1 | 3 | 18

UNIT - 4

→ Intermediate code
→ Notational forms

Syntax directed Translation (SDT):-

Attributes:
Value attached to a grammar symbol is attribute

(or)

Symbol " " " " "

(or)

type of grammar attached to a grammar symbol

(or)

string attached to a grammar symbol

Rules: Rules are attached with productions of the grammar

Semantic Action

Calculation

Concatenation

Print

Two types of attributes. They are:

(1) Synthesized Attribute ⇒ The value of the root node is calculated with the leaf node (or) The value of the grammar symbol is denoted by the calculated leaf of the node.

(2) Inherited "

Eg:- $E \rightarrow E^{(1)} + E^{(2)}$

{ E.val = $E^{(1)}$.val + $E^{(2)}$.value }

E.val
$E^{(1)}$.val
$E^{(2)}$.val
} attributes

(2) Inherited Attribute:

[bottom up is inherited]

Eg:- $A \rightarrow XYZ$

$\{ Y.value = Z * A.val + Z.val \}$

Eg:- $E \rightarrow E^{(1)} + T$ [for string attached to a grammar symbol]

Postfix code assigned to E ← $E.code$

← $E^1.code$ } → Attributes called as Post fix code

Postfix code assigned to E^1 ← $T.code$ → This is postfix code assigned to T

$\{E.code = E^1.code \,||\, T.code \,||\, '+'\}$ ↗ concatinates

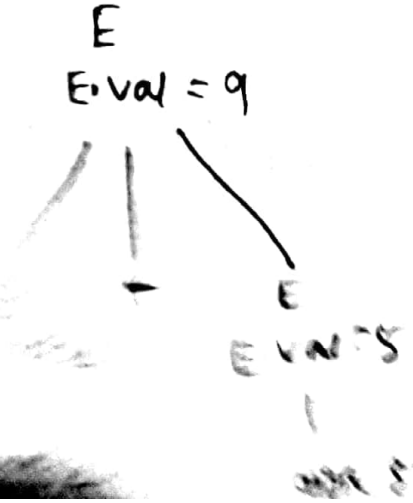
Annotated (or) decorated tree:

Parse tree with value attached at the node is called Annotated (or) decorated parse tree.

Eg:-

$E \rightarrow digit$

$\{E.val = lexval\}$

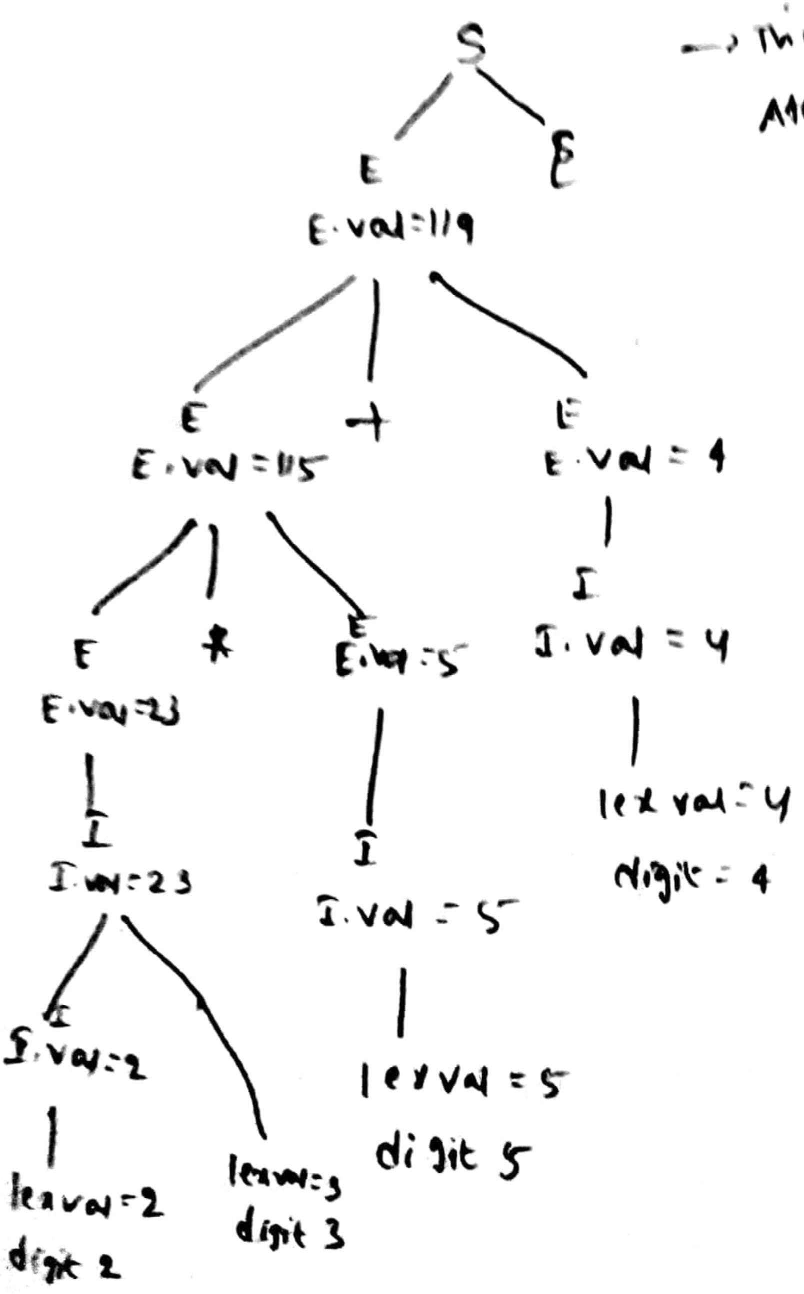


The grammar for this question is Arithmetic Expression desk calculator

Q) S → E $	Print E.val
E → E + E	E.val = E'.val + E''.val
E → E * E	E.val = E'.val * E''.val
E → (E)	E.val = E'.val
E → I	E.val = I.val
I → I digit (val pass; 2 digit num)	I.val = 10 * I'.val + lexval
I → digit	I.val = lexval

23 * 5 + 4 $ → write Annotated tree for this

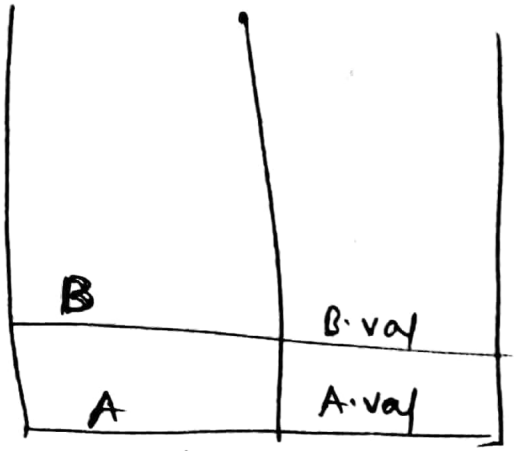
→ This is for synthesized Attributes



Scanned by CamScanner

→ In this, stack contains two fields. They are state field and value field

Eg:-



State ↓
State containing the grammar symbol

Value ↓
contains the attributes of the [A, B] grammar symbols.

Program statement:- [for previous problem]

Print val[TOP]

val[TOP] = val[TOP] + val[TOP-2]

val[TOP] = val[TOP] * val[TOP-2]

val[TOP] = val[TOP-2]

None

val[TOP] = 10 * val[TOP] + lex val

val[TOP] = lex val

Sentence	stack	value	write Production at each step Production
23*5+4$	-	-	
3*5+4$	2	-	I → digit
3*5+4$	I	2	
*5+4$	I3	2-	
*5+4$	I	(23)	
*5+4$	E	(23)	
5+4$	E*	(23)-	
+4$	E*5	(23)--	I → digit
+4$	E*I	(23)-5	E → I
+4$	E*E	(23)-5	
+4$	E	115	
4$	E+	(115)-	
$	E+4	(115)--	
$	E+I	(115)-4	
$	E+E	(115)-4	
$	E	119	
-	E$	119	
	S	print value 119	

de this Program is "arithmetic expression data calculator"

2/3/18

Dependency Graph:- (or order of evaluation)

It is also a PT (or Annoted tree) (parse tree) which shows the order of evaluation.

Q) This question is for inherited attributes.

$T \to F T'$ { $T'.inh = F.val$

$T.val = T'.syn$ }

$T' \to *FT'$ { $T_1'.inh = T'.inh * F.val$

$T'.syn = T_1'.syn$ }

$T' \to \epsilon$ { $T'.syn = T'.inh$ }

$F \to digit$ { $F.val = lex\ val$ }

draw annotated tree for $4 * 5$

derivation for $4 * 5$ is

$T \to F T'$

$digit\ T'$

$digit * F T'$

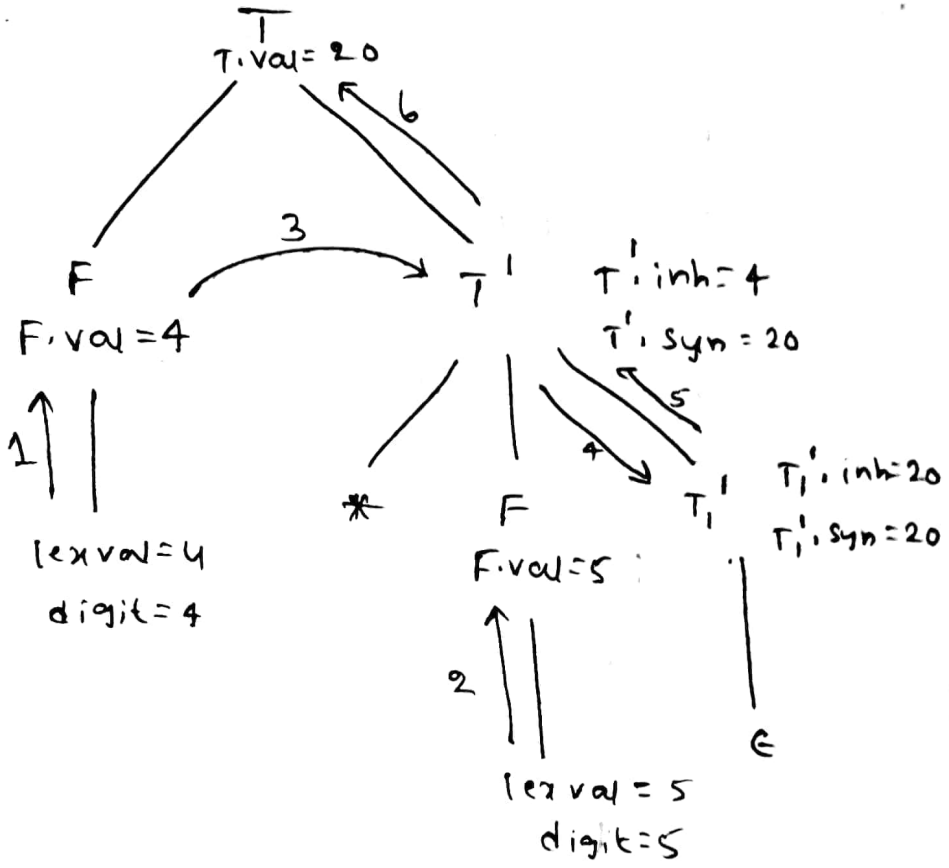
$digit * digit\ T'$

$digit * digit\ \epsilon$

$digit * digit$

$4 * 5$

draw tree based on derivation First



Intermediate code:

(1) Three address code — Quadruples, Triples, Indirect triples

(2) post fix code

(3) Syntax tree

Syntax tree - eliminates the redundant information

↳ It is a modification of semantic tree.

(1) Three address code:

Eg:- 1. A = B op C

2. A = op B

3. Go to L

4. if A relop B GOTO C

5. Cal p(A1, A2, --- An)

Parameter A1

Parameter A2

⋮

Parameter An

Call p,n

6. A[I] = B

7. B = A[I]

Q) write three address code for A = - B * (C+D)

Sol: A = - B * (C+D)

$T_1 = -B$

$T_2 = C+D$

$T_3 = T_1 * T_2$

$A = T_3$

{T1, T2, T3 are temporary variables}

(i) Quadruples:

→ It is a array of record. Each record has 4 fields.

→ 4 fields.

→ Each quadruple is represented with record with 4 fields.

	op	arg 1	arg 2	Result
0	uminus	B		T1
1	+	C	D	T2
2	*	T1	T2	T3
3	=	T3		A

(ii) Triples:-

→ contains 3 fields

→ three fields are op, arg 1, arg 2

	op	arg 1	arg 2
0	uminus	B	
1	+	C	D
2	*	(0)	(1)
3	=	A	(2)

(iii) Indirected Triples:

→ list the pointers to the triples

23	(0)
24	(1)
25	(2)
26	(3)

change the order

23	(0)
25	(2)
24	(1)
26	(3)

skip some terms

23	(0)
24	(1)
26	(3)

→ If you want to change the order of execution (or) skip some terms in execution this Indirected triples will be preferd.

Q) write the three address code for this

$$X = -(a+b) * (c+d) - (a+b+c)$$

and find Quadruples and Triples for this question.

3/2/18

$$X = -(a+b) * (c+d) - (a+b+c)$$

$T1 = a+b$

$T2 = -T1$

$T3 = c+d$

$T4 = T_1 + C$

$T5 = T_2 * T_3$

$T6 = T5 - T4$

$X = T6$

(i) Quadruples:

	op	arg1	arg2	Res
0	+	a	b	T1
1	uminus	T1		T2
2	+	c	d	T3
3	+	T1	c	T4
4	*	T2	T3	T5
5	-	T5	T4	T6
6	=	T6		X

(ii) Triples:

	op	arg1	arg2
0	+	a	b
1	uminus	(0)	-
2	+	c	d
3	+	(0)	c
4	*	(1)	(2)
5	-	(4)	(3)
6	=	X	(5)

(iii) Indirected triples

23	(0)
24	(1)
25	(2)
26	(3)
27	(4)
28	(5)
29	(6)

Q) write three address code for

```
while a<c do
 if a=1 then c=c+1
else
    while a<d do
    a=a+3
```

Soln:

```
Begin : IF a<c GO TO L1
        GO TO End
L1 : If a=1 GOTO L2
     GO TO L3
L2 : C=c+1
     GO TO Begin
```

```
L3 : IF a<d   GO TO L4
     GO TO Begin
L4 : a=a+3
     GO TO L3
END.
```

(2) post fix code:

$a+b \Rightarrow ab+$

if e then x else y $\Rightarrow$ exy?

Q) if a then if c-d then a+c else a*c
else a+b

Sol: a cd- ac+ ac* ? ab+ ?

→ This post fix code is easy to evaluate the expressions.

another simple example

a+b * c - x / y

a b c * x y / + - → doubt

a b c * + x y / -

SDT for post fix code:

$E \to E^{(1)}$ op $E^{(2)}$	{E.code = $E^{(1)}$.code || $E^{(2)}$.code || op}
$E \to (E^{(1)})$	{ E.code = $E^{(1)}$.code }
$E \to$ id	{ E.code = id }

Semantic

print op

{ }

print id

check for a+b*c

Sol: Push a

Reduce a to E E→id print a

Push +

Push b

Reduce b to E E→id print b

Push *

Push c

Reduce c to E E→id print c

Reduce E→E*E print *

Reduce E→E+E print +

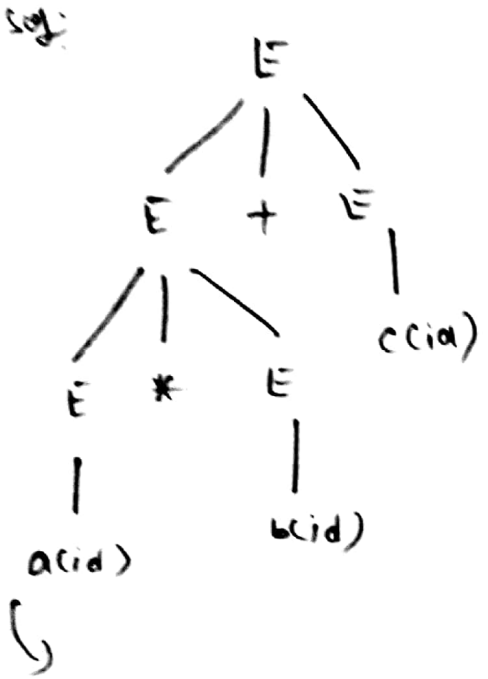
∴ Finally we got abc*+ [last terms]

Syntax tree:

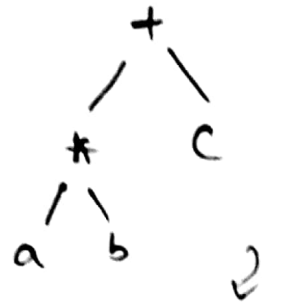
→ which eleminates the redundant information.

E → E+E | E*E | id

w = a*b+c



Parse tree



Syntax tree

SDT for generation Syntax tree

$E \rightarrow E^{(1)} + T \rightarrow E.node = newnode(+, E^{(1)}.node, T.node)$

$E \rightarrow E^{(1)} - T \rightarrow E.node = newnode(-, E^{(1)}.node, T.node)$

$E \rightarrow E * T \rightarrow E.node = newnode(*, E^{(1)}.node, T.node)$

$E \rightarrow T \rightarrow E.node = T.node$

$T \rightarrow (E) \rightarrow T.node = E.node$

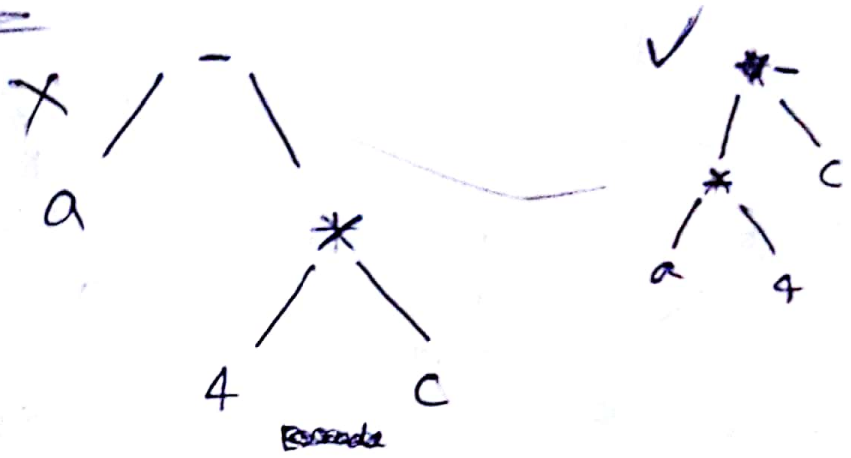
$T \rightarrow id \rightarrow T.node = newleaf(id, id.entry)$

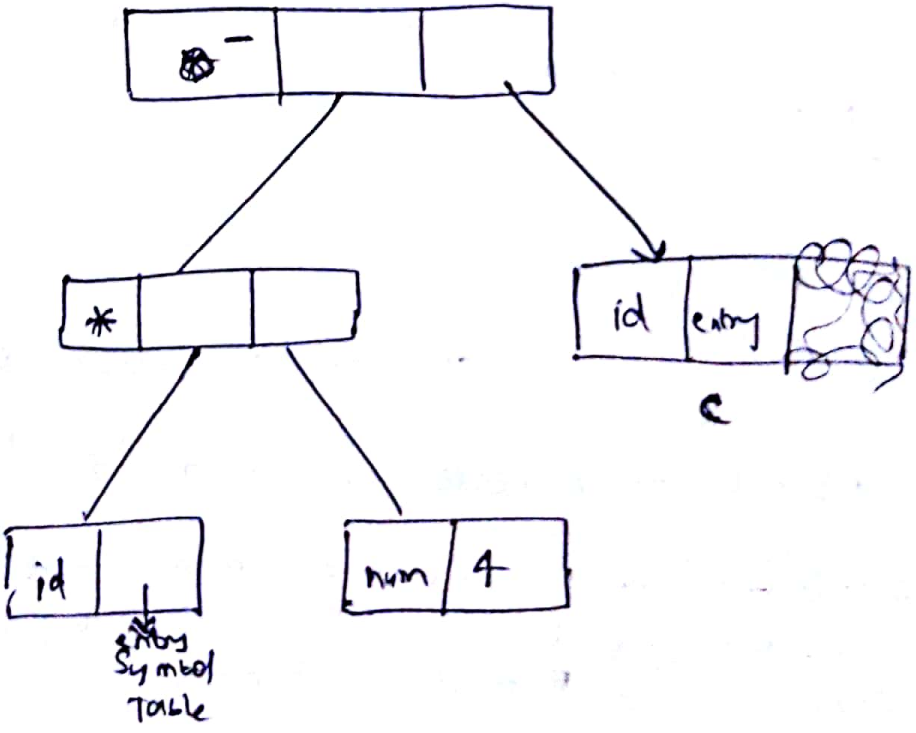
$T \rightarrow num \rightarrow T.node = newleaf(num, num.val)$

6/3/18

Q) Construct the syntax tree for a*4-c

Syntax tree





S-attributed definition:

If all the attributes are synthesized attributes then it is called as "S-attributed definition".

L-attributed definition:

(i) which uses (considering) synthesized attributes as well as inherited attributes, it is called as "L-attributed definition".

↓ of the root node

(or) (ii) inherited & synthesized attributes of grammar symbols which are called before the symbol X_i.

$A \to X_1 X_2 \ldots X_{i-1} X_i \ldots X_n$

SDT for Generating 3 address stmts

(or)

Assignment stmts

(or)

arithmetic stmts

E.code → set of 3 addr stmts

E.place → Location which holds the value of E

New Temp() → Generates the temporary names (i.e., T1, T2, ...)

(1) $A \to id := E$

A.code := E.code ||

id.place := E.place

id.place holds the Location of value of id.

{Generates id.place := E.place}

(2) $E \to E^{(1)} + E^{(2)}$ { T = New Temp()

E.place := T

E.code := $E^{(1)}$.code || $E^{(2)}$.code ||

E.place || := || $E^{(1)}$.place || + || $E^{(2)}$.place

{Gen E.place := $E^{(1)}$.place + $E^{(2)}$.place}

(3) $E \rightarrow E^{(1)} * E^{(2)}$ { T := New Temp()

E.place := T

E.code := $E^{(1)}$.code || $E^{(2)}$.code ||

E.place := $E^{(1)}$.place * $E^{(2)}$.place

{Gen E.place := $E^{(1)}$.place * $E^{(2)}$.place}

Eg (2nd one) E → E1 + E2

$(a * b * c) + (x * y)$

E1 E2

E1.code := T1 = a * b

T2 = T1 + c

E.place = T2

E2.code := T3 = x * y

E.place = T3

T4 := T2 + T3

final temp value

(4) $E \rightarrow - E^{(1)}$

{ T := New Temp()

E.place := T

E.code := $E^{(1)}$.code ||

E.place := - || $E^{(1)}$.place

{Gen E.place := - || $E^{(1)}$.place}

(5) $E \rightarrow (E')$

E.code = $E^{(1)}$.code

E.place = $E^{(1)}$.place {none}

(6) $E \rightarrow id$

E.code = null

E.place = id.place

{ semantic rule }

a) Trace for the following A := - B * (C + D) by using SDT rules.

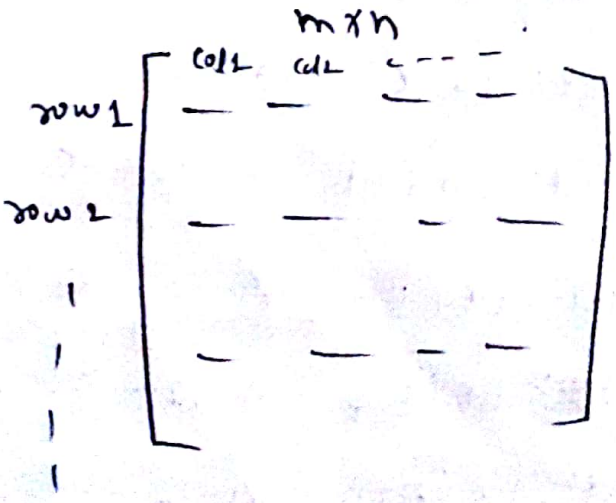
input	stack	place	instruction Generation
A := - B * (C + D)			
:= - B * (C + D)	A	- ↗ hyphen	
:= - B * (C + D)	id	A	
- B * (C + D)	id :=	A -	
B * (C + D)	id := -	A - -	
* (C + D)	id := - B	A - - -	
* (C + D)	id := - id	A - - B	
* (C + D)	id := - E	A - - B	
* (C + D)	id := E	A - T1	T1 := - B
(C + D)	id := E *	A - T1 -	
C + D)	id := E * (	A - T1 - -	
+ D)	id := E * (C	A - T1 - - -	
+ D)	id := E * (id	A - T1 - - C	
+ D)	id := E * (E	A - T1 - - C	
D)	id := E * (E +	A - T1 - - C -	
)	id := E * (E + D	A - T1 - - C - -	

)	id := E * (E + id	A - T1 - - C - D	
)	id := E * (E + E	A - T1 - - C - D	
)	id := E * (E	A - T1 - - T2	T2 = C + D
-	id := E * (E)	A - T1 - - T2 -	
-	id := E * E	A - T1 - T3	T3 = ((+ D)
-	id := E	A - T4	T4 = (- B) * ((+ D)
-	id := E * E	A - T1 - - T2	
-	id := E	A - T3	~~T3 : (T1 * (T2)~~ T3 : T1 * (T2)

8/3/18

SDT for array referencing:

→ used to calculate the address of array elements.



Row major order

column major order

List of attributes:

(1) elist - used to group the index expression (or group the arrays

(2) elist.Array - denotes the Symbol table entry for the arrayname

(3) elist.NDim - NDim means no. of dimensions

(4) Limit(Array, i) - Returns the upper limit along the i^{th} dimension of the array.

(5) elist.place - denotes the Symbol table entry for a name who value is computed by 3 add stmt generated for elist.

(6) L.Offset - gives the starting addr of array (or Null

(7) bpw - bites per word

for Generating
Simple Assignment stmt:

(1) $A \rightarrow L := E$

{ if (L.offset = null) then

L.place := E.place

else

Gen (L.place[offset] := E.place)

(2) $E \rightarrow E^{(1)} + E^{(2)}$

T := NewTemp( )

Gen (T := $E^{(1)}$.place + $E^{(2)}$.place)

E.place := T;

(3) $E \rightarrow (E^{(1)})$

E.place = $E^{(1)}$.place

(4) $E \rightarrow L$

{if L.offset = null) then

E.place := L.place

else

begin

T := NewTemp( )

Gen(T := L.place (L.offset)

end ; E.place := T

(5) L → elist]

{ T := NewTemp();
  U := NewTemp();
} one to hold the location and another one is to calculate the offset

Gen(T := elist.Array - c

Gen(U := bpw * elist.place

L.place := T

L.offset := U }

(6) L → id

L.place = id.place

L.offset = Null }

(7) elist → elist(1), E

{ T := NewTemp();

Gen(T := elist(1).place *

Limit(elist(1).Array, elist(1).Ndim+1);

Gen(T := T + E.place)

elist.Array = elist(1).Array

elist.place := T

elist.NDim := elist.NDim + 1

(8) elist → id [E

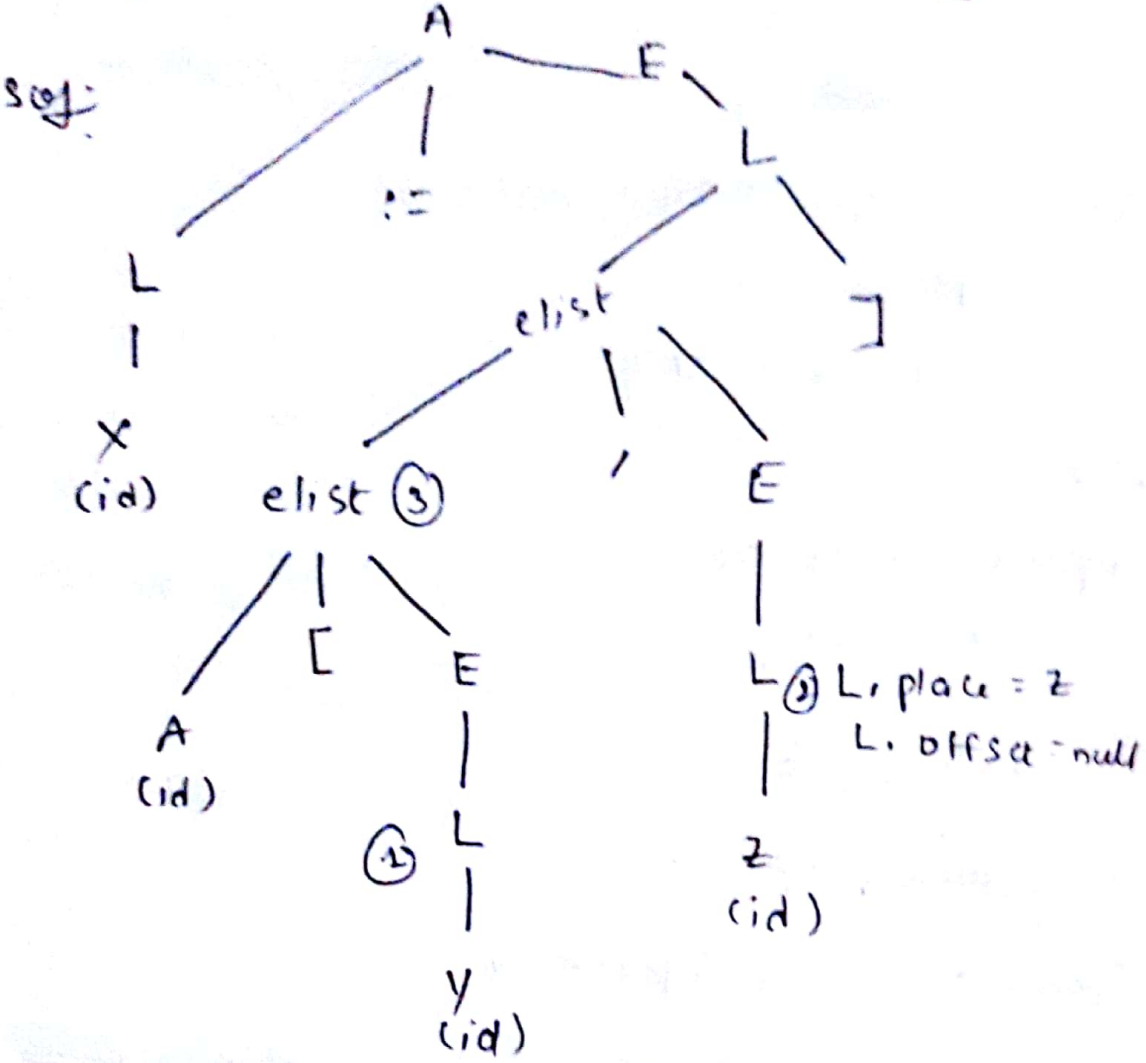
elist.place := E.place

elist.NDim := 1

elist.Array := id.place

Q) write the Annotated tree (OR) decorated parse tree for $x := A[y, z]$

Sol:



(1) L.place = y [L → id]

L.offset = null

(2) L.place = z [L → id]

L.offset = null

(3) elist.place = y (elist → id [E

elist.NDim = 1

elist.Array = A

9/3/18

Boolean expressions:

1. Condition exp

   IF ( )

2. To compute the Logical values

   A (or) B and C

   and - more priority
   or - less

   ~~less~~ priority than and.

→ two methods to evaluate for this. They are:

(1) Simple Arithmetic expression

(2) Flow of control

(1) Simple Arithmetic expressions:

Eg:-(i) A or B and C

T1 := B and C

T2 := A or T1

(ii) A < B

1. if A < B GOTO 4
2. T := 0
3. GO TO [redacted]
4. T := 1

[∵ 1 + 3 goes to 4]

(iii) A < B or C

1. if A<B Go to 4 // Next Quad Quadruple array
2. T := 0
3. Go To 5
4. T := 1
5. T2 := T1 or C

SDT's:

(1) $E \rightarrow E^{(1)}$ or $E^{(2)}$

{ T := NewTemp( )

E.place := T

Gen (T := $E^{(1)}$.place or $E^{(2)}$.place) }

(2) $E \rightarrow id^{(1)}$ relop $id^{(2)}$

T := New Temp( );

E.place := T;

Gen (if $id^{(1)}$.place relop $id^{(2)}$.place goto NextQuad + 3)

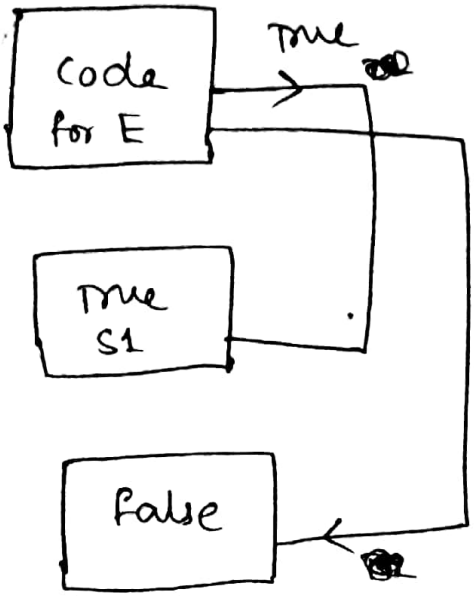
Gen T := 0;

Gen (Go to NextQuad + 2)

Gen T := 1;

(2) Flow of control:

(i) if E then S1 else false



Eg:- A and (B or C) ⇒ If A is true then check (B or C)

A or B and C ⇒ If A false then check the (B and C)

A or B

→ To evaluate flow of control. we have three different procedures. They are :-

(1) Backpatch (P, i)

(2) Merge (P_1, P_2)

(3) Makelist (i)

Makelist(i)

→ Makelist(i) creates a new list containing only i an index into the array of Quadruples being generated.

→ A makelist returns a point to the list it has made.

Merge(p1, p2)

→ Takes the lists pointed by p1 and p2 concadinates them into one list and returns a pointer to the concatinated list.

Backpatch(p, i)

→ Makes each of the Quadruples on the list pointed by 'p' to take Quadruple 'i' as a target.

SDT's:

(1) $E \rightarrow E^{(1)}$ or $M E^{(2)}$ → write as $E \rightarrow E^{(1)}$ or $M E^{(2)}$

M = Markers Non terminal to contain address of $E^{(2)}$

If E(1) is true no need to check $E^{(2)}$

If $E^{(1)}$ is false then check for $E^{(2)}$

$E \rightarrow E^{(1)}$ or $M E^{(2)}$

Backpatch($E^{(1)}$.False, M.Quad)

E.true := Merge($E^{(1)}$.true, $E^{(2)}$.true)

E.false := $E^{(2)}$.false

(2) $E \rightarrow E^{(1)}$ and $M E^{(2)}$

Backpatch ($E^{(1)}$.true, M.Quad)

E.false := Merge ($E^{(1)}$.false, $E^{(2)}$.false)

E.true := $E^{(2)}$.true

(3) $E \rightarrow$ not $E^{(1)}$

E.true := $E^{(1)}$.false

E.false := $E^{(1)}$.true

(4) $E \rightarrow (E^{(1)})$

E.true := $E^{(1)}$.true;

E.false := $E^{(1)}$.false;

(5) $E \rightarrow$ id

E.true := Makelist (NextQuad)

E.false := Makelist (NextQuad + 1)

Gen(if id.place GoTo —)

Gen(GoTo —)

(1) $E \rightarrow id^{(1)}$ relop $id^{(2)}$

E.true := Makelist(NextQuad)

E.false := Makelist(NextQuad+1)

Gen(if $id^{(1)}$.place relop $id^{(2)}$.place GoTo —)

Gen(GoTo —)

(7) $M \rightarrow \epsilon$

(M.Quad := Next Quad)

Q) $P < Q$ or $R < S$ and $T < U$

write the intermediate code and Annotated parse tree.

Soln: Intermediate code:

100: if $P < Q$ Go To 106

101: Go To 102

102: if $R < S$ Go To 104

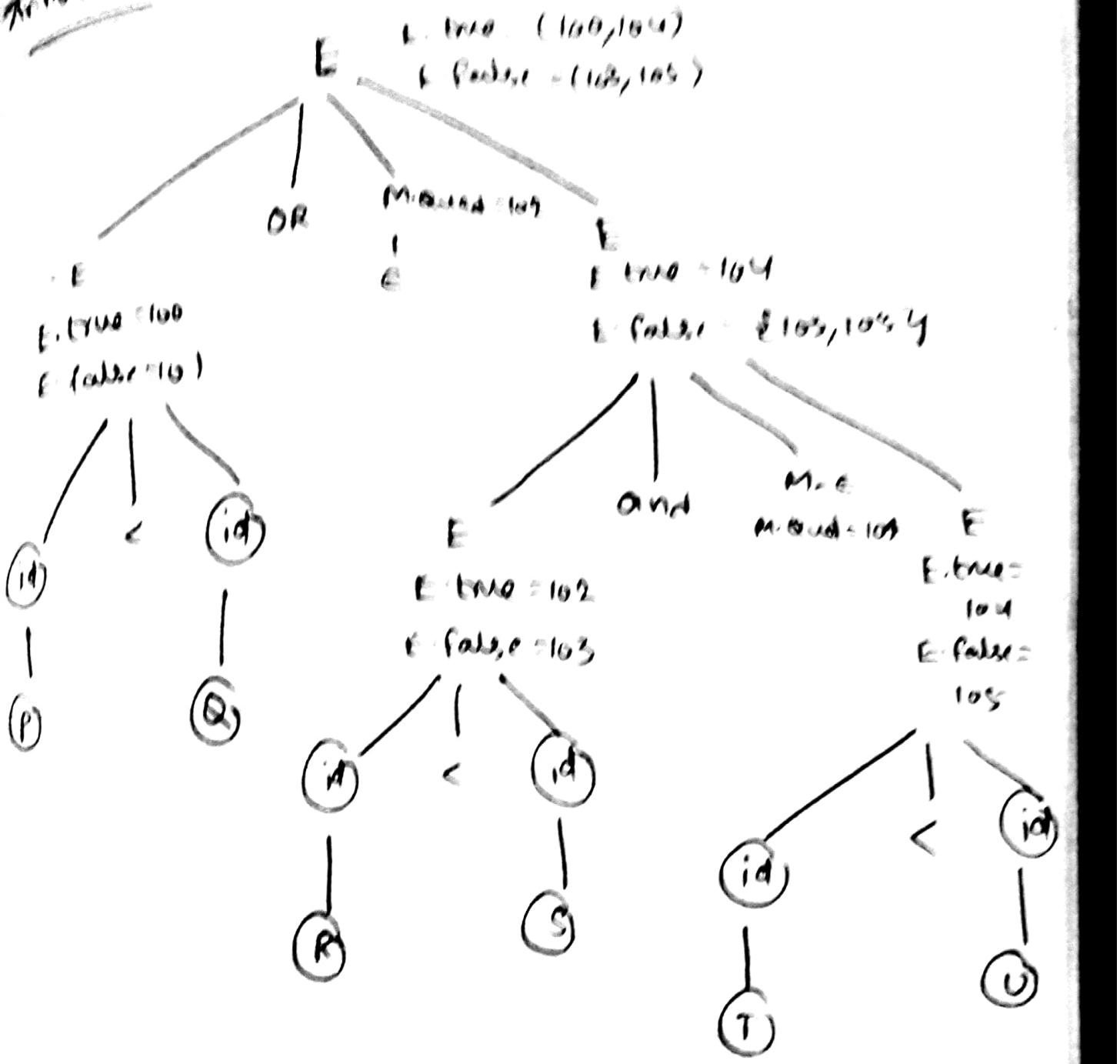
103: Go To 106

104: if $T < U$ Go To 106

105: Go To 106

10/3/18 106: END

With Annotated parse tree



(a) $S \rightarrow if\ (B)\ S_1$

B.true = Newlabel( )

B.false = S.next = S_1.next

S.code = B.code || Label(B.true) || S_1.code

Newlabel( )

↓

It is a label which generates a label for set of stmts.

Q) S → if (B) S1 else S2

B.true = NewLabel()

B.false = NewLabel()

$S_1.next = S_2.next = S.next$

S.code = B.code || Label (B.true) || S_1.code ||
Gen(goto S.next) || Label (B.false) ||
S_2.code

Q) S → while(B) S1

Mixed mode expression:-

// E.mode

T := New Temp( )

if $E^{(1)}$.mode = int and $E^{(2)}$.mode = int

begin

Gen (T = $E^{(1)}$.place int op $E^{(2)}$.place)

E.mode = ~~int~~

end

else

if $E^{(1)}$.mode = real and $E^{(2)}$.mode = real

begin

(T = $E^{(1)}$.place realop $E^{(2)}$.place)

E.mode := real

else

if (E'.mode = int) then

U := NewTemp()

U := intoreal E(1).place

Gen(T := U realop E(2).place)

E.mode := Real

else

U := NewTemp()

U := intoreal(E(2).place)

Gen(T := E(1).place realop U)

E.mode := Real

END.

Eg for S-attributed and L-attributed definitions:

(a) $T \rightarrow FT'$ $\quad T'.inh = F.val$

$T' \rightarrow *FT_1'$ $\quad T_1'.inh = T'.inh * F.val$

$A \rightarrow BC$

$B.val = C.val$

Q) A[I] = B

B = A[J]

	Op	Arg 1	Arg 2
0	[]=	A	I
1	assign	(0)	B

Post Fix translation;

Eg:- $E \rightarrow E^1 + E^2$ { } ← semantic rules

→ If the semantic rules are at the right side of the productions then it is called "post fix translation"

→ If post fix translation is S-attributed then simplify it by using stack.