Covered Topics Under UNIT-3 of "CD- COMPILER DESIGN (KIT-052)"

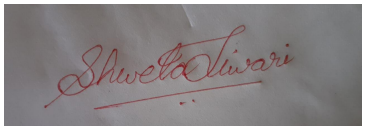
shwetatiwari08@recabn.ac.in
shwetatiwari08aug@gmail.com

CD: UNIT-3

Syntax-Directed Translation

FALL SEMESTER, YEAR (V/VI, 3rd)

FALL SESSION (2022-23)
(CD)
MS. SHWETA TIWARI
Published: September, 2022



shwetatiwari08@recabn.ac.in
shwetatiwari08aug@gmail.com

TOPIC On : UNIT-3

Method of Generating Intermediate Code Generation

—

By SHWETA TIWARI

Under On: Syntax-Directed Translation

PREPARED FOR
Engineering Students
All Engineering College

PREPARED BY
SHWETA TIWARI
Guest Faculty

TOPIC On :

Method of Generating Intermediate Code Generation

There are many methods to generate intermediate code generation.

- **Machine Independent**
- **Abstract Syntax Tree**
- **Directed Acyclic Graph**
- **Postfix**
- **3-Address Code**

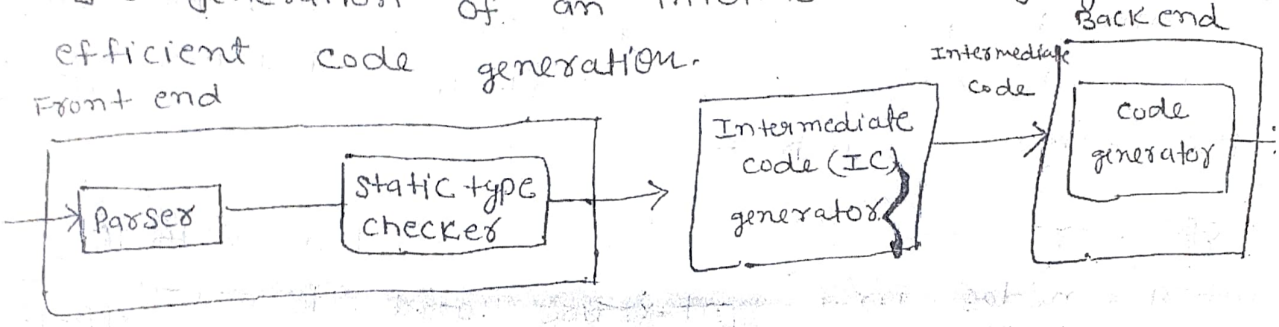
Here, we discuss

- **Abstract Syntax Tree**
- **Directed Acyclic Graph**
- **Postfix**

Intermediate Code Generation

- The task of a compiler is to convert the source program into machine program. This activity can be done directly, but it is not always possible to generate such a machine code directly in one pass then typically compilers generate an easy to represent form of source language which is called intermediate language.

- The generation of an intermediate language leads to efficient code generation.



Intermediate code generator in compiler

Benefits of Intermediate Code generation :-

(1.) A compiler for different machines can be created by attaching different backend to the existing front ends of each machine.

(2) A compiler for different source languages on the same machine can be created by providing different front ends for corresponding source languages to existing back end.

(3.) A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

Forms of Intermediate Code

There are mainly three types of intermediate code representations:

1. Abstract syntax tree
2. Polish notation
3. Three address code

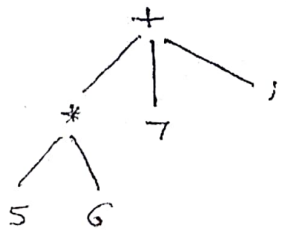
Abstract Syntax Tree

- A syntax tree is a condensed form of parse tree which is used to represent the syntactic structure of the language construct.
- operators and keywords are associated with the interior nodes in a syntax tree and do not appear as leaves.
- Direct Acyclic Graph (DAG) is very much similar to syntax trees but they are in more compact form.
- A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.

Syntax Tree

- The syntax tree is a tree that shows the syntactic structure of a program, while omitting irrelevant details present in a parse tree
- The syntax tree is an abstract representation of the language constructs.
- Each node of the syntax tree represents "what to do" at that point, that is computation. The children of the node correspond to the objects to which that computation is applied.

eg 5 * 6 + 7 ;



syntax tree

Structure of Syntax Trees

Expression : In case of expression leaves, internal nodes and children can be :

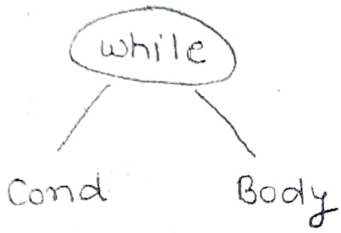
- leaves : identifiers or constants
- Internal nodes : labelled with operations.
- Children : of a node are its operands

Statement : In case of statements structure will

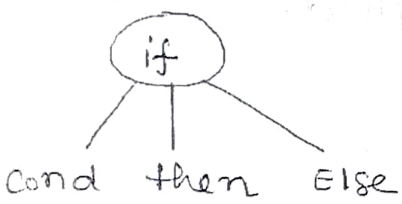
be as follows:

- A node's label indicates what kind of statement it is
- The children of a node correspond to the components of the statement

examples

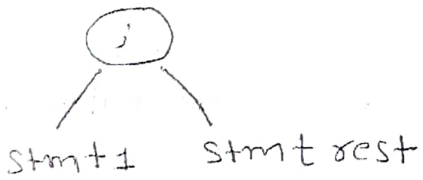


For: while (cond)
{
}



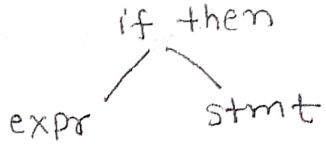
For: if cond
then ...
Else

For:
LHS := RHS Kind Statement

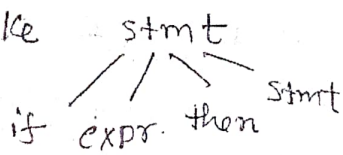


For:
Stmt1 ; Stmt rest
Kind Statements

eg.
The production stmt → if expr then stmt appears in a syntax tree like:



But in a parse tree like



Construction of Syntax Tree

- Each node in a syntax tree can be implemented as a record with several fields
- Constructing syntax tree for an expression means translation of expression into post fix form
- The nodes for each operator and operand are created
- The following functions are used to create the nodes of syntax tree

(1) mknode (op, left, right): This function creates a node with the field operator having operator as label, and two pointers to left and right.

(2) mkleaf (id, entry): This function creates an identifier node with label id and a pointer to symbol table is given by 'entry'

(3) mkleaf (num, val): This function creates node for number with label num and val is for value of that number

example expression $a * y - 5 + z$

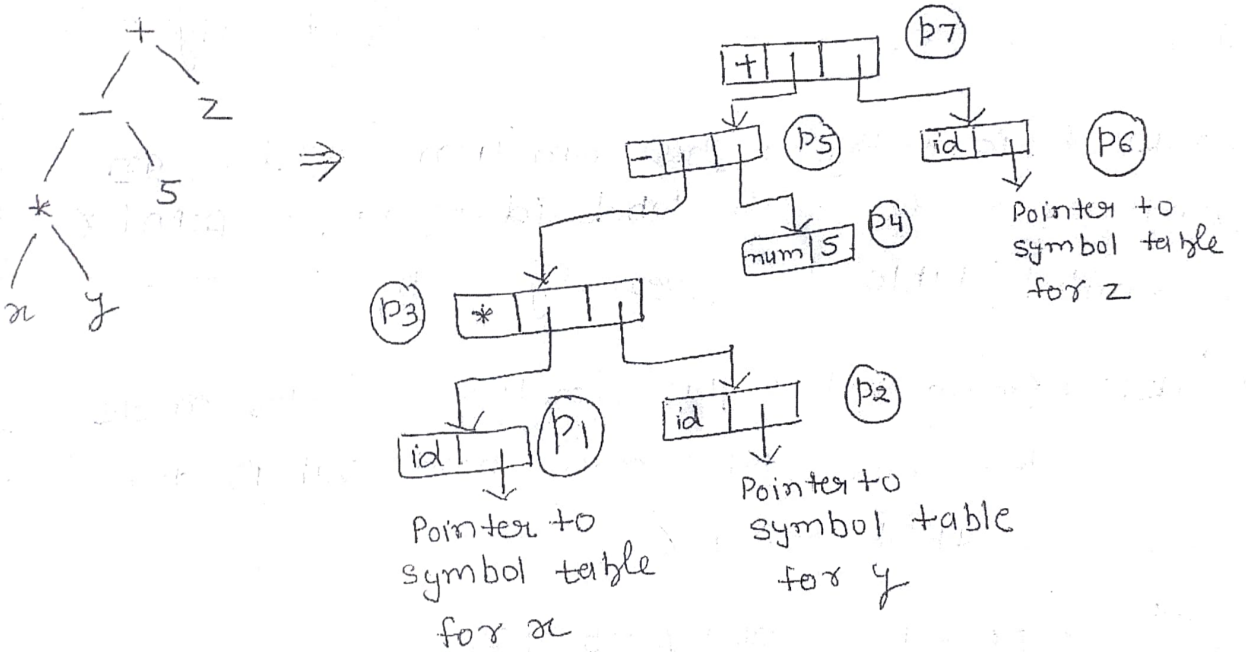
step I: Convert the expression from infix to postfix $ay * 5 - z +$

step II: Make use of the functions mknode(),

mkleaf (id, ptr) and mkleaf (num, val)

step3 The sequence of function calls is given by Postfix expression x y * 5 - z +

Symbol	operation
x	p_1 = mkleaf (id, ptr to entry x)
y	p_2 = mkleaf (id, ptr to entry y)
*	p_3 = mknode (*, p1, p2)
5	p_4 = mkleaf (num, 5)
-	p_5 = mknode (-, p3, p4)
z	p_6 = mkleaf (id, ptr to entry z)
+	p_7 = mknode (+, p5, p6)



syntax tree

A DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Directed Acyclic Graph (DAG) for Expression

- The directed acyclic graph (DAG) is a directed graph drawn by identifying the common subexpressions in the expression.
- Like syntax tree DAG has nodes representing the subexpressions in the expression. These nodes have operator, operand 1 and operand 2, where operands are the children of that node.

The difference between DAG and Syntax tree is that common subexpressions has more than one parent and in syntax tree the common subexpression would be represented as duplicated subtree.

example

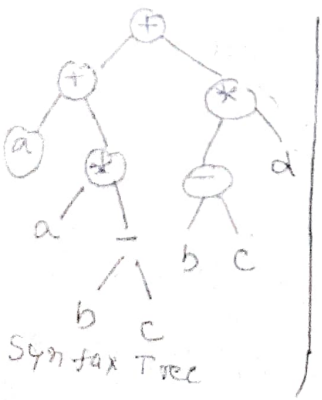
consider the expression

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

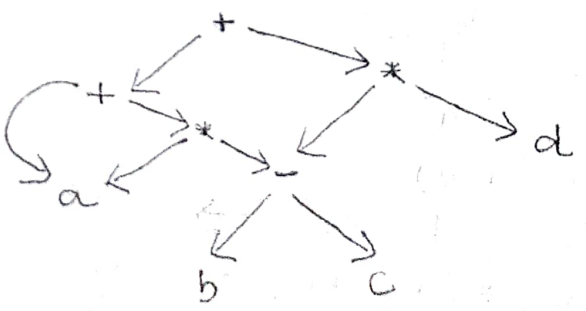
Post fix :-
$aabc- * + bc-d * +$

Here leaf for 'a' has two parent because 'a' is common in two sub expressions a and $a * (b-c)$.

✓ $(b-c)$ is also common in $a * (b-c)$ and $(b-c) * d$ subexpressions



Syntax Tree



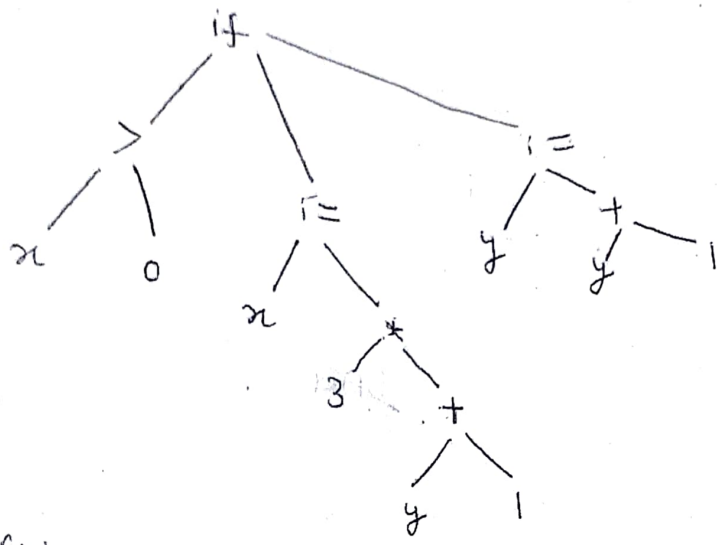
DAG

Q. Draw Syntax tree and DAG

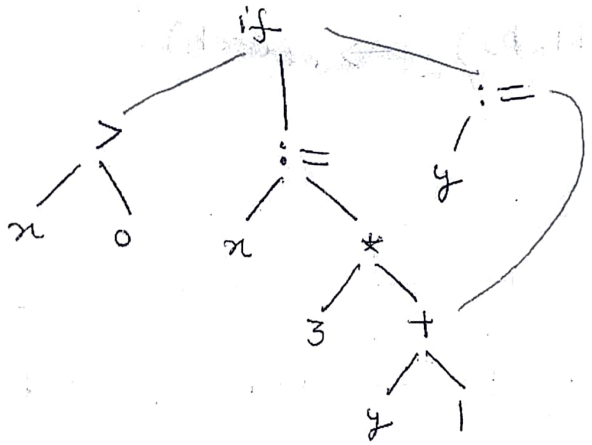
(a) if (x>0) then x := 3*(y+1) else y := y+1 ;

(b) x := -a * b + -a * b

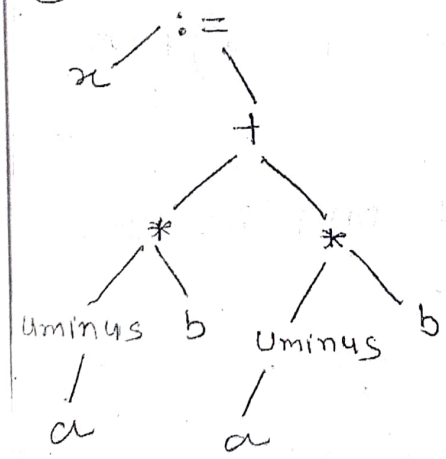
Syntax tree :-



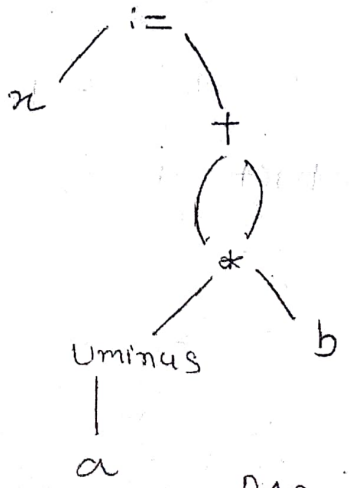
DAG :



(b)



Syntax tree



DAG

Construction of (Syntax Tree and DAG) using function

Step-1 Convert the expression from infix to postfix.

Step-2 Make use of the function mknode(op, left, right) mkleaf(id, ptr) and mkleaf(num, val)

Step-3 The sequence of function calls is given by postfix expression.

Step-4 Make (Syntax Tree And DAG) of the expression

Shweta Tiwari

Question-1 Construction of Syntax Tree of expression using function. (A+B)*C

Answer- Step1 Convert the expression from infix to postfix.

(A+B)*C infix

(A+B)*C

(AB+)*C

AB+C* postfix

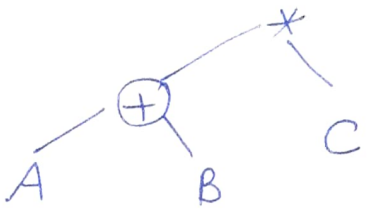
Step-2 Make use of the function mknode(op, left, right) mkleaf(id, entry) and mkleaf(num, val)

Step-3 The sequence of function calls is given by postfix expression AB+C*

Symbol	operation
A	p_1 = mkleaf(id, A)
B	p_2 = mkleaf(id, B)
+	p_3 = mknode(+, p_1, p_2) → (A+B)
C	p_4 = mkleaf(id, C)
*	p_5 = mknode(*, p_3, p_4) → (A+B)*C

PS-2

Step-4



Syntax Tree

Question-2 Construction of Syntax Tree of expression using function A*B-(C+D)+E

Answer Step-1 Convert the expression from infix to postfix.

A*B-(C+D)+E infix

A*B-(CD+)+E

AB*-(CD+)+E

(AB*CD+-)+E

AB*CD+-E+ postfix

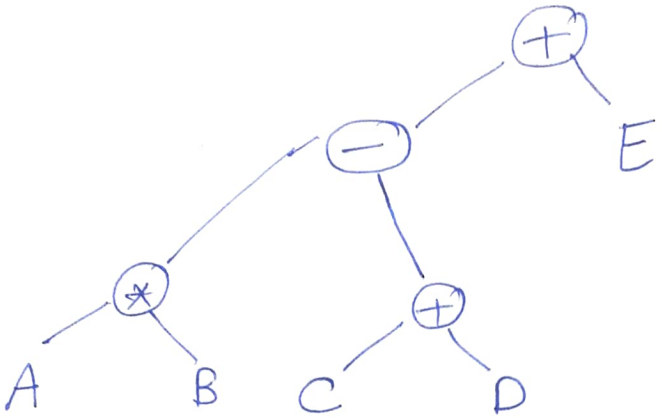
Step-2 Make use of the function (mknode(op, left, right) mkleaf(id, entry) and mkleaf(num, val)

Step-3 The sequence of function calls is given by postfix expression. AB*CD+-E+

Symbol	operation
A	p_1 = mkleaf(id, A)
B	p_2 = mkleaf(id, B)
*	p_3 = mknode(*, p_1, p_2) → (A*B)
C	p_4 = mkleaf(id, C)
D	p_5 = mkleaf(id, D)

+	p6 = mknode(+, p4, p5) → (C+D)
−	p7 = mknode(− p3, p6) → (A*B) − (C+D)
E	p8 = mkleaf(id, E)
−	p9 = mknode(−, p7, p8) → A*B − (C+D)+E

Step-4 Make Syntax tree of given expression



Question 3 Construction of Syntax tree of expression using function ((A+B)−C*(D/E))+F

Answer Step-1 Convert the expression from infix to postfix.

((A+B)−C*(D/E))+F infix

((AB+)−C*(DE/))+F

((AB+)−(DE/*))+F

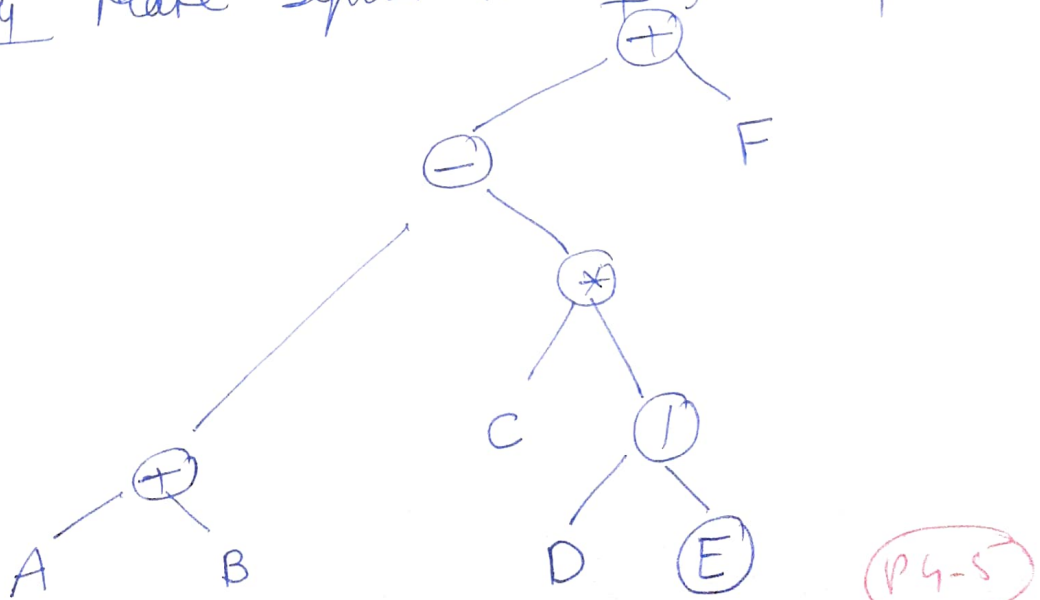
AB+DE/*−F+ postfix

Step-2 Make use of the function mknode(op, left, right), mkleaf(id, entry) and mkleaf(num, val).

Step-3 The sequence of the function calls is given by postfix expression. AB+CDE/*-F+

Symbol	operation
A	p_1 = mkleaf(id, A)
B	p_2 = mkleaf(id, B)
+	p_3 = mknode(+, p_1, p_2) → (A+B)
C	p_4 = mkleaf(id, C)
D	p_5 = mkleaf(id, D)
E	p_6 = mkleaf(id, E)
/	p_7 = mknode(/, p_5, p_6) → (D/E)
*	p_8 = mknode(*, p_4, p_7) → C*(D/E)
-	p_9 = mknode(-, p_3, p_8) → ((A+B)-C*(D/E))
F	p_{10} = mkleaf(id, F)
+	p_{11} = mknode(+, p_9, p_{10}) → (((A+B)-C*(D/E))+F

Step 4 Make Syntax tree of given expression



P4-5

Question-4 Construction of syntax tree and DAG for expression $(((a*b)+(c-d))*(a*b))+b$

Step-1 Convert the expression from infix to postfix.

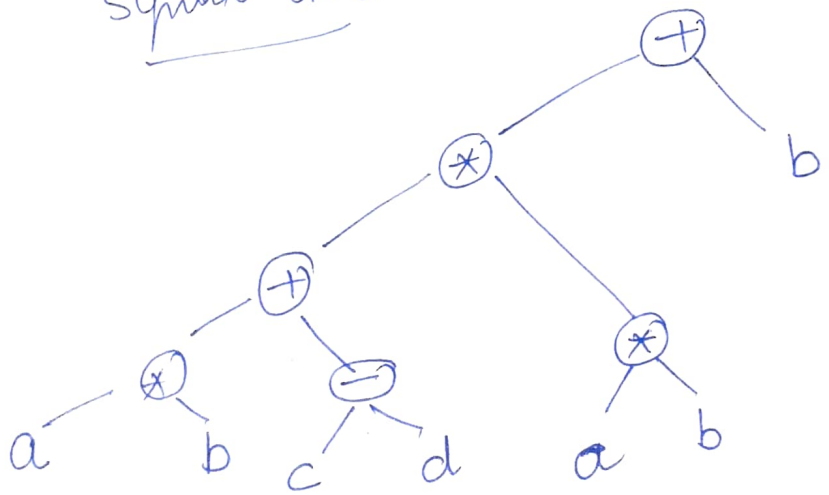
$(((a*b)+(c-d))*(a*b))+b$ infix

$(((ab*)+(cd-))*(ab*))+b$

$((ab*\ cd-+)*(ab*))+b$

$ab*\ cd-+\ ab**b+$ postfix

syntax tree

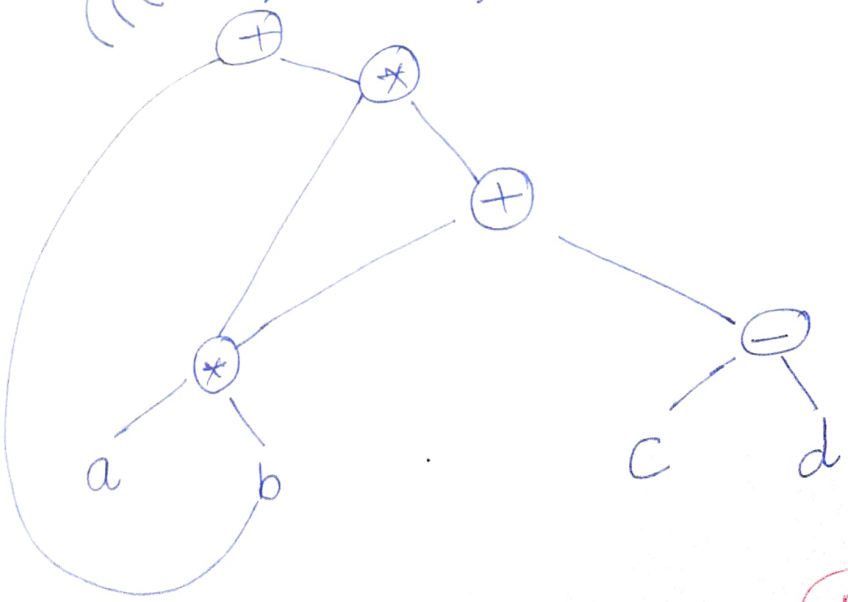


The sequence of operation for syntax tree

Symbol	operation
a	$p_1 = mkleaf(id, a)$
b	$p_2 = mkleaf(id, b)$
*	$p_3 = mknode(*, p_1, p_2) \rightarrow (a*b)$
c	$p_4 = mkleaf(id, c)$
d	$p_5 = mkleaf(id, d)$
-	$p_6 = mknode(-, p_4, p_5) \rightarrow (c-d)$
+	$p_7 = mknode(+, p_3, p_6) \rightarrow (a*b)+(c-d)$
a	$p_8 = mkleaf(id, a)$
b	$p_9 = mkleaf(id, b)$
*	$p_{10} = mknode(*, p_8, p_9) \rightarrow (a*b)$
*	$p_{11} = mknode(*, p_7, p_{10}) \rightarrow ((a*b)+(c-d))*(a*b)$
b	$p_{12} = mkleaf(id, b)$
+	$p_{13} = mknode(+, p_{11}, p_{12}) \rightarrow (((a*b)+(c-d))*(a*b))+b$

DAG Directed Acylic Graph

expression $(((a*b)+(c-d))*(a*b))+b$



The sequence of operation for DAG.

Symbol	operations
a	p_1 = mkleaf (id, a)
b	p_2 = mkleaf (id, b)
*	p_3 = mknode (*, p_1, p_2) → (a*b)
c	p_4 = mkleaf (id, c)
d	p_5 = mkleaf (id, d)
-	p_6 = mknode (-, p_5, p_6)
+	p_7 = mknode (+, p_3, p_6) → (a*b) + (c-d)
*	p_8 = mknode (*, p_7, p_3) → ((a*b)+(c-d))*(a*b)
+	p_9 = mknode (+, p_8, p_2) → (((a*b)+(c-d))*(a*b))+b

Shweta Tiwari