Date / /

9/11/22

Syntax Directed Translations

Syntax Directed { Definition / Translation }

Attribute → value, memory space

Produce semantic rules.

$$E.val = E.\underline{val} + T.val$$

↓ attribute val which is of integer type

A syntax directed definition is a generalisation of a context free grammar in which each grammar symbol has an set of attributes, partioning into two subsets called the synthesized and inherited attributes of that grammar symbol.

The attribute can represent value, number, type, string, memory location etc.

The value of an attribute at a parse tree node is defined by a semantic rule associated with the production used at that node.

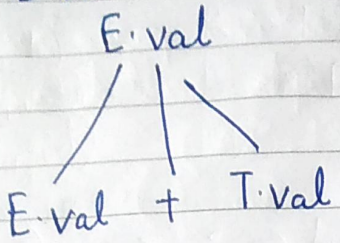
The value of a synthesized node attribute is computed by the children of that node at a parse tree.

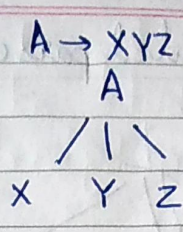
Value of a inherited attribute is computed from the values of attributes at the siblings and the parents at that node.

Saathi

Date ___/___/_____

$E.val = E.val + T.val$

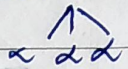




$A \to \alpha$

$b = f(c_1, c_2, c_3 \ldots c_n)$

b is attribute of A then A can be represented as A



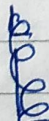
↳ inherited attribute

Semantic rule, $b = f(c_1, c_2, c_3 \ldots c_n)$ This is the form of a syntax directed definition

* **Desk Calculator Program**

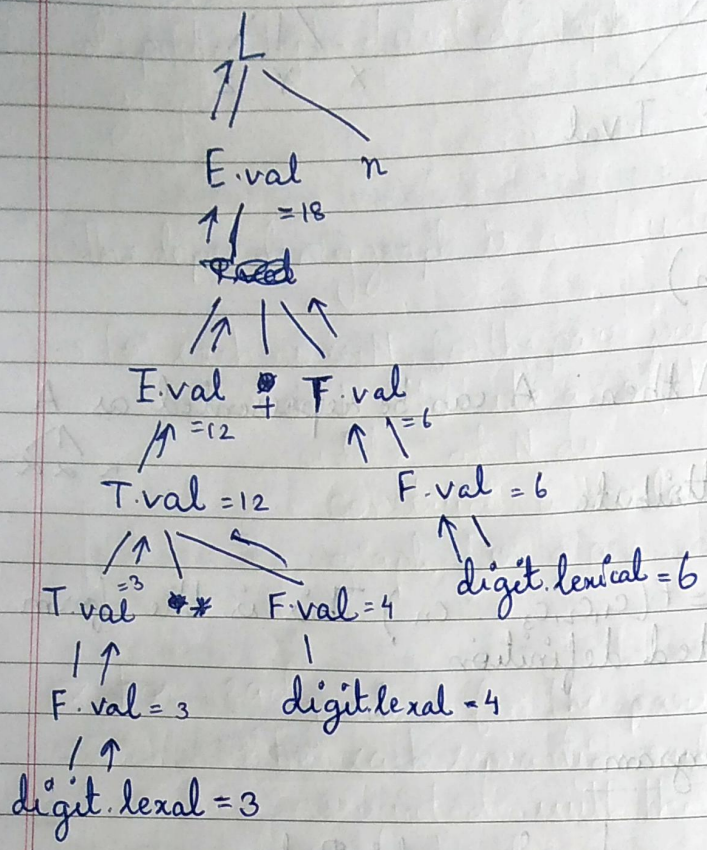
Production	Semantic Rules
	print (En)
$E \to En$	$E.val = E.val + T.val$
$E \to E_1 + T$	$E.val = T.val$
$E \to T$	$T.val = T_1.val * F.val$
$T \to T_1 * F$	$T.val = F.val$
$T \to F$	$F.val = E.val$
$F \to (E)$	$F.val = digit.lexval$
$F \to id$	

We calculate 3*4+6n

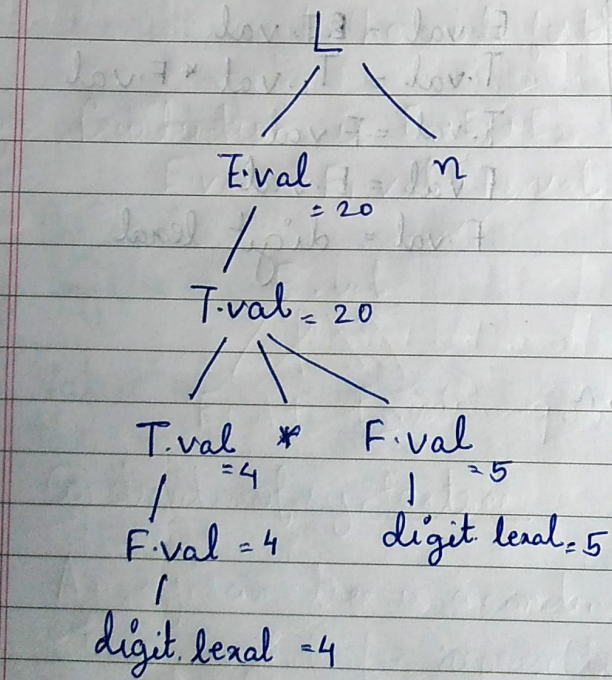


Date ___/___/_____

Annotated parse tree:



Annotated parse tree :4 * 5 n



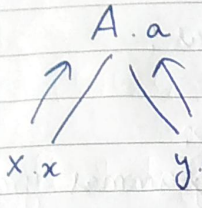
Date ___/___/___

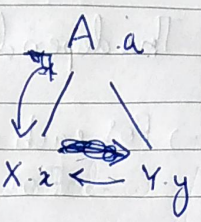
Saathi

Depedency graph

$A.a = f(x.x, y.y)$ → synthesised attribute

$X.x = f(a.a, y.y)$ → inherited attribute



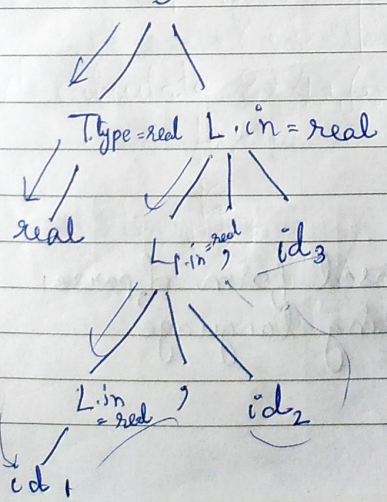


Semantic Rules (List of identifiers)

Production	Semantic Rules
$D \to TL$	$L.in = T.type$
$T \to int$	$T.type = int$
$T \to real$	$T.type = real$
$L \to L_1, id$	$L_1.in = L.in$
$L \to id$	$addtype(id.entry, L.in)$

$L \to$ read id_1, id_2, id_3

Annotated Parse Tree



Dependency graph

In an attribute b, at a node in a parse tree depends on the attribute c, then the semantic rule for b, then the semantic rule for b at that node must be evaluated after the semantic rule, that defines c.

Page No. ____

Date ___/___/___

The interdependencies among the inherited & synthesized attribute at the nodes in a parse tree can be depicted by a directed graph called a dependency graph.

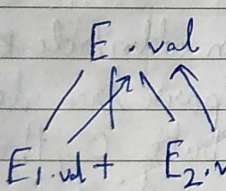
The dependency graph is constructed as:

→ For each node N, for the parse tree do:
  → For each attribute a of the grammar symbol ~~added~~ at N do
    → construct a node in the dependency graph for a;

→ For each node n for the parse tree do
  → For each semantic rules $b = f(c_1, c_2 \ldots c_k)$ associated with the production used at n do
  → for $i = 1$ to k do
    construct an edge from the node for c_i to the node for b

eg: Production is $E \to E_1 + E_2$

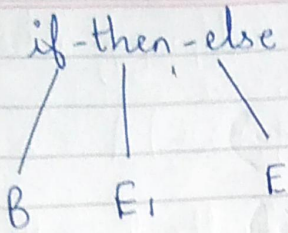
$E.val = E_1.val + E_2.val$

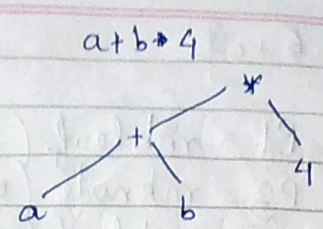


Construction of syntax tree

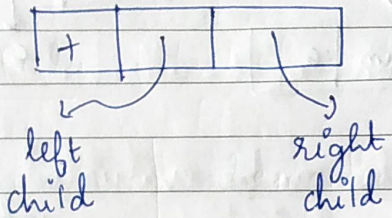
An abstract syntax tree is a condensed form of parse tree useful for representing language construction.

Page No.





Each node ~~of~~ in a syntax tree can be implemented as a record with several fields. In the node for an operator, one field identifies the operator and the remaining fields contain pointers to the nodes for the operators.



We use the following functions to create the nodes of syntax trees for expression with binary operators. Each function returns a pointer to a newly created node.

1) mknode (op, left, right)
creates an operator node with ~~label of~~ label op and a field containing pointers to left and right.

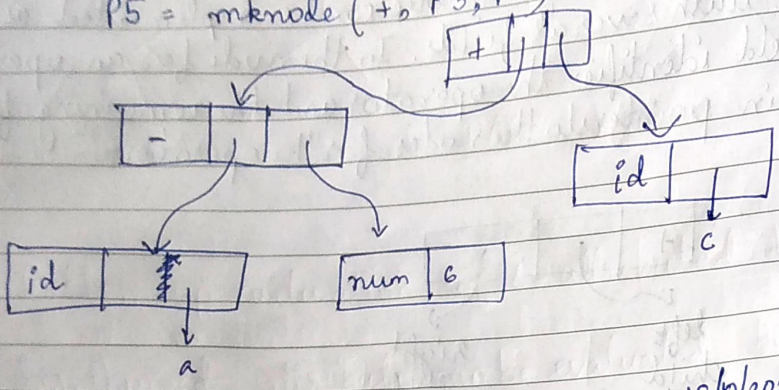
2) mkleaf (id, entry)
creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry to the identifier.

3) mkleaf (~~id~~ num, val) → creates a number node with ~~label~~ num and a field containing val, the value of the number.

Date ___/___/___

a - 6 + c

P1 = mkleaf (id, entry for a)
P3 = mknode (-, P1, P2)
P2 = mkleaf (num, 6)
P4 = mkleaf (id, entry c)
P5 = mknode (+, P3, P4)

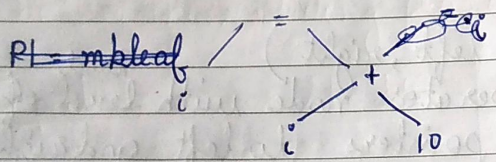


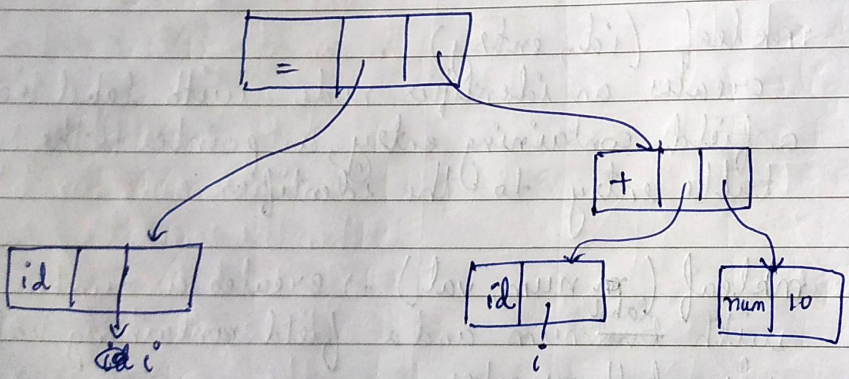
18/10/2022

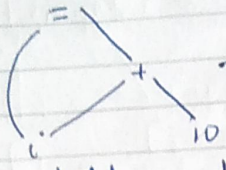
Syntax Tree

Directed Acydic Graph

i = i + 10 i.val = i.val + 10





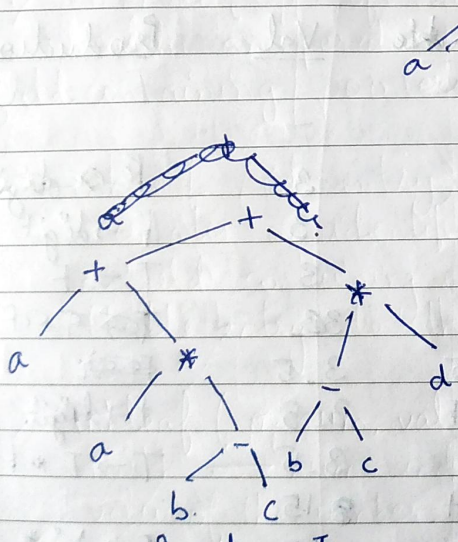


In DAG, a child can have more than one parent.

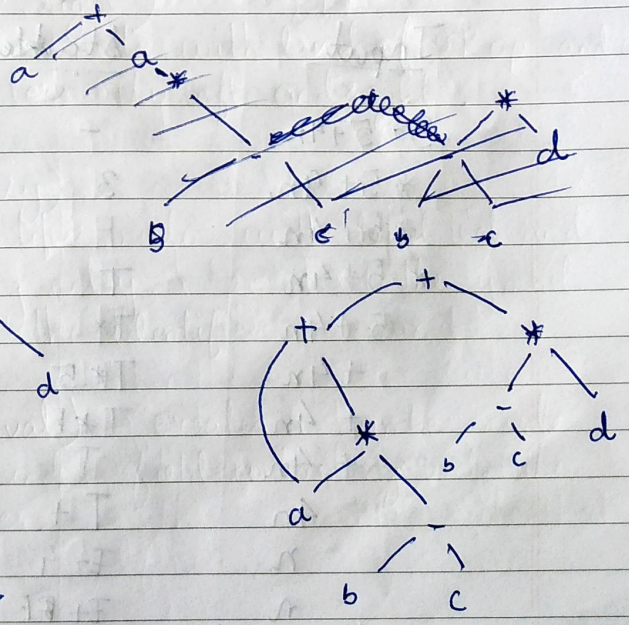
A directed acyclic graph for an expression identifies the common subexpression in the expression like a syntax tree.
Like a syntax tree a DAG has a node for every sub expression of the expression.
An interior node represents an operator and its children represents its operator / operand

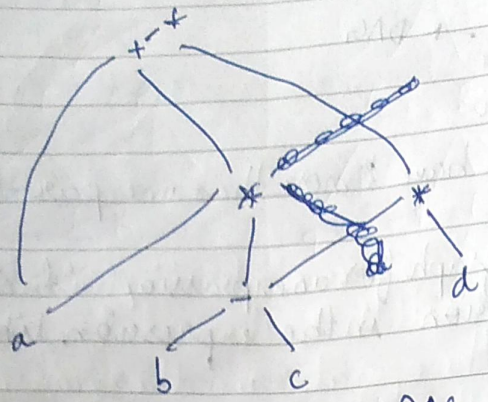
The difference is that a node in a DAG representing a common sub-expression has more than one parent in a syntax tree. The common sub-expression should be represented as a duplicate subtree.

Q) a → a * (b - c) + (b - c) * d



Syntax Tree





DAG

Bottom up evaluation of s-attributed definition

Synthesized attribute can be evaluated by a bottom up parser as the input is being parsed. The parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack. Whenever a reduction is made the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbol of the reducing production.

Input	Stack	Val	Production used
3*5+4n	-	-	
*5+4n	3	3	~~E → digit~~
*5+4n	F	3	F → digit
*5+4n	T	3	T → F
5+4n	T*	3_	~~T → T * F~~
+4n	T*5	3_5	~~F → digit~~
+4n	T*F	3_5	F → digit
+4n	T	3_5	T → T * F
4n	T+	15	
n	E+4	15	F → digit
n	E+F	15	
n	E		

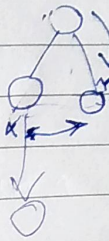
Saathi

Date

DFS evaluation order for attributes in a parse tree.

```
dfirst(n:node)
begin
    for each child m of n, from left to right do:
evaluate inherited attributes of m;
DF visit(m)
END
  Evaluate synthesized attribute of n
END
```

L-attributed definitions



A syntax directed definition is L attributed if each inherited attribute of X_j, $1 \le j \le n$ on the right side of production $A \to x_1 x_2 \ldots x_n$, depends only on the attributes of the symbol $X_1, X_2, \ldots X_{j-1}$ to the left of X_j in the production.

(ii) The inherited attributes of A

(iii) Translation scheme

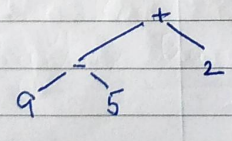
A translation scheme is a CFG in which attributes are associated with the grammar symbol and semantic action enclosed between braces ({}) are inserted within the right side of the production.

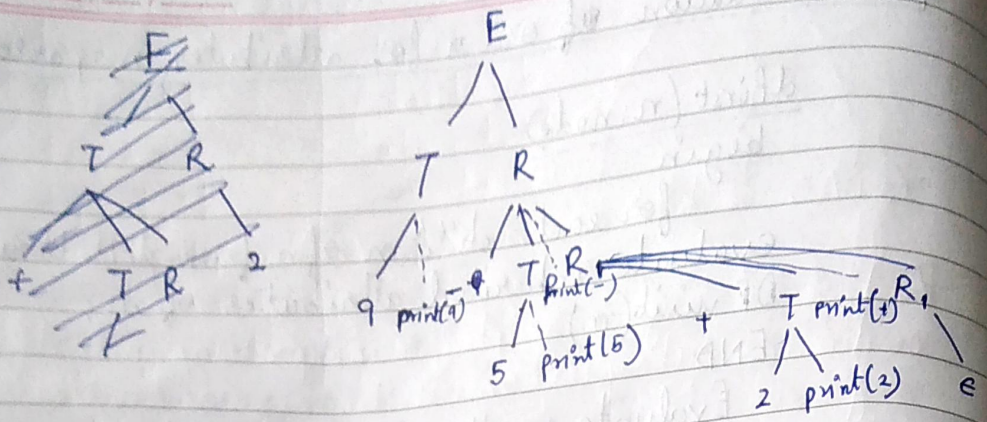
$E \to TR$

$R \to addop\ T\ \{print(addop.lexeme)\}R_1 \mid \epsilon$ ← semantic action

$T \to num\ \{print(num.val)\}$

Given 9-5+2 show action for infix to postfix using parse tree.





Output: 95-2+ (postfix) Input: 95+2 (Infix)

Bottom up evaluation of inherited attribute

D → TL
T → int
T → real
L → L, id
L → id

Input	State	Production used
real p, q, r	-	-
p, q, r	real	T → real
p, q, r	T	-
, q, r	Tp	-
q, r	TL	L → id
, r	TL,	-
r	TL, q	-
-	TL.	L → L, id
-	TL,	-
-	TL, r	-
-	TL	L → L, id
	D	D → TL

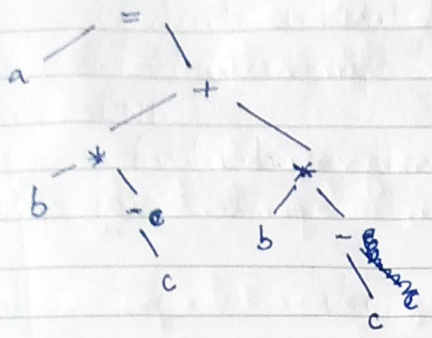
Date

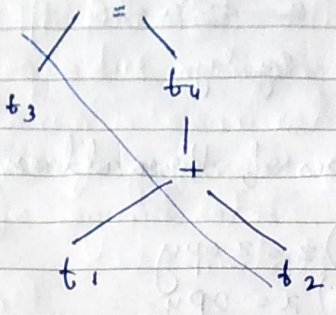
21/10/2022

Intermediate Code Generation

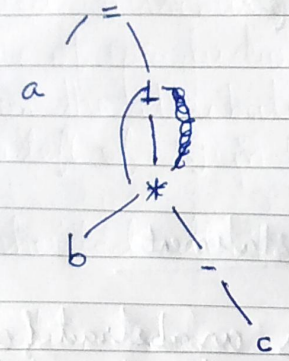
a) $a = b * -c + b * -c$

t_3 : t_4 ; t_1 : t_1 ; t_2 : t_2





Syntax Trees



DAG

Three address code

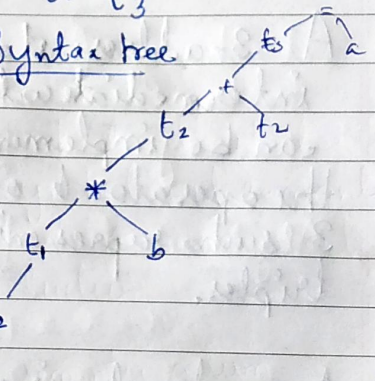
$t_1 = -c$

$t_2 = t_1 * b$

$t_3 = t_2 + t_2$

$a = t_3$

Syntax tree



Postfix Notation

a b c - * b c - * + =

$Z = x \ op \ y$ → Three address code since three operands & operator two.

$a = x + y + z$

$t_1 = y + z$

$a = x + t_1$

Date

ICG (from syntax tree)

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

ICG (from DAG)

$t_1 = -c$
$t_2 = t_1 * b$
$a = t_2 + t_2$

DAG

Saathi

Types of three address code:

i) $z = x \text{ op } y$
ii) $x = \text{op } y$
iii) $x = y$
iv) goto L
v) if x relop y goto L
vi) $x = y[i]$
vii) $x[i] = y$
viii) $x = *y$ | $x = *y$

Implementation of 3 Address statement

A 3-address statement is an abstract form of intermediate code. In a compiler these statements can be implemented as records with fields for the operator & operands.
3 such representation quadruples, triples and indirect triples.

A quadruple is a recorded structure with 4 fields operator(op), arg 1, arg2 and result.

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

Implementing using quadruple
The op field contains an internal code for the operator. The 3 address statement $x = y \text{ op } z$

is represented by placing y in arg1, z in arg2 and x in result.

Quadruple

	op	arg1	arg2	result
(0)	-	c		t1
(1)	*	t_1	b	t2
(2)	-	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t_4	t5
(5)	=	t5		a

triple

	op	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(3)	(1)

Transformation on basic block

A basic block compute a set of expressions. These expressions are the values of the names or exit from the block. 2 basic blocks are said to be equivalent if they compute the same set of expression.

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block.

But many of the transformations are useful for improving the quality of code, that will be generated from a basic block.

There are 2 important classes of local transformation that can be applied to bb. These are the ① structure

preserving transformation
② Algebraic transformation

① SPT → (Structure preserving transformation)
The primary SPT on bb are: →
i) common subexpression elimination
ii) deadcode elimination
iii) Renaming of temporary variable
iv) Interchange of two independent adjacent statements.

a = b+c
b = a-d
c = b+c
d = a-d

Renaming common sub expression

a = b+c
b = a-d
c = b+c
d = b

② Renaming the local variable
t = b+c
u = b+c

③ Interchange of two independent adj. st. →
t1 = b+c ⇒ ~~t1 = b+c~~ t2 = x+y
t2 = x+y ~~t2 = x+y~~ t1 = b+c

Algebraic transformation

x = x * 1 ⇒ x = x ??
x = x + 0

x = y ** 2
⇓
x = y * y

Date ___/___/___

Saathi

DAG representation of basic block:

(1) $t_1 = 4 * i$
(2) $t_2 = a[t_1]$
(3) $t_3 = 4 * i$
(4) $T4 = b[t3]$
(5) $t5 = t2 * t4$
(6) $t6 = prod + t5$ (prod = t_6)
(8) $t_7 = i + 1$
9(8) $i = t_7$
10(9) if $i <= 20$ goto(1)

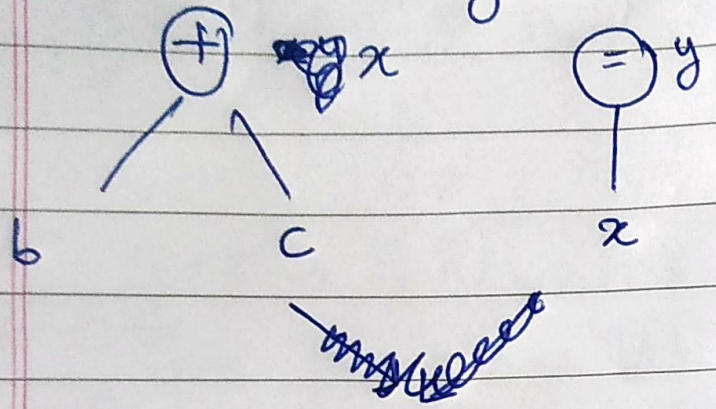
Directed Acyclic graphs are useful data structures for implementing transformation on basic block. A DAG gives a picture of how the value computed by each statement in a basic block is used in subsequent statements of the block. Constructing a DAG from 3 add-statements is a good way of determining common sub expression within a block, determining which names are used inside the block, but evaluated outside the block and determining which statements of the block could have their computed value used outside the block.

A DAG for a basic block is a directed acyclic graph for a following level of nodes:→

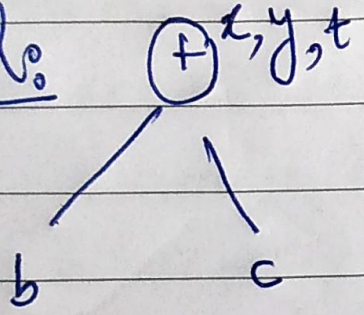
i) leaves are labelled by unique identifiers either variable names or constants.
ii) interior nodes are labelled by an operator symbol.
iii) nodes are also optionally given a sequence of identifier for labels.

Date ___/___/_____

$x = b + c$ $\quad y = x \quad t = y$



Sol:



$t_1 = 4 * i$