function. Variables of the same name may be declared and used within other functions without any conflicts. For instance,

```
int fun1()
{
   int a;
   int b;
   ....
}

int fun2()
{
   int a;
   int c;
   ....
}
```

Visibility: The visibility of a variable determines how much of the rest of the program can access that variable. You can arrange that a variable is visible only within one part of one function, or in one function, or in one source file, or anywhere in the program.

Local and Global variables: A variable declared within the braces {} of a function is visible only within that function; variables declared within functions are called local variables. On the other hand, a variable declared outside of any function is a global variable , and it is potentially visible anywhere within the program.

Automatic Vs Static duration: How long do variables last? By default, local variables (those declared within a function) have automatic duration : they spring into existence when the function is called, and they (and their values) disappear when the function returns. Global variables, on the other hand, have static duration : they last, and the values stored in them persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.) By default, local variables have automatic duration. To give them static duration (so that, instead of coming and going as the function is called, they persist for as long as the function does), you precede their declaration with the static keyword: static int i; By default, a declaration of a global variable (especially if it specifies an initial value) is the defining instance. To make it an external declaration, of a variable which is defined somewhere else, you precede it with the keyword extern: extern int j; Finally, to arrange that a global variable is visible only within its containing source file, you precede it with the static keyword: static int k; Notice that the static keyword can do two different things: it adjusts the duration of a local variable from automatic to static, or it adjusts the visibility of a global variable from truly global to private-to-the-file.

Contents in a symbol table

Possible entries in a symbol table for each identifier/variable:

- Name: a string.
- Attribute:
    - Reserved word
    - Variable name
    - Type name
    - Procedure name
    - Constant name
- Data type.
- Storage allocation, size, . . .
- Scope information: where and when it can be used.

How names are stored

Fixed-length name: allocate a fixed space for each name

If the size allocated is too little: names must be short.

If the size is too much: waste a lot of spaces.

Variable-length name:

- A string of space is used to store all names.
- For each name, store the length and starting index of each name.

Symbol Table Entries

- each entry for a declaration of a name
- format need not be uniform because information depends upon the usage of the name

- each entry is a record consisting of consecutive words
- to keep records uniform some entries may be outside the symbol table
- information is entered into symbol table at various times
    - keywords are entered initially
    - identifier lexemes are entered by lexical analyzer
- symbol table entry may be set up when role of name becomes clear
- attribute values are filled in as information is available

For each declaration of a name, there is an entry in the symbol table. Different entries need to store different information because of the different contexts in which a name can occur. An entry corresponding to a particular name can be inserted into the symbol table at different stages depending on when the role of the name becomes clear. The various attributes that an entry in the symbol table can have are lexeme, type of name, size of storage and in case of functions - the parameter list etc.

Operations:
The basic operations defined on a symbol table include:
- **allocate** – to allocate a new empty symbol table
- **free** – to remove all entries and free the storage of a symbol table
- **insert** – to insert a name in a symbol table and return a pointer to its entry
- **lookup** – to search for a name and return a pointer to its entry
- **set_attribute** – to associate an attribute with a given entry
- **get_attribute** – to get an attribute associated with a given entry

Other operations can be added depending on requirement

For example, a delete operation removes a name previously inserted

Some identifiers become invisible (out of scope) after exiting a block

Implementation and Management:

Basic Implementation Techniques:
- First consideration is how to insert and lookup names

Variety of implementation techniques
1. Unordered List
2. Ordered List (Array)
3. Binary Search Tree
4. Hash tables

1. **Unordered List**
    - Simplest to implement
    - Implemented as an array or a linked list
    - Linked list can grow dynamically – alleviates problem of a fixed size array Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

2. **Ordered List**
    - If an array is sorted, it can be searched using binary search – O(log2 n)
    - Insertion into a sorted array is expensive – O(n) on average
    - Useful when set of names is known in advance – table of reserved words

3. **Binary Search Tree**
    - Can grow dynamically
    - Insertion and lookup are O(log2 n) on average

4. **Hash Tables and Hash Functions:**
    - A hash table is an array with index range: 0 to TableSize – 1
    - Most commonly used data structure to implement symbol tables
    - Insertion and lookup can be made very fast – O(1)
    - A hash function maps an identifier name into a table index
    - A hash function, h(name), should depend solely on name
    - h(name) should be computed quickly
    - h should be uniform and randomizing in distributing names
    - All table indices should be mapped with equal probability
    - Similar names should not cluster to the same table index.

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

The global symbol table contains names for global variable and procedure names, which should be available to all the child nodes.

To determine the scope of a name, symbol tables are arranged in hierarchical structure. 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.