10th March '17
Lecture - 37 Unit - 3

Semantic Analysis

* Syntax directed translation : (SDT)

It is a systematic process of assigning meaning to program which can also be viewed as computation of some special information (attributes) associated with non-terminals of the program.

To accomplish the task of specification of syntax directed translation there are 2 approaches :

a.) Syntax directed Definations
b.) Syntax Directed Translation Schemes

* Syntax directed definations & translation schemes:

When we associate semantic rules with productions we use 2 notations:

i.) Syntax directed defination : → Gives high level specification for translation.
→ Hides many implementation details such as : Order of evaluation of semantic actions.

→ We associate a production rule with a set of semantic actions & we do not say when they will be evaluated.

ii) Syntax directed translation schemes :

→ Indicate the order of evaluation of semantic actions associated with production rule.

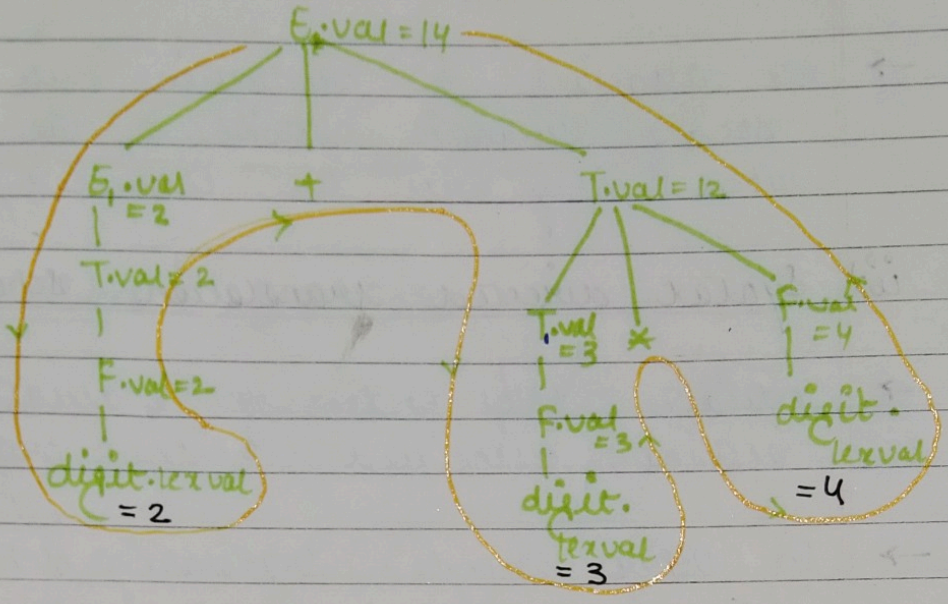
→ In other words, translation scheme gives little bit information about implementation details.

- Synthesized attributes are the attributes of the parents & this value depends upon the value of its children.
- Inherited attributes are the attributes of the children & this value depends upon the value of parents & its sibling.
- Annotated parse tree : when attributes are associated with the parse tree it is called Annotated parse tree.

Example :

	Semantic rule
$E \rightarrow E + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow digit$	$F.val = digit.lexval$

lexical value
It provides scanner
represent in the leaf.



→ Whenever we have reduction we go to production & carry out action.

17th March'17
Lecture-38

Synthesised & Inherited attributes:

In a syntax directed defination each production i.e; $A \to \alpha$ is associated with a set of semantic rule of forms $b = f(c_1, c_2, \cdots c_n)$ where f is a function and b can be one of the following:

i) b is a <u>synthesised attribute</u> of A & $c_1, c_2, \cdots c_n$ are attributes of the grammer symbols in the production $A \to \alpha$.

ii) b is a inherited attribute one of the grammar symbols in α & $c_1, c_2, \ldots c_n$ are attributes of the grammar symbols in the production $A \to \alpha$.

iii) Terminals have only synthesized attributes whose values are provided by the scanner or LA.

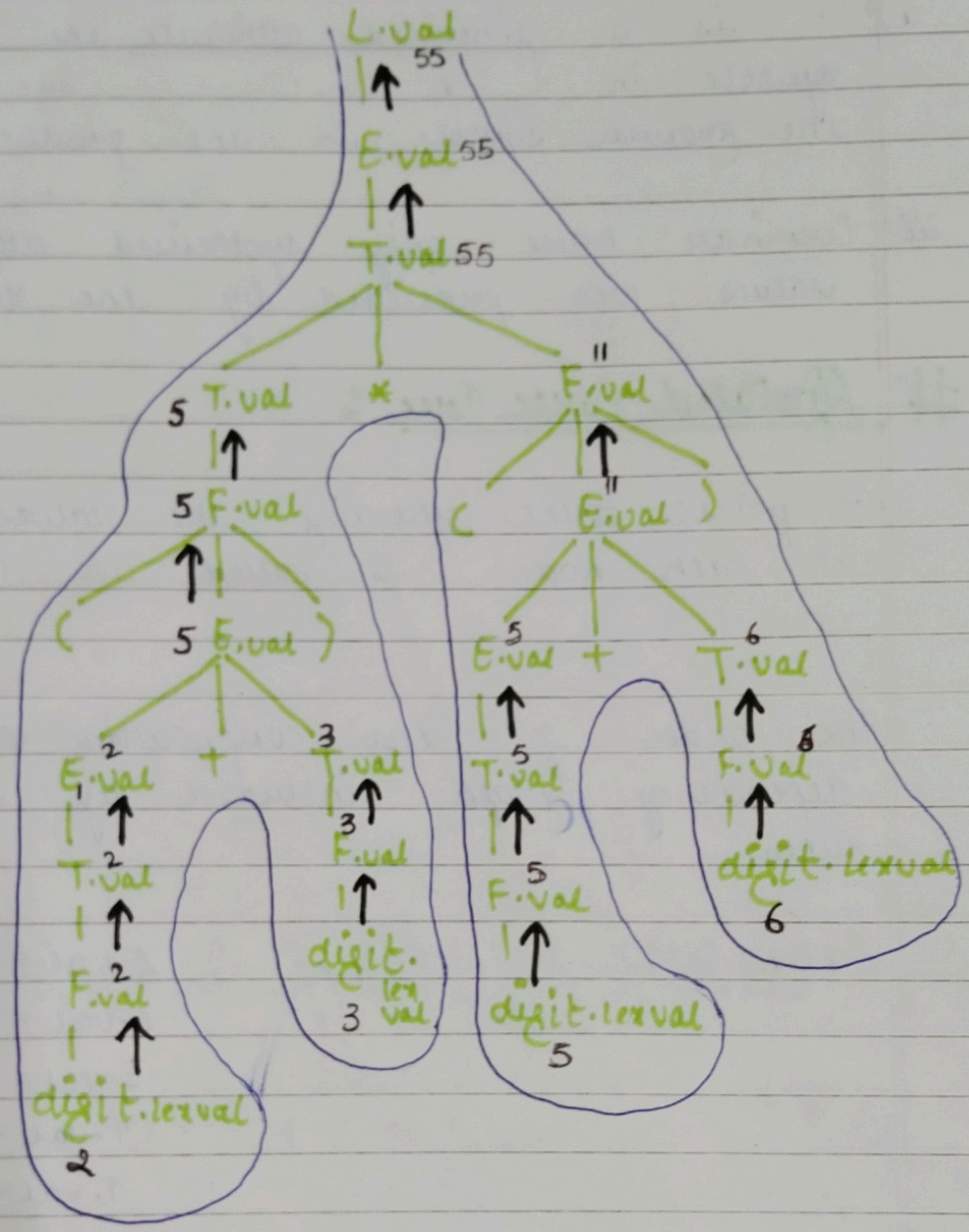
Annotated Parse Tree :

A parse tree showing the values of attributes at each node is called an Annotated parse tree.

The order of the computation depends on the dependency graph induced by the semantic rules.

Example :

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


L.val
55
E.val 55
T.val 55
5 T.val
* 11 F.val
5 F.val
(11 E.val)
(5 E.val)
5 E.val + 6 T.val
2 E1.val + 3 T.val
T.val 5
F.val 6
T.val 2
F.val 3
F.val 2
digit.lexval 5
digit.lexval 6
digit.lexval 2
digit.lexval 3

24th March'17
Lecture-39

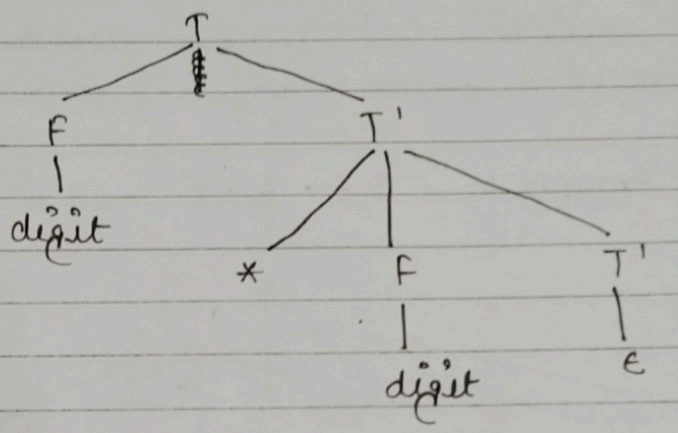
Example : $T \to FT'$ $T.val = T'.syn$ | 3 * 5 $T'.inh = F.val$

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

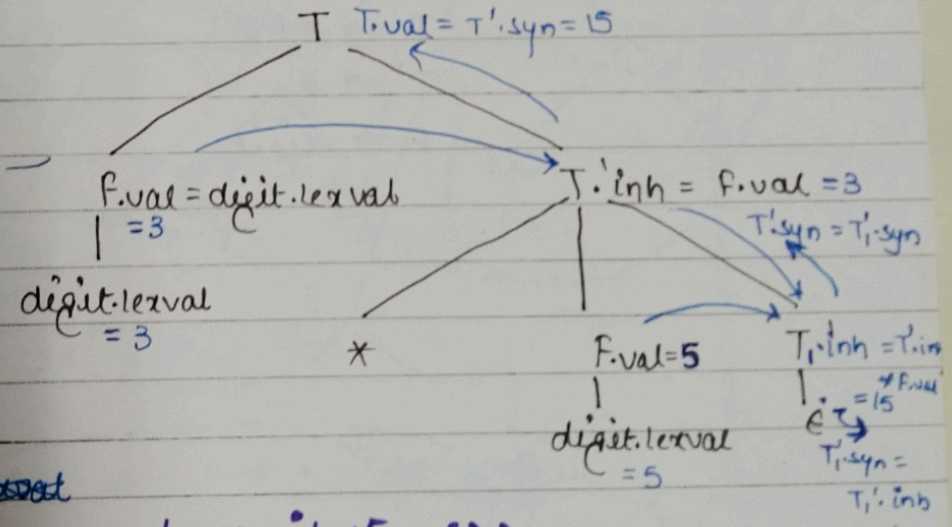
$T' \to \epsilon$ $T'.syn = T'.inh$

$F \to digit$ $F.val = digit.lexval$

Parse Tree :



Annotated P.T :



~~F.val = digit.lexval~~

L-attribute SDD

* $A \to BC$ $A.s = B.b$ Synthesized / Inherited ?

$B.i = f(C.c, A.s)$

Not S-attribute & L-attribute SDD.

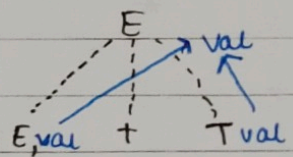
".i" is present, so not S-attribute

Value of B.i is inherited from parent "A.s" & right sibling "C.c" so not L-attribute

Evaluation orders for SDD :

- Dependency Graphs are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.
- An annotated parse tree shows the values of attributes & the dependency graph helps how those values can be computed.

Example : $E \rightarrow E_1 + T$ $\quad E.val = E_1.val + T.val$

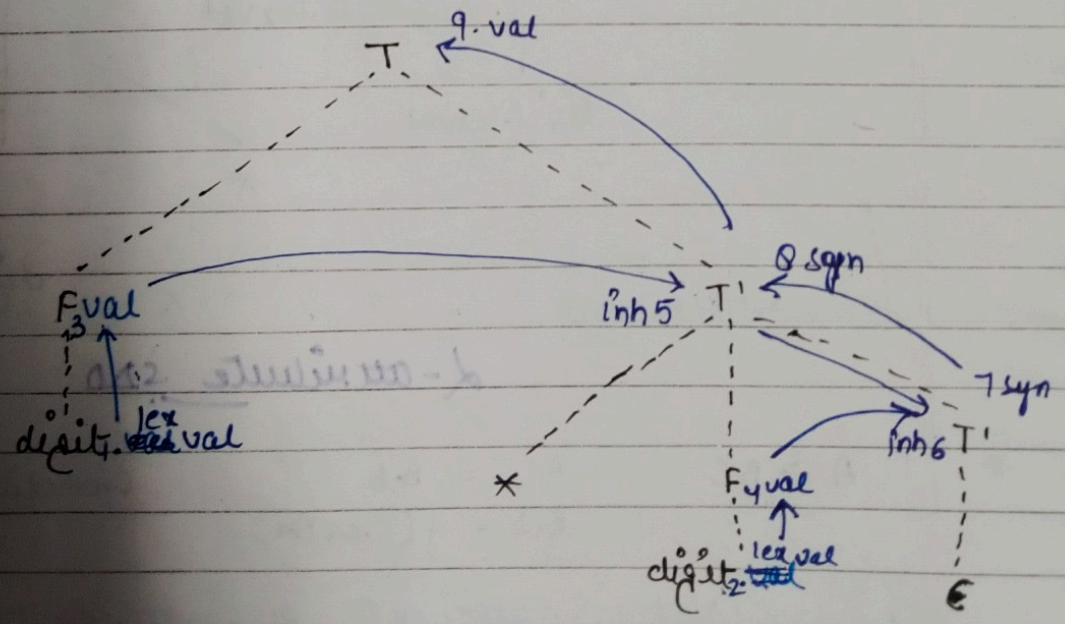


Example : $T \rightarrow FT'$ $\quad T.val = F.val$

$T' \rightarrow *FT'$

$T' \rightarrow \epsilon$

$F \rightarrow digit$



25th March'17

Lecture-40

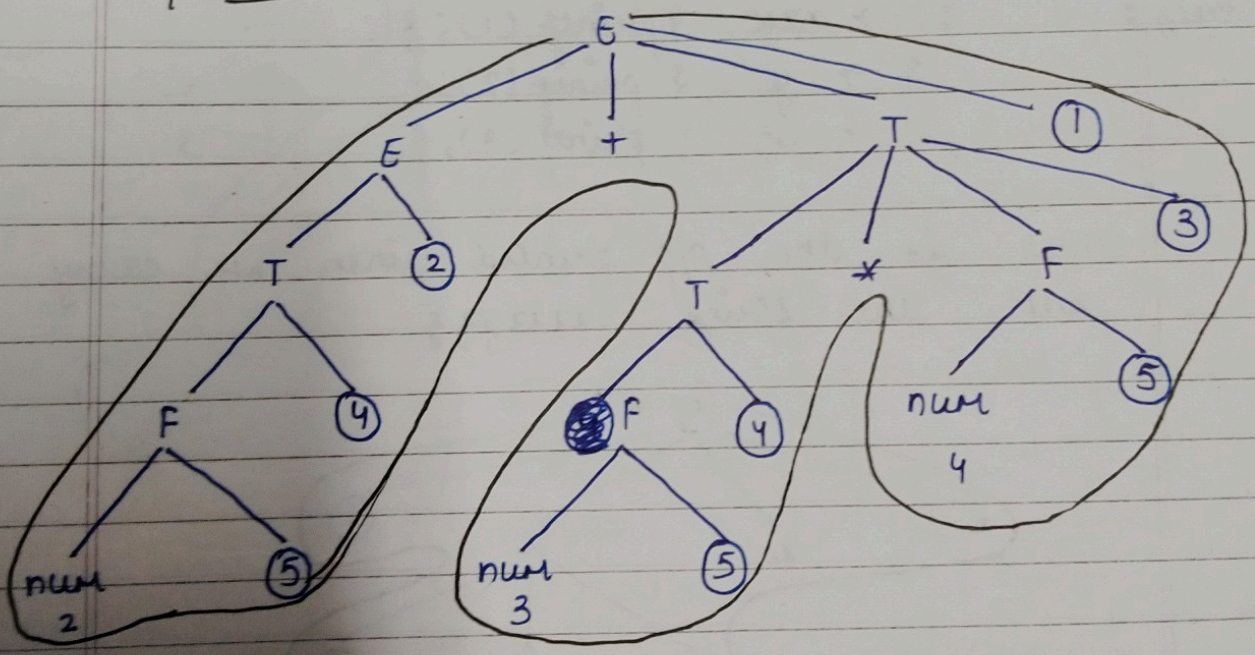
Syntax directed translation : (SDT)

Ques :

$E \rightarrow E+T$	{ print ("+"); } - (1)
$E \rightarrow T$	{ } - (2)
$T \rightarrow T*F$	{ print ("*"); } - (3)
$T \rightarrow F$	{ } - (4)
$F \rightarrow num$	{ print (num.lexval); } - (5)

2 + 3 * 4

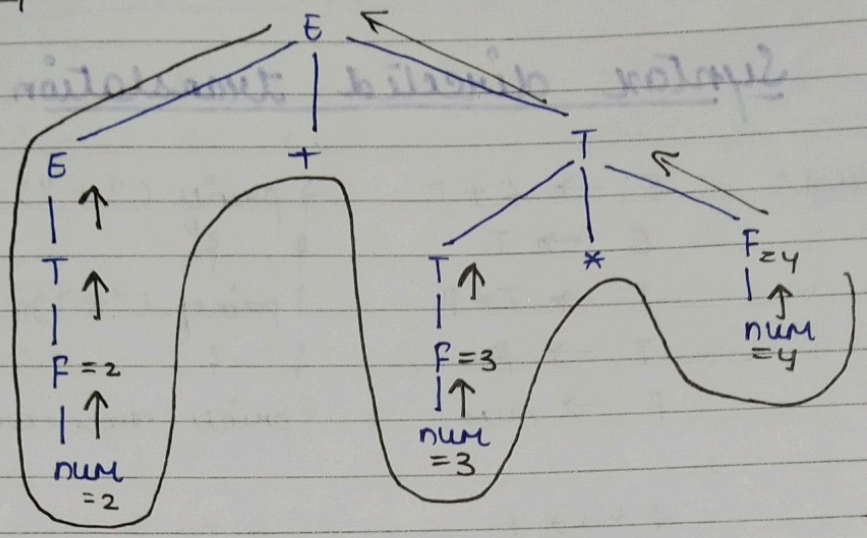
Top-down :



2 3 4 * + → postfix

Bottom-up :

(Reduce)



2 3 4 * +

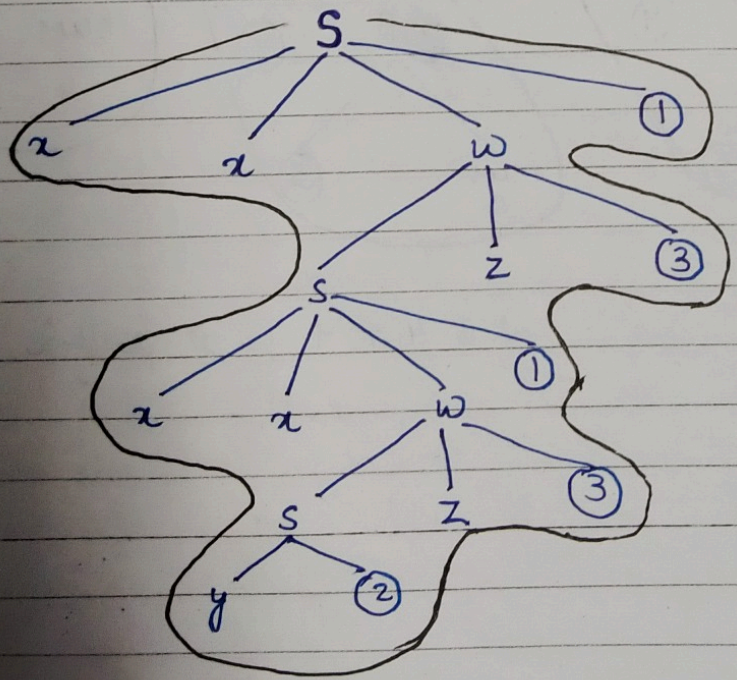
Ques :

$S \rightarrow xxw$ { printf (1); } — 1

$S \rightarrow y$ { printf (2); } — 2

$W \rightarrow Sz$ { printf (3); } — 3

What is the o/p printed when we carry out the string xxxyzz.



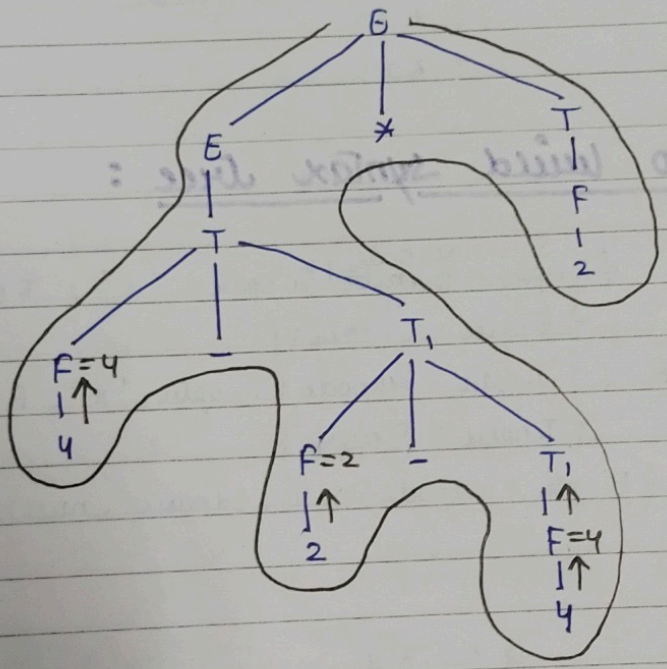
2 3 1 3 1 Ans.

Date: __/__/__
Pg:

Ques:

$E \rightarrow E_1 * T$	{ E.val = E_1.val * T.val; }
$E \rightarrow T$	{ E.val = T.val; }
$T \rightarrow F - T_1$	{ T.val = F.val - T_1.val; }
$T \rightarrow F$	{ T.val = F.val; }
$F \rightarrow 2$	{ F.val = 2 }
$F \rightarrow 4$	{ F.val = 4 }

what is the value of w = 4-2-4*2 & how many no. of reductions we will use?



precendence - ↑ *
- right asso.
* left asso.

(4-(2-4)) * 2
(4+2) * 2
6 * 2
12

No. of reductions = 10

28th March'17
Lecture-41

$E \rightarrow E \# T$	{ E.val = E.val * T.val; }
$\mid T$	{ E.val = T.val; }
$T \rightarrow T \& F$	{ T.val = T.val + F.val; }
$\mid F$	{ T.val = F.val; }
$F \rightarrow num$	{ F.val = num.lval; }

what is the o/p generated for string 2#3&5#6&4

precedence : '&' > '#' as & is away from E.

⇒ '+' > '*' & both are left associative.

if 5 marks then parse tree

2#3&5#6&4

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

((2 * 8)* 10) (∵ left ass so start from left)

16 * 10

160

SDT to build syntax tree :

E → E1 + T { E.nptr = ~~E.nptr~~ Mknode (E1.nptr, '+', T.nptr); }

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

T → T1 * F { T.nptr = Mknode (T1.nptr, '*', F.nptr); }

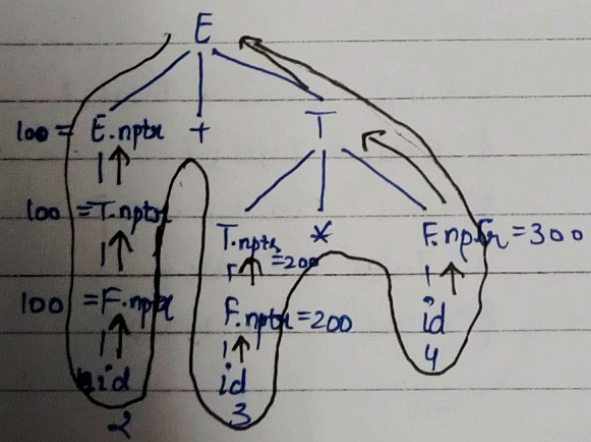
| F { T.nptr = F.nptr; }

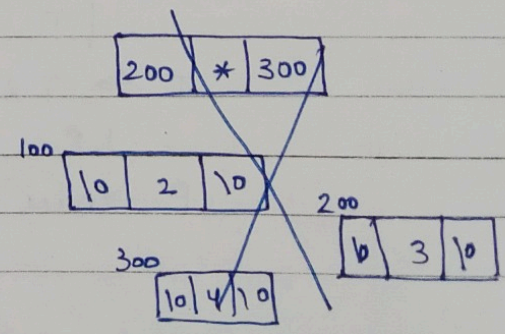
F → id { E.nptr = mknode(null, idname, null); }

node pointer make node

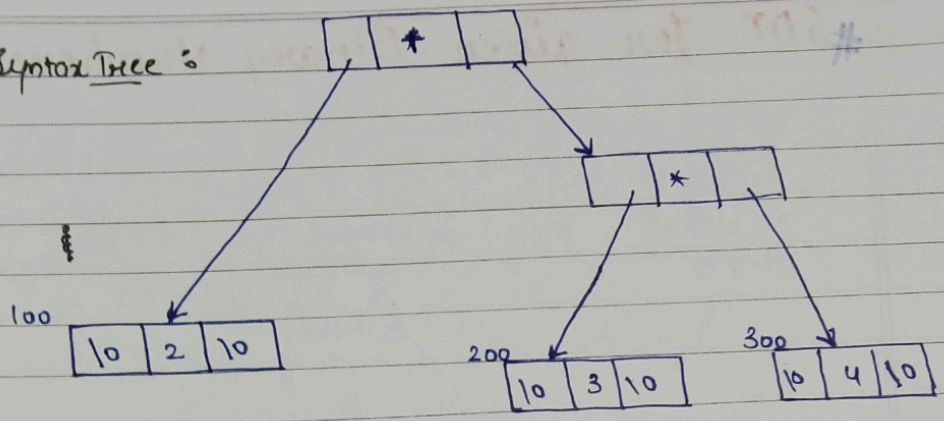
Example : 2 + 3 * 4

Parse Tree :





Syntax Tree :



* SDT for typecheck :

$E \rightarrow E_1 + E_2$ { if ((E1.type == E2.type) && (E.type = int)) then E1.type = int, else error; }

| $E_1 == E_2$ { if ((E1.type == E2.type) && (E1.type = int/bool)) then E.type = bool else error; }

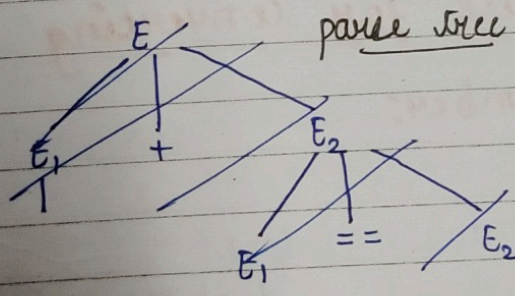
| (E_1) { E.type = E1.type }

| num { E.type = int; }

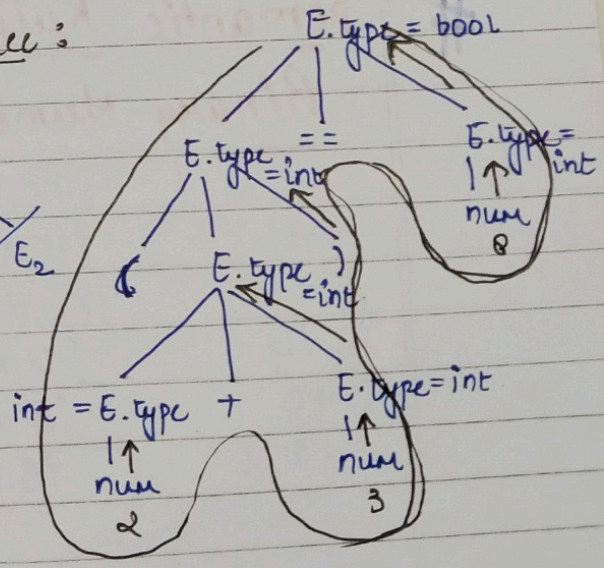
| True { E.type = bool; }

| False { E.type = bool; }

Example : (2+3) == 8



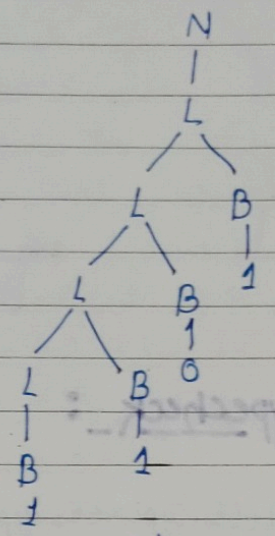
parse tree :



29th March '17
Lecture-42

SDT for given Binary Number:

$N \to L$
$L \to L\ B \mid B$
$B \to 0 \mid 1$



1101

1's	0's	Total no. of bits
$\{N.L = L.C\}$	$\{N.L = L.C\}$	$\{N.L = L.C\}$
$\{L.C = L_1.C + B.C\}$	$\{L.C = L_1.C + B.C\}$	$\{L.C = L_1.C + B.C\}$
$\{L.C = B.C\}$	$\{L.C = B.C\}$	$\{L.C = B.C\}$
$\{B.C = 0\}$	$\{B.C = 1\}$	$\{B.C = 1\}$
$\{B.C = 1\}$	$\{B.C = 0\}$	$\{B.C = 1\}$

Semantic Rule for converting into Binary Number:

$N \to L$	$\{N.dval = L.dval\}$
$L \to LB$	$\{L.dval = L_1.dval * 2 + B.dval\}$
$\mid B$	$\{L.dval = B.dval\}$
$B \to 0$	$\{B.dval = 0\}$
$\mid 1$	$\{B.dval = 1\}$

1010 = 5

10

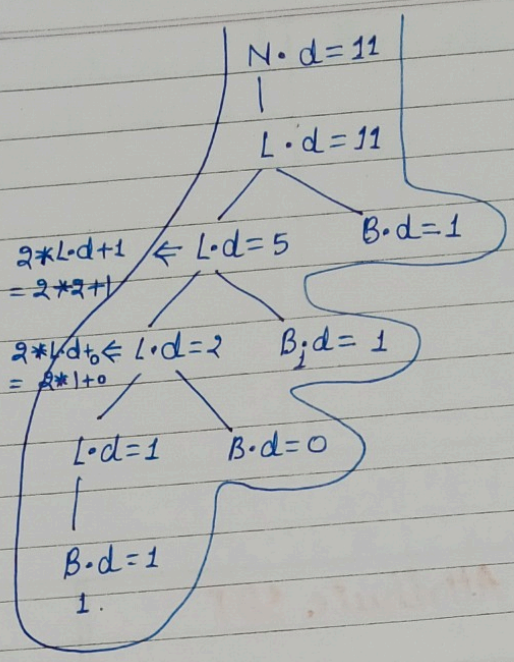
1x2+0 = 2

101

2x2+1 = 5

1011

5x2+1 = 11



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

$L \rightarrow L_1 B$ { L.c = L_1.c + B.c ; L.dval = L_1.dval * 2 + B.dval }

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

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

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

1101.101 .101 = 5

5 0.

0.

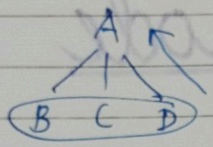
Q: 1101.101

#

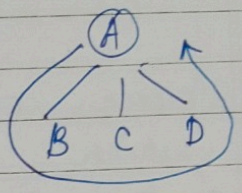
S-Attribute SDT	L-Attribute SDT
uses only Synthesized attribute	uses both inherited & Synthesized attributes Each inherit att. is restricted to inherit either from parent or left sibbling only. $A \to XYZ \{ Y.S = A.S,$ $Y.S = X.S$ $Z.S = Y.S \}$
Semantic actions are placed at right End production. $A \to B\ C\ \{\}$	Semantic actions are placed anywhere on R.H.S $A \to \{\}\ B\ C$ $A \to B\ \{\}\ C$ $A \to B\ C\ \{\}$

Date: / /
Page:

Attributes are Evaluated during Bottom up parsing.



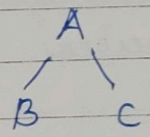
Att. are Evaluated by traversing the parse tree .- depth first, left to right.



Q:1 $A \to LM$ $\{L.i = f(A.i);\ M.i = f(L.s);\ A.s = f(m.s);\}$
$A \to QR$ $\{R.i = f(A.i);\ Q.i = f(R.i);\ A.s = f(a.s);\}$
right

(a) $\S$-att
(b) L-att
(c) both
(d) none ✓

Q:2 $A \to BC$ $\{B.s = A.s;\}$
Lattribute



31st March '17
Lecture - 43

Unit - 3 (5)

Intermediate Code

Source code is translated into a language which is intermediate in complexity b/w a programming language & machine code, such a language is called Intermediate code or Intermediate text.

There are 4 kinds of intermediate code used in compilers:

1. Postfix Notation
2. Syntax Tree
3. Quadruples } 3 address code
4. Triples

* 3-address code, Quadruples & Triples:

→ 3-address code is a sequence of statements of a form A := B op C where A, B & C are either programmer defined names or constants or compiler generated temporary names. Op stands for any operator such as fixed or floating point arithmetic operator or a logical operator.

Example: 1.) A := B op C → atmost 1

diff. address (there will be diff. add. of A, B & C i.e; 3 add. will be there so, 3 add. code).

2.) $x = y + z * k$ not a 3 add. code

$t1 = y + z$

$x = t1$

$t_1 = z * k$

$t_2 = y + t_1$

$x = t_2$

→ 3 add. code for the above example

copy instruction it is 3 add. bcz there must be atmost 1 op. i.e; 1 or less than 1 op. can be there

3.) $x = -b * c$

'-' has more precedence

$t1 = -b$

$x = t1 * c$

4.) $a = b * -c + b * -c$, convert into 3 add. code

optimized:

$t1 = -c$

$t2 = b * t1$

$a = t2 + t2$

$t1 = -c$

$t2 = b * t1$

$t3 = -c$

$t4 = b * t3$

$t5 = t2 + t4$

$a = t5$

→ Quadruples has 4 fields i.e; "op", "arg1", "arg2" & "result".

Example :

```
t1 = -c
t2 = b * t1
t3 = -c          → 3 add. code
t4 = b * t3
t5 = t2 + t4
a = t5
```

quadruple ⇒

	op	arg1	arg2	result
0	minus	c		t1
1	*	b	t1	t2
2	minus	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

triple ⇒

direct triple

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

→ Triples have only 3 fields i.e; "op", "arg1" & "arg2". Using triples we refer to the result of an operation x op y by its position rather than by an explicit temporary name.

1st April '17
Lecture-44.

Indirect triple :

	op	arg 1	arg 2
135	minus	c	
136	*	b	(135)
137	minus	c	
138	*	b	(137)
139	+	136	(138)
140	=	a	(139)

(0)	135
(1)	136
(2)	137
(3)	138
(4)	139
(5)	140

Indirect triples consists of a listing of pointers to triples rather than a listing of triples themselves.

Q.

1.) $x = a*b+c*d+e*f+g*h$

2.) $x = a+(a+(b-c))+((b-c)*d))$

3.) $A = -B*(C-D)$

4.) $w = (x*y)+(y+z)+(x+y+z)$

5.) $m = c+d*e/f+g+h$

1.) $x = a*b+c*d+e*f+g*h$

$t1 = a*b$
$t2 = t1+c$
$t3 = t2*d$
$t4 = t3+e$
$t5 = t4*f$
$t6 = t5+g$
$t7 =$

$t1 = a*b$
$t2 = c*d$
$t3 = e*f$
$t4 = g*h$
$t5 = t1+t2$
$t6 = ~~t5+t3~~ ~~t3+t4~~ \; t5+t3$
$t7 = ~~t5+t6~~ \; t6+t4$
$x = t7$

quadruple :

	op	arg1	arg2	result
0	*	a	b	t1
1	*	c	d	t2
2	*	e	f	t3
3	*	g	h	t4
4	+	t1	t2	t5
5	+	t5	t3	t6
6	+	t6	t4	t7
7	=	t7		x

triple :

	op	arg1	arg2
0	*	a	b
1	*	c	d
2	*	e	f
3	*	g	h
4	+	(0)	(1)
5	+	(4)	(2)
6	+	(5)	(3)
7	=	x	(6)

2. $x = \{a + (a + (b - c)) + ((b - c) * d))$

$t1 = b - c$

$t2 = a + t1$

$t3 = b - c$

$t4 = t3 * d$

$t5 = a + t2$

$t6 = t5 + t4$

$x = t6$

quadruple :

	op	arg 1	arg 2	result
0	−	b	c	t1
1	+	a	t1	t2
2	−	b	c	t3
3	*	t3	d	t4
4	+	a	t2	t5
5	+	t5	t4	t6
6	=	t6		x

triple :

	op	arg 1	arg 2
	−	b	c
	+	a	(0)
	−	b	c
	*	(2)	d
	+	a	(1)
	+	(4)	(3)
	=	x	(5)

3. A = −B * (C − D)

t1 = −B

t2 = C − D

t3 = t1 * t2

A = t3

quadruple :

	op	arg 1	arg 2	result
(0)	minus	B		t1
(1)	−	C	D	t2
(2)	*	t1	t2	t3
(3)	=	t3		A

triple :

op	arg 1	arg 2
minus	B	
−	C	D
*	(0)	(1)
=	A	(2)

4. w = (x+y) + (y+z) + (x+y+z)

t1 = x+y ✓

t2 = y+z ✓

t3 = x+y

t4 = t3+z ✓

t5 = t1+t2

t6 = t5+t4

w = t6 ✓

quadruple :

	op	arg1	arg2	result
(0)	+	x	y	t1
(1)	+	y	z	t2
(2)	+	x	y	t3
(3)	+	t3	z	t4
(4)	+	t1	t2	t5
(5)	+	t5	t4	t6
(6)	=	t6		w

triple :

op	arg1	arg2
+	x	y
+	y	z
+	x	y
+	(2)	z
+	(0)	(1)
+	(4)	(3)
=	w	(5)

5. m = c + a * e / f + g + h

t1 = d * e

t2 = t1 / f

t3 = c + t2

t4 = t3 + g

t5 = t4 + h

m = t5

quadruple :

	op	arg1	arg2	result
(0)	*	d	e	t1
(1)	/	t1	f	t2
(2)	+	c	t2	t3
(3)	+	t3	g	t4
(4)	+	t4	h	t5
(5)	=	t5		m

triple :

op	arg1	arg2
*	d	e
/	(0)	f
+	c	(1)
+	(2)	g
+	(3)	h
=	m	(4)

* 3-address code for FOR & WHILE loop :

→ **FOR :**

```
for(i=1; i<=20; i++)
{
    x = x+y;
}
```

3-address code for FOR :

1. i = 1
2. if i<=20 go to 7
3. go to 10
4. t1 = i + 1
5. i = t1
6. go to 2
7. t2 = x+y
8. x = t2
9. go to 4
10. Next statement

3rd April '17
Lecture-45

→ **While :**

```
While(a<c and b<d)
{
    if (a=1)
        c=c+1
    else
        while(a<=d)
            a = a+3
}
```

1. if (a<c) go to 3
2. go to 15
3. if (b<d) go to 5
4. go to 15
5. if (a=1) go to 7
6. go to 10
7. t1 = c+1
8. c = t1
9. go to 1
10. if (a<=d) go to 12
11. go to 1
12. t2 = a+3
13. a = t2
14. go to 10
15. Next statement

* 3-address code for Boolean :

→ A < B or C

1. if A < B go to 4
2. t1 = 0
3. go to 5
4. t1 = 1
5. t2 = t1 or C

→ a < b and c > d

1. if a < b go to 4
2. t1 = 0
3. go to 5
4. t1 = 1
5. if c > d go to 8
6. t2 = 0
7. go to 9
8. t2 = 1
9. t3 = t1 and t2

→ a < b or c < d and e < f

1. if a < b go to 4
2. t1 = 0
3. go to 5
4. t1 = 1
5. if c < d go to 8
6. t2 = 0
7. go to 9
8. t2 = 1
9. if e < f go to 12
10. t3 = 0

11. go to 13
12. t3 = 1
13. t4 = t2 and t3
14. t5 = t1 or t4

→
```
if (x<y)
{
  while (z>d)
      a = a+b;
  { do
      p = p+q;
      while(e<=f);
  }
}
```

if (x<y) go to 3
~~if (z>d) go to~~
~~t1=0~~ t1 = a+b
~~go to~~ a = t1
~~t1=1~~

1. if (x<y) go to 3
2. go to 11
3. if (z>d) go to 5
4. go to 8
5. t1 = a+b
6. a = t1
7. go to 3
8. t2 = p+q
9. p = t2
10. if e<=f go to 8
11. next statement

3-address code for same number

→
```
if (x<y)
{
  while(z<d)
  a = a+b;
}
else
{
   do
     p = p+q;
     while(e<=f);
}
```

1. if (x<5) go to 3
2. go to 8
3. if z>d go to 5
4. go to 11
5. t1 = a+b
6. a = t1
7. go to 3
8. t2 = p+q
9. p = t2
10. if e<=f go to 8
11. next statement

4th April '17
Lecture-46

3-address code for array element :

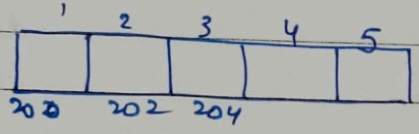
Q.

```
sum = 0;
for(i = 1; i <= 2; i++)
{
  sum = sum + a[i];
}
```

$Loc\ [a[i]] = base + (i - lb) * w$

$= base + (i - 1) * w$

$= i * w + base - w$



$i = 2,\ w = 2,\ base = 200$

$(200 - 2) + (2 \times 2) = 202$

1-Darray

1. sum = 0
2. i = 1
3. if i <= 2 go to 8
4. go to 14
5. $t_1 = i + 1$
6. $i = t_1$
7. go to 3
8. $t_2 = i * w$
9. $t_3 = add(a) - w$
10. $t_4 = t_3[t_2]$
11. $t_5 = sum + t_4$
12. $sum = t_5$
13. go to 5
14. next statement

int a[5] = {5, 10, 15, 20, 30}

0	1	2	3	4
5	10	15	20	30

base address = 5000

5000 + 120

a[2] = * (5000 + 0)

base addrn + w * i

$x = a[i]$

$t1 = 4 * i$

$t2 = a + 4 * i$

$t3 = *t2$

t1 = 4 * i

t2 = addr(a) + t1

t3 = *t2

Ques: 3-address code for SDT

```
for (i=1; i<=2; i++)
{
  sum = sum + a[i] * b[i];
}
```

Row wise:

1. sum = 0
2. i = 1
3. if i<=2 go to 8
4. go to 17
5. $t_1 = i + 1$
6. $i = t_1$
7. go to 3
8. $t_2 = i * w$
9. $t_3 = add(a) - w$
10. $t_4 = t_3[t_2]$
11. $t_5 = i * w$
12. $t_6 = add(b) - w$
13. $t_7 = t_6[t_5]$
14. $t_8 = t_4 * t_7$
15. $t_9 = sum + t_8$
16. $sum = t_9$
17. go to 5
18. next statement

Col wise:

1. sum = 0
2. i = 1
3. if i<=2 go to 8
4. go to 17
5. $t_1 = i + 1$
6. $i = t_1$
7. go to 3.
8. $t_2 = i * w$
9. $t_3 = add(a) - w$
10. $t_4 = t_3[t_2]$
11. $t_5 = i * w$
12. $t_6 = add(b) - w$
13. $t_7 = t_6[t_5]$
14. $t_8 = t_4 * t_7$
15. $t_9 = sum + t_8$
16. $sum = t_9$
17. go to 5
18. next statement

1. i = 0
2. if i<10 go to 5
3. go to 5
4. $t1 = i * i$
5. $t2 = 4 * i$
6. $t3 = b + t2$
7. $*t3 = *t7$
8. .

$a[4] = *(a+4)$

$t2 = i * w$

$t3 = addr(b) - w$

$t4 = t3[t2]$

3-address code for 2D array :

Row major :

loc (a[i, j]) = base + [(i - lb) * n_2 + (j - lb)] * w

where :

n_2 - no. of columns

n_1 - no. of rows

= [(i * n_2) + j] w + base - [(n_2 + 1) * w]

column major :

loc (a[i, j]) = base + [(j - lb) * n_1 + (i - lb)] * w

= [(j * n_1) + i] w + base - [(n_1 + 1) * w]

Q.

```
add = 0;
i = 1;
j = 1;
do
{
    add = add + a[i, j] + b[j, 1];
    i = i + 1
    j = j + 1
} while (i <= 20 and j <= 20)
```

21/4

#10

w = 4

1. add = 0;
2. i = 1
3. j = 1
4. $t_1 = i * 20$ $(i * n_2)$
5. $t_2 = t_1 + j$ $(t_1 + j)$
6. $t_3 = t_2 * 4$ $* w$
7. $t_4 = add[a] - 84$ $+ base - [(n_2+1) * w]$

— Row major

8. $t_5 = t_4[t_3]$
9. $t_6 = j * 20$ $(j * n_1)$
10. $t_7 = t_6 + i$ $(t_6 + i)$
11. $t_8 = t_7 * 4$ $(* w)$
12. $t_9 = add[b] - 84$ $(base - [(n_1+1) * w]$
13. $t_{10} = t_9[t_6]$
14. $t_{11} = add + t_5$
15. $t_{12} = t_{11} + t_{10}$
16. add = t_{12}
17. $t_{13} = i + 1$
18. $i = t_3$
19. $t_{14} = j + 1$
20. $j = t_{14}$
21. if $i <= 20$ go to 23
22. go to 25
23. if $j <= 20$ go to 4
24. go to 25
25. Next statement

— column major

5th April '17
Lecture - 47

* 3- address code for switch case statement :

→
```
switch(i+j)
{
 case 1: x=y+z;
         break;
 case 2: u=v+w;
         break;
 default: p=q+r;
         break;
}
```

1. $t_1 = i+j$
2. if $t_1 = 1$ go to 5
3. if $t_1 = 2$ go to 8
4. go to 11
5. $t_2 = y+z$
6. $x = t_2$
7. go to 13
8. $t_3 = v+w$
9. $u = t_3$
10. go to 13
11. $t_4 = q+r$
12. $p = t_4$
13. next statement

→
```
switch(a+b)
{
 case 2: {
          x=y;
          break;
         }
 case 5: {
          switch(x)
          {
           case 0: {
                    a=b+1;
                    break;
                   }
```

1. $t_1 = a+b$
2. if $t_1 = 2$ go to 8
3. if $t_1 = 5$ go to 11
4. if $t_1 = 9$ go to 23
5. $t_2 = 2$
6. $a = t_2$
7. go to 25
8. $t_3 = y$
9. $x = t_3$
10. go to 25
11. $t_4 = x$
12. if $t_4 = 0$ go to 17
13. if $t_4 = 1$ go to 20

```
case 1 : {
            a = b+3;
            break;
            }
default : {
            a = 2;
            break;
          }
        }
        break;
      }
case 9 : {
            x = y-1;
            break;
          }
default : {
            a=2;
            break;
          }
        }
```

14. $t_5 = 2$
15. $a = t_5$
16. go to 25
17. $t_6 = b+1$
18. $a = t_6$
19. go to 25
20. $t_7 = b+3$
21. $a = t_7$
22. go to 25
23. $t_8 = y-1$
24. $x = t_8$
~~go to 25~~
25. Next statement

* 3-address code for Procedures :

```
add(int a, int b)
{
  int c;
  c = a+b;
  return(c);
}
main()
{
```

Proc_begin_add
1. $t_1 = a+b$
2. $c = t_1$
3. return c
4. go to L0
5. Proc_end

L0 : Proc_begin_main
1. $v_1 = 10$

```
int v1, v2, v3, v4;
v1 = 10
v2 = 20
v3 = add(v1, v2);
v4 = v3 + 5;
}
```

2. $v_2 = 20$
3. param v_1
4. param v_2
5. t_2 = call add, 2 → function, → 2 parameters
6. receive t_2
7. $v_3 = t_2$
8. $t_3 = v_3 + 5$
9. $v_4 = t_3$

→ SDT for assignment, boolean, array. (remaining topics).