SYSTEM SOFTWARE AND COMPILERS
18CS61

MODULE-5

Dr.Sanchari Saha
Assistant Professor
Dept. of CSE,
CMRIT, Bangalore

Text Books

1. System Software by Leland. L. Beck, D Manjula, 3rd edition, 2012
Module 1

2. Alfred V Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman , Compilers-Principles, Techniques and Tools, Pearson, 2nd edition, 2007 .
Module 2, 3, 5

3. Doug Brown, John Levine, Tony Mason, lex & yacc, O'Reilly Media, October 2012.
Module 4

Course Learning Objectives:

This course (18CS61) will enable students to:

- Define System Software.
- Familiarize with source file, object file and executable file structures and libraries
- Describe the front-end and back-end phases of compiler and their importance to students

Module 5 - syllabus

- Syntax Directed Translation
- Intermediate code generation
- Code generation

Text book 2: Chapter 5.1, 5.2, 5.3, 6.1, 6.2, 8.1, 8.2
RBT: L1, L2, L3

VTU QP

Aug -2022

Module-5

9 a. Define S – Attribute and I – Attribute with respect to SDD and construct Syntax tree, Parse tree and annotated tree for string 5 *6 + 7 by using given grammar.

$S \rightarrow En$
$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F$
$T \rightarrow T \mid F$
$T \rightarrow F$
$F \rightarrow (E) \mid digit \mid$
$n \rightarrow ;$

(10 Marks)

b. What are the different three address code instructions? Translate the arithmetic expression a + b – (-c) into quadruples , triplets and indirect triples. **(10 Marks)**

OR

10 a. Define SDD. Give SDD for simple type declaration. Construct a dependency graph for the declaration int a, b ; **(10 Marks)**

b. Explain the issues in design of code generation. **(10 Marks)**

VTU QP

Jan -2023

Module-5

9	a.	What is syntax directed definition? Write the grammer and SDD for a simple desk calculation and show annotated Parse tree for the expression (3 + 4) * (5 + 6).	(08 Marks)
	b.	What is an attribute? Explain the different types of attributes with example.	(08 Marks)
	c.	What is the difference between syntax tree and parse tree?	(04 Marks)

OR

10	a.	Explain the Intermediate Code Generation (ICG) and type of method used to convert ICG.	(10 Marks)
	b.	Explain the issues in the design of code generation.	(10 Marks)

Question Bank-Module- 5

Refer previous year VTU Question papers (Slide 5 & 6)

Sample Questions:

- What is semantic analysis? What is syntax directed definition (SDD)?
- What are the different types or classifications of attributes, Explain with example. (synthesized attribute & inherited attribute)
- What is a semantic rule? Explain with example
- What is annotated parse tree? Explain with example
- Write the SDD for the following grammar: S → En where *n* represent end of file marker

T → T * F | F

E → E + T | T

F → (E) | **digit**

- Write the grammar and syntax directed definition for a simple desk calculator and show annotated parse tree for the expression (3+4)*(5+6)
- Obtain SDD and annotated parse tree for the following grammar using top-down approach:

T → T * F | F

F → digit

- Obtain SDD for the following grammar using top-down approach:

S → En
E → E + T | T
T → T * F | F
F → (E) | digit
and obtain annotated parse tree for the expression (3 + 4) * (5 + 6)n
*** *Practice all the given problems in note related to annotated parse tree*

- What is a dependency graph. Explain with an Example.
- What is S-attributed definition & L-attributed definition
- Write the SDD for a simple type declaration and write the annotated parse tree and the dependency graph for the declaration "**float a, b, c**" and "**int a,b**"
- Write the SDD for a simple desk calculator. Write the annotated parse tree for the expression 3*5+4n
- What is a syntax tree? What is the difference between syntax tree and parse tree?"
- For the following grammar show the parse tree, syntax tree and annotated parse tree considering the expression

3 *5 + 4
E → E + T | E – T | T
T→ T*F | T/F | F
F→ (E) | digit | id

- Create a syntax tree for the expression “a – 4 + c”
- Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions with operators -, +, * and / using top-down approach
- Give the syntax directed translation of type **int [2][3]** and also given the semantic rules for the respective productions
- What is syntax directed translation scheme
- What are the different 3-address code instructions? Construction a dependency graph for the declaration **int a,b**
- Explain the issues in design of a code generator
- Translate the arithmetic expression a+b- (-c) into quadruples, triples, indirect triples, syntax tree

(Practice for all the expressions as given in note)

- Generate code for the following three-address statements assuming all variables are stored in memory locations.

a) x = 1
b) x = a
c) x = a + 1
d) x = a + b
e) The two statements
x = b * c
y = a + x

- Generate code for the following three-address statements assuming a and b are arrays whose elements are 4-byte values.

The four-statement sequence
x = a[i]
y = b[j]
a[i] = y
b[j] = x

❑ Generate code/ Machine code for the following sequence assuming that n is in a memory location:

```
s = 0
i = 0
L1: if i > n goto L2
s = s + i
i = i + 1
goto L1
L2:
```

❑ Determine the costs of the following instruction sequences:

a)
```
LD R0, y
LD R1, z
ADD R0, R0, R1
ST x, R0
```

b)
```
LD R0, i
MUL R0, R0, 8
LD R1, a(R0)
ST b, R1
```

c)
```
LD R0, c
LD R1, i
MUL R1, R1, 8
ST a(R1), R0
```

d)
```
LD R0, p
LD R1, 0(R0)
ST x, R1
```

Module : (5)

(A)

(*) Syntax directed definition (SDD) :-

SDD :- A SDD is a context-free grammar (CFG) with semantic rules.

→ Attributes are associated with grammar symbols and semantic rules associated with productions.

→ If 'x' is a symbol and 'a' is one of its attribute, then x.a denotes value at node 'x'.

→ Attributes may be numbers, strings, references, datatypes etc.

Productions	Semantic rule
$E \rightarrow E+T$	$E.val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$

(B)

(*) Types of Attributes in SDD :-

(i) **Synthesized attribute** :- If a node takes value from its children then it is synthesized attribute.

Ex: $A \rightarrow BCD$, A be a parent node, B, C, D are children nodes.

S = synthesized attribute notation

$A.S = B.S$
$A.S = C.S$
$A.S = D.S$

} Parent node A taking value from its children B, C, D.

(2) **Inherited Attribute**:- If a node takes value from its parent- or siblings then it is inherited attribute.

Ex: $A \to BCD$

i = inherited attribute notation
- $C.i = A.i$ ---> Parent node
- $C.i = B.i$ ---> Sibling "
- $C.i = D.i$ ---> Sibling "

(c)

(*) Types of SDD:-

(1) S- Attributed SDD / S-Attributed Grammar
(2) L- Attributed SDD / L Attributed Grammar

S Attributed SDD

→ A SDD that- uses only synthesized attributes is called as S-attributed SDD.

Ex: $A \to BCD$

i.e
- $A.S \Rightarrow B.S$
- $A.S \Rightarrow C.S$
- $A.S \Rightarrow D.S$

→ Semantic actions are always placed at right-end of Production ∴ also called as post-fix SDD

→ Attributes are evaluated with bottom-up parsing

L- attributed SDD

→ A SDD that- uses both synthesized & inherited attributes is called L- attributed SDD. But it is restricted for inherited attribute, considering that- each inherited attribute is restricted to inherit- from Parent- or left- side sibling only.

Ex: $A \to XYZ$ i.e
- $Y.S = A.S$ ✓
- $Y.\$ = X.i$ ✓
- $Y.i = Z.i$ ✗

→ Semantic actions are placed anywhere of RHS i.e anywhere in production.

→ Attributes are evaluated al- dept-first → L to R ⇒ top down L to R.

(*) SDD for Simple Desk Calculator
(or)
SDD for Evaluation of Expression
(or)
Annotated Parse Tree

(1) Construction of Annotated Parse Tree

Example (1): Construct Annotated Parse Tree for 3*5+4

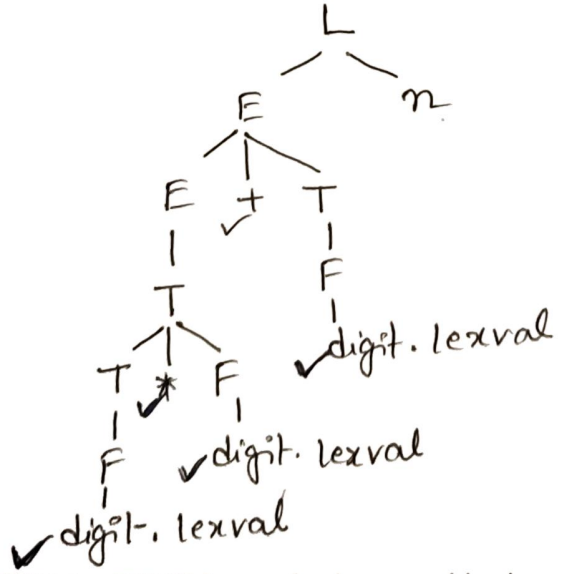
Productions	Semantic Rules
$L \rightarrow En$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow digit$	$F.val = E.val$
$F \rightarrow (E)$	$F.val = digit.lexval$
$n \rightarrow ;$	

Here n is an endmarker for expression E.

Here E and E_1 ; T and T_1 are same. Just to separate as parent and child node written as E, E_1 and T, T_1.

SDD of a simple desk calculator

→ 1st step: Construct - Parse tree for the given expression.

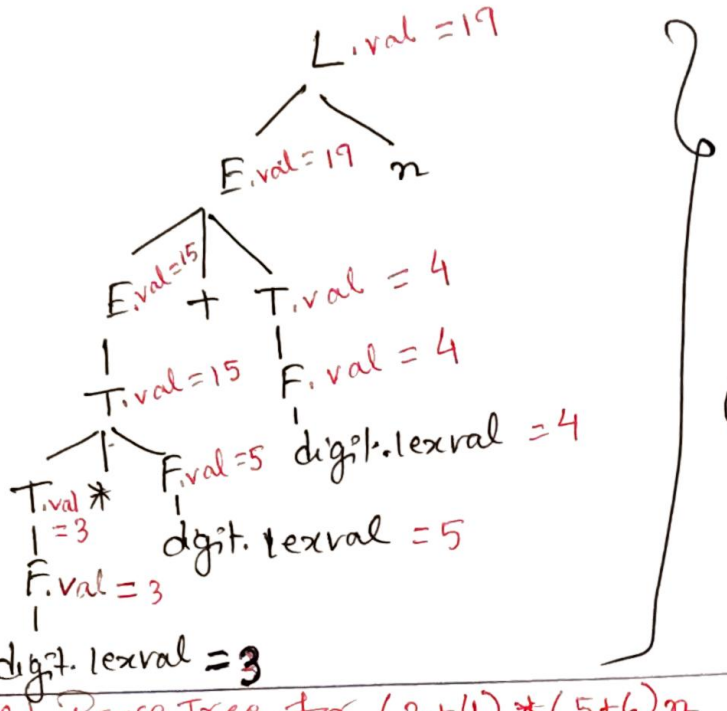


(Top-down, L to R Parsing)

→ **2nd step** : Convert the parse tree to annotated parse tree.

PT ⟶ APT

Parse tree containing values of attributes at each node is called annotated Parse Tree.



This is annotated Parse Tree.

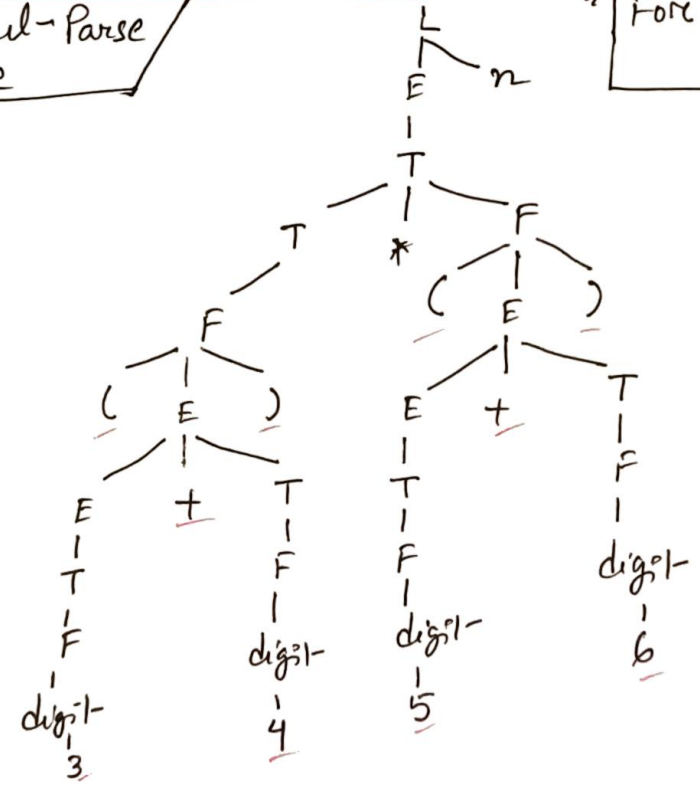
(Top-down, & L to R Parsing)

Annotated Parse Tree for (3+4)*(5+6)n

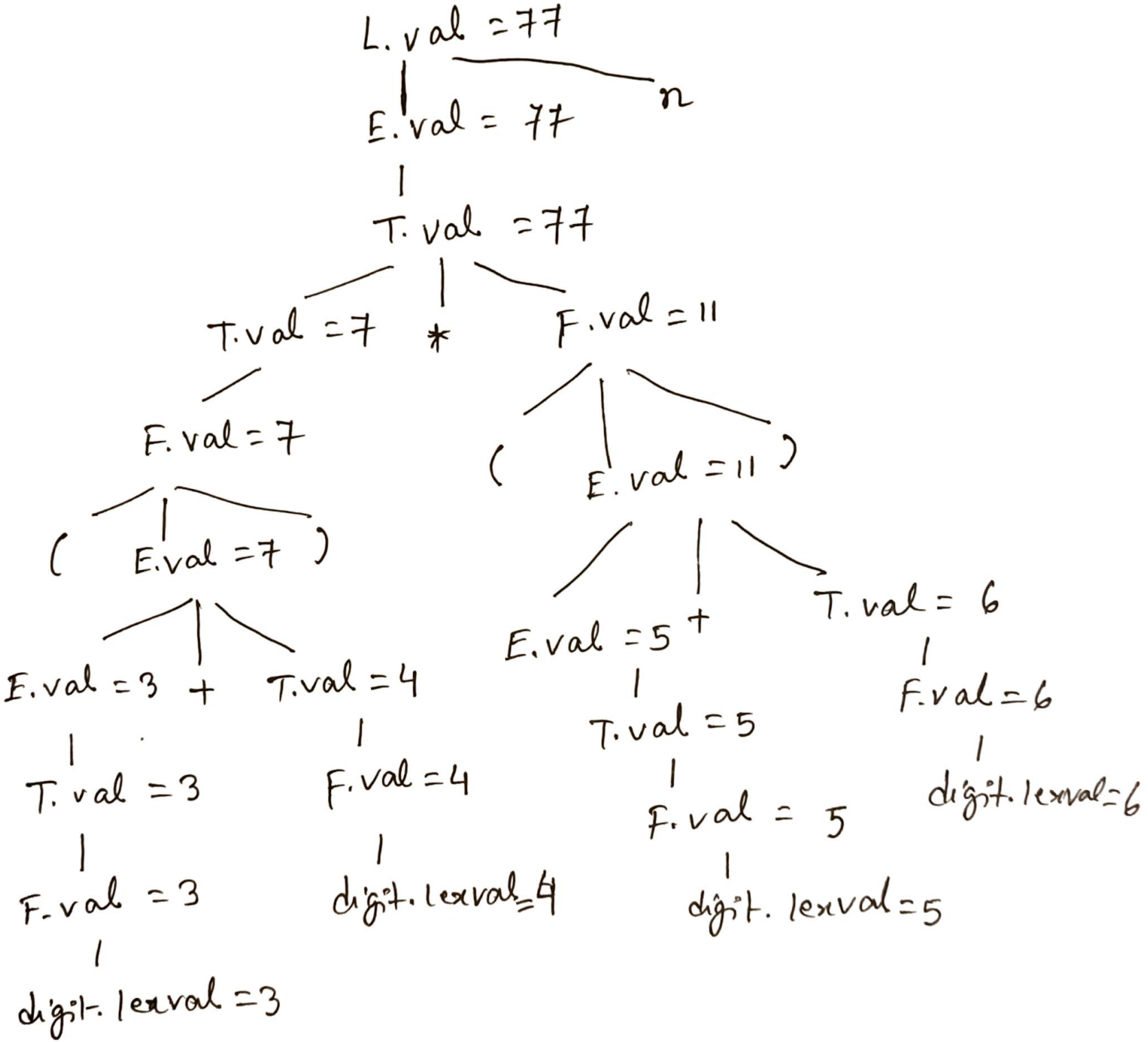
Example (2)

For same Simple Desk calculator SDD

Step (1) - Construct Parse Tree



Step2: Convert Parse Tree to Annotated Parse Tree

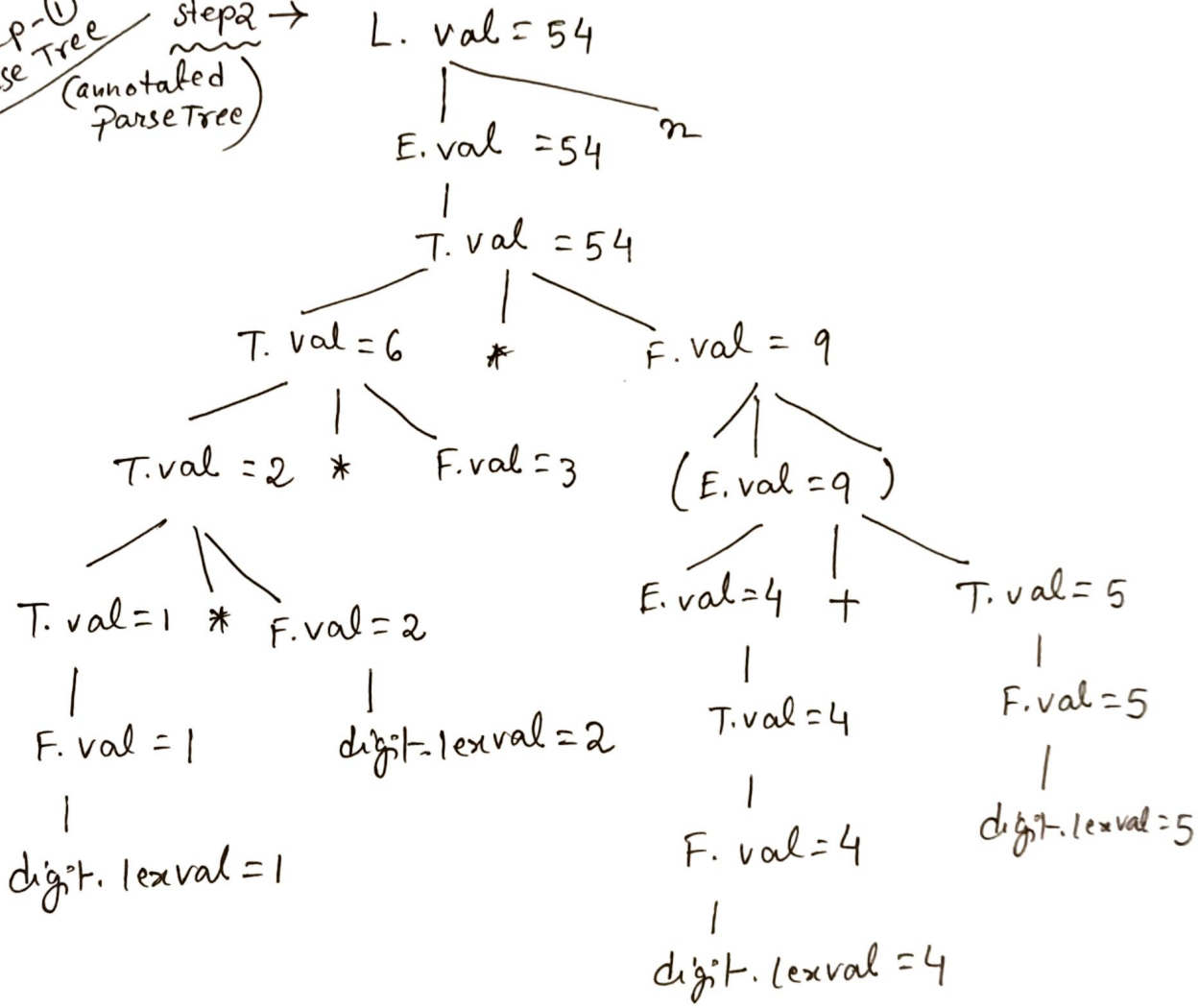


Example-3

For the same Simple Desk Calculator SDD, construct annotated Parse Tree for the expression

1 * 2 * 3 * (4+5)n

Do Step-① Parse Tree / Step2 → (annotated Parse Tree)

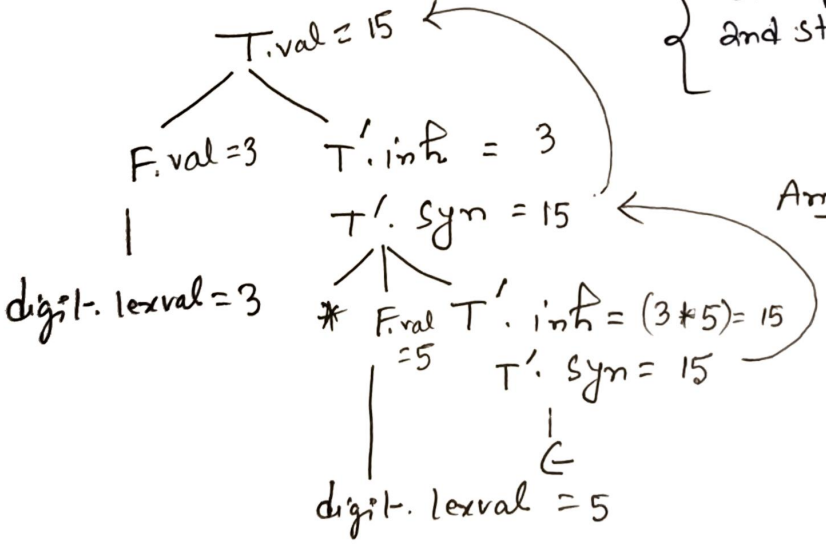


(*) Annotated Parse Tree :- A parse tree showing the values of its attributes is called an Annotated Parse Tree

Example 4: SDD for a Grammar (Grammar + Semantic Rules)

Productions	Semantic Rules
1) $T \rightarrow FT'$	$T'.inh = F.val$; $T.val = T'.syn$
2) $T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh * F.val$; $T'.syn = T_1'.syn$
3) $T' \rightarrow \in$	$T'.syn = T'.inh$
4) $F \rightarrow digit$	$F.val = digit.lexval$

1st Step: Construct Parse Tree
2nd step: Convert Parse Tree to Annotated Parse Tree.



Annotated parse tree for 3 * 5

Perform Top-down L to R parsing.

⇓

Annotated Parse Tree 3 * 5

(*) what is SDT? Explain with example

→ SDD specifies the value of attributes by associating semantic rules with productions, while SDT puts program fragments within the production bodies themselves.

→ SDD is easier to read and specify. But SDT is more efficient and is also easy to implement.

→ SDT is used for executing arithmatic expressions. It can also be used to convert infix to postfix / prefix expressions. It is also used to convert binary numbers to decimal numbers.

SDT Scheme

$E \to E+T$	{print '+'}
$E \to E-T$	{print '-'}
$E \to T$	
$T \to 0$	{Print '0'}
$T \to 1$	{print '1'}
⋮	
$T \to 9$	{print '9'}

SDD

$E \to E+T$	E.code = E.code ‖ T.code ‖ '+'
$E \to E-T$	E.code = E.code ‖ T.code ‖ '-'
$E \to T$	E.code = T.code
$T \to 0$	T.code = '0'
$T \to 1$	T.code = '1'
⋮	
$T \to 9$	T.code = '9'

Topic

Dependency Graph :- (Evaluation Ordering for SDD's)

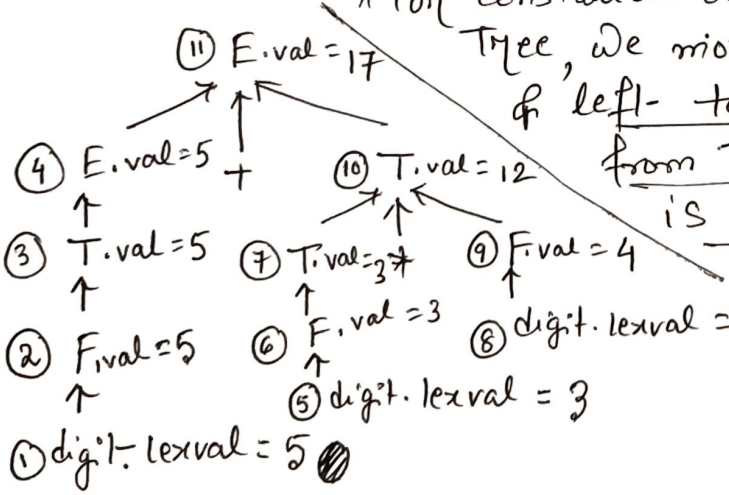
→ Dependency Graph represent the flow of information among the attributes in a parse Tree.

→ Dependency Graphs are useful for determining evaluation order for attributes in a parse Tree.

→ While any annotated parse tree shows the values of attributes, a dependency graph determines how these values can be computed.

Expression Grammar	Semantic Rule
1) $E \rightarrow E+T$	$E.val = E_1.val + T.val$
2) $E \rightarrow T$	$E.val = T.val$
3) $T \rightarrow T*F$	$T.val = T_1.val * F.val$
4) $T \rightarrow F$	$T.val = F.val$
5) $F \rightarrow digit$	$F.val = digit.lexval$

(*) Construct Dependency Graph for 5+3*4 :-



* For construction of annotated Parse Tree, we move from Top to bottom & left to right. While moving from Top to bottom, if there is any reduction, we execute the corresponding semantic rule of that production..

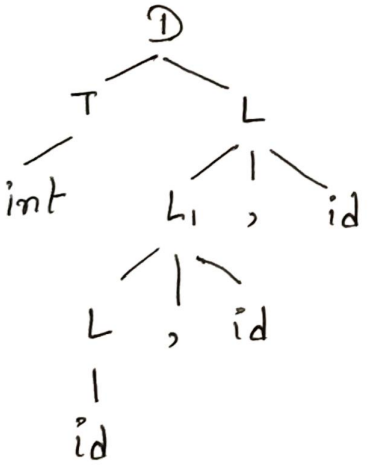
Question: Define SDD. Give SDD for simple type declaration. Construct a dependency Graph for the declaration int a, b, c.

(*) SDD for Simple type declaration

Grammar	Semantic Rule
1) $D \rightarrow TL$	$L.inh = T.type$
2) $T \rightarrow int$	$T.type = int$
3) $T \rightarrow float$	$T.type = float$
4) $L \rightarrow L_1, id$	$L_1.inh = L.inh$ addType (id.entry, L.inh)
5) $L \rightarrow id$	addType (id.entry, L.inh)

(*) Construct Annotated Parse Tree for int a, b, c and show dependency Graph.

Step (1) - Construct Parse Tree :-



→ as our input string contains int, we have chosen 2nd production as extension of T (not the 3rd one as it will extend to float)

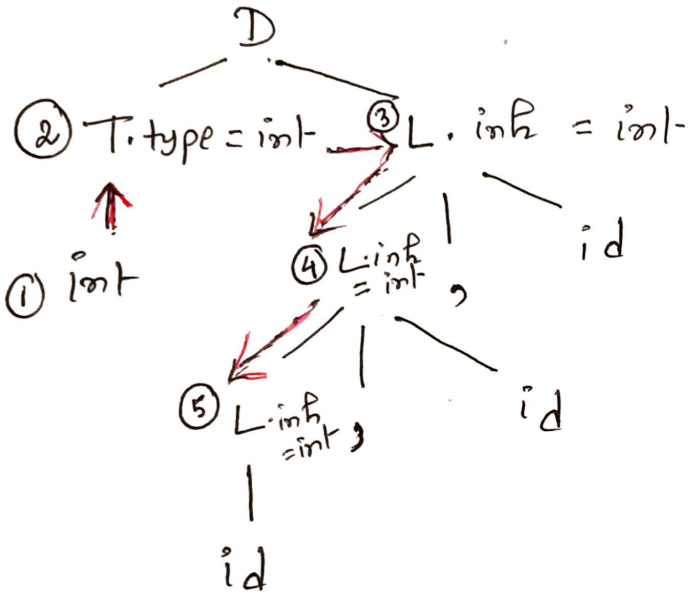
→ To extend L, we have chosen 4th production, as in the i/p string we have 3 identifiers 'a, b, c'. (not 5th prod. as it will give only one id)

→ Same way we extended L_1 using 4th production.

Step (2):- Convert Parse Tree to annotated Parse tree & dependency Graph.

(Start at top, move towards down, - left to right)

to see the Medution



arrows and sequence numbers are showing order of computation in dependency Graph.

Topic: Applications of Syntax Directed Translation

Applications of Syntax-Directed Translation → Type checking
→ Intermediate code Generation

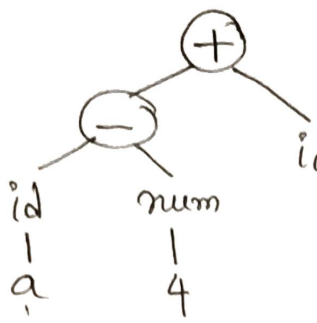
(1) * Construction of Syntax Trees:

Syntax Tree represents a constaul-, the children represents the meaningful compenents of the construct. Syntax Tree is one form of intermediate code representation.

Productions	Semantic Rules
1) $E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
2) $E \rightarrow E_1 - T$	E.node = new Node ('-', E_1.node, T.node)
3) $E \rightarrow T$	E.node = T.node
4) $T \rightarrow (E)$	T.node = E.node
5) $T \rightarrow id$	T.node = new leaf (id, id.entry)
6) $T \rightarrow num$	T.node = new leaf (num, num.val)

Steps for the Construction of Syntax Tree for a-4+c

Syntax Tree



P_1 = new leaf (id, entry-a)
P_2 = new leaf (num, 4)
P_3 = new Node ('-', P_1, P_2)
P_4 = new leaf (id, entry-c)
P_5 = new Node ('+', P_3, P_4)

(2)

* The Structure of a Type :-

Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input; attributes can then be used to carry information from one part of the parse tree to another.

In C, The type int [2][3] can be read as, "array of 2 arrays of 3 integers". The corresponding type expression array (2, array (3, integer)) is represented by the tree as shown in following figure:-

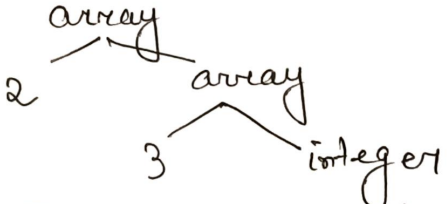


Fig: Type expression for int [2][3]

Productions	Semantic Rules
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ int	$B.t$ = integer
$B \rightarrow$ float	$B.t$ = float

$C \rightarrow [\ num\ ]\ C_1$ $\quad C.t = array(num.val, C_1.t)$

$C_1.b = C.b$

$C \rightarrow \epsilon$ $\quad C.t = C.b$

Here T generates either a basic type or an array type. The nonterminals B and T have a synthesized attribute 't' representing a type. The nonterminal C has 2 attributes → inherited attribute 'b' and synthesized attribute 't'. The inherited b attributes pass a basic type down the tree, and the synthesized t attributes accumulates the result.

Annotated Parse Tree for int [2][3]:-

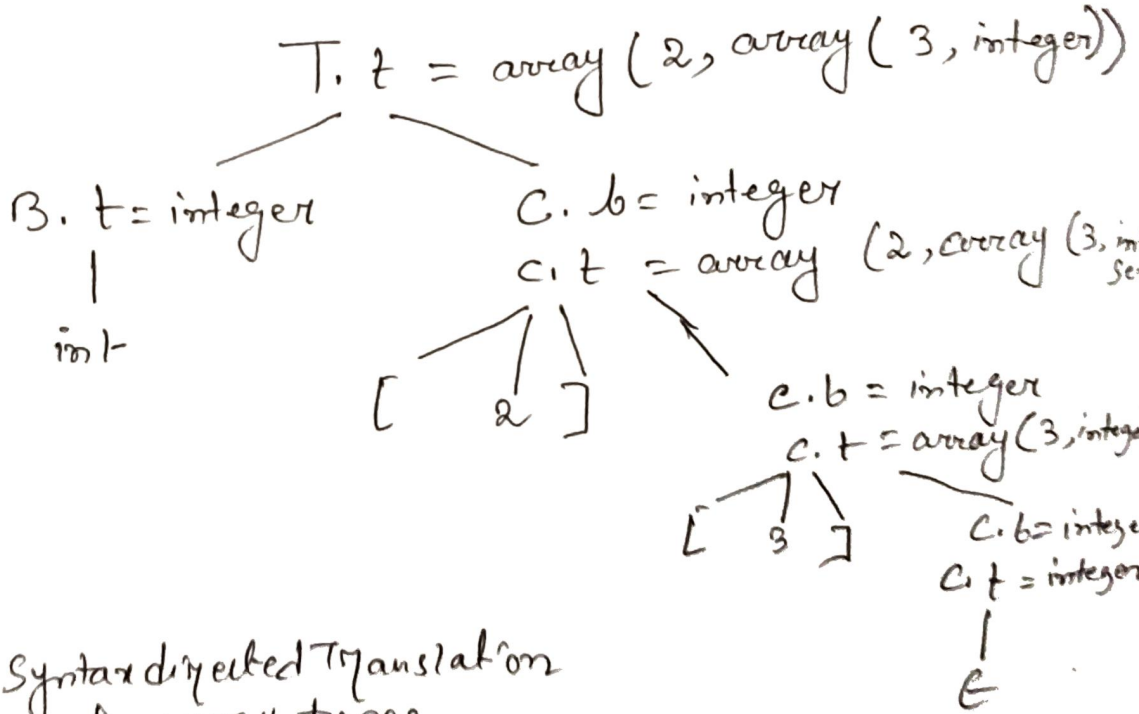
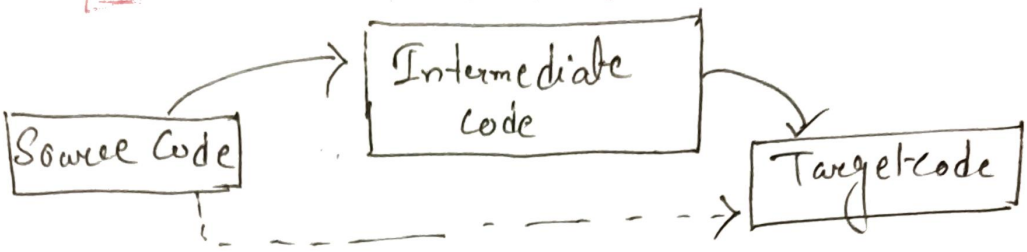


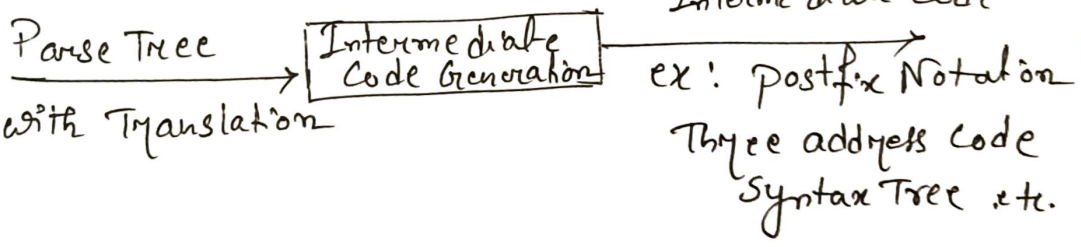
fig:- Syntax directed Translation of array types (Annotated parse Tree)

Topic

Intermediate Code Generation



During the Translation of a source code/program into the object code for a target-machine, a compiler may generate a middle-level language code, which is known as intermediate code or intermediate text. The intermediate code can be represented in the form of post-fix notation, syntax tree, directed acyclic graph (DAG), three address code.



(1) Postfix Notation :-

Also known as suffix notation. The ordinary way of writing the sum of a & b is with an operator in the middle (infix) : 'a+b'. The postfix Notation for the same expression places the operator at the right-end as 'ab+'

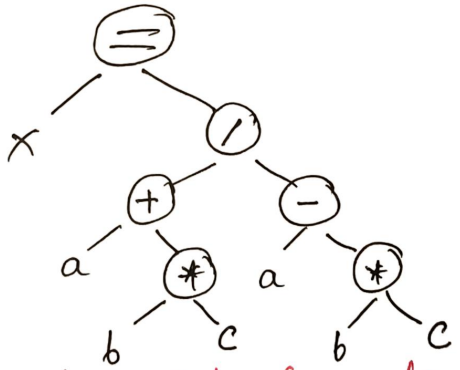
ex 1: The postfix representation of the expression $(a+b)*c$ is : ab+c*

ex 2: The postfix representation of the expression $(a-b)*(c+d)+(a-b)$ is : ab-cd+*ab-+

(11) Syntax Tree :-

A syntax tree is more like a condensed form of a parse Tree. The operator and keyword nodes of the parse Tree are moved to their parents and a chain of single productions is replaced by the single link in the syntax tree. The internal nodes are operator and child nodes are operands.

ex: $x = (a+b*c) / (a-b*c)$



Advantages of Intermediate Code Generation:

1. Easier to implement
2. Facilitates code optimization
3. Platform independence
4. Code reuse
5. Easier debugging.

(*) Three Address Code

Forms of Intermediate code

(3 ways to represent IC (intermediate code) :-

→ (i) using syntax Trees (or) abstract- syntax Trees

→ (ii) Post-fix Notation

→ (iii) Three address code representation.

(*) (iii) 3- address Code

Three address Code is represented in 3 ways:-

(1) Quadruple (2) Triple (3) Indirect-Triple

Let the expression is of the form $x + y * z$

→ This is not a 3-address code instruction as in RHS we have 2 operators. We can have max 1 operator in the RHS and max 3 operands in the expression for a 3-address code (In this expression there is no issue with the operands as we have 3 operands only).

→ Now we have to translate this expression into a 3-address code representation.

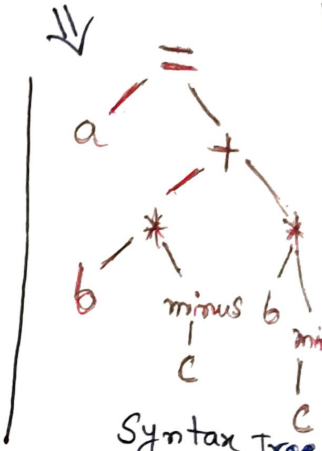
→ Here we have + and * operators. and * is having higher priority.

3-address code:

$t_1 = y * z$, Here t_1 is a temporary storage

$t_2 = x + t_1$, & t_2 " "

(*) Let the instruction is $a = b * -c + b * -c$



Syntax Tree

(1) Quadruple :-

It contains 4 fields → Operator, argument 1, argument 2, Result.

Let us convert- $a = b * -c + b * -c$ to quadruple.

→ here unary minus has higher priority.

→ then * " "

This is 3-address code:

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

Also can be written as

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = b * t_1$

$t_4 = t_2 + t_3$

$a = t_4$

This is 3-address code.

Ex.

i) $t_1 = -c$
ii) $t_2 = b * t_1$
iii) $t_3 = -c$
iv) $t_4 = b * t_3$
v) $t_5 = t_2 + t_4$
vi) $a = t_5$

Now converting to Quadruple :-

Add	op	arg1	arg2	Result
(0)	−	c		t_1
(1)	*	b	t_1	t_2
(2)	−	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

3-address code is represented in the form of a Quadruple.

Disadvantage!-

Too many temporary variables are used & need to store all of them in a symbol table.

To overcome this disadvantage Triples are used.

(2) Triple :-

It contains 3 fields → operator, argument 1, argument 2

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

=> In Triple representation there is no need of temporary variables

(3) Indirect-Triple :-

Here also we use triple only but we need to have an extra table. The table contains pointer to the triple. This table is also called instruction array to list the pointers to the triples in desired order.

add	OP	arg1	arg2
(0)	−	c	
(1)	*	b	(0)
(2)	−	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

pointers

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)
105	(5)

Instruction array.

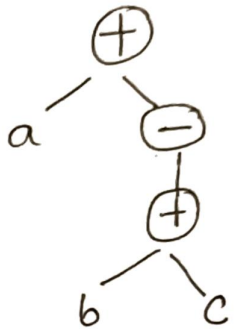
Indirect-Triples consists of a list of pointers to triples, rather than a listing of triples themselves. It allows an optimizing compiler to easily re-position the sub-expression for producing an optimized code. An optimizing compiler can move an instruction by re-ordering the instruction list, without affecting the triples themselves.

Example : (2)

Translate the arithmetic expression $a + -(b+c)$ into : (i) Syntax Tree (ii) Quadruples (iii) Triples (iv) Indirect- Triples

---

(i) Syntax Tree



3 address code

based on priority of operators

$t_1 = b + c$

$t_2 = -t_1$

$t_3 = a + t_2$

(ii) Quadruples

add	op	arg 1	arg 2	Result
(0)	+	b	c	t_1
(1)	unary minus	t_1		t_2
(2)	+	a	t_2	t_3

(iii) Triples

add	op	arg 1	arg 2
(0)	+	b	c
(1)	unary min.	(0)	
(2)	+	a	(1)

(iv) Indirect- Triples

10	(0)
11	(1)
12	(2)

Example : 3

expression : $s = -z / a * (x+y)$ where $-z$ stands for unary minus.

3-address code

$t_1 = x + y$

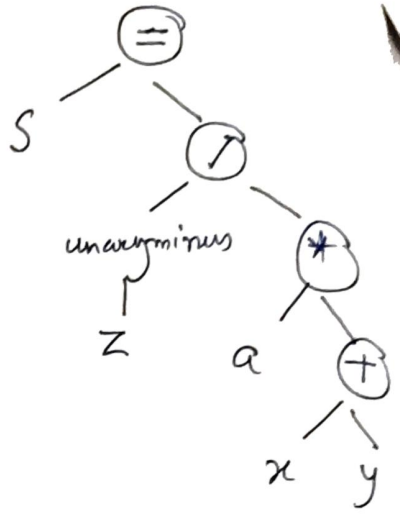
$t_2 = -z$

$t_3 = a * t_1$

$t_4 = t_2 / t_3$

$s = t_4$

Syntax Tree



Quadruple

add	op	operand 1	operand 2	result
(0)	+	x	y	t_1
(1)	unary minus	z		t_2
(2)	*	a	t_1	t_3
(3)	/	t_2	t_3	t_4
(4)	=	t_4		s

Triple

add	operator	operand 1	operand 2
(0)	+	x	y
(1)	unary minus	z	
(2)	*	a	(0)
(3)	/	(1)	(2)
(4)	=	s	(3)

Indirect - Triple

Pointer	address of Triple
10	(0)
11	(1)
12	(2)
13	(3)
14	(4)
:	

draw same Triple Here

Example : (4)

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

$t_1 = a+b$

$t_2 = -t_1$

$t_3 = c+d$

$t_4 = t_2 * t_3$

$t_5 = t_1 + c$

$t_6 = t_4 + t_5$

$X = t_6$

Variants of Syntax Tree :-

① Directed Acyclic Graphs for Expression :-

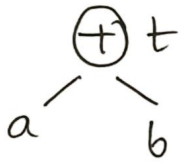
A directed Acyclic Graph (DAG) for an expression identifies the common subexpression (subexpr. that appear more than once) of the expression.

→ DAG represents the structure of a basic block (sequence of instructions / expressions in a particular order)

* In a DAG, internal node represents operators.
* Leaf node represents identifiers, constants.
* Internal node also represents result of expression.

ex: $t = a + b$

DAG :



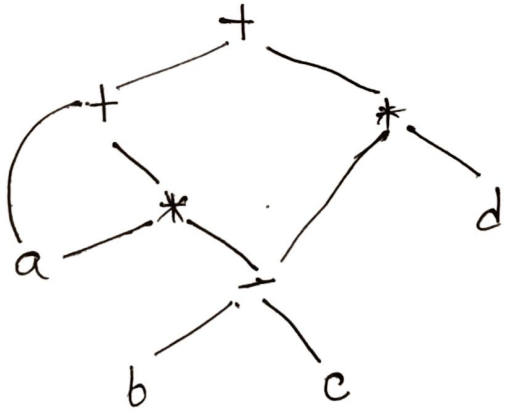
→ Application of DAG :-

① Determining the common sub-expression
② Determining which names are used inside the block & computed outside the block.
③ Determining which statements of the block could have their computed value outside the block.
④ Simplify the list of quadruples by eliminating common sub-expression.

SDD to produce syntax Trees or DAG's for arithmatic expression Grammar / Simple Desk calculator Grammar

Grammar Production	Semantic Rules
1) $E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
2) $E \rightarrow E_1 - T$	E.node = new Node ('-', E_1.node, T.node)
3) $E \rightarrow T$	E.node = T.node
4) $T \rightarrow (E)$	T.node = E.node
5) $T \rightarrow id$	T.node = new leaf (id, id.entry)
6) $T \rightarrow num$	T.node = new leaf (num, num.val)

DAG for the expression $a + a * (b-c) + (b-c) * d$



3-address code

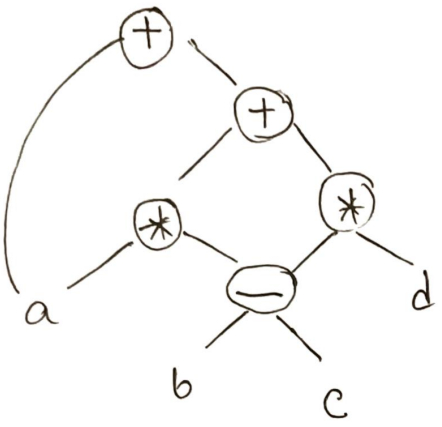
$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

Steps for Constructing the DAG of expression
$a + a * (b-c) + (b-c) * d$ [as shown in Prev. page DAG]

1) P_1 = leaf (id, entry-a)
2) P_2 = leaf (id, entry-a) = P_1
3) P_3 = leaf (id, entry-b)
4) P_4 = leaf (id, entry-c)
5) P_5 = Node ('-', P_3, P_4)
6) P_6 = Node ('*', P_1, P_5)
7) P_7 = Node ('+', P_1, P_6)
8) P_8 = leaf (id, entry-b) = P_3
9) P_9 = leaf (id, entry-c) = P_4
10) P_{10} = Node ('-', P_3, P_4) = P_5
11) P_{11} = leaf (id, entry-d)
12) P_{12} = Node ('*', P_5, P_{11})
13) P_{13} = Node ('+', P_7, P_{12})

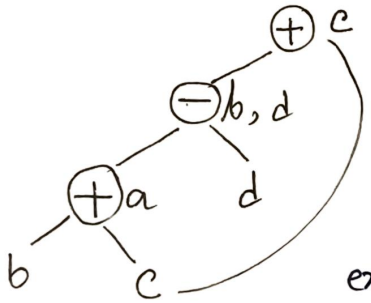
Construction of DAG for the basic block :-

ex (1) : $a + a * (b-c) + (b-c) * d$



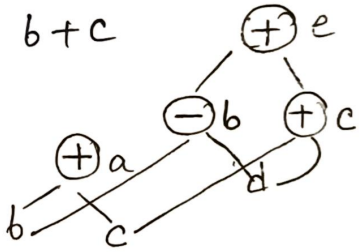
ex (2) :

1) $a = b+c$
2) $b = a-d$
3) $c = b+c$
4) $d = a-d$



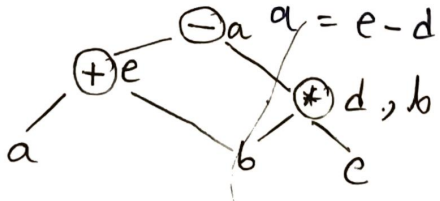
ex (3) :

$a = b+c$
$b = b-d$
$c = c+d$
$e = b+c$



ex (4) :

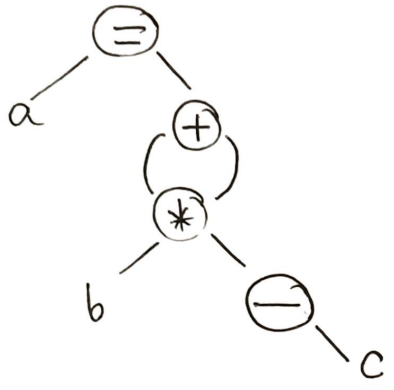
$d = b*c$
$e = a+b$
$b = b*c$
$a = e-d$



ex: (5) Construct DAG for the expression

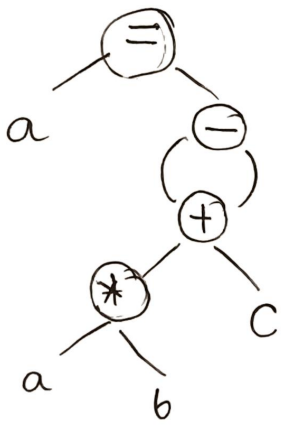
$$a = b * -c + b * -c$$

Here -c represents unary minus



ex: (6)

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



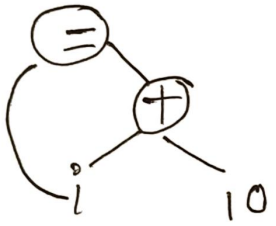
(2) The Value_Number Method for Constructing DAG

Nodes are stored in an array of records.

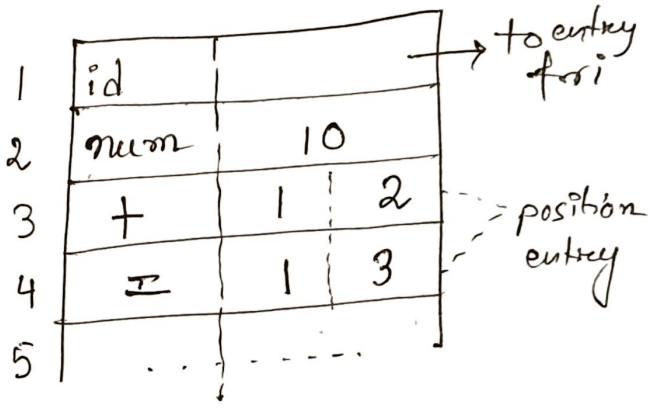
Each row of array → a record (one node)

ex :-

DAG for i = i + 10



a) DAG



(b) Array

Questions :-

(1) What are the different 3-address code instructions ? (10M)

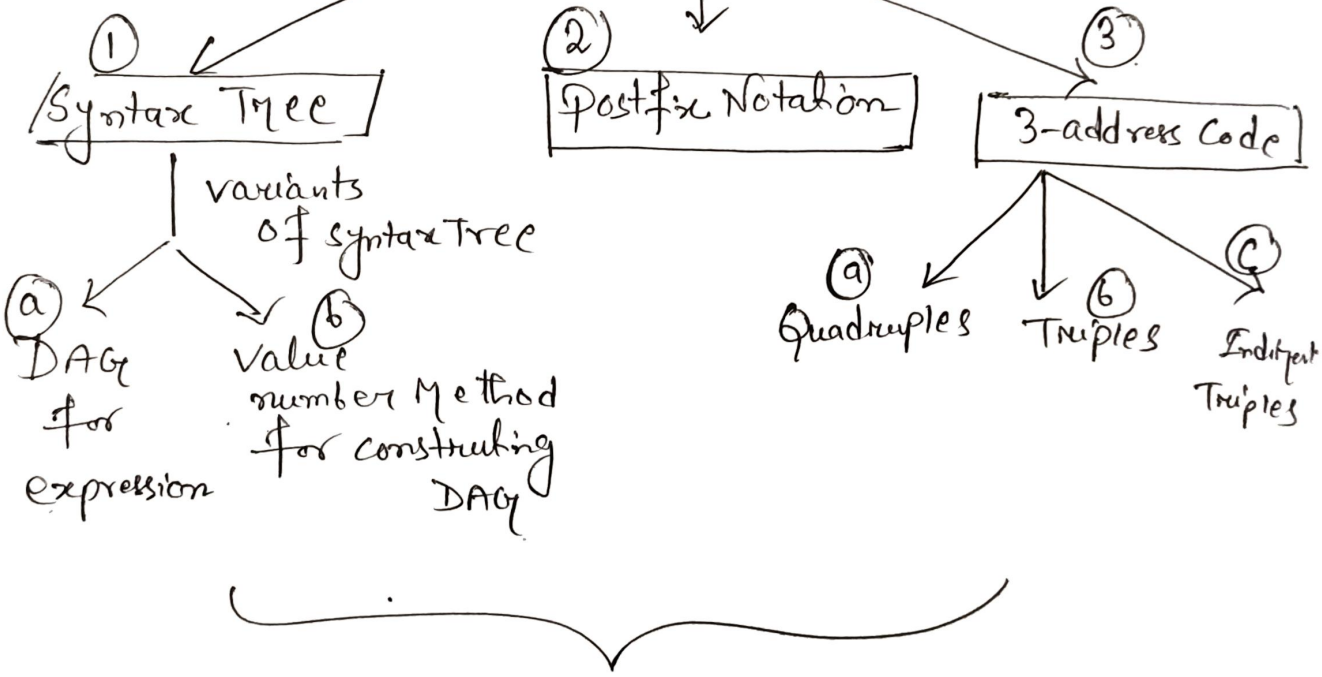
Ans :- (i) Quadruple
(ii) Triple
(iii) Indirect-Triple

} Take an expression and explain them with example.

(2) Explain the intermediate code generation (ICG) and type of method used to convert- ICG (10M)

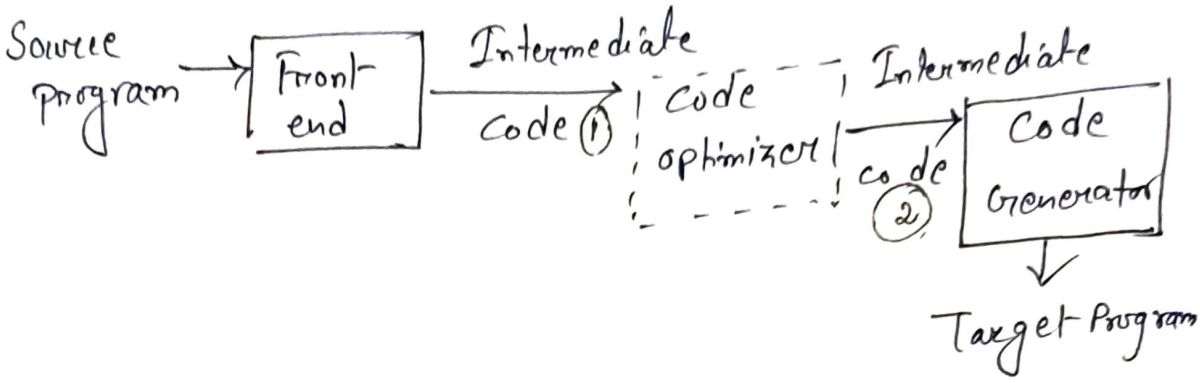
Ans :- (i) Explain ICG with a block diagram.

(ii) Type of Method used to convert-ICG



Explain them in detail with possible example

Topic :- // Code Generator // :-



Compilers that need to produce efficient target programs, include an optimization phase prior to the code generation. The optimizer maps the intermediate code (1) to the intermediate representation (2) (IR) (code) from which more efficient code can be generated. In general the code optimization and code generation phases of a compiler, often refered to as the back-end, may take multiple passes over the IR before generating the target code.

→ A code generator has 3 primary tasks
- → Instruction selection
- → Register allocation & assignment
- → Instruction ordering

Topic: [Code Generator]

Issues in the Design of a Code Generator

1) Input-to code generator
2) Target- Program
3) Memory Management-
4) Instruction selection
5) Register allocation
6) Evaluation order.

The most important criterion for a code generator is that it produce correct code. These are the 6 issues or criteria that a code generator should consider while producing correct code.

1) Input-to code Generator :-

Code Generator accepts i/p from code optimizer or intermediate code generator. Therefore code generator accepts i/ps as intermediate code which can be either a postfix notation or Three address code or as an optimized form of syntax Tree i.e Directed Acyclic Graph (DAG).

2) Target-Program :-

Code Generator accepts intermediate code as input- and produces Target- program as output

Mainly it produces 3 types of target-programs:

(i) Absolute Machine Language
(ii) Relocatable Machine Language / code
(iii) Assembly Language

(i) Absolute Machine Language :- The program will be stored at a fixed location of memory. It is mainly suitable when the program is very small, so that it is compiled & executed very fast.

(ii) Relocatable Machine code (Object code) :- Linker and Loader is used for this kind of codes.

→ Linker links object-codes of several programs to a single program.

→ Loader loads the executable code (file) into main memory.

→ whenever there is a free space in the RAM, it uses the space to store the program.

→ It is suitable for both small & large programs.

(iii) Assembly language Program/code

For code generation assemby language is most preferred, as it produces the code targetcode/in easy manner.

(3) Memory Management :-

Lexical analyzer, syntax analyzer, Semantic analyzer or Code generator, all use Symbol Table for memory management.

→ Program is divided into multiple tokens & those token information is stored in Symbol Table.

(4) Instruction selection :-

Instruction Selection & speed of instruction execution is very imporetant - for a code generator for producing effrient Targel-code.

For example ①: $x \pm y + z$ → It's a 3 address code instruction. let us translate this instruction into assembly language.

Assembly code
```
MOV  y, R0
ADD  z, R0
MOV  R0, x
```

Another example ②: $a = a + 1$

Assembly code.
```
MOV  a, R0
ADD  #1, R0
MOV  R0, a
```

Here instead of writing these 3 lines of assembly code, we can write a single instruction: INC a
increment a.

Increment is a register operation, so it is faster.

Another example ③: 1) $a = b + c$
2) $d = a + e$

```
1) MOV  b, R0
2) ADD  c, R0
3) MOV  R0, a   ]
4) MOV  a, R0   ]
5) ADD  e, R0
6) MOV  R0, d
```
→ We can eliminate these 2 redundant instructions.

After removing Redundant instructions

1) MOV b, R0
2) MOV c, R0
3) ADD e, R0
4) MOV R0, d

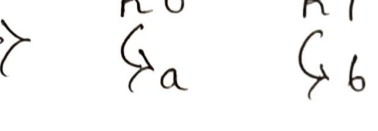
(5) Register Allocation :

Let us assume that- we have more number of operations but- limited no. of registers.

Solution to this scenario is achieved in 2 ways :- (i) Register allocation (ii) Register assignment-

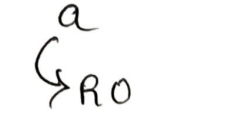
Register allocation specifies which register contains what- variable. say for ex: Register R0 and R_1 contains what- variable.

Let R0 contain a, R_1 " b $\Rightarrow$



Register assignment- is opposite to register allocation. It specifies which variable contains what- register.

Let us assume we have 2 variables a and b that- holds register R0, R1.



ex: Register allocation

```
t = a+b
t = t * c      } =>
t = t/d
```

```
MOV a, R0
ADD b, R0      t = a+b
MUL c, R0      t = t*c
DIV d, R0      t = t/d
MOV R0, t
```

(6) Evaluation Order

The order in which instructions are executed or operations are performed, is called evaluation order and it decides the efficiency of target-code.

ex: efficient order of instruction execution.

Topic:

The target-language (for code generation)

→ Familiarity with the target-m/c and its instruction set is a pre-requisite for designing a good code generator.

→ Here target-language (assembly code) for a simple computer is discussed, which is representative of many register machines.

A simple Target-Machine Model :-

Our target-computer models a "Three-address Machine" with following properties / operations :

* Load operation — LD r, x (load value x in register r)
  LD r1, r2 (load register r2 value in register r1)
  (value always gets loaded into a register)
* Store operation — ST x, r (store register r value to x)
  (value always gets stored in a variable)
* Computation operation — ADD / SUB / MUL / DIV
* Unconditional Jumps — BR L (Branch to label L)
* Conditional Jumps — BLTZ r, L (Branch to label L if value of r is less than zero)

Example ① → x = y - z

M/c Instructions →

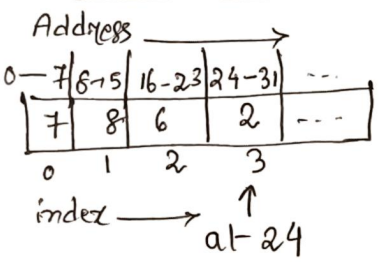
```
LD   R1, y   // (R1 = y)
LD   R2, z   // (R2 = z)
SUB  R1, R1, R2 // (R1 = R1 - R2)
ST   x, R1   // (x = R1)
```

It is not necessary to use all operations. We can try to do this in minimum number of operations.

```
LD   R1, y    // (R1 = y)
SUB  R1, R1, z // (R1 = R1 - z)
ST   x, R1    // (x = R1)
```

Example ② → Get (store) an array value to a variable

Suppose 'a' is an array whose elements are 8 byte values. The m/c instruction for b = a[i] is:



if i = 3
then 3 × 8 = 24
is in Register

```
LD   R1, i   // (R1 = i)
MUL  R1, R1, 8 // (R1 = R1 * 8) = Giving address
LD   R2, a[R1] // (R2 = a[R1/8])
ST   b, R2   // (b = R2)
```

M/c Instruction / code

Example ③ → Store a variable value into Array

a[j] = c

m/c instruction (or) Code =>

```
LD   R1, c   // (R1 = c)
LD   R2, j   // (R2 = j)
MUL  R2, R2, 8 // (R2 = R2 * 8) = Giving address
ST   a(R2), R1 // (a[R2/8] = R1)
```

value of R1, i.e 'c' is stored in 24th address location, i.e 3rd index of array a[].

Example (4) : if $x < y$ goto L

M/C instruction => OR Code

```
LD   R1, x          // (R1 = x)
LD   R2, y          // (R2 = y)
SUB  R1, R1, R2     // (R1 = R1 - R2)   x - y
BLTZ R1, L          // (if (x-y) answer is in -ve, it means that x<y is true, then goto L)
```

Program and Instruction Cost :-

→ The addressing modes involving registers, cost = 0

→ The addressing modes involving memory or constant, cost = 1

Based on addressing mode, we can calculate cost of an instruction.

Instruction Cost :-

1 + cost of Source & Destination addessing mode

Ex: MOV R0, R1 => 1 + 0 + 0 = 1 (cost)

both are registers

MOV, R0, M => 1 + 0 + 1 = 2 (cost)

ADD #1, R0 => 1 + 1 + 0 = 2 (cost)

Consider the expression : a = b + c ⟹ M/C code

```
MOV  b, R0   => Cost = 1 + 1 + 0 = 2
ADD  c, R0   => Cost = 1 + 1 + 0 = 2
MOV  R0, a   => Cost = 1 + 0 + 1 = 2
```

Total cost = 6

Here, a, b, c are memory

R0 is register