__/__/__

SEMANTIC ANALYSIS

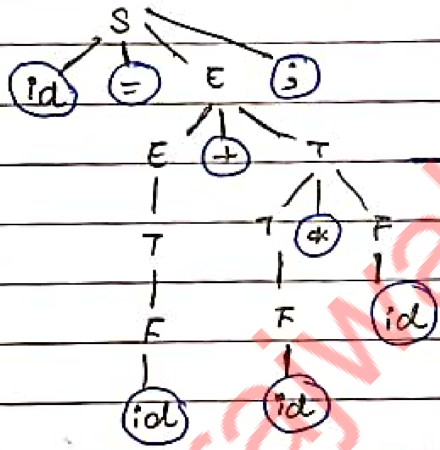
INTRODUCTION TO SDTS

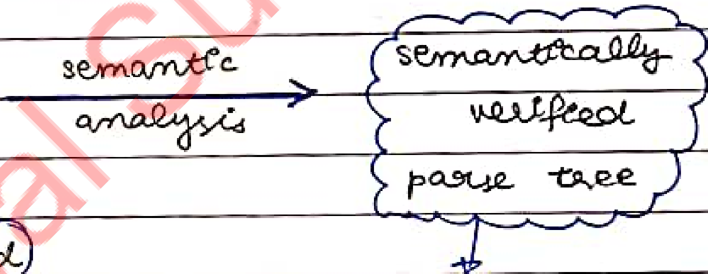
outcome :

(v) understanding syntax directed translation schemes (SDTS)

(v) How to produce semantically verified parse trees

Semantic Analyzer :





eg : The first id must be a variable, while the other 3 must be values.

Syntax Directed Translation scheme :

Using parsers, we produce parse trees, we use parse trees to verify syntax, using the production rules of a grammar.

In SDTS, alongside the grammar, we are going to associate some semantic rules

	Grammar	Semantic Rules
①	$E \rightarrow E + T$	{ E.value = E.value + T.value }
②	$E \rightarrow T$	{ E.value = T.value }
③	$T \rightarrow T * F$	{ T.value = T.value * F.value }
④	$T \rightarrow F$	{ T.value = F.value }
⑤	$F \rightarrow id$	{ F.value = id.value }

__/__/__

while producing the parse tree, simultaneously, we can get to do a lot of different stuff like :

- ✓ code generation.
- ✓ symbol table updation
- ✓ Expression evaluation
  (most important job of the syntax analysis phase)

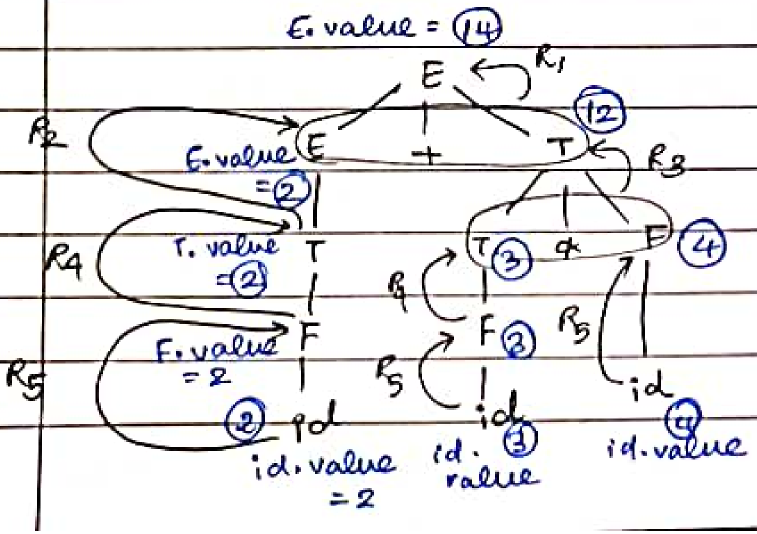
$\boxed{\text{Grammar + semantic Rules} \Rightarrow \text{SDT}}$

Given SDT previously, and the expression
2 + 3 * 4 → evaluates to 14, considering higher precedence of multiplication over addition,

using SDT, check if the expression evaluates to 14 or not.

Procedure :

1. construct the parse tree and associate attribute to every non-terminal.
2. During the (top bottom and left right) traversal, whenever we come across a (production) reduction, we are to go to the production rule and carry out the associated semantic action.



Summary :

- ✓ understanding SDTS
- ✓ How to produce semantically verified parse trees.

_/_/_

SDT - INFIX TO POSTFIX

Outcome :

(V) Top-Down parsing of SDT.

(Ⓥ) Bottom-up parsing of SDT.

SDT

Grammar	Semantic Rules
① $E \rightarrow E + T$	① { printf ("+"); }
② $E \rightarrow T$	② { }
③ $T \rightarrow T * F$	③ { printf ("*"); }
④ $T \rightarrow F$	④ { }
⑤ $F \rightarrow id$	⑤ { printf (id.lval); }

SDT - Infix to postfix (Top-Down parsing)

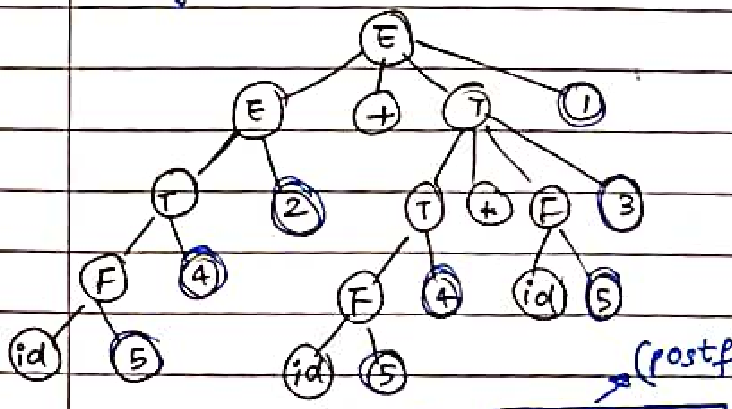
Procedure :

① During Top-Down parsing, the semantic action is first given a value and is considered as the rightmost element in the RHS of the production.

eg $E \rightarrow E + T$ ①
$E \rightarrow T$ ② } (SDT)

② During traversal, action is taken whenever any semantic action number is encountered.

Eg: 2 + 3 * 4



Performing a Top-Down parsing generates the postfix equivalent of the given infix expression.

(postfix)

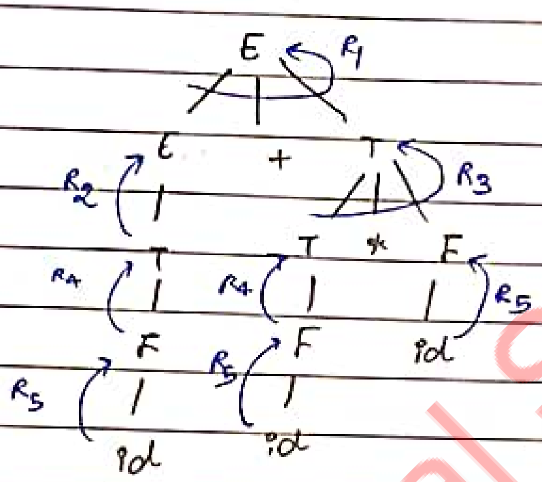
o/p $\boxed{2\ 3\ 4 * +}$

__/__/__

SDT - Infix to postfix (Bottom-Up parsing)

Procedure :

1. Bottom-up parsers mainly focus on reduction
2. During reduction, the appropriate production is chosen and the associated action is performed.



output :

2 3 4 * +

summary :

- ✓ Top-Down parsing of SDT.
- ✓ Bottom-up parsing of SDT

SDT - SOLVED PROBLEMS (SET 1)

outcome :

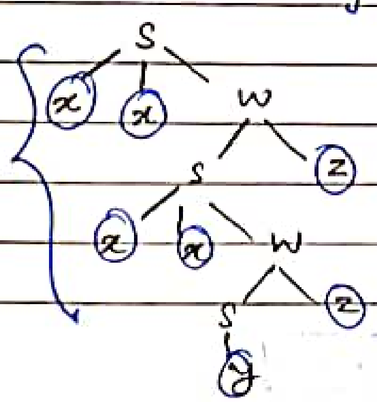
- ✓ 3 problems on determining the output of SDTs

Q: Determine the output of the following SDT wrt LR parsers

1. S → xxW { printf("1"); }
2. S → y { printf("2"); }
3. W → Sz { printf("3"); }

Input : xxxxyzz

parse tree



output :

2 3 1 3 1

__/__/____

Q : consider the following SDT :

① $E \rightarrow E * T$ { E. val = E. val * T. val }

② $E \rightarrow T$ { E. val = T. val }

③ $T \rightarrow F - T$ { T. val = F. val − T. val }

④ $T \rightarrow F$ { T. val = F. val }

⑤ $F \rightarrow 2$ { F. val = 2 }

⑥ $F \rightarrow 4$ { F. val = 4 }

If input is w : 4 − 2 − 4 * 2

(a) what is the output ?

(b) Find the number of reductions during bottom-up parsing.

On looking at the given grammar :

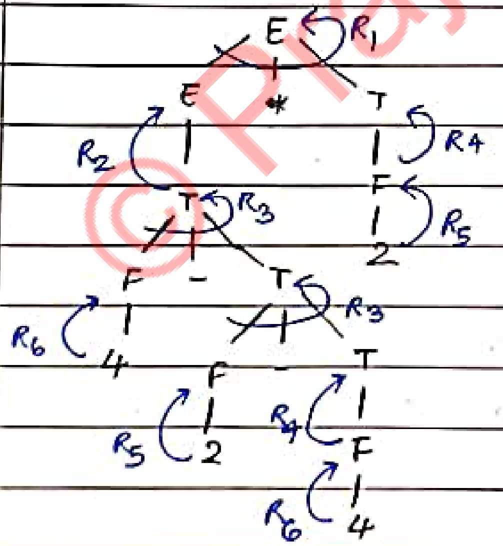
✓ '−' has higher precedence & is right associative

✓ '*' is left associative.

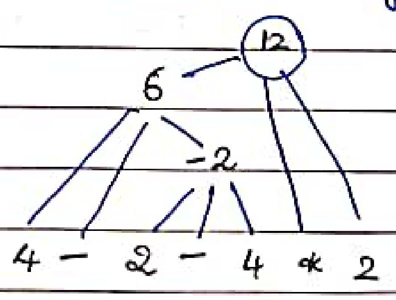
$$w : (4 - (2 - 4)) * 2 = (4 - (-2)) * 2$$

$$= (4 + 2) * 2 = 6 * 2 = 12$$

12 → output



Bottom-up parsing :



R6,	R5,	R6,	R4,	R3,
R3,	R2,	R5,	R4,	R1

↓

(reductions in order)

no. of reductions = 10

Q: considering the following SDT and the input string, generate the output :

$E \rightarrow E \# T$ { E.val = E.val * T.val }
$E \rightarrow T$ { E.val = T.val }
$T \rightarrow T \& F$ { T.val = T.val + F.val }
$T \rightarrow F$ { T.val = F.val }
$F \rightarrow id$ { F.val = id.val }

Input: 2 # 3 & 5 # 6 & 4

(or) 2 * 3 + 5 * 6 + 4

* → left associative
+ → left associative, higher precedence

$$\left(2 * (3+5)\right) * (6+4)$$
$$= (2 * 8) * 10 = 16 * 10 = \boxed{160}$$
output

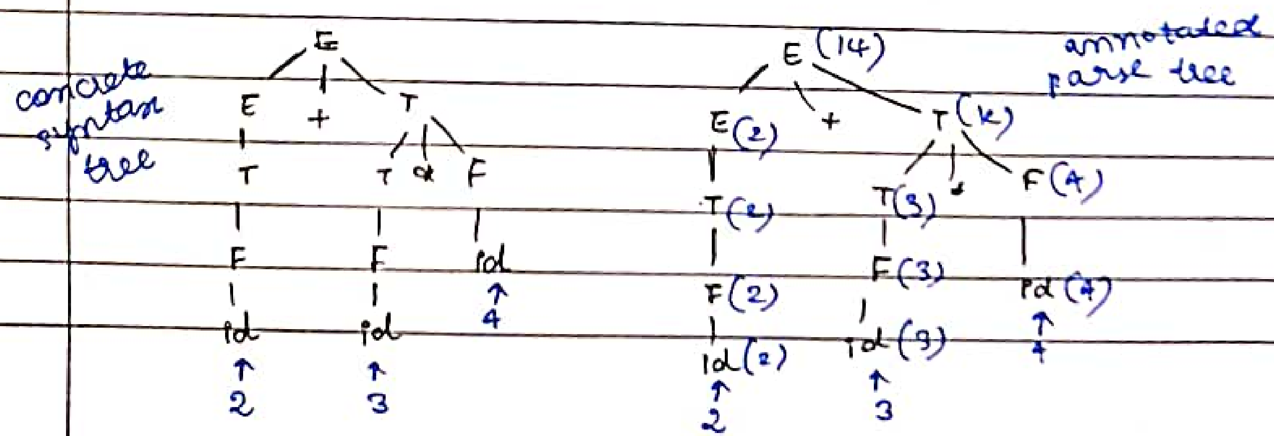
Summary :

✓ 3 solved problems on determining the output of SDTs.

| TYPES OF SYNTAX TREES |

outcome :

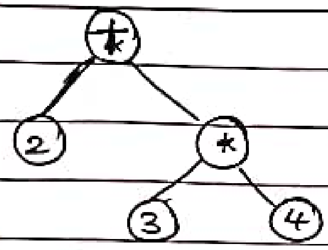
✓ concrete syntax tree and annotated parse tree.
✓ Usefulness of abstract syntax tree



__/__/____

Abstract syntax tree :

Input = 2 + 3 * 4



Advantages :

- no details are shown.
- faster evaluation

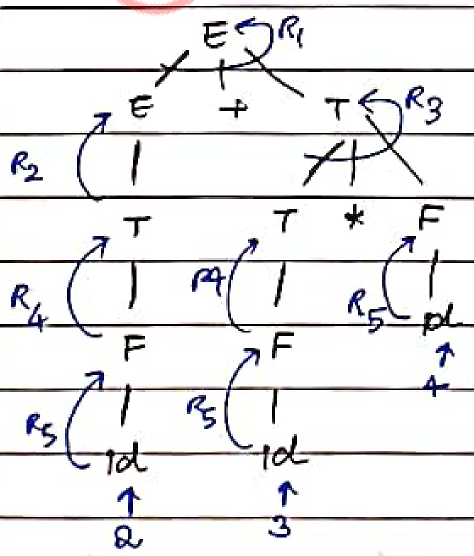
Summary :

- concrete syntax tree & annotated parse tree.
- usefulness of abstract syntax tree.

SDT- CONSTRUCTION OF ABSTRACT SYNTAX TREE

	Grammar	Semantic Rules
1	E → E + T	{ E.nptr = mknode (E.nptr, '+', T.nptr); }
2	E → T	{ E.nptr = T.nptr; }
3	T → T * F	{ T.nptr = mknode (T.nptr, '*', F.nptr); }
4	T → F	{ T.nptr = F.nptr ; }
5	F → id	{ F.nptr = mknode (null, id.value, null); }

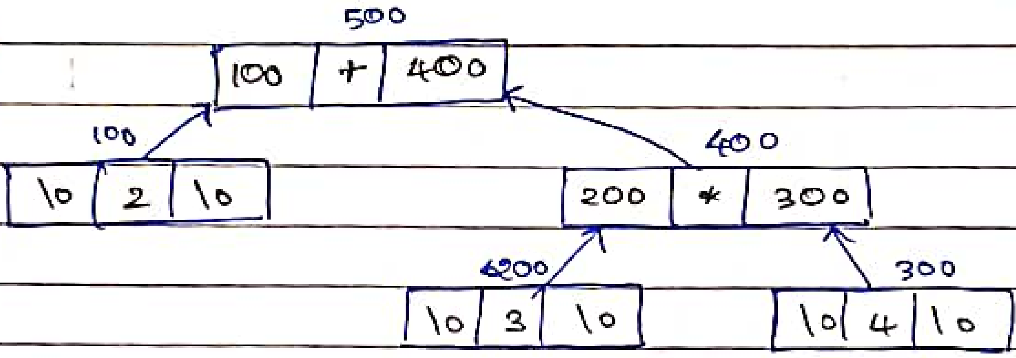
Expression : 2 + 3 * 4



Actions :

1. R_5 : F.nptr = 100
2. R_4 : T.nptr = F.nptr = 100
3. R_2 : E.nptr = T.nptr = 100
4. R_5 : F.nptr = 200
5. R_4 : T.nptr = F.nptr = 200
6. R_5 : F.nptr = 300
7. R_3 : T.nptr = 400
8. R_1 : E.nptr = 500

8 reduction actions performed



Need of Abstract Syntax Tree :

1. popular intermediate code representation.
2. parsing and intermediate code generation are done simultaneously.

Summary :

(v) visualization of the construction of the abstract syntax tree of an expression

SDT- TYPE CHECKING

Outcome :

(v) Analysis of how type checking works using SDT

SDT- Type checking :

Grammar :

(ambiguous) (E1, E2 used for understanding purposes)

1. $E \rightarrow E_1 + E_2$
2. $E \rightarrow E_1 == E_2$
3. $E \rightarrow (E_1)$
4. $E \rightarrow num$
5. $E \rightarrow True$
6. $E \rightarrow False$

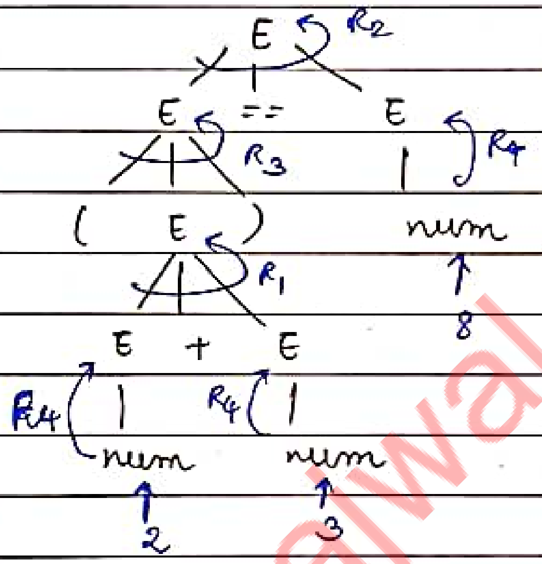
Semantic Rules :

1. { if ((E_1.type == E_2.type) && (E_1.type == int)) then E.type = int else ERROR ; }

_/_/_

2. { if ((E_1. type == E_2. type) && (E_1. type == int | bool)) then E. type = bool else ERROR; }
3. { E. type = E_1. type; }
4. { E. type = int; }
5. { E. type = bool; }
6. { E. type = bool; }

Expression : (2 + 3) == 8



Actions :

1. R_4 : E_1's type : int
2. R_4 : E_2' type : int
3. R_1 : E's type : int
4. R_3 : E_1's type : int
5. R_4 : E_2's type : int
6. R_2 : E's type : bool

Type checking SUCCESSFUL

Summary :

- ✓ Analysis of how type checking works using SDT.

SDT - DEALING WITH BINARY STRINGS

Outcome :

Different SDTs using the same grammar :

- ✓ counting the number of 1's in a binary string.
- ✓ counting the number of 0's in a binary string.
- ✓ counting the number of bits in a binary string.

__/__/____

SDT - Dealing with Binary strings

Grammar :

N → L ⟶ A binary number can be a list of bits

L → LB | B ⟶ A list of bits can either be a list of bits followed by a single bit or a bit single

B → 0 | 1 ⟶ A bit can be either 0 or 1

	Grammar	counting 1's	counting 0's	counting all bits
①	N → L	{ N.C = L.C; }	"	"
②	L → LB	{ L.C = L.C + B.C; }	"	"
③	L → B	{ L.C = B.C; }	"	"
④	B → 0	{ B.C = 0; }	{ B.C = 1; }	{ B.C = 1; }
⑤	B → 1	{ B.C = 1; }	{ B.C = 0; }	{ B.C = 1; }

$\boxed{.C \;\equiv\; .\text{count}}$

Summary :

Different SDTs using the same grammar to count no. of ✓ 0's ✓ 1's ✓ bits

SDT - BINARY TO DECIMAL :- PART ①

Outcome :

✓ conversion of integer binary to integer decimal

	Grammar	semantic Rules
①	N → L	{ N.dvalue = L.dvalue; }
②	L → LB	{ L.dvalue = L.dvalue * 2 + B.dvalue; }
③	L → B	{ L.dvalue = B.dvalue; }
④	B → 0	{ B.dvalue = 0; }
⑤	B → 1	{ B.dvalue = 1; }

__/__/____

$\boxed{L.dvalue = L.dvalue * 2 + B.dvalue}$

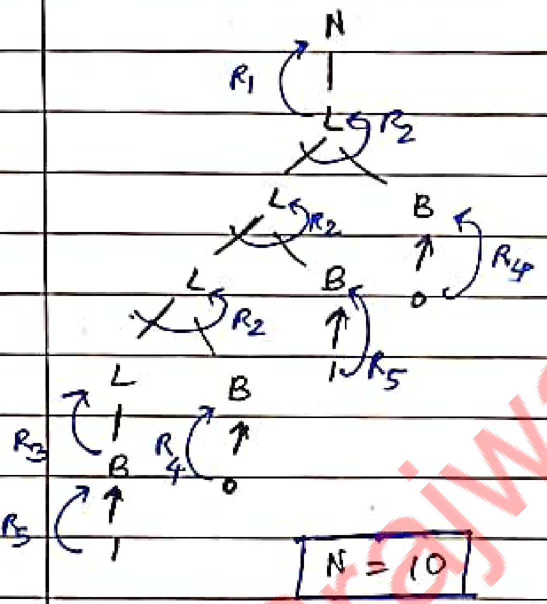
eg : convert $(110)_2$ to decimal.

$(1)_2 \rightarrow (1)_{10}$

$(11)_2 \rightarrow (1 \times 2) + 1 = 3$

$(110)_2 \rightarrow (3 \times 2) + 0 = 6$ (decimal)

Analyzing for input $(1010)_2$



Actions :

1. R_5 : B.dvalue = 1
2. R_3 : L.dvalue = 1
3. R_4 : B.dvalue = 0
4. R_2 : L.dvalue = $(1*2)+0 = 2$
5. R_5 : B.dvalue = 1
6. R_2 : L.dvalue = $(2*2)+1 = 5$
7. R_4 : B.dvalue = 0
8. R_2 : L.dvalue = $(5*2)+0 = 10$
9. R_1 : N.dvalue = 10

summary :

(✓) conversion of integer binary to integer decimal.

SDT - BINARY TO DECIMAL (PART 2)

outcome :

(✓) conversion of floating point binary to decimal.

Procedure :

1. Evaluate the binary bit stream without considering the radix point.
2. count the number of digits after the radix

point (say n) and divide the previously determined value by 2^n.

Example : $(0.011)_2$

① $(0011)_2 \rightarrow 3$

② $n = 3 \rightarrow 3/2^3 = 3/8 = \boxed{(0.375)_{10}}$

Grammar	Semantic Rules
① $N \rightarrow L_1 . L_2$	{ N.dval = L_1.dval + L_2.dval / $2^{L_2.count}$; }
② $L \rightarrow L_1 B$	{ L.count = L_1.count + B.count ; L.dval = L_1.dval * 2 + B.dval ; }
③ $L \rightarrow B$	{ L.dval = B.dval; }
④ $B \rightarrow 0$	{ B.dval = 0 ; }
⑤ $B \rightarrow 1$	{ B.dval = 1 ; }

Example : $(11.01)_2 = 3 + 1/2^2 = (3.25)_{10}$

Summary :

✓ conversion of floating point binary to decimal

SDT- GENERATION OF TAC

Outcome :

✓ Analysis of the SDT for generating TAC (Three Address code)

Grammar	Semantic Rules
① $S \rightarrow id := E$	{ gen (id.name = E.place); }
② $E \rightarrow E_1 + T$	{ E.place = newTemp(); gen (E.place = E_1.place + T.place); }
③ $E \rightarrow T$	{ E.place = T.place; }
④ $T \rightarrow T_1 * F$	{ T.place = new Temp(); gen (T.place = T_1.place * F.place); }
⑤ $T \rightarrow F$	{ T.place = F.place; }
⑥ $F \rightarrow id$	{ F.place = id.name ; }

__/__/____

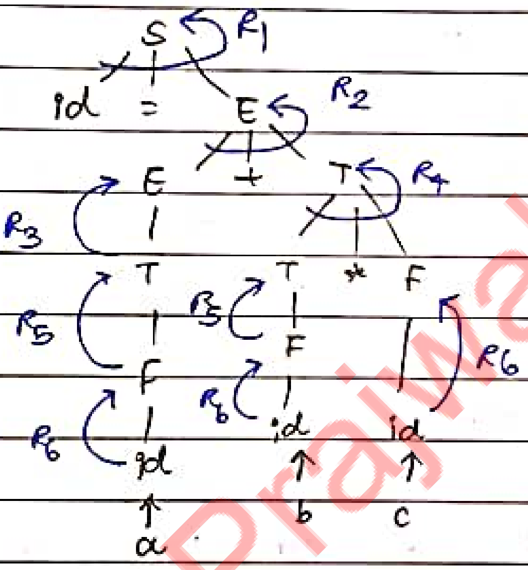
X.place : It is a placeholder variable, the variable remembers the place of the RHS.

newTemp() : It is a function that creates temporary variables. Eg t_1, t_2.

gen() : It is a function that generates an expression.

Expression :

$x = a + b * c$



Actions :

1. R_6 : F.place = a
2. R_5 : T.place = a
3. R_3 : E.place = a
4. R_6 : F.place = b
5. R_5 : T.place = b
6. R_6 : F.place = c
7. R_4 : T.place = t_1
   $t_1 = b * c$
8. R_2 : E.place = t_2
   $t_2 = a + t_1$
9. $x = t_2$

$$\boxed{\begin{array}{l} t1 = b * c \\ t2 = a + t1 \\ x = t2 \end{array}}$$

↓

TAC (Three Address code) for the given expression has been generated successfully

Summary :

(v) Analysis of the SDT for generating TAC (Three Address code)

__/__/__

TYPE OF ATTRIBUTES IN SDTs

outcome :

- ✓ synthesized Attributes & Inherited Attributes.
- ✓ Types of SDTs.

Types of Attributes :

In SDTs, every variable (non-terminal) in the production rules are associated with attributes.

Eg $B \rightarrow 0$ { B.dvalue = 0 ; }

① synthesized Attributes :

Attributes deriving values from non-terminal children (mentioned by the production rules).

Eg: $A \rightarrow BCD$ then $A.attr = f(B.attr, C.attr, D.attr)$

The A.attr is a synthesized attribute

② Inherited Attributes :

Attributes deriving values from the parent and siblings of the non-terminal.

Eg : $A \rightarrow BCD$ then $C.attr = f(A.attr) | f(B.attr) | f(D.attr)$

The C.attr is an inherited attribute

Types of SDTs :

(I) S- attributed SDTs :

1. uses only synthesized attributes.
2. semantic actions are placed at the rightmost end of the productions.

   Eg : $A \rightarrow BC$ { }
3. Attributes are evaluated during bottom up parsing.

__/__/____

(II) L-attributed SDTs :

① Uses both synthesized and inherited attributes. Each inherited attribute is restricted to inherit either from parent or left sibling(s) only.

eg : $A \rightarrow XYZ$

$\{Y.attr = A.attr,\ Y.attr = X.attr,\ Y.attr \neq Z.attr\}$

② Semantic actions can be placed anywhere in the RHS.

eg : $A \rightarrow \{\}\,BC \mid D\{\}E \mid FG\{\}$

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

outcome summary :

✓ synthesized & Inherited attributes.

✓ Types of SDTs

$\boxed{\text{SDT - SOLVED PROBLEMS (SET 2)}}$

outcome :

✓ 2 solved problems on determining the type of SDTs.

Q : Examine the SDT and determine the type of it :

$A \rightarrow LM\ \{L.i = f(A.i);\ M.s = f(L.s);\ A.s = f(M.s);\}$

$A \rightarrow QR\ \{R.i = f(A.i);\ \underbrace{Q.i = f(R.i)}_{L \times};\ \underbrace{A.s = f(Q.s)}_{S \times};\}$

(a) S-attributed SDT

(b) L-attributed SDT

(c) Both

(d) None ✓

$\boxed{\text{(d) None}}$

_/_/_

Q : Examine the SDT and determine the type of it :

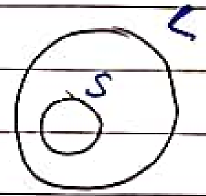
$A \rightarrow BC \quad \{ B.S = A.S; \}$

(a) S- attributed SDT

(b) L- attributed SDT ✓ (L)

(c) Both

(d) None



S- attributed ⊂ L- attributed

Summary :

✓ 2 solved problems on determining the type of SDTs.

SDT – STORING TYPE INFORMATION IN SYMBOL TABLE (PART 1)

Outcome :

✓ How type information is stored in symbol table using L- attributed STD.

L- attributed SDT – storing type information in a symbol table

int a;
int b;
int c;

(type) → int a, b, c;
(type)

Grammar	Semantic Rules
① $D \rightarrow TL$	$\{ L.in = T.type; \}$
② $T \rightarrow int$	$\{ T.type = int; \}$
③ $T \rightarrow char$	$\{ T.type = char; \}$
④ $L \rightarrow L_1, id$	$\{ L_1.in = L.in,$ addtype(id.name, L.in); }
⑤ $L \rightarrow id$	{ addtype (id.name, L.in); }

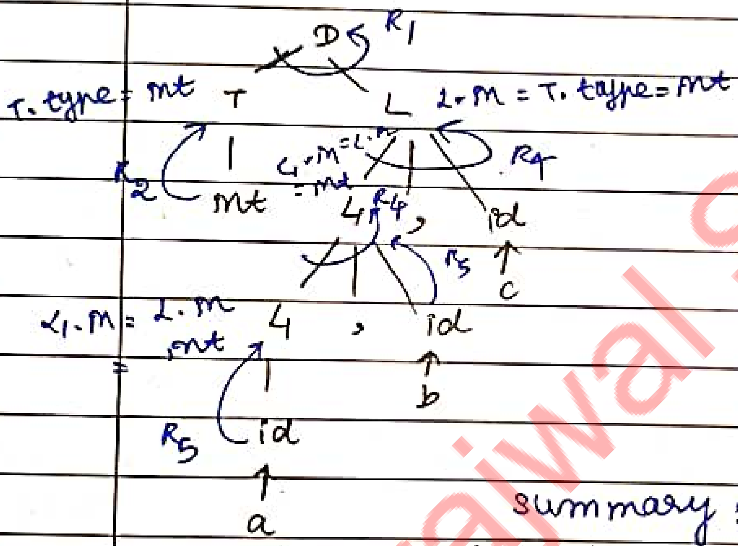
_/_/__

addtype() : It is a function that adds the identifier and its type to the symbol table.

(• in → • information)

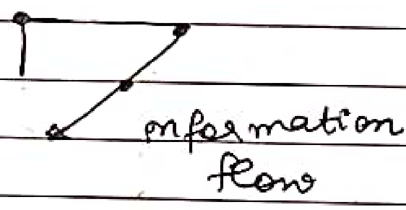
1. synthesized Attributes (info. moves up)
2. Inherited Attributes (info. moves from parent to children)

Expression: int a, b, c;



Actions:

(#1) addtype (a, int)
(#2) addtype (b, int)
(#3) addtype (c, int)



summary:
How type information is stored in symbol table using L-attributed SDT.

SDT - STORING TYPE INFORMATION IN SYMBOL TABLE (PART 2)

outcome :

1. How type information is stored in symbol table using s-attributed SDT.

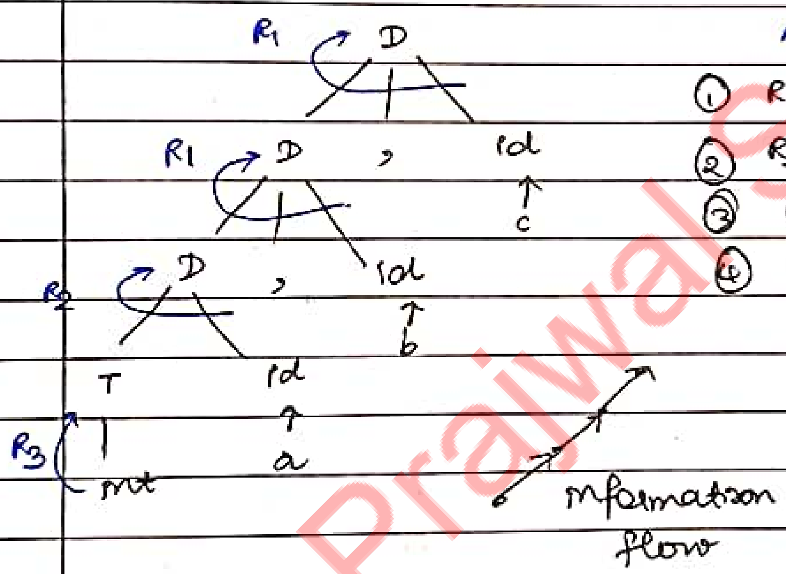
S-attributed SDT - storing Type Info in symbol Table

$\underbrace{\text{int}}_{\text{declaration}}\ \ a, \underbrace{b;}_{\text{id}}$ $\underbrace{\text{int}}_{\text{type}}\ \underbrace{a;}_{\text{id}}$

Grammar	Semantic Rules
① D → D, id	{ addtype (id.name, D.type), D.type = D.type }
② D → T id	{ addtype (id.name, T.type), D.type = T.type }
③ T → int	{ T.type = int; }
④ T → char	{ T.type = char; }

addtype( ) : It is a function that adds the identifier and its type in the symbol table

Expression int a, b, c;



Actions :

① R_3 : T.type = int
② R_2 : addtype (a, int)
③ R_1 : addtype (b, int)
④ R_1 : addtype (c, int)

a	int
b	int
c	int

← symbol table

Summary :

① How type information is stored in symbol table using S-attributed SDT.