Date 17/10/06

Syntax Directed Translation (SDT)

→ It is nothing but grammer + semantic rules.

Semantic rules — Semantic actions
Translatn rules.

eg

Prodn	rule
$A \to \alpha$	{ Printf("reduced by using $A \to \alpha$"); }

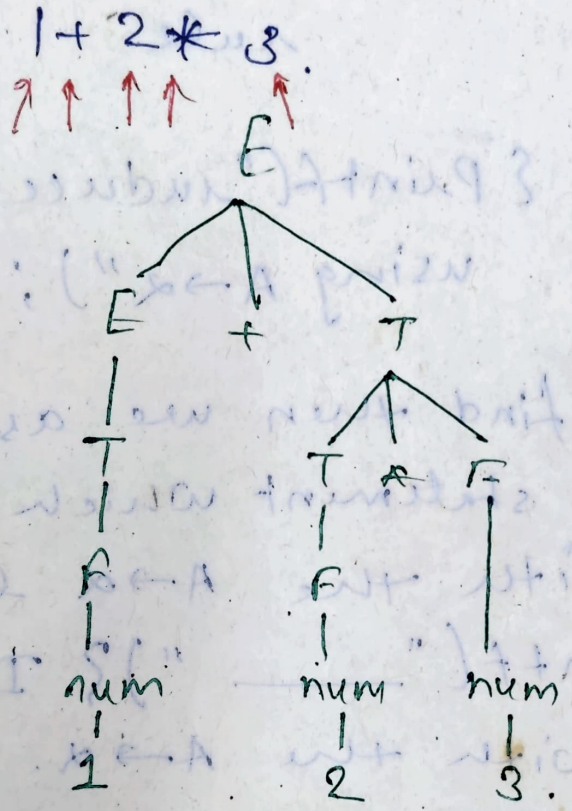
* When $A \to \alpha$ find then we assume that the statement which is attached with the $A \to \alpha$ like here { Printf("_____") } It is executed with the $A \to \alpha$. is reduced.

Examples of SDT → Semantic rules can be added —

1. To store / retrieve type information into symbol table.
2. To perform consistency checks like type checking parameters checking etc.

3. to build syntax trees.
4. to issue error messages.
5. to generate intermediate code / Target code.

Eg Bottom-up Parser.



$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow num$

→ If we want to evaluating the value of the terminal then we have the need to propogate the along with the nodes of parser, or with the gramer's attributes.

How to propogate the lexime value with the attributes —

eg { F.val = num.lval ; }

lexime value of num token.

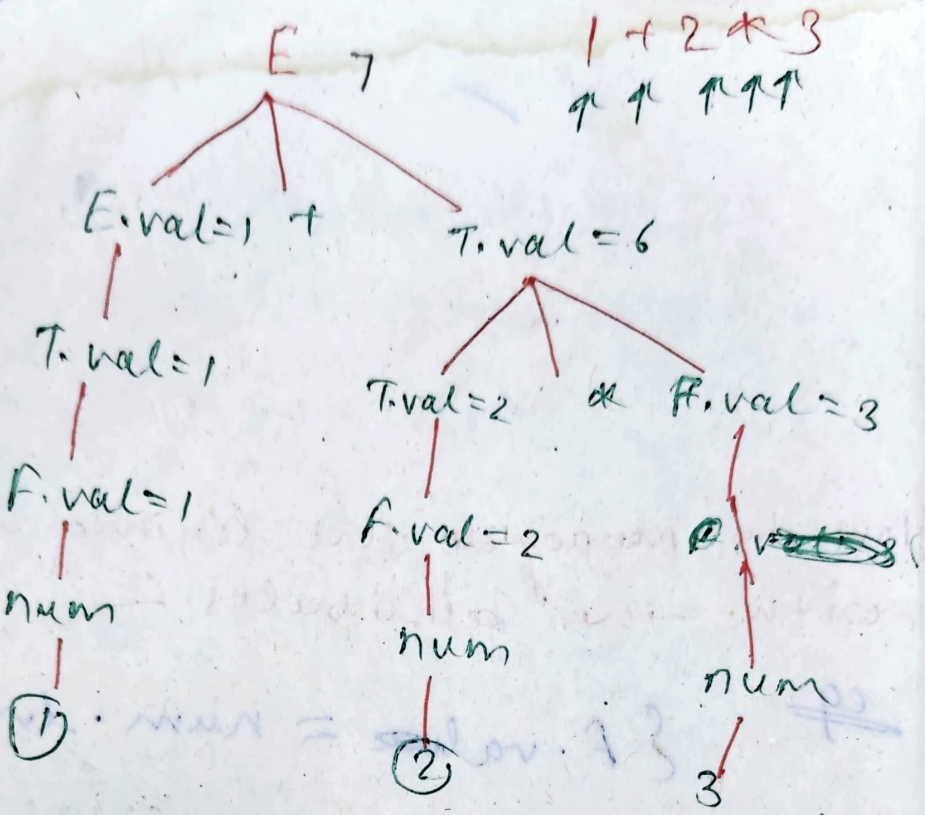
→ Order must be followed.

→ When the F, T, or E is reduced the F.val = 1. or E.val = 1, or T.val = 1.

→ when

then the same prodⁿ is propogate to the parent so that the parent has the value same as the child's value.

Annotated/Decorated parse tree ⇒ A parse tree that shows attribute values at each node is called Annotated or Decorated Parse tree.



$E \rightarrow E + T$ { E.val = E1.val + T.val; }

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

$T \rightarrow T_1 * F$ { T.val = T1.val * F.val; }

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

$F \rightarrow num$ { F.val = num.lval; }

The above eg. is the synthesized attribute b'coz the child's value is give to the parent.

Steps for SDT

$G \rightarrow i/p$

actions $\rightarrow$ o/p ✓

→ look at the i/p defined the grammar

→ take a simple i/p string & Draw the parse tree.

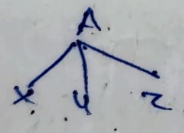
→ define the rules for

Attributes

Synthesized

> attribute value at a node is evaluated in terms of attribute values of its children.

eg $A \to XYZ$



$\cdot s$ → synthesized attribute.

$A.s = f(X.s / Y.s / Z.s)$

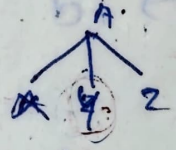
$A.s = X.s * Y.s$

$A.s = X.s + Y.s * Z.s$

Inherited

attribute value at a node is evaluated in terms of attribute of its parent or siblings.

eg $A \to XYZ$



$\cdot i$ → inherited attribute

$Y.i = f(A.i / X.i / Z.i)$

→ A grammer may has many no. of attributes.

→ When we use any attribute like synthesized or Inherited we have to maintained this attribut in each steps.

Annotated / Decorated Parse tree

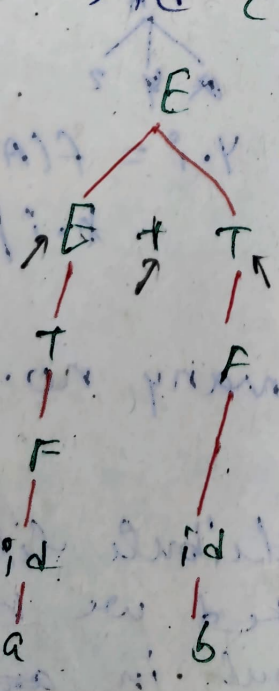
A parse tree showing attribute value at each node is called as the Annotated / Decorated parse tree.

eg Converting infix to postfix →

i/p : a + b * c

o/p : a b c * +

```
E → E + T   { print(+); }
  | T       { }
T → T * F   { print(*); }
  | F       { }
F → id      { print(id.name); }
```

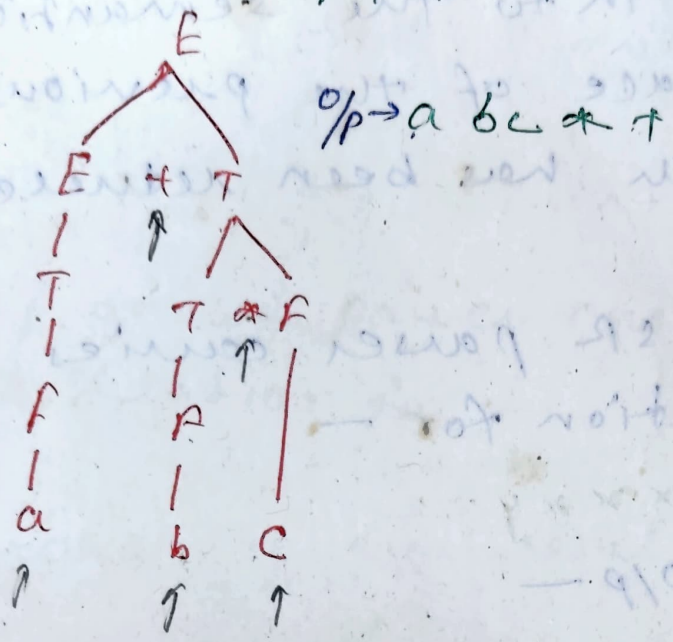


i/p → a + b

o/p → a b +

↳ Here we print the desire o/p so we don't need to choose any attribute b'coz we don't need to propogate the value of a to the previous nodes.

eg i/p a + b * c



o/p → a b c * +

→ A bottom up parser uses the a stack. to store the intermediate symbols we use the stack.

→ Here we have another stack "Semantic stack" along with the parser stack.

→ At any time whatever the action which performed in the semantic stack is the mirror of the parser stack. means when the parser stack perform the shift then the semantic stack also perform shift action. When the parser stack performs the reduce action & places the resultant in the stack. Same as the semantic stack is

also reduces the prodⁿ & places its result into the semantic stack is place of the previous prodⁿ which has been reduced.

1995

Que If an SR parser carries out the reduction for —

$w = xxxxyzz$.

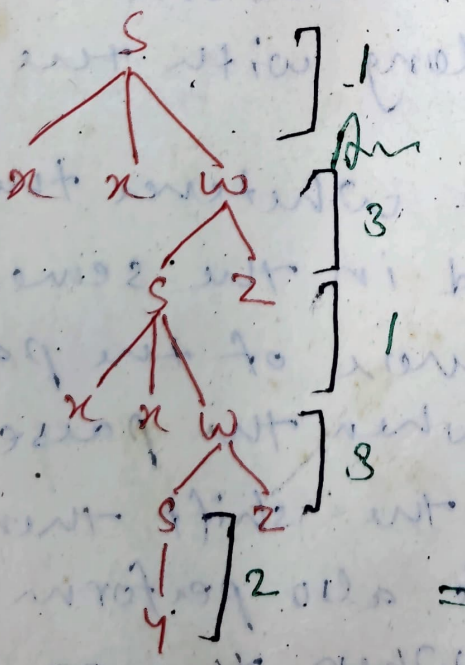
what is o/p —

$S \to xxw$ { print ("1"); }

$\quad | \; y$ { print ("2"); }

$w \to Sz$ { print ("3"); }

$w = xxxxyzz$.



$= 23131$ Ans

1996

Que.

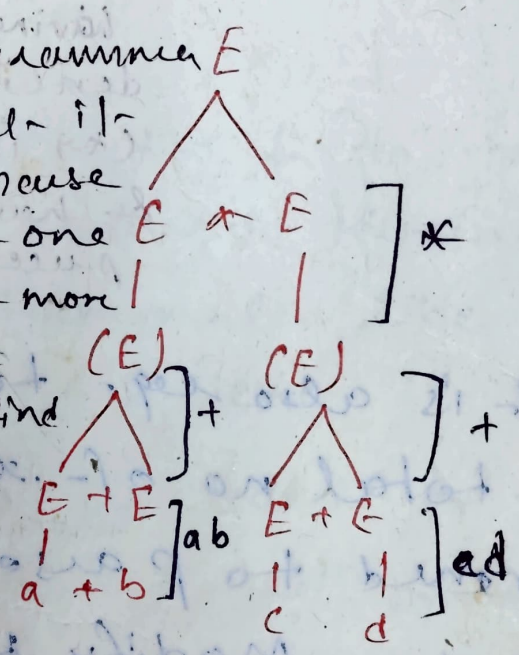
$E \rightarrow E+E$ { print('+'); }
$\mid E*E$ { print('*'); }
$\mid (E)$
$\mid id$ { print(id.name); }

If an LR parser carries out the translation for —

$w = (a+b)*(c+d)$

what is the result →

→ Though this grammar is ambiguous but it gives the one parse tree. So we get one o/p. If we have more than one parse tree then we find more than one o/ps:



Ans $ab+cd+*$

2020

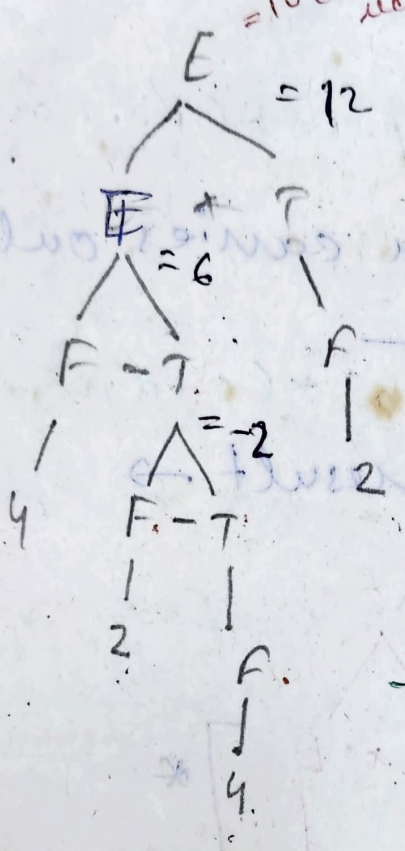
Que a)

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

$T \rightarrow F-T$ { T.val = F.val - T.val }
$\mid F$ { T.val = F.val }

$F \rightarrow 2$ { F.val = 2 }
$\mid 4$ { F.val = 4 }

Bottom up parser for w = 4-2-4*2
then o/p —



$w = 4-2-4*2$
$= 4-(-2)*2$
$= 6*2$
$= 12$

→ (-) is ~~left~~ Right associative & having ~~not~~ high precedence.
→ (*) is left associative & having less precedence.

b) It is also req. to compute the total no of reductions performed to (parse) the given i/p string Modify the SDT to find the no. of reductions performed.

Soln we have to count the no. of reduction performed by the parse tree = 10 Ans

→ So we use a global variable count which is incremented by when the any reduction is performed. But this is not the advisible soln b'coz the global variable is updated by the any body at any time.

→ So use using the another ~~use~~ alternative —

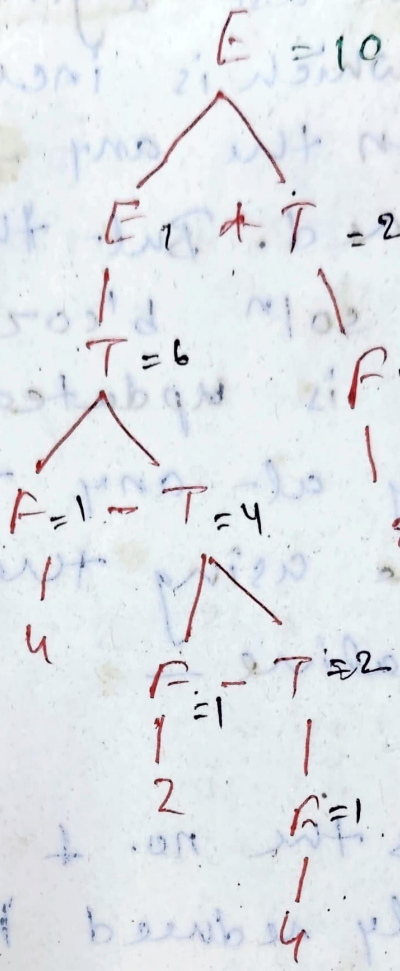
Rules —

1) Assigns the no. 1 to the each initially reduced terminal to the non terminal. like F → 14. & F → 2 & etc.

2) The no. of reductn of the parent is the no. of redn of the child + 1.

eg: when the child has the 1 then the parent has 2.

3) The no. of redn of any node is the no. of redn of right child + no. of redn of left child + 1.



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

$\mid T$ { E.val = T.val; Enr = T.nr + 1; }

$T \rightarrow F - T$ { T.val = F.val - T.val;
T.nr = F.nr + T.nr + 1; }

$\mid F$ { T.val = F.val; Tnr = F.nr + 1; }

$F \rightarrow 2$ { F.val = 2; F.nr = 1; }

$\mid 4$ { F.val = 4; F.nr = 1; }

SDT to count # of 1's in a given bin

i/p : 10101

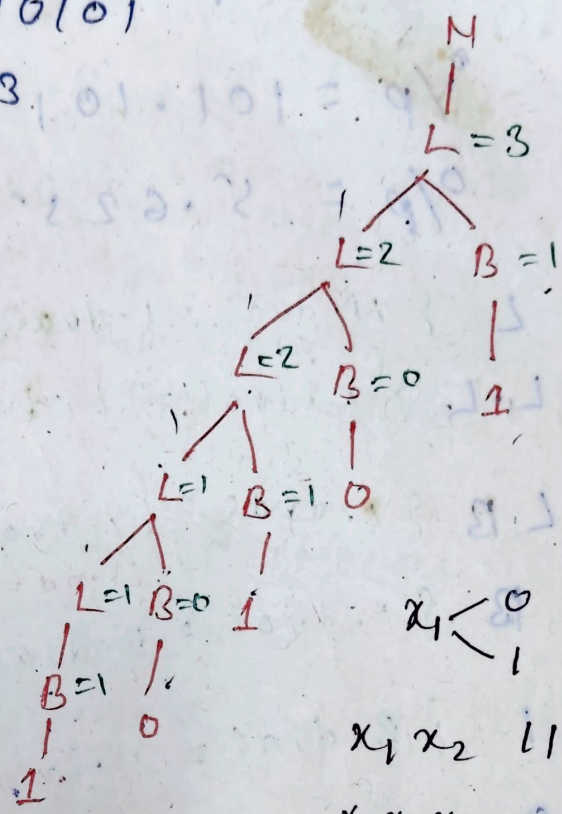
o/p : 3

i/p → 10101

N → L

L → LB | B

B → 1 | 0



decimal eq.

x_1 < 0 → 0
1 → 1

$x_1 x_2$ 11 → 1*2+1=3

$x_1 x_2 x_3$ 111 → 3*2+1=7

previous * 2 + latest value = decimal eq.

N → L { N.cnt = L.cnt ; }

L → L1B { L.cnt = L1.cnt + B.cnt; }

| B { L.cnt = B.cnt; }

B → 1 { B.cnt = 1; }

| 0 { B.cnt = 0; }

→ for any unipiodn the cnt value of the child is propogated to the parent.

→ when it has more than one child the the left child + right child.

Que Write SDT to convert binary to decimal.

Sol

i/p = 101.101

o/p = 5.625

.dval – digital value

.nd – no. of digits.

N → L { N.dval = L.dval; }

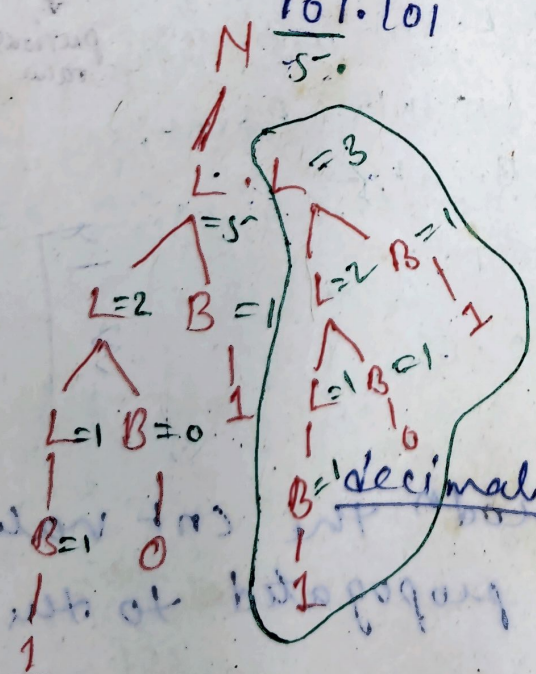
| L1.L2 { L.dval = L1.dval * (L2.dval / 2^L2.nd); }

L → L1B { L.dval = L1.dval * 2 + B.dval; L.nd = L1.nd + B.nd; }

| B { L.dval = B.dval; L.nd = B.nd; }

B → 1 { B.dval = 1; B.nd = 1; }

| 0 { B.dval = 0; B.nd = 1; }



x4 – 0 – 0

1 – 1

x1 x2 – 1.1 – 1*2+1

111 – 3*1+1

decimal equivalent of decimal part of binary –

decimal eq. of decimal part / 2^no. of digits. = 10.101 / 5 = 5 / 2^3 = 5/8

= 0.625

Date 19/10/06

SDT for Simple type checker (int, bool, err)

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

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

/ (E) { E.type = E_1.type ; }

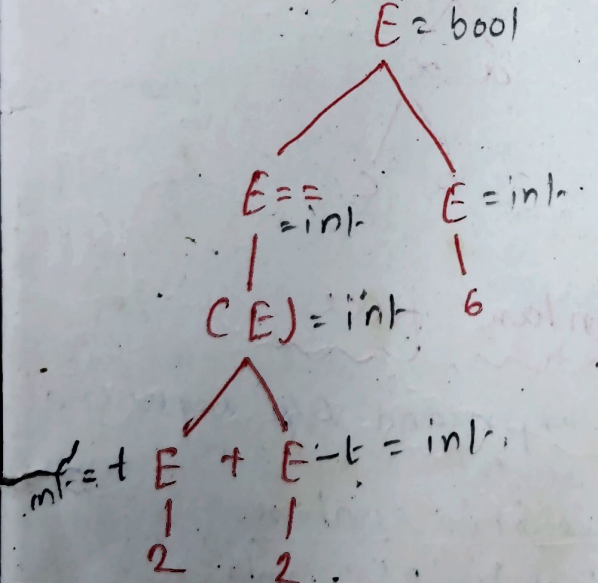
/ num { E.type = int; }

/ true { E.type = bool; }

/ false { E.type = bool; }

→ We can add additional operators to the grammers.

→ When we deriving the i/p string to the parse we are going to do some semantic operations takes place.



i/p = (2 + 2) == 6

o/p = bool

i/p (2 + 2) == 6

↑ type

→ In this type compability is imp.

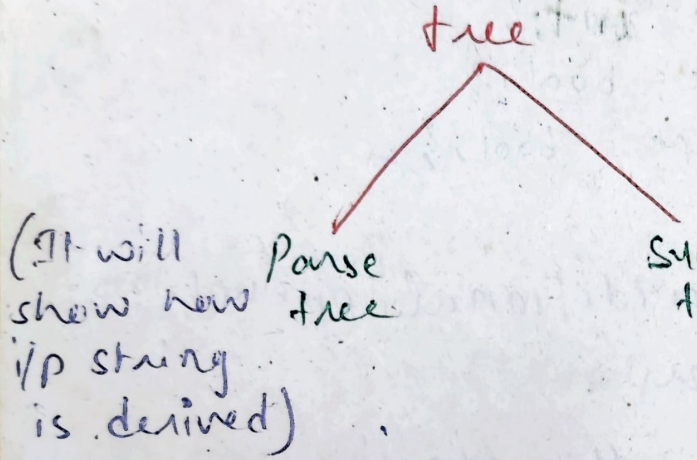
→ Whenever subexpresion is reduced then check the type compatibility of the E.

If both are same return type then else error.

→ Thus is a Synthesized attributes.

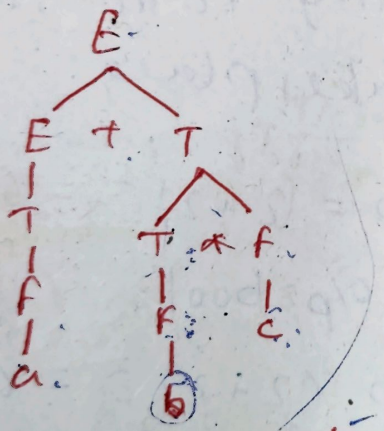
S- Attributed Definition →

The SDT which uses only synthesized attributes is called S-attributed definition or postfix SDT.



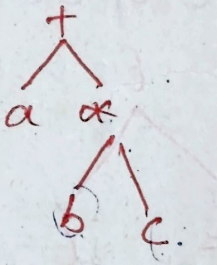
→ Concrete Syntax tree

Ex



→ Abstract syntax tree (AST)
Unnecessary terminals are abstracted away.

Ex



SDT for building syntax tree

Given some arithmetic expr and O/p required is syntax tree.

Translation rules for execution syntax.

$E \rightarrow E_1 + T \quad \{E.nptr = mknode\{E_1.nptr, '+', T.nptr\}\}$

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

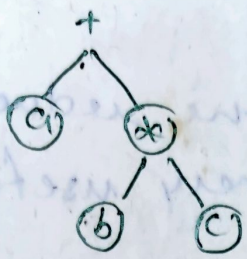
$T \rightarrow T_1 * f \quad \{T.nptr = mknode(T_1.nptr, '*', f.nptr);\}$

$\quad | \; F \quad \{T.nptr = f.nptr;\}$

$F \rightarrow id \quad \{f.nptr = mknode(NULL, id.name, NULL)$

i/p – a + b * c

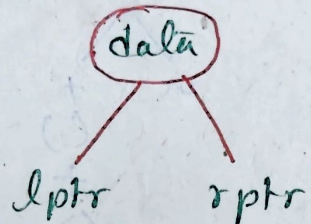
o/p –



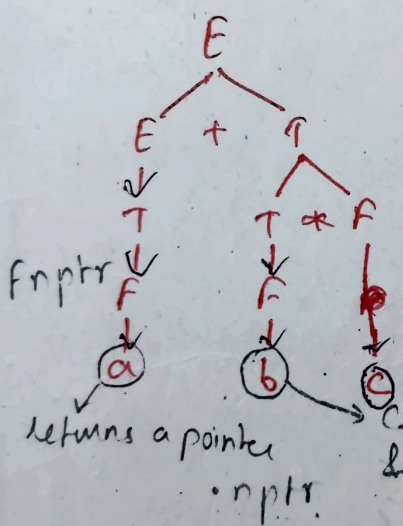
nptr stands for the node ptr.

Procedure –

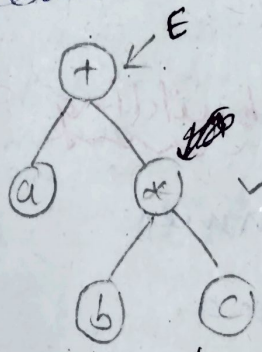
mk node [lptr , data , rptr]



The order in which reductions are performed –



parse tree.



when the lptr & rptr is null it means this is the leaf node

This pointer is preserved for F

And it will be node pointer • nptr

→ Whenever any subexpression is reduced i.e. creates i.e. mknode data pointer and makes right- pointer & left- pointer.

DAG (Directed ACyclic Graph).

↳ Used for optimized code.

→ By using DAG it will identify the duplicates.

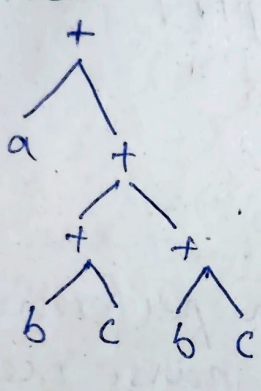
→ It will reduces the redundancies and it will not allow.

→ Tree is also a DAG but it makes allow the repeated exp.

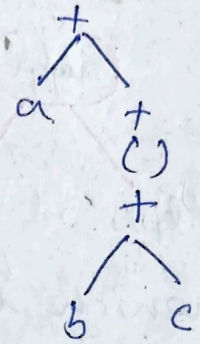
→ It eliminates the redundant code & reuses and very useful in optimizatn.

a + (b + c) + (b + c).

Tree



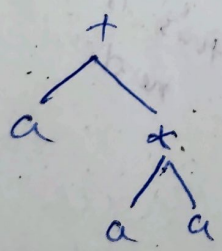
DAG



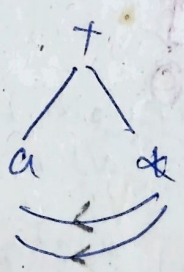
SDT for building DAG.

Given grammer

i/p a + a * a



o/p



→ No change in the translation rules but the change is in the implementat?

→ The Mk node must be modified. Whenever mknode is created, it should be verified whether it is created or not. If it created it will point to old one, if not then it will create new node.

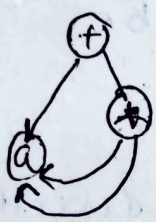
$E \rightarrow E + T$ { E.nptr = mknode(E.nptr, '+', T.nptr); }

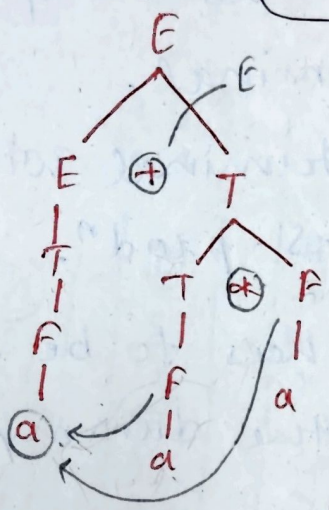
/ T { E.nptr = T.nptr; }

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

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

F → id { F.nptr = mknode(NULL, id.name, NULL) }





In this DAG the mknode implementation is diff. with the mknode use previous. In previous mknode this mknode blindly creates a node with two ptr.s. But in the DAG we use mknode which check that the node is already made or not. if made then point to that node

a + a * a

2005 Que

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

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

$T \rightarrow T \& F$ { T.val = T.val + F.val; }

/ F { T.val = F.val }

F → num { F.val = num.val.

If it is bottom up parser then 2 # 3 $ 5 # 6 & 4

Soln + > * [shortcut]

2 * 3 + 5 * 6 + 4

2 * 8 * 10

= 160.

(or) we can draw parse tree.

Still now we have assumed SDT for Bottom up parser.

→ It can also be done by top down parsing

Top-down Parser

→ Initially it assumes every action as dummy non terminal.

→ The dummy non terminal will be placed in the R.H.S. prodⁿ.

→ At whenever it has to be reduced it will take out the dummy non terminal.

SDT to count no. of balanced parameters (parenthesis)

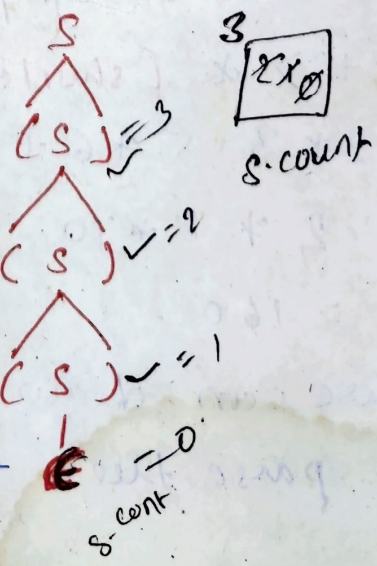
Bottom-up-parser —

(1) $S \rightarrow (S)$ { S.cnt = S.cnt + 1 ; }

/ ∈ Epsilon { S.cnt = 0; }

i/p = ((()))

o/p = 3



→ Initially the count = 0 after reading left-paranthesis.

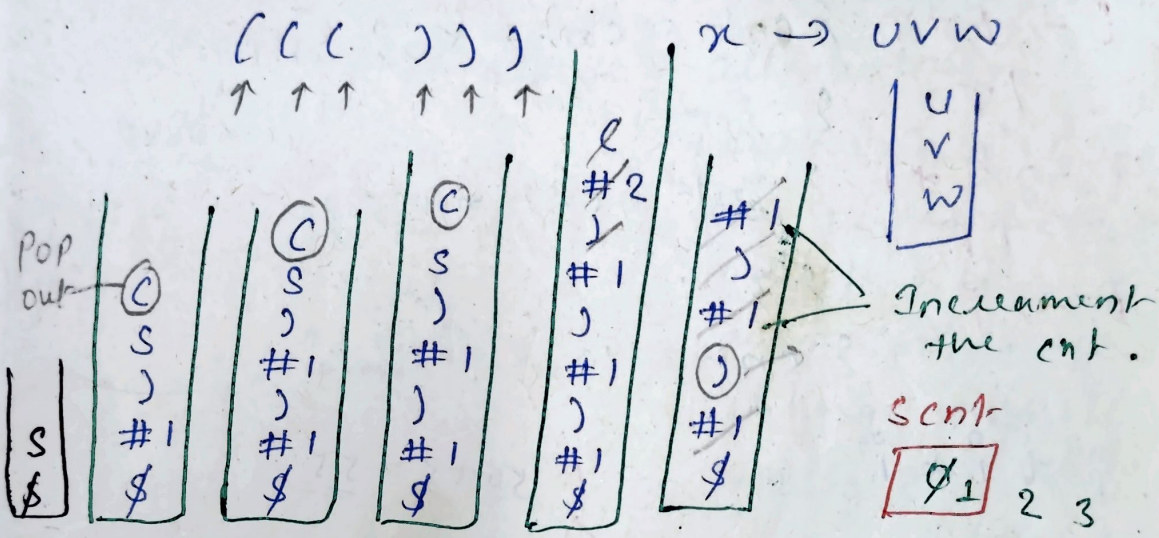
→ When it reads the right parenthesis it will increament the count.

Top-down Parser

→ Start with initial i/p

$S \rightarrow (S)$ { S.cnt = S.cnt + 1; }
#1

| ϵ { S.cnt = 0; }
#2



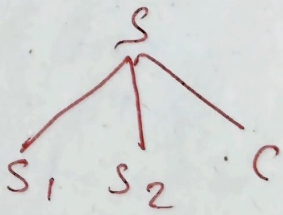
When the dummy variable like #1 or #2 we take out it & replaced with the translat^ which it define.

(2) $S \rightarrow S_1 S_2 C$ { S.T = S_1.t + S_2.t }

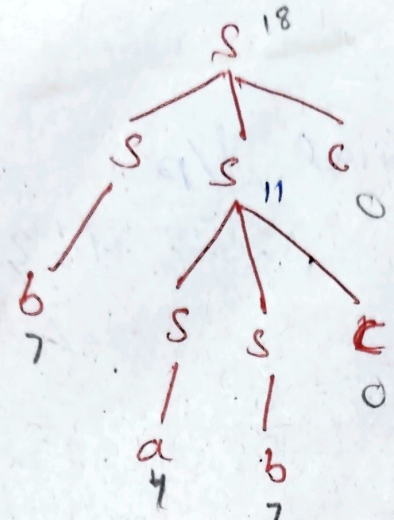
$S \rightarrow a$ { S.T = 4; }

$S \rightarrow b$ { S.T = 7; }

w = babcc



In this translation is carried by parser to parse i/p string babcc what is o/p.



babcc

7 + 4 + 7

= 18

Ans = 18

(2) $E \to E+T$ { cnt = cnt + 5; }

$|T$ { }

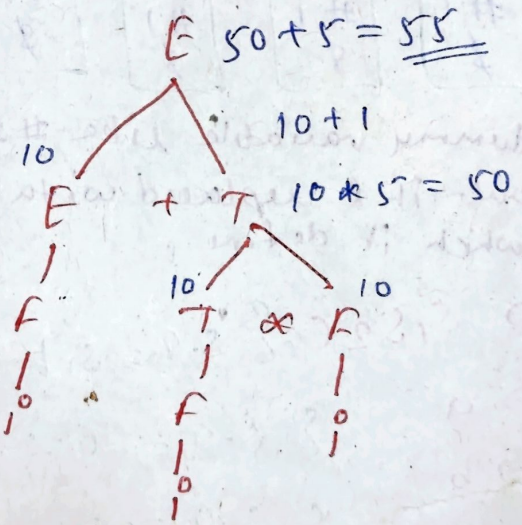
$T \to T*F$ { cnt = cnt * 5; }

$|F$

$F \to i$ { }

{ cnt = 10; }

i + i * i



Ans = 55

SDT

(Post fix) ↑ SDT

S-attributed Definition

1) Which allow only one type of attributes i.e synthesize attributes.

2) Semantic actions are placed at right end of the prodn

3) Semantic rules are evaluated during the Bottomup parser.

L-attributed Definition

1) Allow both synthesized & inherited attributes

Each inherited attribute is restricted to inherit either from parent (or) left siblings only.

inherited < Parent, left sublings

Ex $x \rightarrow x_1 x_2 x_3 x_4 x_5$ ($x_1 x_2$: left subling)

$x_3.i = f(x, i)$ ✓

$x_3.i = f(x_1.i / x_1.s)$ ✓

$= f(x_2.i)$ ✓

$x_2.i / x_2.s$ $= f(x_1.i)$ ✓

It is also allow

$= f(x_4.i)$ ✗

$= f(x_5.i)$ ✗

2) Semantic actions can be placed anywhere on the R.H.S.

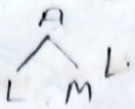
Ex $A \rightarrow \{\} BC$

$| \; 0 \{\} E$

$| \; F \; G \{\}$

3) Attributes r evaluated by traversing parse tree depth first & left to right.

Date
23/10/06



L.i = f(A.i)
M.i = f(L.S) i S
M.i = f(A.i) inherited

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

$A \rightarrow QR$ { R.T = f(A.i); ✓
Q.T = f(R.s); ✗ Inherited is not allowed from right sibling.
A.s = f(Q.s); }

In this, SDT is —

a) S.attribute b) L.attribute c) both d) None

S-attribute definition (i.e. Converting infix to postfix).

↳ In this no attributes are allowed only sementics are allowed.

$E \rightarrow E + T$ { Print(+); } #1
A A α
| T

$T \rightarrow T * F$ { Print(*); } #2
A A α
| F

$F \rightarrow num$ { Print(num.lval); } #3

→ Assume that it is left recursion & eliminate —

$E \rightarrow TE'$

$E' \rightarrow +T\#1\,E' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *F\#2\,T' / \epsilon$

$F \rightarrow num\#3$.

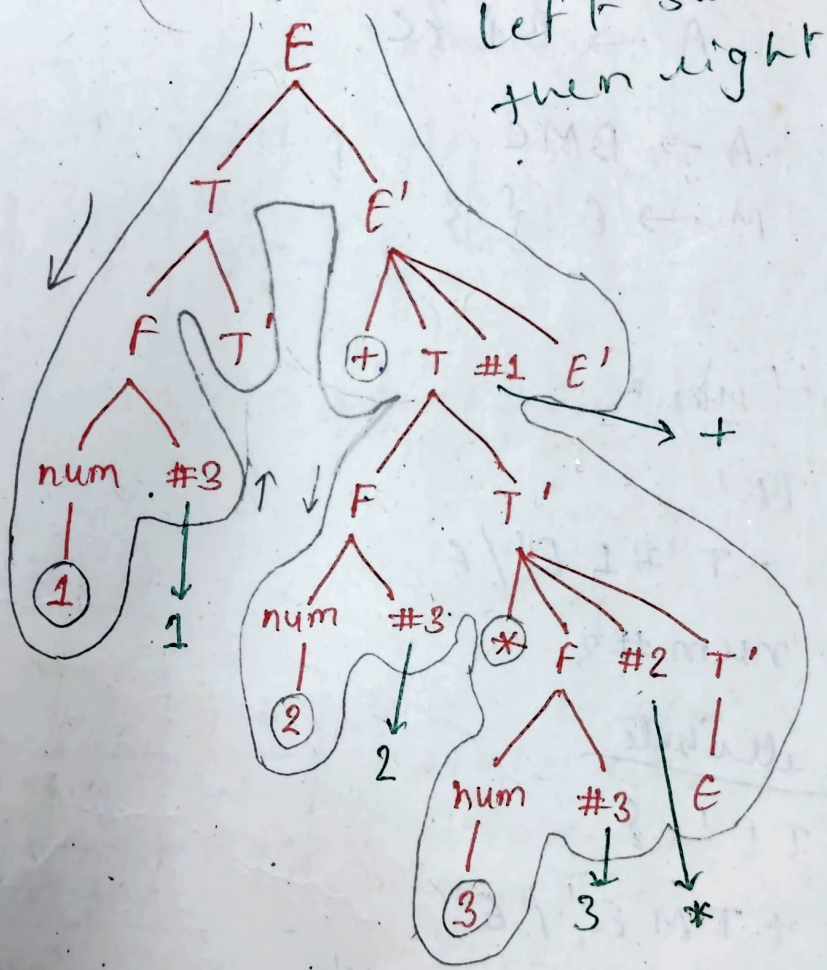
Since semantic actions are at middle we can say that it is L-attribute definition.

→ L-attribute.

Infix – i/p - 1+2*3

Postfix – 1 2 3 * +

Traversal first left subtree & then right subtree.



1 2 3 * +

Converting L-attribute to S-attribute

→ S-attribute is only defined when the semantic action is at R.H.S.

→ To move the semantic action into R.H.S., introduce some dummy variable and move the semantic action.

$A \rightarrow B \{\} C$

$A \rightarrow BM\epsilon$	$\{\} = M$
$M \rightarrow \epsilon \{\}$	$M \rightarrow \epsilon \{\}$

L-attribute

$E \rightarrow TE'$

$E' \rightarrow +T \#1 E' / \epsilon$

$T \rightarrow num \#3$

S-attribute

$E \rightarrow TE'$

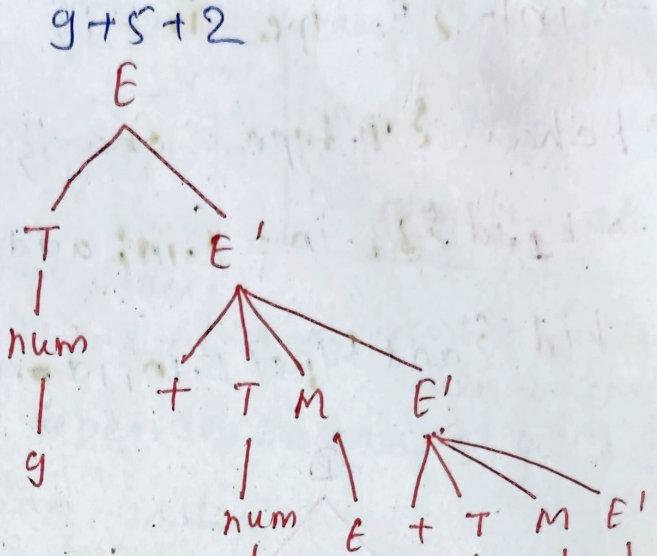
$E' \rightarrow +TME' / \epsilon$

$M \rightarrow \epsilon \{ Print(+); \}$ #1

$T \rightarrow num \{ Print(num.lval); \}$ #3

2003

Que 1) What is the o/p for given i/p 9+5+2.



i/p – Infix 9+5+2

↓

o/p – Postfix 95+2+

SDT to store type Information into symbol Table →

i/p - int x, y, z

o/p →

x	int
y	int
t	int

here the o/p is the symbol table which store the type info. about the ~~variables~~ identifier. like x, y & z

Que 1992

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

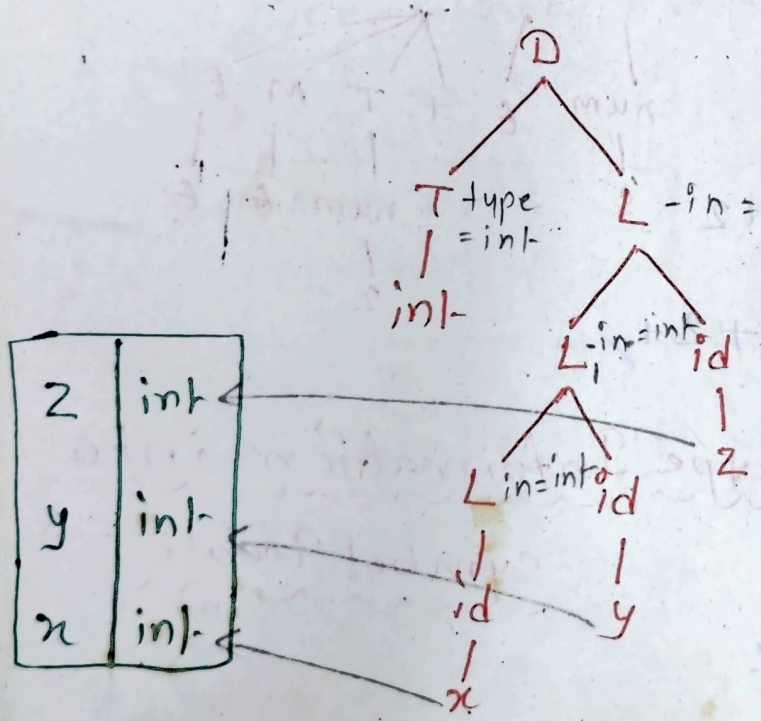
$T \rightarrow$ int { T.type = int; }

| char { T.type = char; }

$L \rightarrow L_1$, id { L_1.in = L.in; add_type(id.name, L_1.in); }

| id { add_type(id.name, L.in); }

• type is the synthesised attribute.



int x, y, z

• type is the attribute which is very useful to store the type info.

→ Propagating info. from one node to another node is only possible in inherited attribute.

→ Since it is given in synthesized attribute it is not possible to store the type info.

Synthesized attribute $D \rightarrow T\ L$ { L.type = T.type } ✗

Not possible so we take inherited.

{ L.in = T.type; }

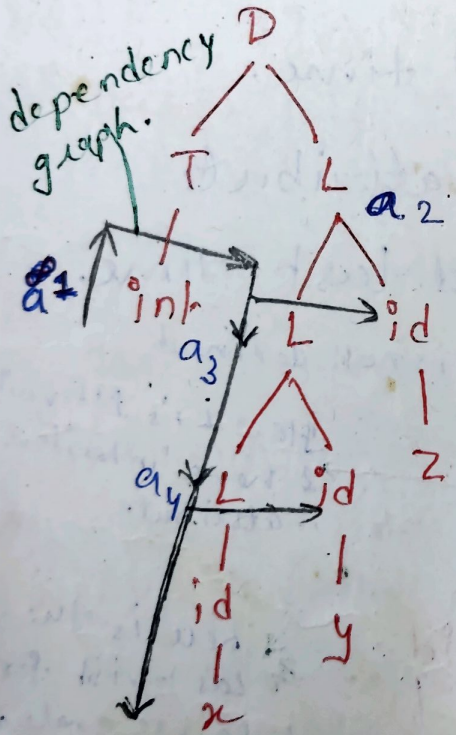
General Procedure

i/p - Parse tree
↓
dependence graph
↓
evaluation order

graph that shows interdependencies among the attributes.

A order in which the translatn rules are carried out.

→ By using dependency graph goes through evaluation order.



a_1 = int
a_2 = a_1
a_3 = a_2
addtype (z)
a_4 = a_3
addtype (y)
addtype (x).

→ It is very difficult to find dependency graph so we use S & L attributes.

S-attribute

→ Carry out translations using BUP

L-attribute

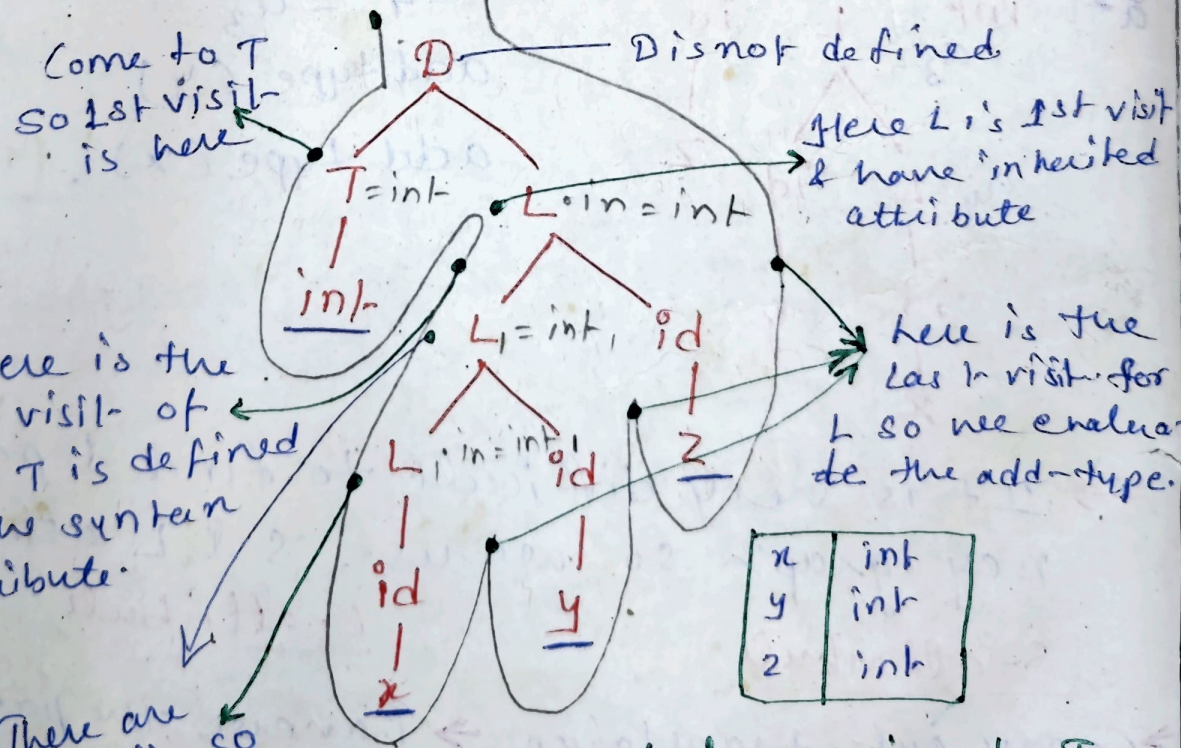
→ Traverse the parse tree depth first- evaluatn the attributes (or) translatn rules.

Evaluation of attributes in L-attributed definition that uses both synthesized as well as inherited attributes →

1) Traverse the parse tree depth first left to right.

2) Evaluate inherited attribute when node is visited 1st time.

3) Evaluate synthesized attribute when node is visited last time.



D → TL { L.in = T.type; }

T → int { T.type = int; }

| char { T.type = char; }

L → L1 id { L1.in = L.in; addtype(L1.in, id.name); }

| id { add.type(L.in, id.name); }

S-attribute definition for storing type information into symbol table →

→ The grammer is reduced to use the synthesized grammer & the equivalent grammer is —

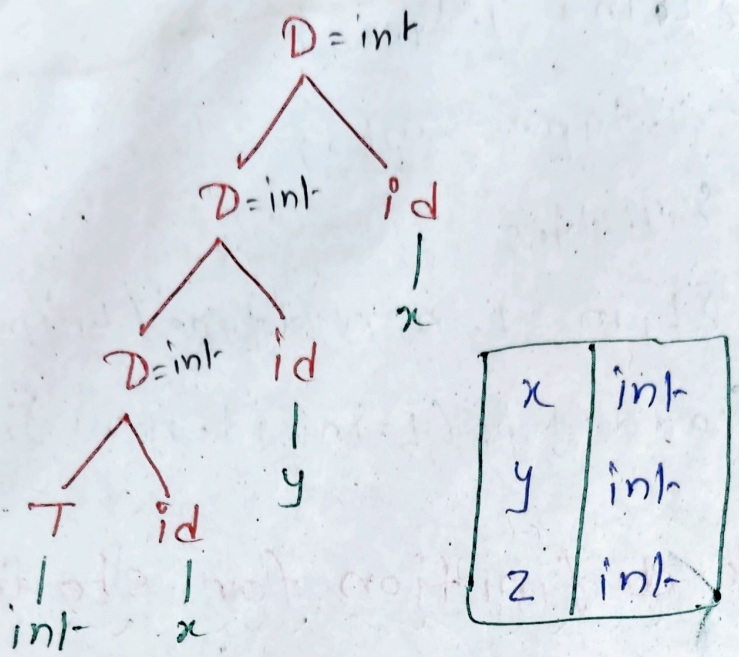
* → It is only possible through rewitting the grammer & sometimes it is not possible.

D → D, id { D.type = D.type; addtype(D.type, id.name); }

| T.id { type = T.type; add type(T.type, id.name); }

T → int { T.type = int; }

| char { T.type = char; }



→ Propagate ~~int~~ the information to the parent and store type with identifier in the symbol table.

L-attribute. – s-attribute → Can convert only by re-writing the grammer.

Que $A \rightarrow BC \{ C.s = f(B.s); \}$

a) Synthesis. b) L-attribute

c) both d) None.

Qu: $A \rightarrow BC \{ A.i = f(B.i); \}$

a) S-attribute.

b) L-attribute

c) both

d) None.

* Every S-attribute is L-attribute

Qu: Convert this SDT to an equivalent SDT i.e.

(i) a post-fix SDT and

(ii) has no left-recursion in the grammer.

$A \rightarrow A \{a\} B / B \{b\}$

$B \rightarrow 0 \{c\}$

Soln → First eliminate left recursion b'coz while eliminating we may get S-attributes.

$A \rightarrow \underbrace{B \overset{M}{\{b\}}}\ A'$

$A' \rightarrow \{a\} B A' / E$

$B \rightarrow 0 \{c\}$

→ Translating into s-attributes.

$A \rightarrow BMA'$ $\{b\}$

$M \rightarrow \epsilon$ $\{b\}$

$A' \rightarrow RBA'$ $\{a\}$

$R \rightarrow \epsilon$ $\{a\}$

$B \rightarrow 0$ $\{c\}$

or

$A \rightarrow MA'$ $\{b\}$

$M \rightarrow B$ $\{b\}$

$A' \rightarrow RBA'$ $\{a\}$

$R \rightarrow \epsilon$ $\{a\}$

$B \rightarrow 0$ $\{c\}$