Internal Test II, April 2018

Sub: System Software & Compiler Design

Sub code: 15CS63, Prepared by:

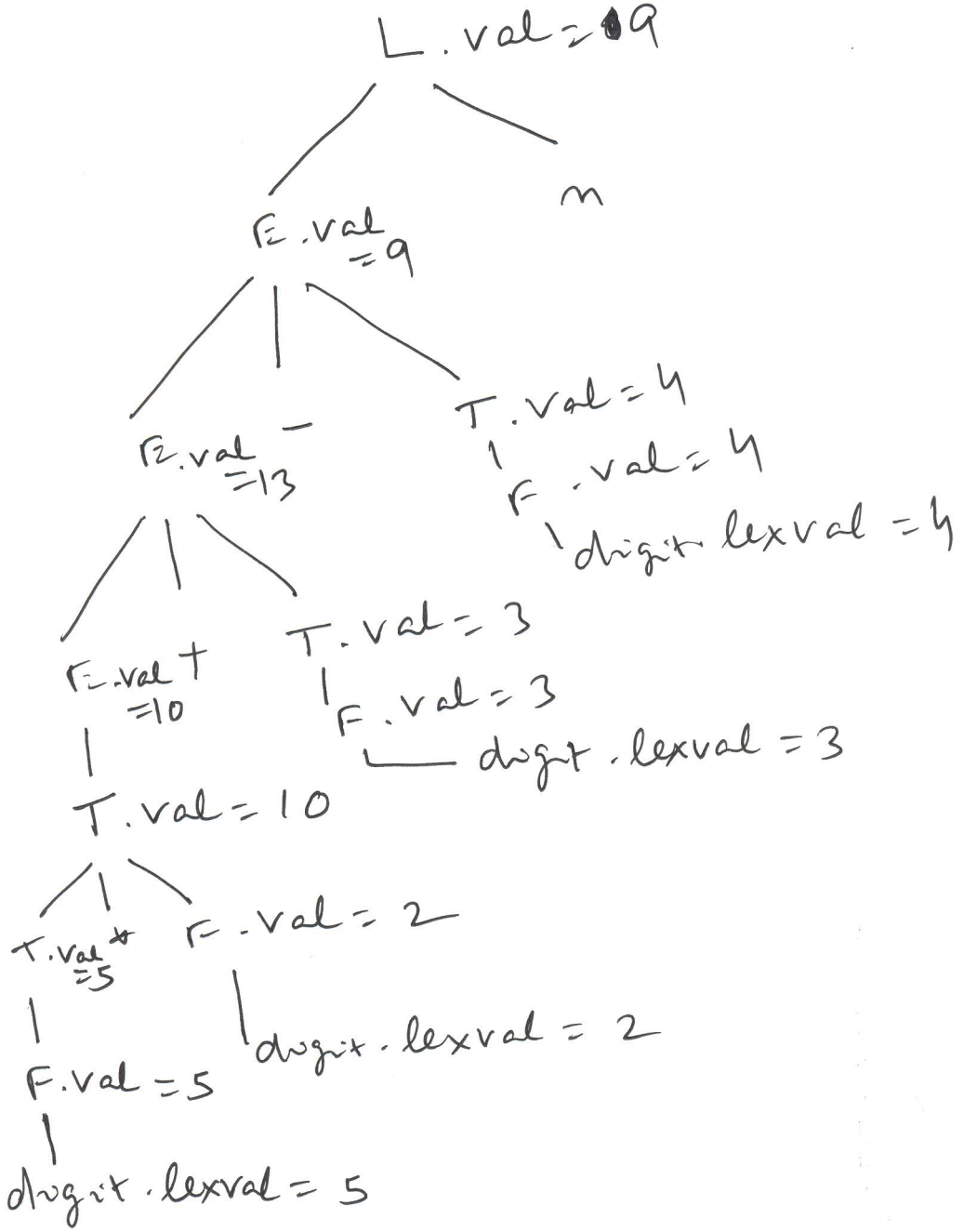
Sagarika Behera
Dept. of CSE

(1) A Syntax-directed Definition (SDD) is a context free grammar, together with attributes & rules. (1M)

SDD for desk calculator (4M)

Production	Semantic Rules
1) $L \rightarrow E\ n$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T.val * F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow digit$	$F.val = digit.lexval$

Annotated Parse tree for 5*2+3-4 (5M)



2.(a) Synthesized attribute [2X2.5]

A synthesized attribute at node N is defined only in terms of attribute values at the children of N & N itself.

Eg: E → E+T E.val = E.val + T.val

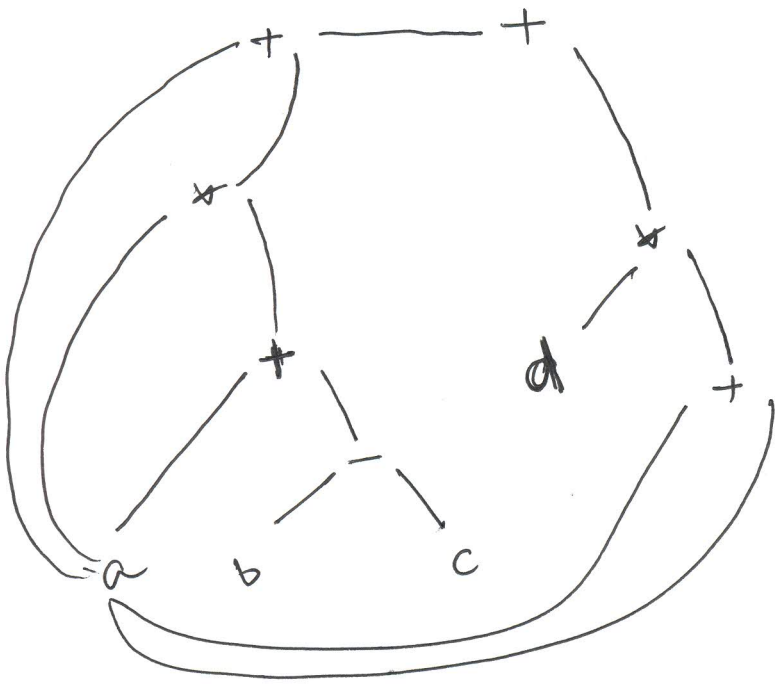
Inherited attribute

Inherited attribute for a nonterminal B at a parse tree node N is defined by a

semantic rule associated with the production at the parent of N. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings.

Eg: $E \rightarrow TE'$ $\quad E'.inh = T.val$

2.(b) $a + a * (a + (b-c)) + d * (a+a)$ [5M]



3.(a) After removing left recursion [5+2]

$E \rightarrow TE'$

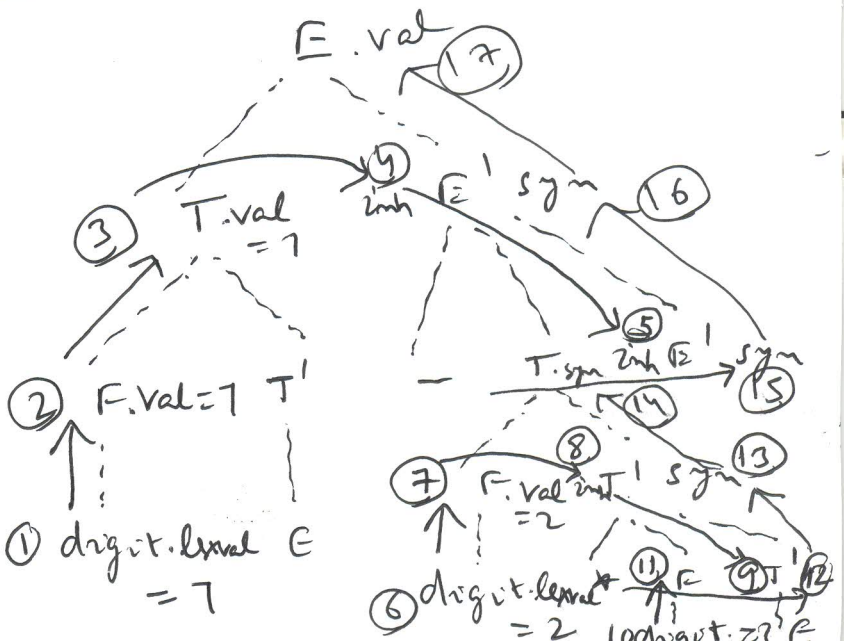
$E' \rightarrow -TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow digit$

1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17



3.(b) Need of Intermediate Code generation [2]

→ In the analysis-synthesis model of a computer, the front end analyzes a source program & creates an intermediate representation, from which the back end generates target code.

→ If there is no intermediate form, then for m source language & n target machine, it requires m x n translations.

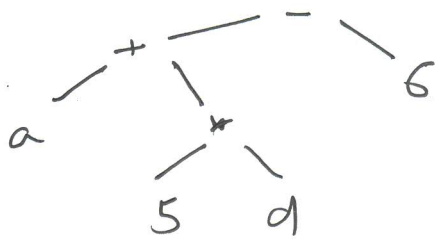
But with intermediate form it requires m+n translations.

[4+6]

4.

Productions	Semantic Rules
$E \rightarrow E + T$	E.node = new Node('+', E.node, T.node)
$E \rightarrow E - T$	E.node = new Node('-', E.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow T * F$	T.node = new Node('*', T.node, F.node)
$T \rightarrow T / F$	T.node = new Node('/', T.node, F.node)
$T \rightarrow F$	T.node = F.node
$F \rightarrow (E)$	F.node = E.node
$F \rightarrow$ digit	F.node = new Leaf(digit, value)

steps to construct tree for a+5*d-6



P_1 = new Leaf (digit, 5)

P_2 = new Leaf (id, entry for d)

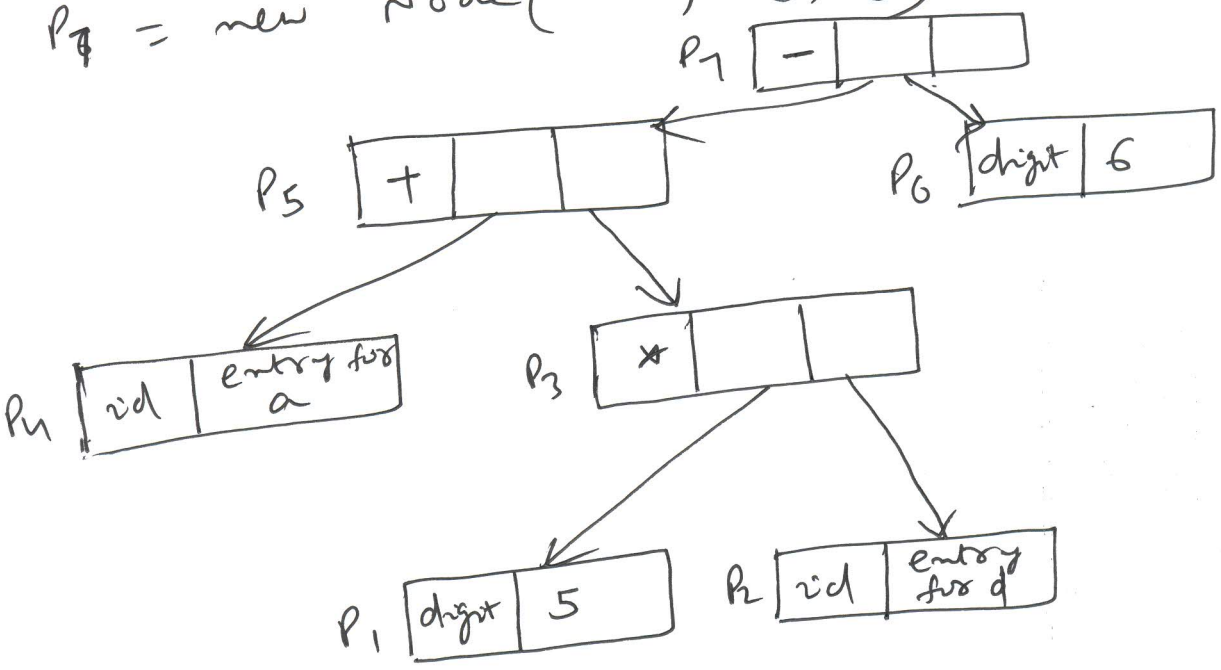
P_3 = new Node ('*', P_1, P_2)

P_4 = new Leaf (id, entry for a)

P_5 = new Node ('+', P_4, P_3)

P_6 = new Leaf (digit, 6)

P_7 = new Node ('-', P_5, P_6)



5.(a) Different ways of representing Intermediate form

---

① 3-address code

② Directed Acyclic Graph (DAG)

③ Postfix Notation or Polish Notation

3-add code $(a-b) * (c+d) - (a+b)$ [2x3]

$t_1 = a - b$

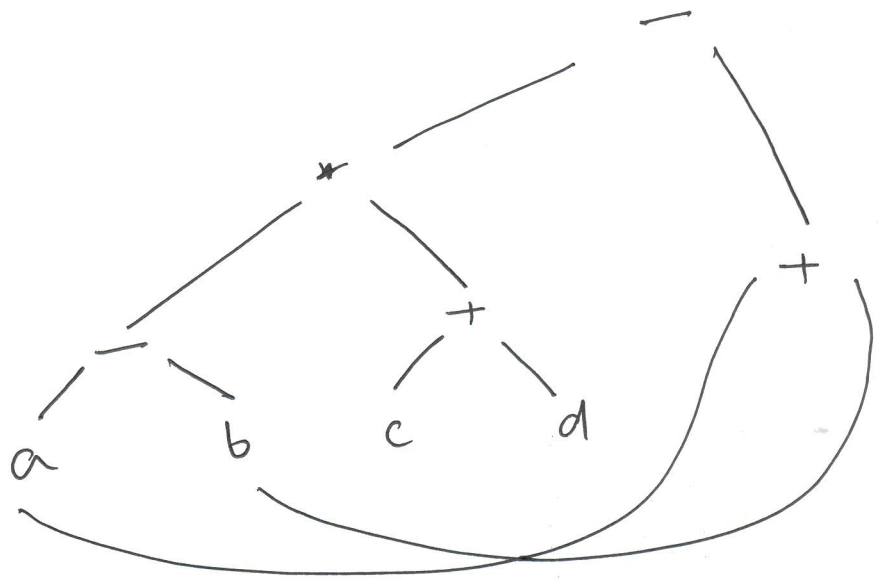
$t_2 = c + d$

$t_3 = a + b$

$t_4 = t_1 * t_2$

$t_5 = t_4 - t_3$

DAG



Postfix or Polish notation

$ab - cd + * ab + -$

(b) [04]

```
switch (expression)
{
 case v1 : s1 ; break;
 case v2 : s2; break;
  :
 case vn-1 : sn-1 ; break;
}default : sn ;
```

3- add code for switch statement

```
    code to evaluate E into t
    goto test
L1 : code for S1
     goto next
L2 : code for S2
     goto next :
     ...
Ln-1 : code for Sn-1
       goto next
Ln : code for Sn
     goto next
test : if t = V1 goto L1
       if t = V2 goto L2
       ...
       if t = Vn-1 goto Ln-1
       goto Ln
next :
```

6. Intermediate code for [10]

```
for i=1 to 10 do
    for j=1 to 10 do
        A[i,j] = 0.0
for i=1 to 10 do
    A[i,i] = 1.0
```

1) $i = 1$

2) $j = 1$

3) $t_1 = 10 * i$

4) $t_2 = t_1 + j$

5) $t_3 = 8 * t_2$

6) $t_4 = t_3 - 88$

7) $a[t_4] = 0.0$

8) $j = j + 1$

9) if $j <= 10$ goto (3)

10) $i = i + 1$

11) if $i < 10$ goto (2)

12) $i = 1$

13) $t_5 = i - 1$

14) $t_6 = 88 * t_5$

15) $a[t_6] = 1.0$

16) $i = i + 1$

17) if $i <= 10$ goto (13)

7. (a) 3-add code for $a + (b - c * d)/e$ [2 x 3]

$t_1 = c * d$

$t_2 = b - t_1$

$t_3 = t_2 / e$

$t_4 = a + t_3$

Quadruples

Index	Operator	Operand 1	Operand 2	Result
0	*	c	d	t_1
1	−	b	t_1	t_2
2	/	t_2	e	t_3
3	+	a	t_3	t_4

Triples

Index	Operator	Operand 1	Operand 2
0	*	c	d
1	−	b	[0]
2	/	[1]	e
3	+	a	[2]

Indirect Triples

Triple table + Index table

	Operator	Opnd 1	Opnd 2
0	*	c	d
1	−	b	[0]
2	/	[1]	e
3	+	a	[2]

Index table

Index	Pointer
100	[0]
101	[1]
102	[2]
103	[3]

7.(b) 3-add code for the expression

if (a > b || b < 10 && c > 2) x = 1

[4M]

3-add code

```
    if b < 10 goto L1
       goto L3

L1: if c > 2 goto L2
       goto L3

L2: x = 1
       goto Last
L3: if a > b goto L2

Last:
```