unit-4

Semantic Analysis:

* Semantic analysis phase is the Third phase of the compiler. Semantic analysis checks the source program for semantic Errors.

* It uses the hierarchical structure determined by the syntax analysis phase to identify the operators and operands of Expressions and statements.

* Semantic analysis performs type checking, ie, it checks that, whether each operator has operands that are permitted by the source language specification.

Ex:- If a Real number is used to index an array i.e, a[1.5] then the compiler will Report an error. This error is handled during semantic analysis.

Semantic errors:

1, Missing parenthesis in expr
2, Matching if-else statement
3, Typechecking etc.

* After syntax Analysis, It produce hierarchical structure, which is determined by the parse tree, called AbST.

ie, It is nothing but a parse tree, variables are not shown in parse tree.

Syntax-directed translation:- (SDT)

* till Now As we know how an input string is syntatically checked by the syntax analyser, now, we should know how to analyse the input semantically.

Need of semantic analysis:

The semantic analysis is done in order to obtain the precise meaning of programming construct.

SDT it is, to translate a programming language construct, i.e, In Compiler Design, along with a grammar we have to give some meaningful Rule (Formal Notation).

i.e, **Grammar + Semantic Rule = SDT**

So, a compiler has to keep track of syntax analysis, I-C-G, C-O, symbol table Generation, Expression evaluation many things can be done parallel at the time of parsing.

(or)

* SDT is to construct a Parse Tree (or) syntax Tree, and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree.

* A formal Notation list called as "syntax-directed definitions" is used for specifying translations for programming language constructs.

* 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.

* Set of attributes can be associated with each terminal and Non-terminal symbol. and attribute can be a string, number, a type, a memory location & anything else.

* we shall deal with TWO Kinds of attributes for non-terminals.

1, synthesized Attribute

2, Inherited Attribute.

synthesized Attribute:

Consider $X \rightarrow \alpha$ be a context free Grammar and $a := f(b_1, b_2 \cdots b_k)$ where 'a' is the attribute.

then, (1) the Attribute 'a' is called synthesized attribute of X and $b_1, b_2, .. b_k$ are attributes belonging to the production symbols.

the value of synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

* A synthesized Attribute is an attribute of the non-terminal on the left-hand side of a production.

(2) the attribute 'a' is called inherited attribute of one of the grammar symbol on the R.H.S of the production i.e, (α) and $b_1, b_2 \dots b_k$ are belonging to either X or α.

In herited attributes can be computed from the values of the attributes at the siblings and parent of that node.

* An attribute of a non-terminal on the right hand side of a production is called an inherited attribute.

SDT for evalution of Expression:

$E \rightarrow E_1 + T$ { E.value = E_1.value + T.value }

$\quad | \; T$ { E.value = T.value }

$T \rightarrow T_1 * F$ { T.value = T_1.value * F.value }

$\quad | \; F$ { T.value = F.value }

$F \rightarrow$ num { F.val = num.lvalue }

the above Grammar, which is used to evalute some expression

2+3*4

Syntax-Directed Translation schemes:-

* The syntax directed translation scheme is a context-free-grammar, It is used to evaluate the order of semantic Rules.

* In translation scheme, the semantic Rules are embedded within the Right side of the productions.

* The position at which an action is to be executed is shown by enclosed between braces. it is written within the Right side of the productions.

* So, a translation scheme is like a syntax directed definition, except that the order of evaluation of semantic Rules is explicitly shown.

* The order in which, the actions are executed is the order in which they appear during a depth-first traversal of a parse-tree. (left-right depth first order)

$N = L_1 \cdot L_2$ { $N.dval = L_1.dval + L_2.dval / 2^{L_2.c}$; } 11.01

$L = L_1 B$ { $L.c = L_1.c + B.c$, $L.dval = L_1.dval + B.dval$; }

$| B$ { $L.c = B.c$; $L.dval = B.dval$; }

$B \to 0$ { $B.c = 1$; $B.dval = 0$ }

$B \to 1$ { $B.c = 1$; $B.dval = 1$ }

Syntax Directed Translation scheme

E → E+T { printf("+"); } — ①

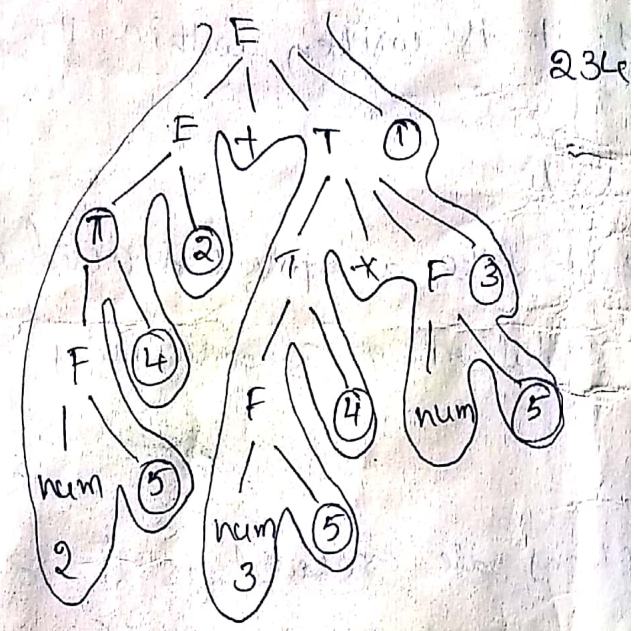
| T { } — ②

T → T*F { printf("*"); } — ③

| F { } — ④

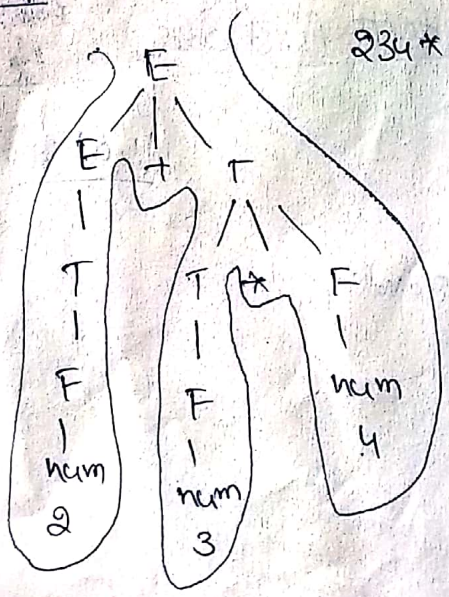
F → num { printf(num.val); } — ⑤

Top down parse:



234*+

bottom-up

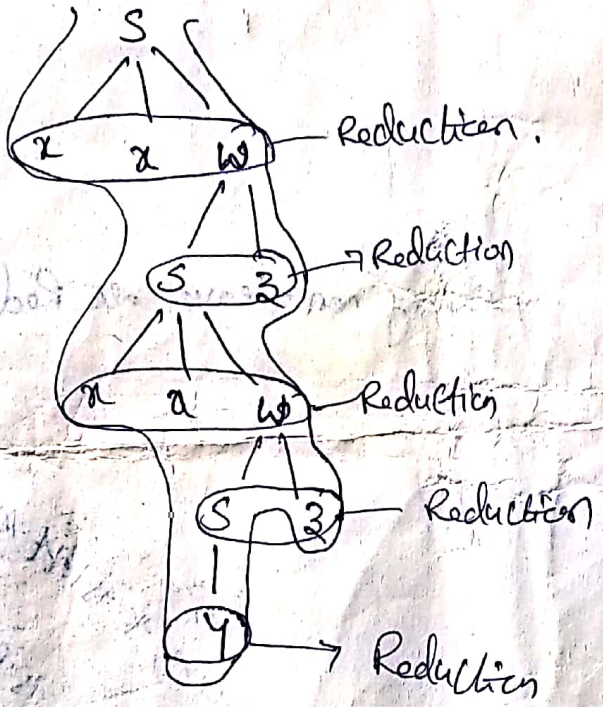


Translation scheme for Generating a string from SDT:

S → xxw { printf(1); }
| y { printf(2); }
w → Sz { printf(3); }

String w = xxxyzz.
find the output of this string.

If we use LR parser, In order to Generate a particular string from SDT, how will a tree active.



2 3 1 3 1 output.

E → E#T { E.val = E.val * T.val; }
| T { E.val = T.val; }
T → T&F { T.val = T.val + F.val }
| F { T.val = F.val; }
F → num { F.val = num.val; }

w = 2#3&5#6&4 — output — ?
= 2 * (3+5)/1 * (6+4)/2
= ((2*8)*10) = 160

$E \to E * T$ { E.val = E.val * T.val; }

$\quad | T$ { E.val = T.val; }

$T \to F - T$ { T.val = F.val - T.val; }

$\quad | F$ { T.val = F.val; }

$F \to$ (2) { F.val = 2; }

$\quad |$ (4) { F.val = 4; }

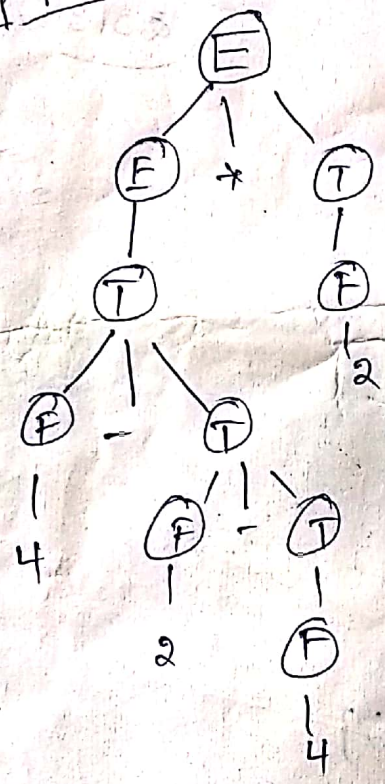
lexeme (if we observed only lexemes are using directly).

$$w = \left(\frac{\overset{2}{\overbrace{4 - (2-4)}}}{\underset{3}{\underbrace{\quad\quad}}} \times 2 \right)$$

$4 - (-2) \times 2$

$6 \times 2 = 12$

bottom-up parse:



Every non-leaves are Reductions.

$E \to E * T$ (it is left Recursive, * is left Associative)

$T \to F - T$ (it is Right Recursive, - is Right Associative)

(- has higher precedence than *.)

Std to build syntax Tree:

$E \rightarrow E_1 + T$ { E.nptr = mknode (E_1.nptr, '+', T.nptr); }

| T { E.nptr = T.nptr; }

$T \rightarrow T_1 * F$ { T_1.nptr = mknode (T_1.nptr, '*', F.nptr); }

| F { T.nptr = F.nptr; }

$F \rightarrow$ id { F.nptr = mknode { null, id.name, null }; }

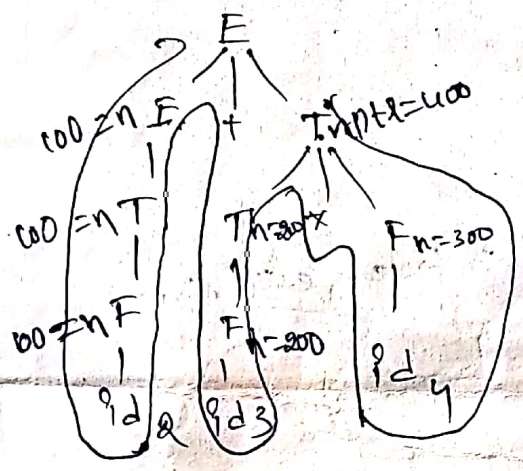
If I have an expn. 2+3*4. (value)

nptr – nodePointer

↓

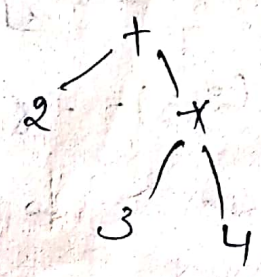
attribute

mknode – Function



Parse Tree (or)

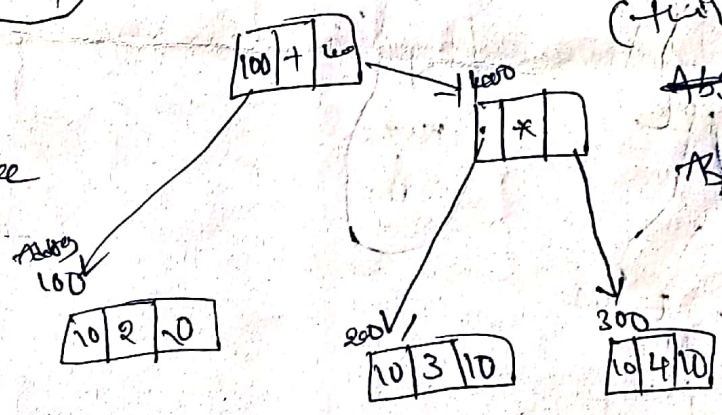
Concrete syntax Tree



Abstract syntax Tree

(Tree is nothing but ~~Abs~~ Data structure ✓

Abstract Parse Tree).



Type check

$E \to E_1 + E_2$ { if ($E_1.type == E_2.type$) && ($E_1.type == int$)) then $E.type = int$ else error; }

| $E_1 == E_2$ { if ($E_1.type == E_2.type$) && ($E_1.type = int/boolean$)) then $E.type = boolean$ else error; }

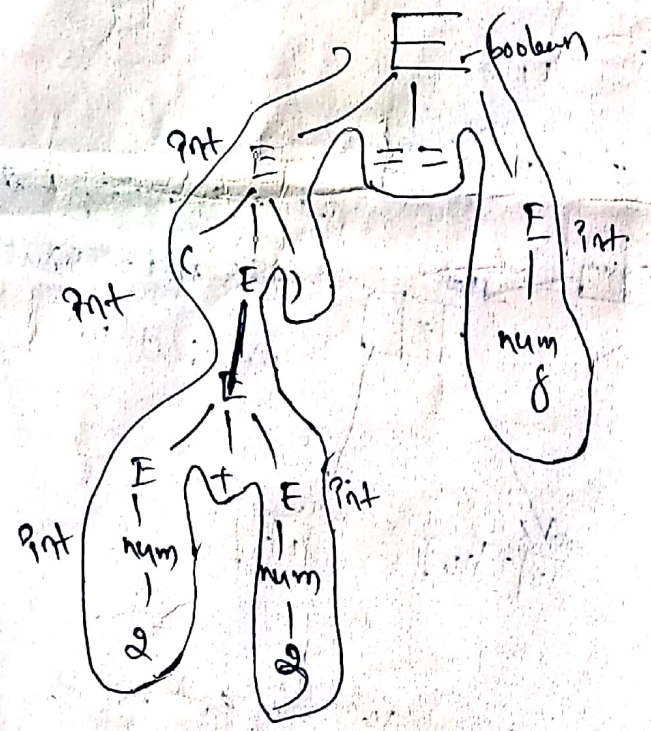
| (E_1) { $E.type = E_1.type$ }

| num { $E.type = int$; }

| True { $E.type = boolean$; }

| False { $E.type = boolean$; }

Ex: I have take an example $(2+3) == 8$

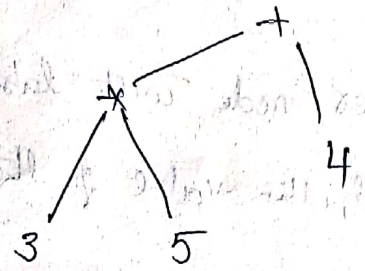


So, it is boolean expression.

Construction of Syntax Tree :-

* A syntax Tree, It is a compressed Representation of a parse Tree.
* In which operators are appear as parent node, and operands are appear as children nodes of that operators in syntax Tree.
* Syntax-directed definitions can be used to specify the construction of syntax trees and other graphical Representations of language constructs.
* syntax tree is one of the Intermediate Representation.
* In syntax tree, operators and keywords do not appear as leaves, but rather they are associated with Interior node that would be the parent of those leaves in the parse Tree.

The syntax Tree for the expression 3*5+4 is given by



* Construction of a syntax tree for an expression is similar to the translation of the expression into postfix form.

* Subtrees are constructed for sub expression by creating a node for each operator and operand.

* In the node for an operator, one field identifies the operator and the remaining fields contain pointers to the nodes for the operands.

* The operator is called the label of the node.

The following functions are used to create the nodes of syntax trees for expressions.

1) mknode(op, left, right): creates an operator node with label op and two fields containing pointers to left and right operands.

2) mkleaf(id, entry): creates an identifier node with label id and a field containing entry, which is a pointer to the symbol-table entry for the identifier.

3) mkleaf(num, val): creates a number node with label num and a field containing val, the value of the number.

For example, To construct a syntax-tree for the expression, a-4+c, the sequence of function calls

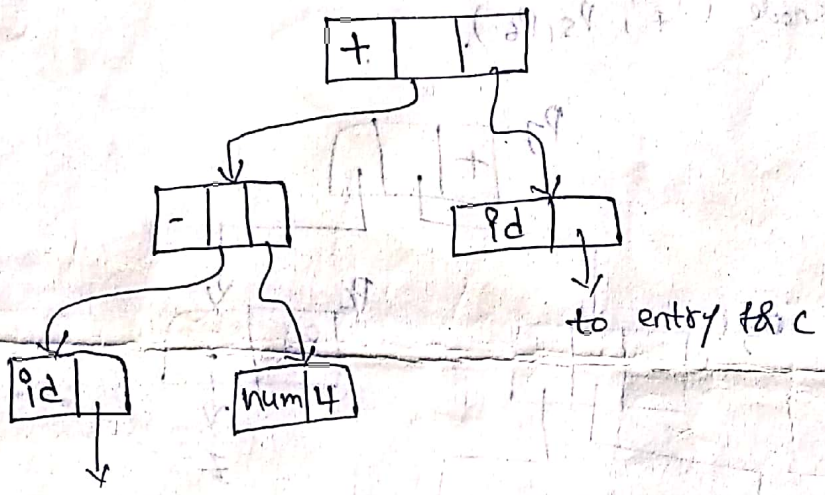
— $P_1, P_2 \cdots P_5$ are the pointers to nodes

— entry a and entry c are pointers to the symbol-table

entries for identifiers a and c, respectively.

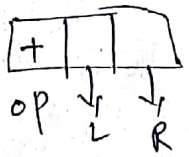
1) $P_1 := mkleaf(id, entry\ a)$;
2) $P_2 := mkleaf(num, 4)$;
3) $P_3 := mknode('-', P_1, P_2)$;
4) $P_4 := mkleaf(id, entry\ c)$;
5) $P_5 := mknode('+', P_3, P_4)$;

the syntax tree is shown in below

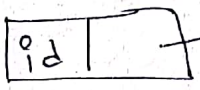


Ex: Constructing a syntax tree for an expression Grammar

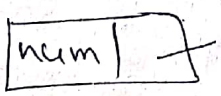
$E \to E_1 + T$
$E \to E_1 - T$
$E \to E_1 * T$
$E \to T$
$T \to id$
$T \to num$

1. 

1. mknode(op, left, Right)

2. 

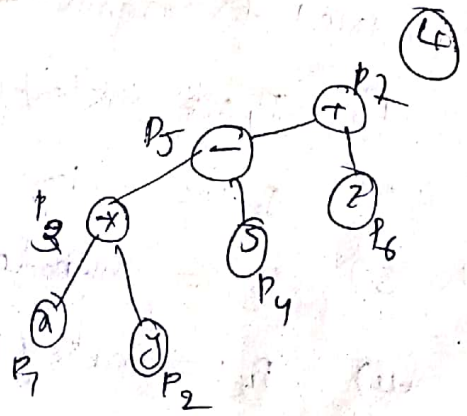
2. mkleaf (id, entry)

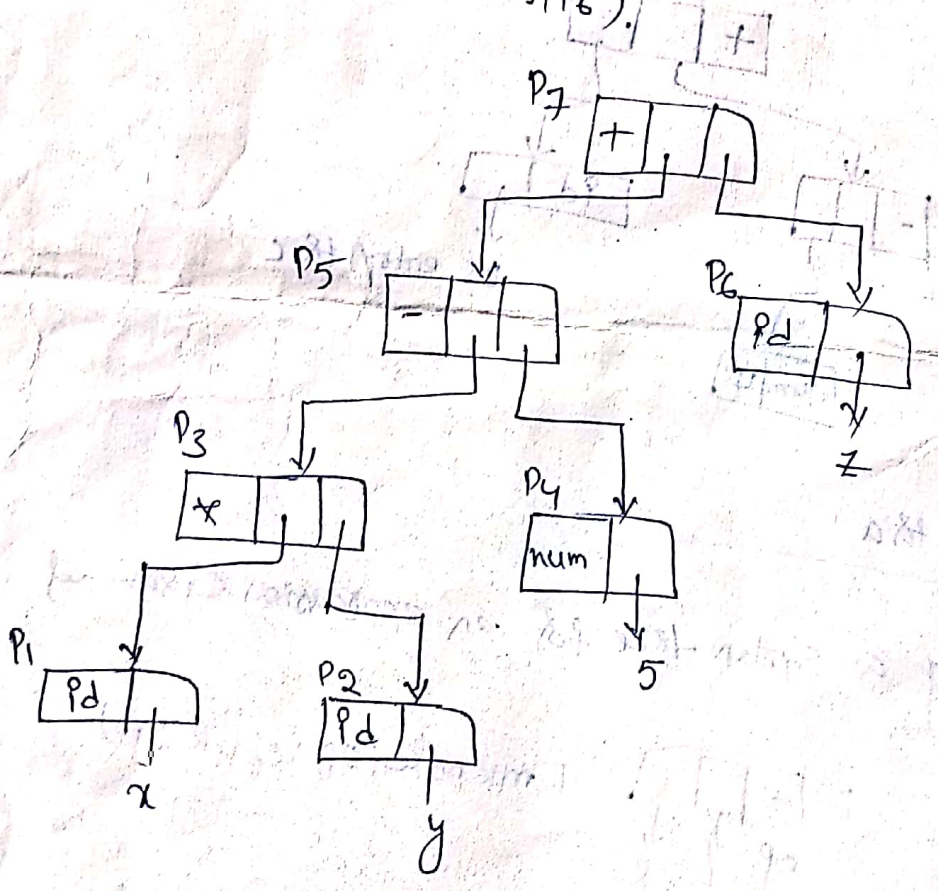
3 

3. mkleaf (num, val)

Ex: x*y - 5 + z ⇒ xy*5 - z +
infix postfix

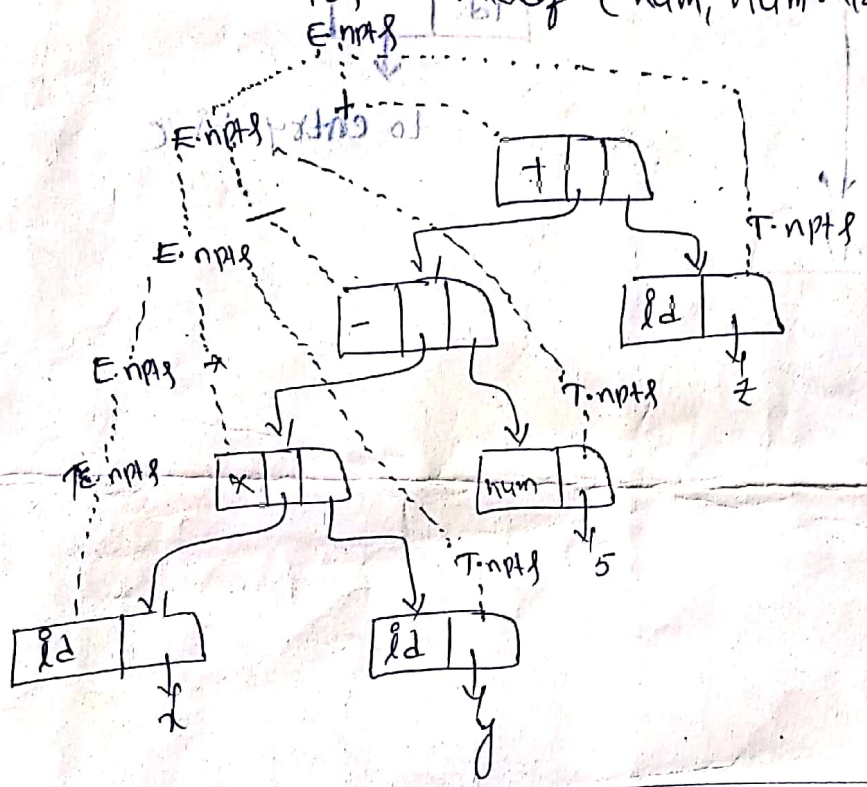
P_1 = mkleaf (Id, ptr to entry x)
P_2 = mkleaf (Id, ptr to entry y)
P_3 = mknode ('*', P_1, P_2)
P_4 = mkleaf (num, 5)
P_5 = mknode ('-', P_3, P_4)
P_6 = mkleaf (Id, ptr to entry z)
P_7 = mknode ('+', P_5, P_6).





SDD

$E \to E_1 + T$	$E.nptr = mknode('+', E_1.nptr, T.nptr)$
$E \to E_1 - T$	$E.nptr = mknode('-', E_1.nptr, T.nptr)$
$E \to E_1 * T$	$E.nptr = mknode('*', E_1.nptr, T.nptr)$
$E \to T$	$E.nptr = T.nptr$
$T \to id$	$T.nptr = mkleaf(id, id.ptr\_entry)$
$T \to num$	$T.nptr = mkleaf(num, num.val)$

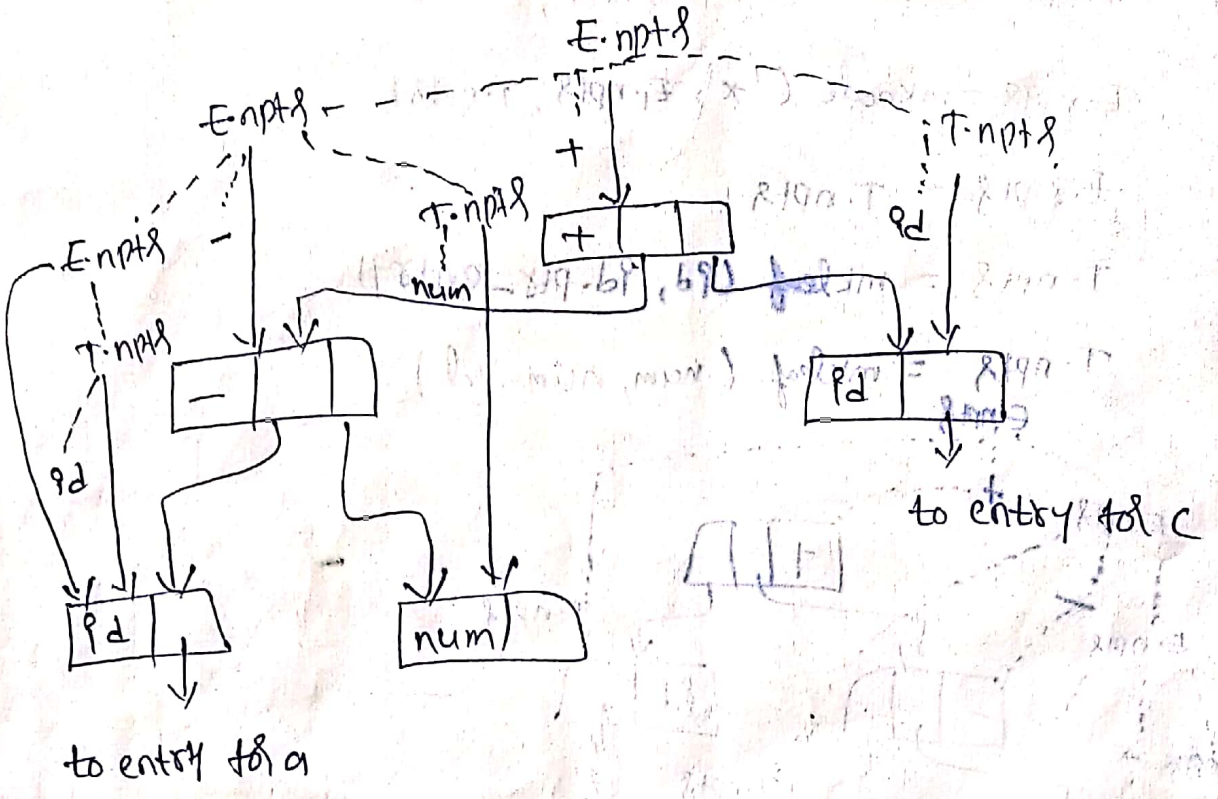


Syntax-directed Definition for constructing a syntax tree for an expression a-4+c

$E \to E_1 + T$	{ $E.nptr := mknode('+', E_1.nptr, T.nptr)$
$E \to E_1 - T$	{ $E.nptr := mknode('-', E_1.nptr, T.nptr)$
$E \to T$	{ $E.nptr := T.nptr$ }
$E \to (E)$	{ $T.nptr := E.nptr$ }

E → id { T.nptr := mkleaf (id, id.entry)

T → num { T.nptr := mkleaf (num, num.val)



Dependency Graph

Annotated Parse Tree:-

* The Rules of an SDD are applied by first constructing a parse tree and then using the Rules to 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.
* How do we construct an annotated parse tree? In what order do we evaluate attributes? Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends.
* If for Example, 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;

Example:

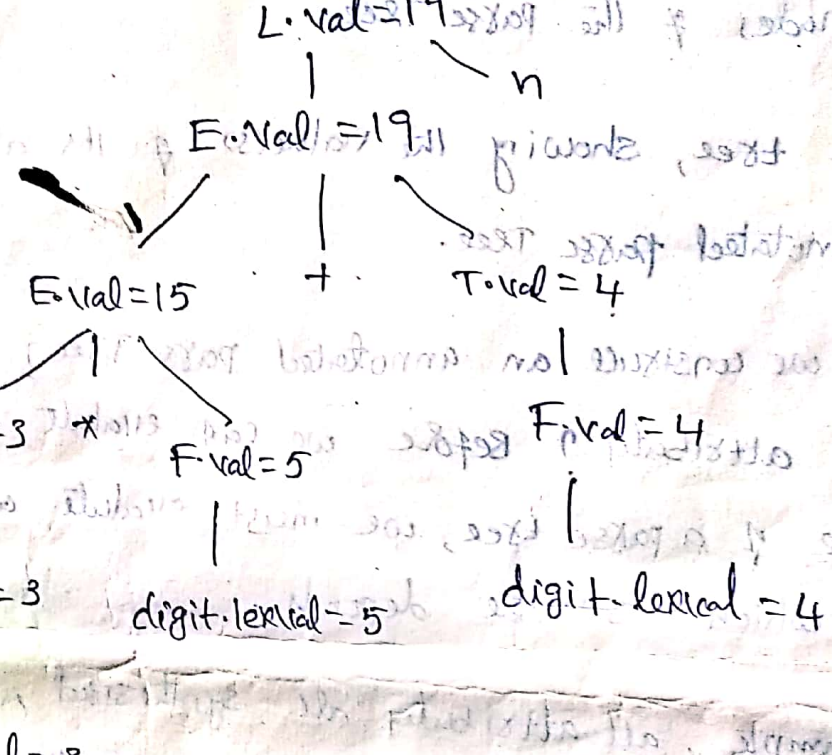
$L \rightarrow E\ n$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

dependence Graph:-

→ The Rule for production, $L \to En$, sets L.val to E.val, which we shall see is the numerical value of the entire expression.

Ex: 3*5+4n

Annotated parse Tree for 3*5+4n



Dependency Graph:

* A dependency Graph depicts the flow of information among the attribute instances in a particular parse tree.

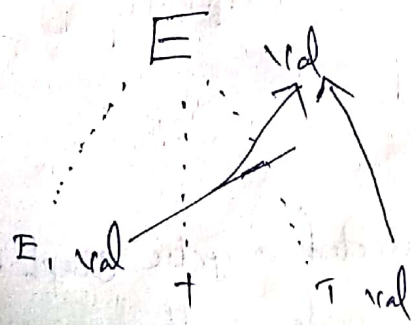
* an edge from one attribute instance to an other, means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.

* For each parse tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.

*** the directed graph that represents the interdependencies b/w the synthesized and inherited attribute at nodes in the parse tree.

Consider the following production Rule:

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



E.val is synthesized from $E_1.val$ and $E_2.val$.

Synthesized + Inherited Attributes:

production	Semantic Rule
$D \rightarrow TL$	L.in := T.type
$T \rightarrow int$	T.type := 'integer'
$T \rightarrow real$	T.type := 'real'
$L \rightarrow L_1, id$	L.in := L.in; addtype (id.entry, L.in)
$L \rightarrow id$	addtype (id.entry, L.in)

Synthesized : T.type, id.entry

Inherited : L.in

L-attribute

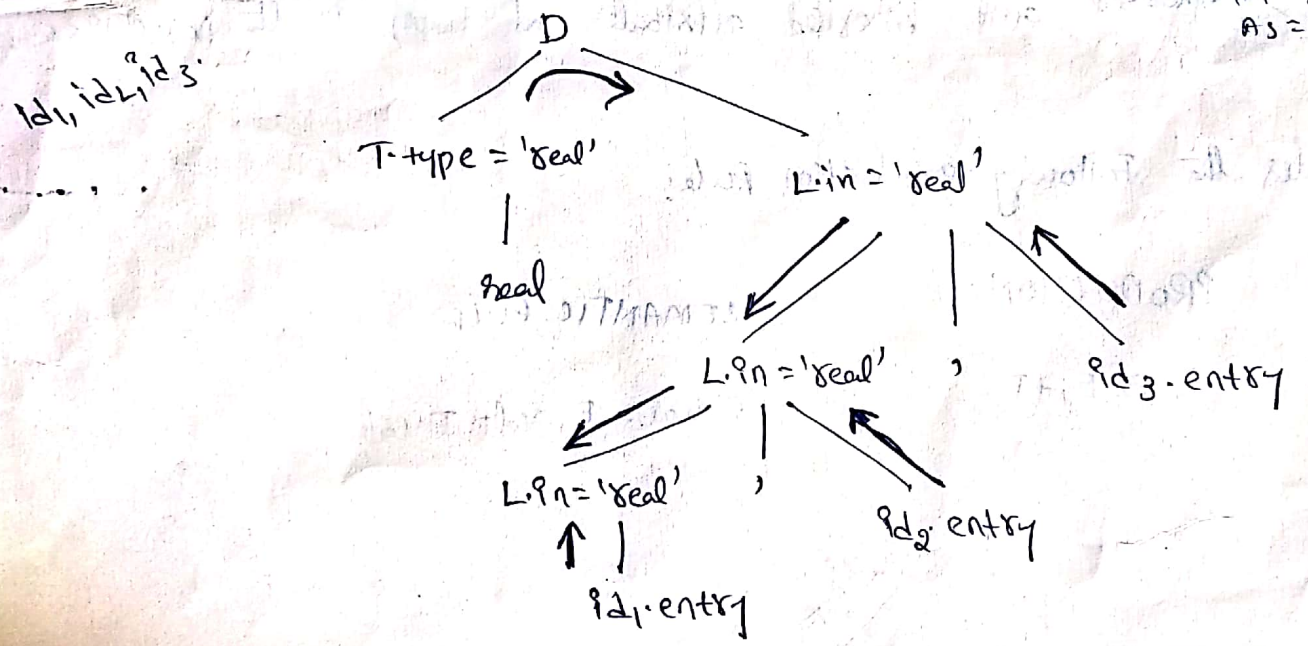
Ex:

$A \rightarrow XY$

$X.i = A.i$ $\quad$ $A.s = Y.j$

$Y.i = X.s$

Dependency Graph for S-attribute and L-attribute

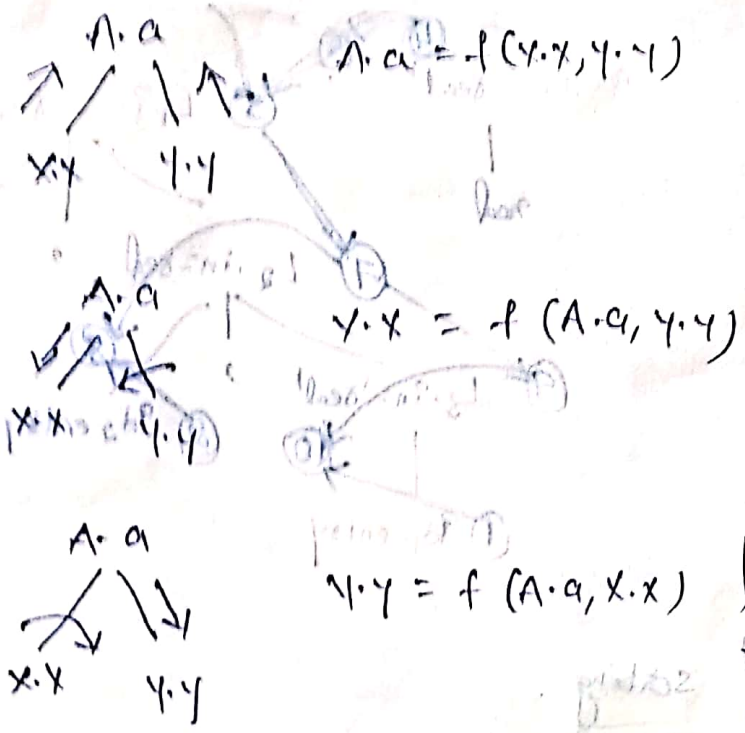


Evaluation Order:

A topological sort of a directed acyclic graph (DAG) is any ordering $m_1, m_2 \ldots, m_n$ of the nodes of the graph, such that if $m_i \rightarrow m_j$ is an edge then m_i appears before m_j.

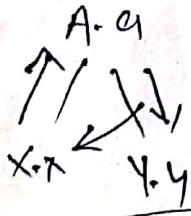
dependence graph:-

$A \rightarrow XY$



Direction of → value dependence.

cyclic dependence graph



to avoid computation difficulties, it has to be a acyclic graph

dependence of a Graph, shows the direction of dependence of one variable (or) one attribute on the other.

---

where the non-terminals.

D represents declaration

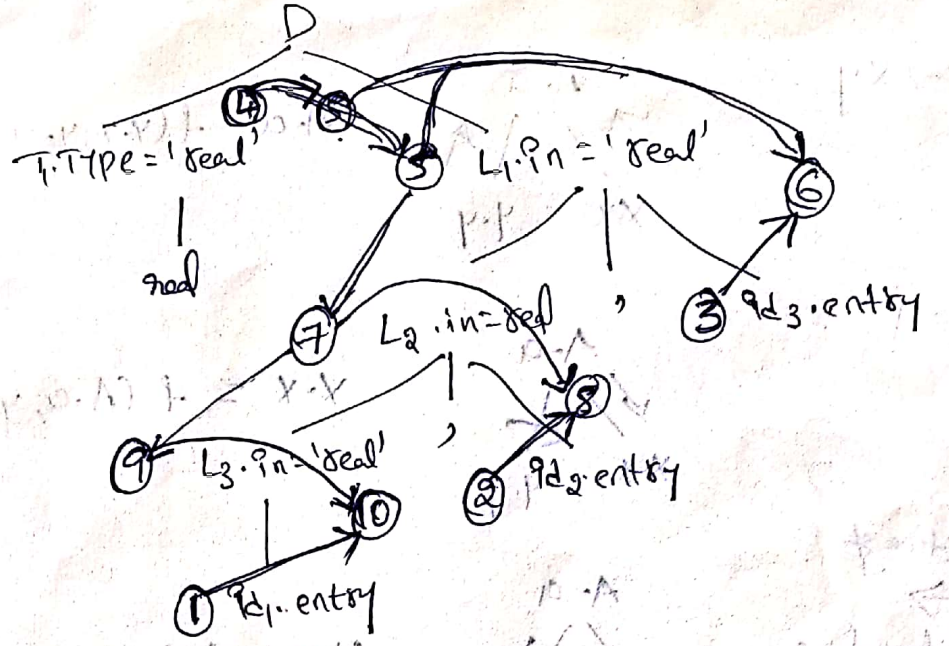
T Represents the keywords int (or) real,

L Represents list of identifiers.

- the declaration is generated by D which consists of keywords int (or) real and list of identifiers.

- the non-terminal T has a synthesized attribute "type", whose value is determined by the keyword in the declaration.

- the non-terminal L has an inherited attribute "in", whose value is defined interms of declaration of T.type.



Topological sorting.

1, Get id_1.entry
2, Get id_2.entry
3, Get id_3.entry
4. T_1.type = 'real'
5. L_4.in = T_1.type
6. addtype (id_3.entry, L_4.in)
7. L_2.in = L_4.in
8. addtype (id_2.entry, L_2.in)
9. L_3.in = L_2.in
10. addtype = (id_1.entry, L_3.in)

Semantic Rule
SDT scheme

$D \rightarrow TL$

$D \rightarrow T \{L.in := T.type\} L$

$T \rightarrow int \{T.type = 'integer'\}$

$T \rightarrow real \{T.type = 'real'\}$

$L \rightarrow \{L_1.in = L.in\} L_1, id \{addtype(id.entry, L.in)\}$

$L \rightarrow id \{addtype(id.entry, L.in)\}$

SDT to Generate three address Code:-

Statement can be identifier and an Expression.

$S \rightarrow id = E$ { gen(id.name = E.place); }

$E \rightarrow E_1 + T$ { E.place = newTemp(); gen(E.place = E_1.place + T.place); }

/T { E.place = T.place; }

$T \rightarrow T_1 * F$ { T.place = newtemp(); gen(T.place = T_1.place * F.place); }

/F { T.place = F.place; }

$F \rightarrow id$ { F.place = id.name; }

Statement $x = a + b * c$

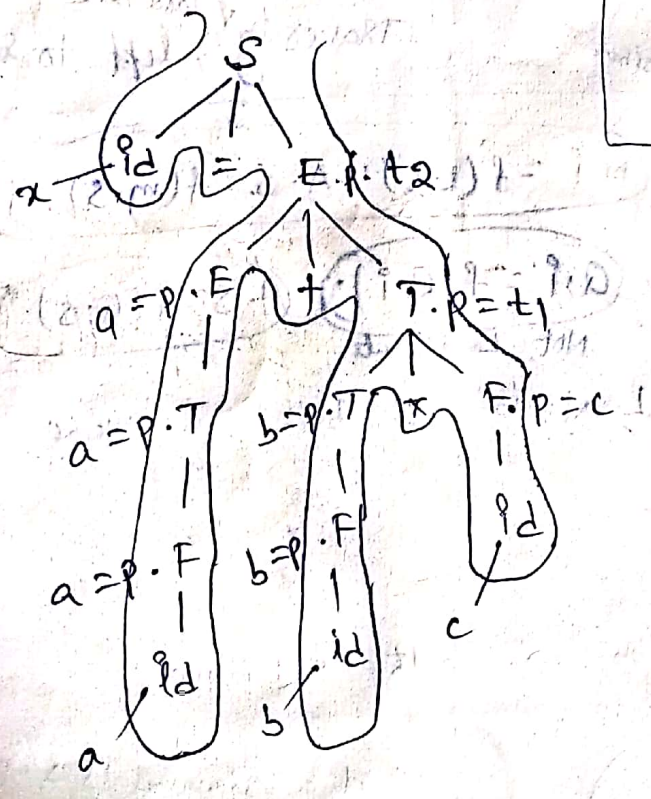
↓ Identifier ↓ Expression

$x = a + b * c$

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$



Bottom up - Evaluation of S-attribute SDT and L-attribute SDT:

S-attribute SDT

1) uses only synthesized Attribute

2) Semantic actions are placed at right end of production
   $A \rightarrow B \{\}$

3) Attributes are evaluated during BUP

L-attribute SDT

1) uses both I-attribute and S-attribute. Each I-attribute is restricted to inherit either from parent or left sibling only.
   Ex: $A \rightarrow XYZ \{ Y.S = A.S, Y.S = X.S, Y.S = Z.S \}$

2) Semantic Actions are placed anywhere on R.H.S
   $A \rightarrow \{\} BC$
   $| D \{\} E$
   $| F G \{\}$

3) Attributes are evaluated by depth first traversal of parse tree, left to right.

Syntax Directed Translation Scheme for Typechecking:-

(we can not add & boolean variable)

$E \to E_1 + E_2$ { if (E_1.type == E_2.type) && (E_1.type = int)) then E.Type = int

/ $E_1 == E_2$ { if (E_1.type == E_2.type) && (E_1.type = int/boolean)) then E.type = boolean else error; }

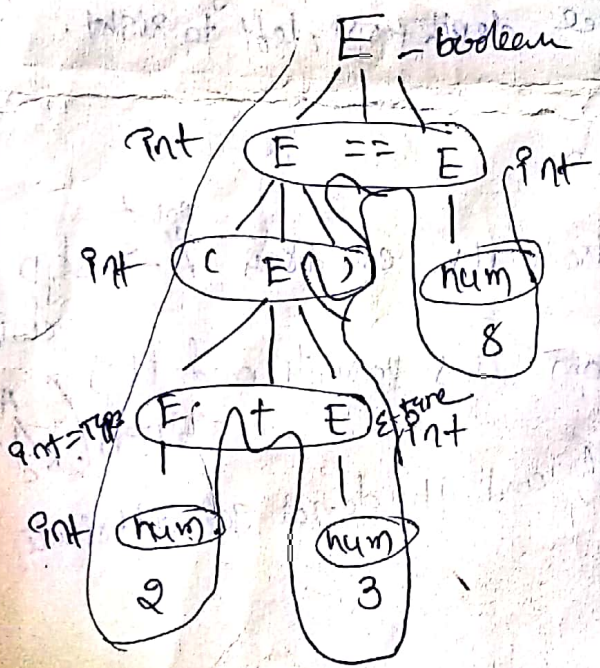
/ (E) { E.type = E_1.Type } (what is the type of inside E, which is equal to outside E)

/ num { E.type = int; } (ie, it is identities)

/ True { E.type = boolean; }

/ False { E.type = boolean; }

Ex: (2+3) == 8, so it's a boolean expression. — it is meaningful & not.

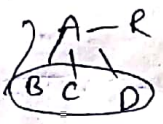


S-attribute SDT

1) uses only synthesized attribute

2) Semantic actions are placed at right end of production
A → BCC { }
i.e, (POSTFIX SD.T)

3) Attributes are evaluated only Bottomup parser.



L-attribute SDT.

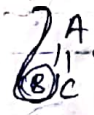
1) uses both Inherited attribute and synthesized Attribute is restricted to Inherit either from parent (or) left sibling only.

Ex: A → XYZ { Y.S = A.S, Y.S = X.S, Y.S = Z.S; }

2) semantic actions are placed anywhere on R.H.S.

A → { } B C
| D { } E
| F G { }

3) Attributes are evaluated by traversing parse tree depth first, left to right.



Synthesized: It a node taking attributes from its childrens

Inherited: it a node taking attributes from it's parent and its siblings

L-attribute: it uses both S-Attr, & Inherited, and it donot get attribute from it's Right siblings.

A → BCD (if i have a production)

A can have attribute $A.S = f(B.S, C.S, D.S)$, if it is getting attribute by Function $f(B.S, C.S, D.S)$, which means A is taking value from it's childrens, B, C, D.

if C is taking Attribute from its parent (or) its sibling. (B C D)
i.e, $C.i = A.i$, $C.i = B.i$, $C.i = D.i$