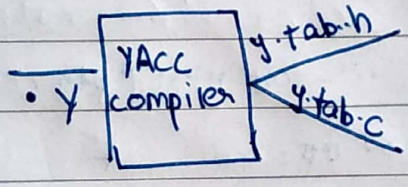
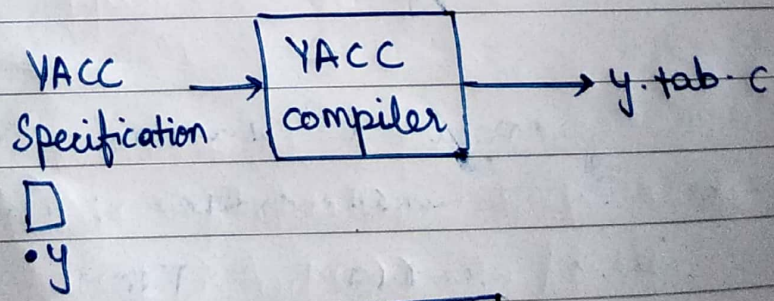
31-01-'19

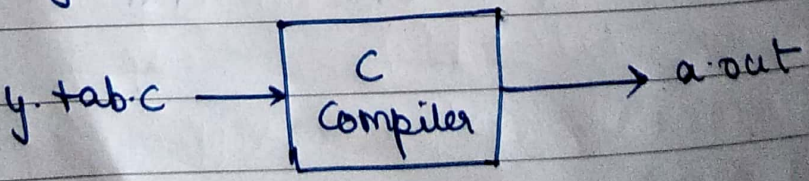
Parser Generators

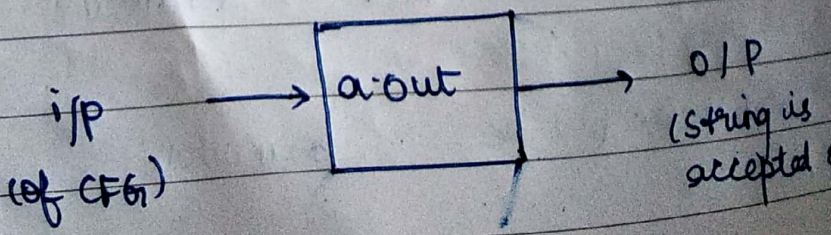
- It is an existing tool.
- It has YACC tool (yet another compiler compiler)
- It is an automatic in-built tool in UNIX environment, like LEX (Lexical Analyzer Generator)
- It consists of three parts:
  (i) Declarations (optional)
  (ii) Context Free Grammar / Semantic Actions (i.e, translation rules) (Compulsory)
  (iii) C functions (optional)
- y.tab.h is a header file containing all tokens and y.tab.c is an executable file



- Structure of YACC tool:







- YACC source program contains three parts:

declarations

%%

translation rules

%%

supporting C functions

- To YACC compiler: YACC file_name.y
  ↳ For compilation

- Translation Rules
  - Context Free Grammar
  - Eg. %token digit

- Declarations are of two types
  (i) Declarations of C
  (ii) Token declaration
  %token digit

- Representing Context Free Grammar
  - <head> → <body 1> / <body 2> / <body 3> / ... / <body n>

<head> : <body 1> {Semantic action 1}
⋮
<head> : <body n> {Semantic action n}

body 1, body 2, body 3, body n etc. are 1st, 2nd, 3rd and nth alternatives for production.

- Sample Program:

```
%{ #include <ctype.h>
%}
%token DIGIT
%%
line : expr '\n' {printf ("%d\n", $1);}
     ;
expr : expr '+' term {$$ = $1 + $3;}
     | term          {$$ = $1}
     ;
expr : term '*' factor {$$ = $1 * $3;}
     | factor          {$$ = $1}
     ;
factor : '(' expr ')'  {$$ = $2}
       | DIGIT
       ;
%%
yylex()
{
   int c;
   c = getchar();
   if (isdigit(c))
   {
      yyval = c - '0';
      return DIGIT
   }
   return c;
}
```

Semantic Analyzer

- The meaning of Content Free Grammar is analyzed.
- It adds additional information to the content free grammar such as type correction, type mismatch and other attributes.
- Rules are included in productions and attributes are added to variables or tokens.
- Parse Tree + Rules + Attributes = Annotated Parse tree
  O/P of Semantic Analyzed
- Eg. CFGs for unsigned number

  digit → 0|1|.....|9

  number → digit | number digit

* Adding attributes

  digit.value = 0 digit → 0

  digit.value = 1 digit → 1

  here, .value = attribute, it is the value of the digit

  digit.value = 9 digit → 9

  number → digit ⇒ number.value = digit.value

  number$_1$ → number$_2$ digit

  number1.value → number2.value * 10 + digit

- Annotated Parse Tree: Include attributes in the tree and calculate the values or type match.

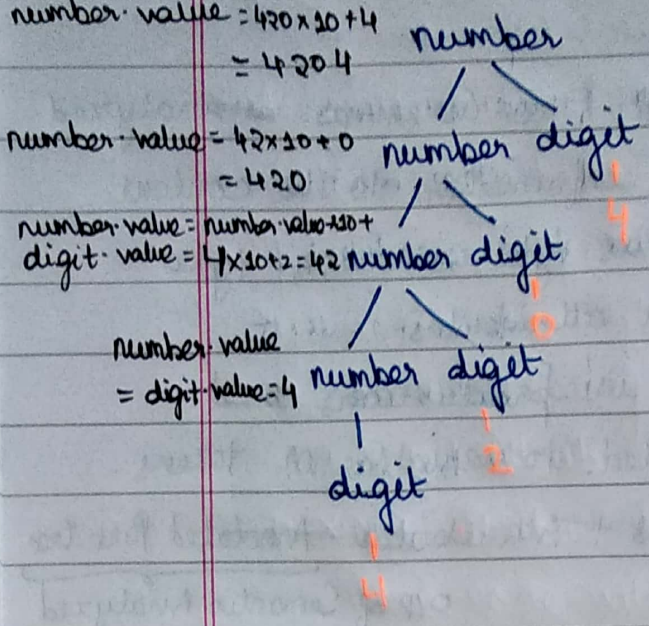
Eg. 4204

number.value = 420 × 10 + 4
= 4204

number.value = 42 × 10 + 0
= 420

number.value = number.value × 10 +
digit.value = 4 × 10 + 2 = 42

number.value
= digit.value = 4



⇒ This CFG contains Synthesized Attribute.

- Next, we have to add attributes.
Attributes are of two types:
(i) Synthesised Attribute ⇒ attribute depends on child node.
(ii) Inherited Attribute ⇒ value is calculated from either sibling node (prefrably left sibling) or parent node.

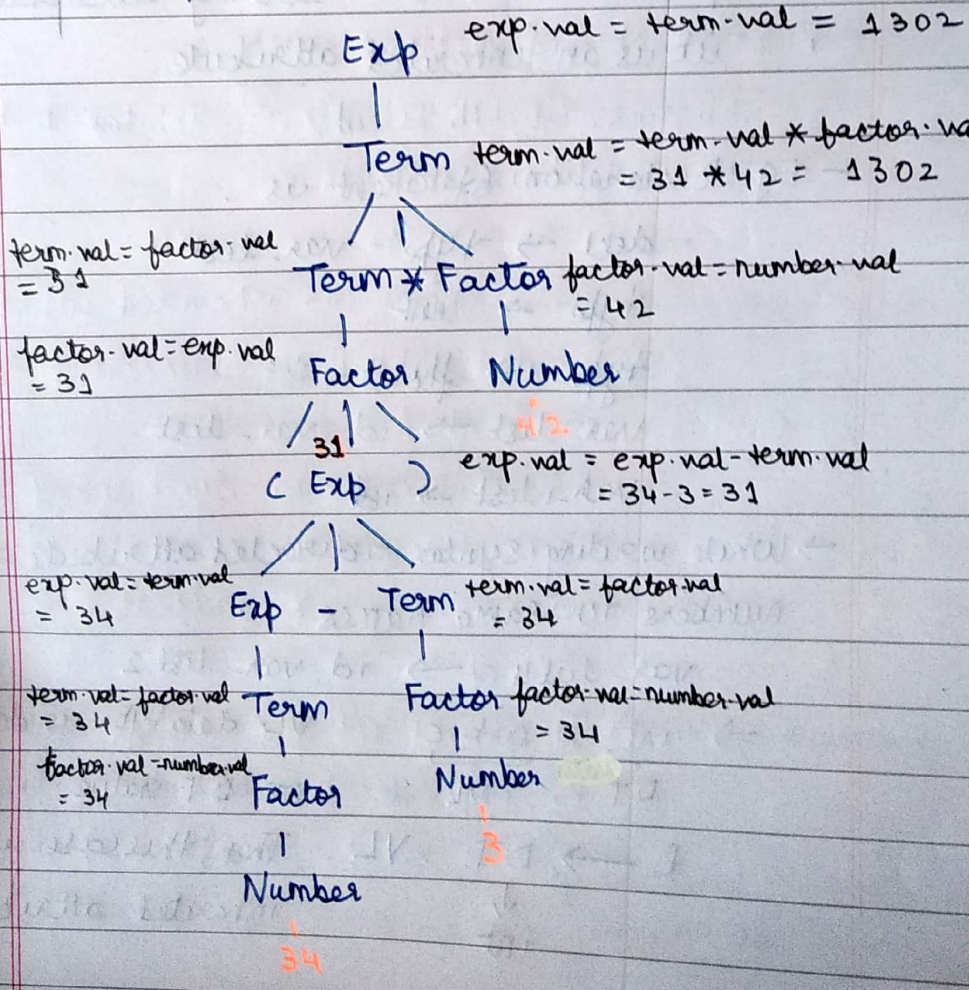
- Eg. CFG for Expression
Exp → Exp + Term / Exp - Term
Term → term * factor / factor
factor → (exp) / number

Writing the CFG with respect to attributes and rules (combination of CFG, attributes, rules) is called as Syntax Directed Definition.

Grammar Rule	Semantic Rule
exp1 → exp2 + term	exp1.val = exp2.val + term.val
exp1 → exp1 - term	exp1.val = exp2.val - term.val
exp → term	exp.val = term.val
term1 → term2 * factor	term1.val = term2.val * factor.val
term → factor	term.val = factor.val
factor → (exp)	factor.val = exp.val
factor → number	factor.val = number.val

Eg. (34-3) * 42

→ Annotated Parse Tree



01-02-'19

UNIT - III: SYNTAX DIRECTED TRANSLATION, INTERMEDIATE CODE GENERATION

Syntax Directed Translation

- A syntax directed definition is a syntax tree + rules + attributes. The value is calculated for parse tree, add rules and attributes, then it gives an annotated parse tree.
- The attributes may be Synthesised Attributes or Inherited Attributes.
  - * Node attribute value is depending on child attributes then node value is synthesized attribute.
  - * If a particular node 'A' is depending on either parent attribute or sibling attributes, then it is an inherited attribute.

- Eg: Declaration Statement is:

  decl → type var_list
  type → int
  type → float
  var_list → id , var_list
  var_list → id

→ While writing syntax-directed attribute assign numbers to same names.

var_list1 → id, var_list2

=> int a,b,c

DT VL

VL datatype is depending on DT value.

D → DT VL

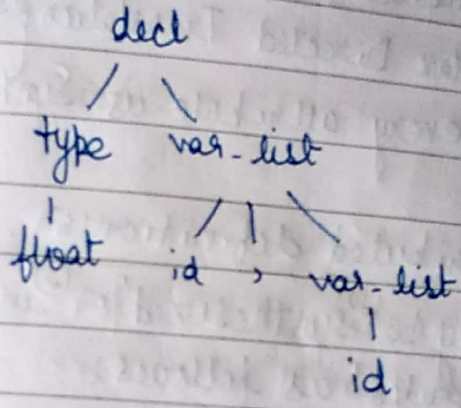
↓

int

Then, this case is inherited attributes.

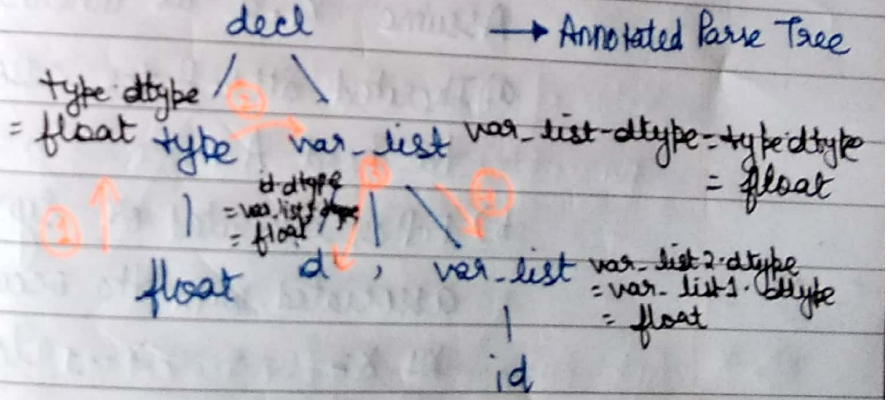
var_list.dtype = type.dtype
type.dtype = integer
type.dtype = float
id.type = var_list.dtype
var_list 2.dtype = var_list 1.dtype
id.dtype = var_list.dtype
↳ Type Declaration Grammar

Eg: float id, id;
⇒



→ Syntax Tree

⇒



→ Annotated Parse Tree

* Evaluation Order of Syntax Directed Definition
* Dependancy Graph

* Synthesised Attributed and ~~Dependancy Graph~~ Inherited Attribute

* S-attribute : Attribute that depends only on the synthesided attributes.
* L-attribute : May be synthesised attribute (depends on child node) or inherited attribute (depends only on variables left to it).
* A dependancy graph must have no cycles for L-attributes.

→ S-attributed definition(s):
- A Syntax Directed Translation (SDT) is S-attributed if every attribute is Synthesized

→ L-attributed definition(s):
- It may be ① 'synthesized' or ② inherited but with the 'rules' as follows:

  Let $A \rightarrow X1\ X2 \cdots\cdots Xn$

  Assume $X_i.a$ is inherited attribute

  a) Inherited attributes associated with the head A

  b) Either inherited or synthesized attributes associated with the occurances of symbols $X1, X2, \cdots\cdots, X_{i-1}$ located to the left of X_i

  c) It should not form cycles - There are no cyles in the dependancy graph formed by the attributes of this X_i.

Eg: CFG for either a simple variable or array variable

→ Eg: a[3] = array (3, int)
a[2,3] = array (2, array (3, int))

Grammar is:

Production	
T → B C	T.t = C.t
	C.b = B.t
B → int	B.t = integer
B → float	B.t = float
C → [num] C_1	C.t = array (num.val, C_1.t)
	C_1.b = C.b
C → ϵ	C.t = C.b

↳ Basic Data Type

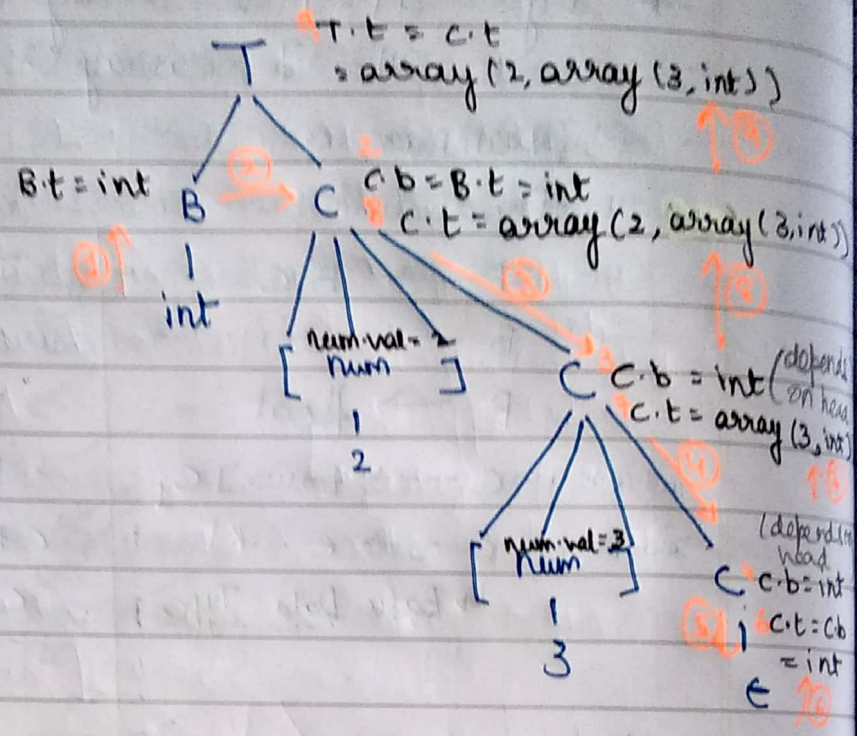
Here,
we have basic data types and derived data types (array)

Here,
we have derived and synthesised attributes
- t - synthesised attribute
- b - inherited attribute

B, T } - t - synthesised

C - b } - inherited

Eg: int [2][3]



Value of B is passed from top to bottom (depends on child node / parent node), hence it is inherited attribute.

Value of T is calculated from child node to parent node, hence it is synthesized attribute.

05-02-'19

* Applications of Syntax Directed Definitions

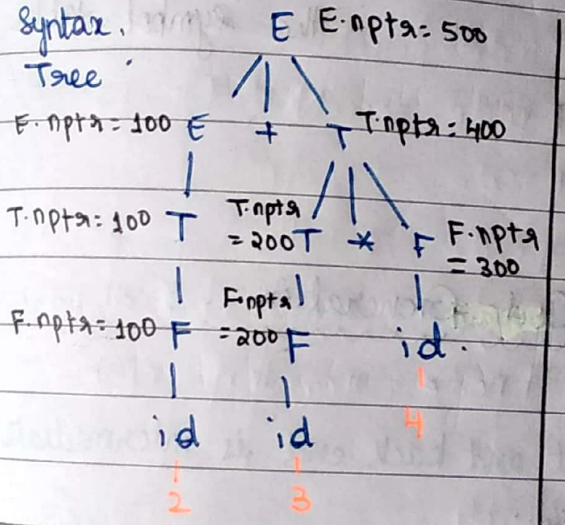
(i) Construction of Syntax Tree
(ii) SDD with Control Side Effects

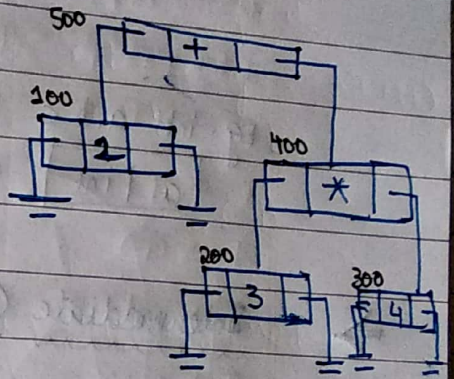
Construction of a Syntax Tree

$E \rightarrow E + T$	{E.nptr = mknode(E1.nptr, '+', T.nptr)}
$E \rightarrow T$	{E.nptr = T.nptr}
$T \rightarrow T * F$	{T.nptr = mknode(T1.nptr, '*', F.nptr)}
$T \rightarrow F$	{T.nptr = F.nptr}
$F \rightarrow id$	{F.nptr = mknode(null, idname, null)}

Eg. 2 + 3 * 4

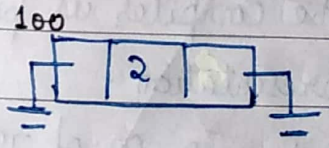
Syntax Tree

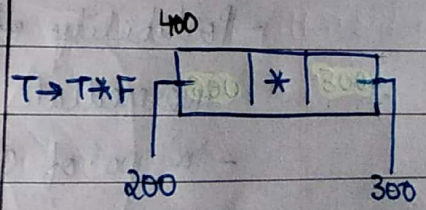




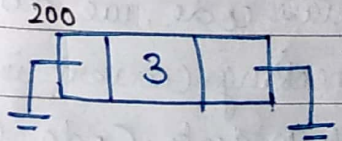
E

F → id



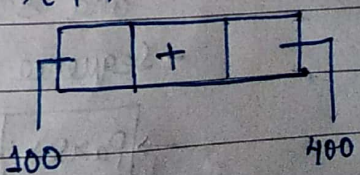


F → id

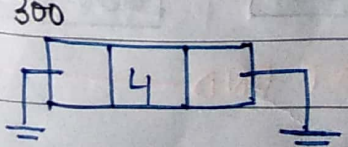


500

E → E + T



F → id



SDD with Control Side Effects

Eg:

$D \longrightarrow T\ L$	$L.inh = T.type$
$T \longrightarrow int$	$T.type = int$
$T \longrightarrow float$	$T.type = float$
$L \longrightarrow L1, id$	$L1.inh = L.inh$ addtype (id.entry, L.inh)
$L \longrightarrow id$	addtype (id.entry, L.inh)

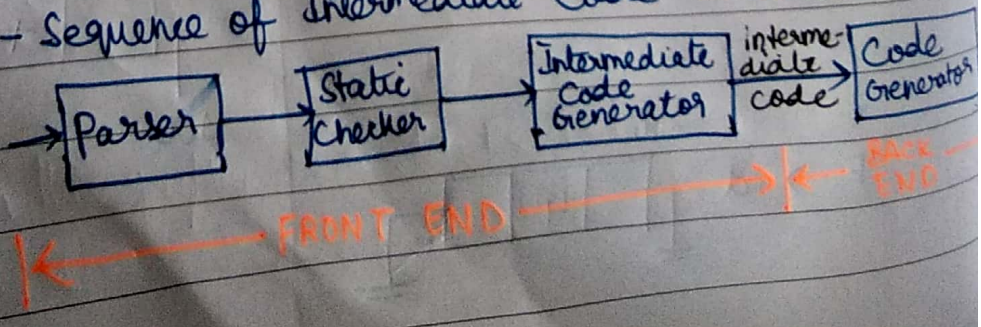
↳ passing information to the symbol table

Eg. int a,b,c

a	int

Intermediate Code Generation

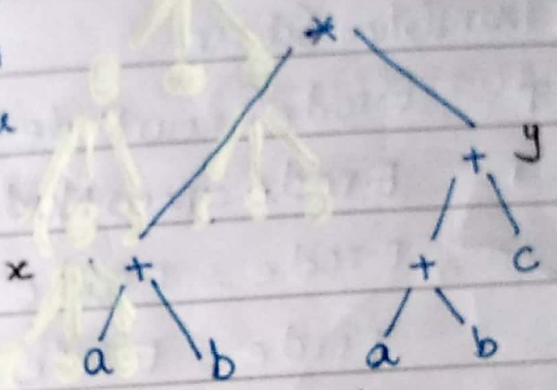
- Between front and back end is intermediate representation.
- Portability of the compiler is possible using intermediate representation.
  - 'n' no. of compilers, 'n' no. of intermediate codes
  - we can write intermediate code in 8085, 8086, etc.
  - Before intermediate code, we do type conversion and static checking. (convert int to float, etc.)
- Sequence of Intermediate Code Generator



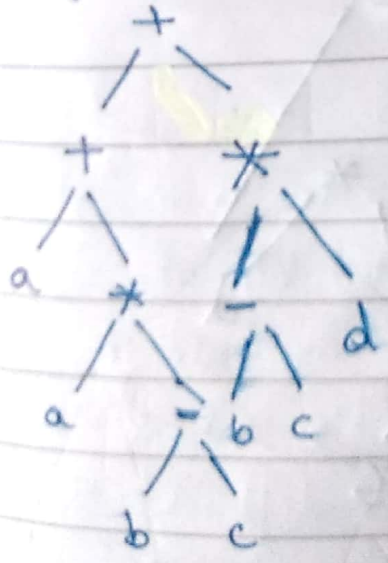
Variations of Syntax Trees

- To get optimised code, we write the variations of the syntax tree
- DAG (Directed Acyclic Graph) Representation
- To eliminate common sub-expression
- Eg. $\underbrace{(a+b)}_{x} * \underbrace{(a+b+c)}_{y}$

Operator Grammar



- a+b is redundant here
- If we represent it only once without redundancy it is called as a directed acyclic graph
- Eg. a + a * (b - c) + (b - c) * d



2 duplicate nodes for 'a'
" " " 'b'
" " " 'c'
" " "
" "

→

P_1 = leaf (id, entry-a)

~~P_2 = leaf (id, entry-a)~~

P_3 = leaf (id, entry-b)

P_4 = leaf (id, entry-c)

P_5 = node ('-', P_3, P_4)

P_6 = node ('*', P_1, P_4)

P_7 = node ('+', P_1, P_6)

Here, we can take P_2 also. Since P_1 is not modified we have P_1. else P_2.

~~P_8 = leaf (id, entry-b)~~

~~P_9 = leaf (id, entry-c)~~

~~P_{10} = node ('-', P_3, P_4)~~ (P_8, P_9)

P_{11} = leaf (id, entry-d)

P_{12} = node ('*', P_{10}, P_{11})

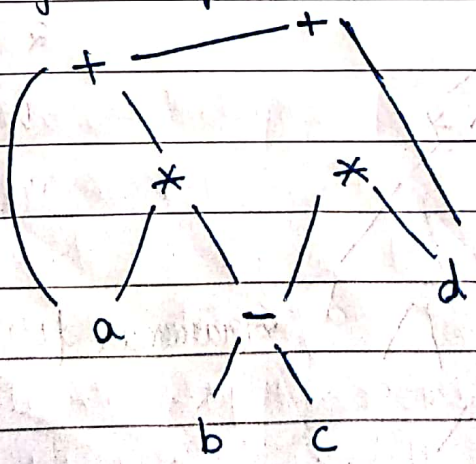
P_{13} = node ('+', P_7, P_{12})

Grammar / Translation Scheme.

$E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	E.node = new Node ('-', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = new leaf (id, id.entry)
$T \rightarrow num$	T.node = new leaf (num, numval)

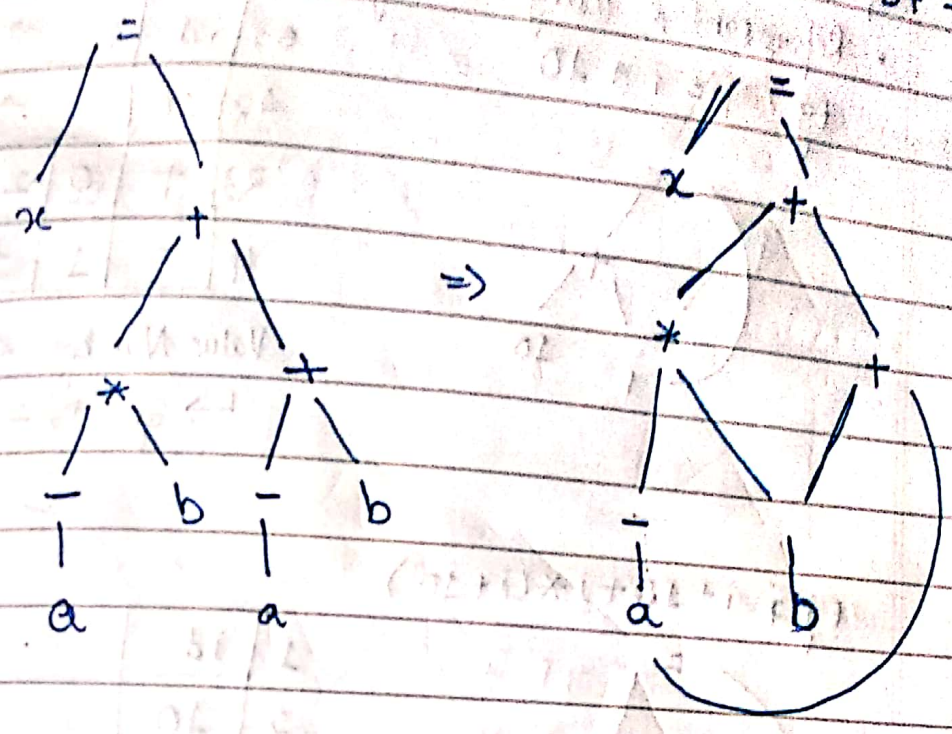
06-02-19

Directed Acyclic Graph



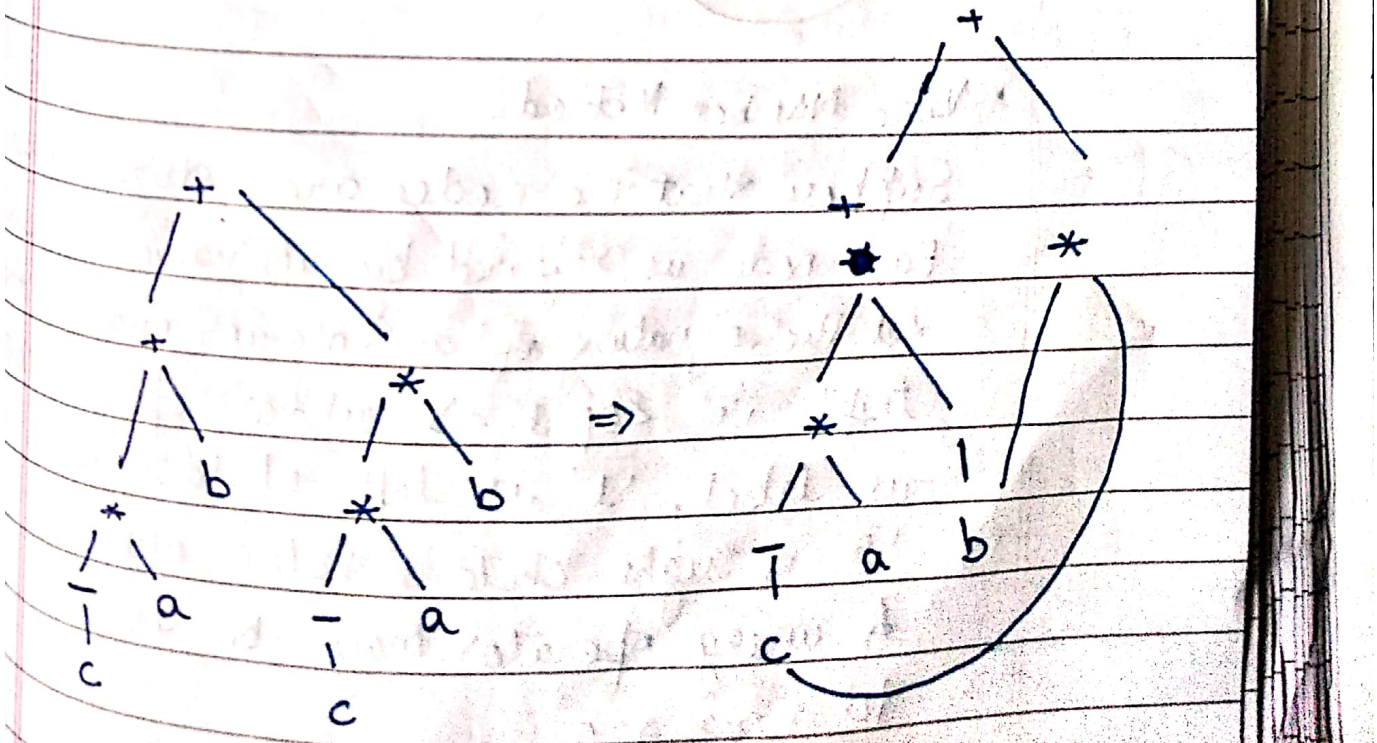
For this graph, nodes P_2, P_8, P_9, P_{10} are not required.

• Eg: Write the DAG representation for: x = -a * b + -a + b



- => replace with uminus (unary minus)
= => replace with assign

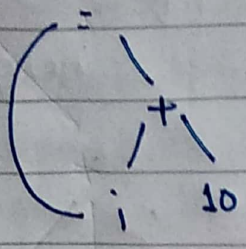
• Eg. Write the DAG representation for: (-c * a) + b + (-c * a) * b



→ Representation of DAG using Value Number Method

- Represented using arrays

Eg. $i = i + 10$

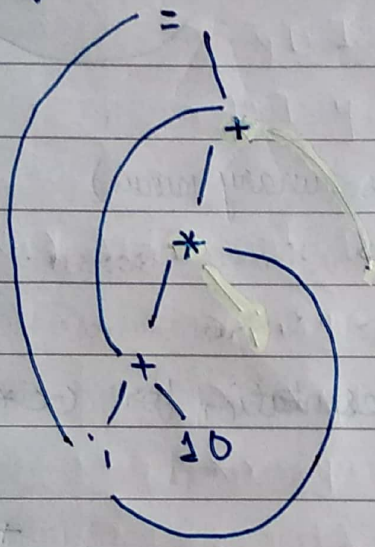


1	id		→ Entry into symbol table
2	10		→ Entry number 10
3	+	1	2
4	=	1	3

Value Number: <op, l, r> l, r

↳ Eg. +, =

Eg. $i = i + 10 + i * (i + 10)$



1	id		
2	10		
3	+	1	2
4	*	1	3
5	+	3	4
6	=	1	6

- Value Number Method:

Suppose that the nodes are stored in an array. Each node is referred by its value number. Let the signature of an interior node be the triple, i.e. <op, l, r> where operator (op) is label, 'l' is left child's value number 'r' is right child's value number. A unary operator may be assumed to have r = 0.

- Algorithm for Value Number Method (Node Construction)

I/P: Label op, node and noder

O/P: The value number of a node in the array with signature <op, l, r>

Method:

→ Three Address Code / IP Code

- It has address and instructions
- Max three addresses.
- Address: Name, Constant, Compiler Generated Tool, Constant Value
- The three address code is in different formats

(i) Assignment Instructions

x = y op z → Binary Operator

x = op y → Unary Operator

Eg. -c + a

t1 = -c

t2 = t1 + a

(ii) Copy Instructions

x = y

(iii) Unconditional Jump Instructions

goto L (L is next address to be executed)

(iv) Conditional Jump Instructions

if x goto L (L is next address to be executed)

if false x goto L

if x relop y goto L

(v) Procedure Calls and Returns

```
param x
calls p,n        (p = fn., n = no. of arguments)
return y
```

p(x1, x2, ..., xn)

```
param x1
  :
param x
param p,n
```

(vi) Indexed Operations / Indexed Copy Instructions

```
x = y[i]
x[i] = y
```

(vii) Address and Pointer Assignment

```
x = &y
x = *y
*x = y
```

Eg. a = b * -c + b * -c

Let t1 = -c t3 = -c

t2 = b * t1 t4 = b * t3

a = t2 + t2 t5 = t2 + t4

↳ Optimized ↳ Straight Forward

Eg. a + a * (b - c) + (b - c) * d

Let t1 = b - c t4 = t2 + t3

t2 = a * t1 t5 = a + t4

t3 = t1 * d

Eg. $x = -a * b + -a + b$

Let: $t1 = -a$

$t2 = t1 * b$

$t3 = t1 + b$

$x = t2 + t3$

Eg. $(-c * a) + b + (-c * a) * b$

Let: $t1 = -c$

$t2 = t1 * a$

$t3 = t2 * b$

$t4 = b + t3$

$t5 = t2 + t4$

Eg. $i = i + 10$

Let: $t1 = i + 10$

$i = t1$

Eg. $i = i + 10 + i * (i + 10)$

Let: $t1 = i + 10$

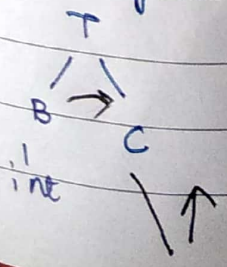
$t2 = i * t1$

$i = t3 = t1 + t2$

Eg. $T \rightarrow BC$ $B \rightarrow int$ $B \rightarrow float$ $C \rightarrow \epsilon$

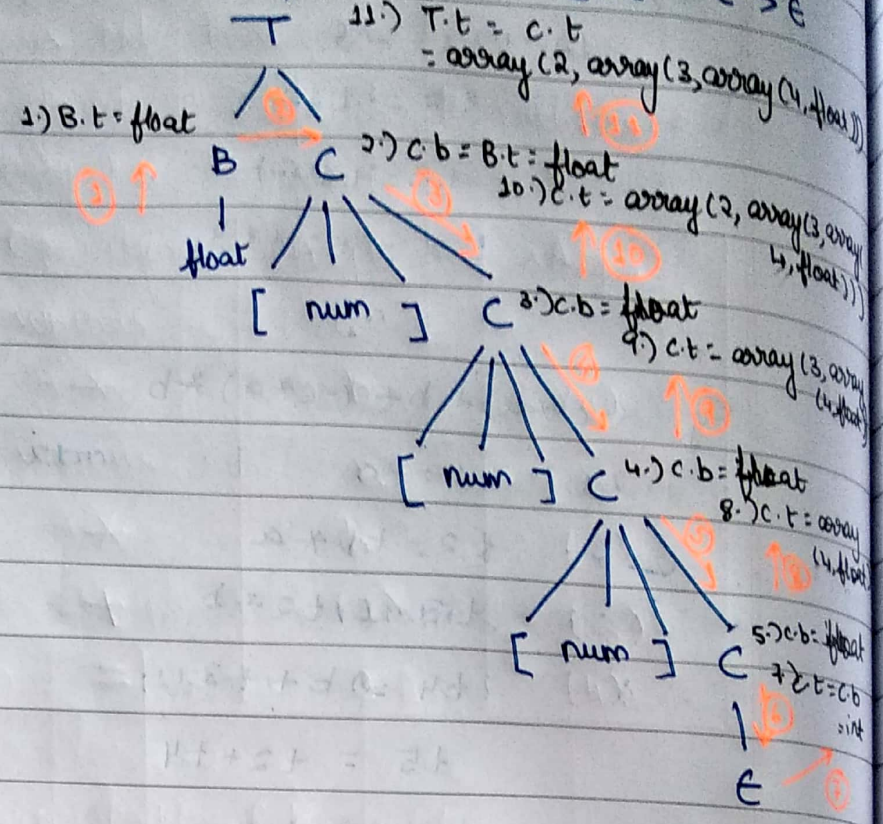
float [2][3][4]

$\Rightarrow$ array (2, array (3, array (4, float)))



B has one attribute to retreive the basic data type, so it is $\rightarrow$ synthesised attribute. Basic DT is sent to C, so it is inherited attribute, it is passed to bottom. Then, it is passed up (synthesized)

T → BC B → int B → float C → [num] C1 C → ϵ



07-02-'19

Eg. Write the three address code for:

```
do i = i+1
while (a[i] < v);
```

```
L : t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a[t2]
    while t3 < v goto L
```

12	10	3	4

a[0] = a + 0 * 8 = a = 2000 ↳ base addr

a[1] = a + 1 * 8 = a + 8 = 2008

a[2] = a + 2 * 8 = a + 16 = 2016

↳ width, for referencing array

```
100: t1 = i + 1
101: i = t1
102: t2 = i * 8
103: t3 = a[t2]
104: while t3 < v goto 100
```

Eg. Write the three address code for: x = a[i] + x + y

Let: t1 = i * 8

t2 = a[t1]

t3 = t2 + x

t4 = t3 + y

x = t4

→ Notations to represent Three Address Code:

(i) Quadrable Notation

(ii) Triple Notation

(iii) Indirect Triple Notation

Eg. a = b * -c + b * -c

Let: t1 = minus c

t2 = b * t1

t3 = minus c

t4 = b * t3

t5 = t2 + t4

a = t5

* Quadraple Notation:

	Operation	Arg1	Arg2	Result
1	minus	c		t1
2	*	b	t1	t2
3	minus	c		t3
4	*	b	t3	t4
5	+	t2	t4	t5
	=	t5		a

- Uses four arguments

* Triple Representation:
- Result is not there
- It is stored as a pointer to each record
- Uses only three arguments.

	Operation	Arg1	Arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

* Indirect Triple Representation:
- Similar to triple representation
- But a pointer is stored to denote the pointer.

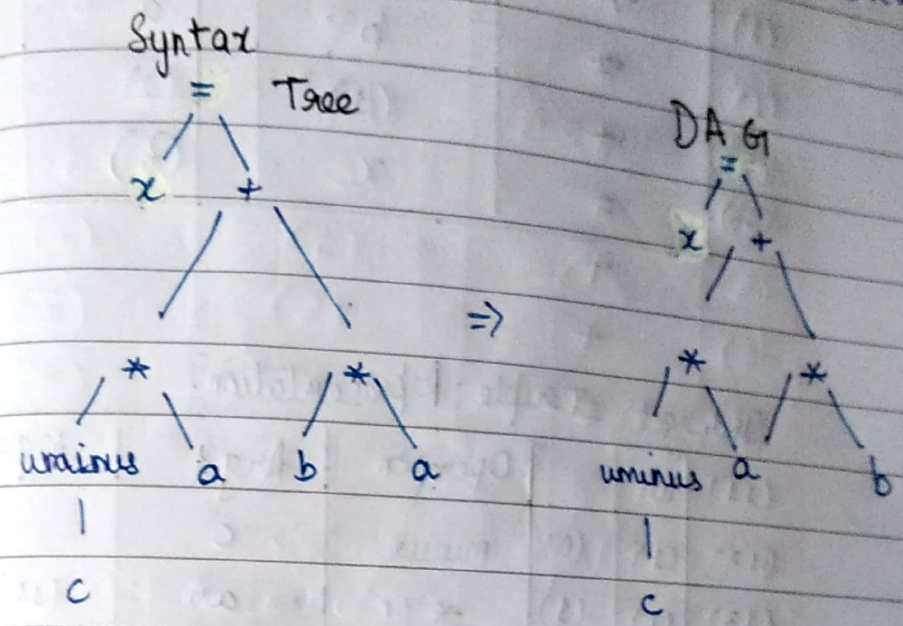
Ptr	Ptr
(11)	(0)
(12)	(1)
(13)	(2)
(14)	(3)
(15)	(4)
	(5)

	Operation	Arg1	Arg2
(0)	minus	c	
(1)	*	b	(11)
(2)	minus	c	
(3)	*	b	(13)
(4)	+	(12)	(14)
(5)	=	a	(15)

P.T.O →

Write the Syntax Tree, Dag Representation and Quadruple Representation, Triple Representation and Indirect Triple Representation using Syntax Tree. x = uminus c * a + b * a

x = uminus c * a + b * a



x = uminus c * a + b * a

Let: t1 = uminus c

t2 = t1 * a

t3 = b * a

t4 = t2 + t3

x = t4

Quadruple Representation:

	Operation	Arg1	Arg2	Result
1	uminus	c		t1
2	*	t1	a	t2
3	*	b	a	t3
4	+	t2	t3	t4
5	=	t4		x

Triple Representation:

	Operation	Arg 1	Arg 2
(0)	uminus	c	
(1)	*	a	(0)
(2)	*	b	a
(3)	+	(1)	(2)
(4)	=	x	(3)

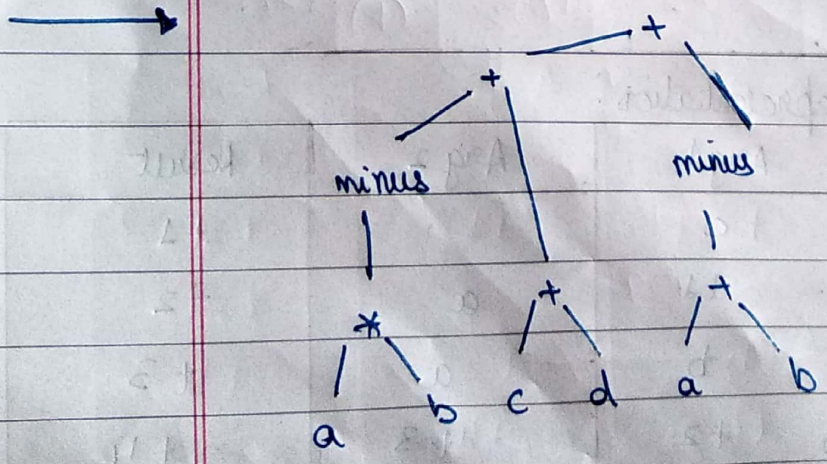
Indirect Triple Representation:

(11)	(0)
(12)	(1)
(13)	(2)
(14)	(3)
(15)	(4)

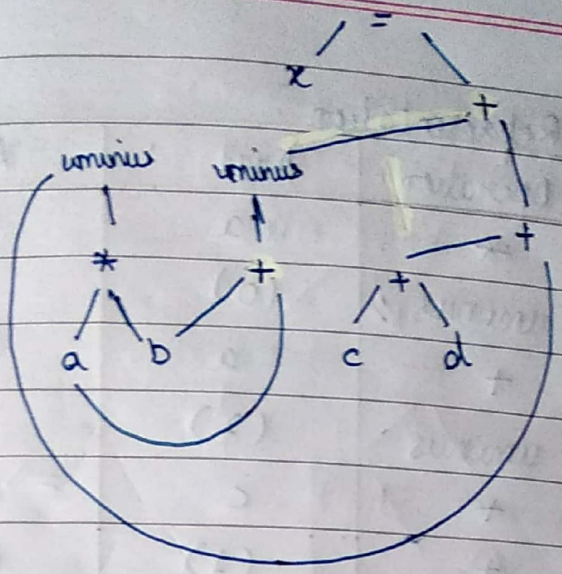
	Operation	Arg 1	Arg 2
(0)	minus	c	
(1)	*	a	(11)
(2)	*	b	a
(3)	+	(12)	(13)
(4)	=	x	(14)

→ Write DAG, S^t, TR, QR, ITR, 3 address code for:

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



=> Syntax Tree



⇒ DAG

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

Let: $t1 = a*b$

$t2 = \text{uminus } t_1$

$t3 = a+b$

$t4 = \text{uminus } t_3$

$t5 = c+d$

$t6 = t2+t3$

$t7 = t6+t4$

$x = t7$

Quadrapole Representation:

	Operation	Arg1	Arg2	Result
1	*	a	b	t1
2	uminus	t1		t2
3	+	a	b	t3
4	uminus	t3		t4
5	+	c	d	t5
6	+	t2	t3	t6
7	+	t6	t4	t7
8	=	t7		x

Triple Representation:

	Operation	Arg1	Arg2
(0)	*	a	b
(1)	uminus	(0)	
(2)	+	a	b
(3)	uminus	(2)	
(4)	+	c	d
(5)	+	(1)	(2)
(6)	+	(5)	(3)
(7)	=	x	(7)

Indirect Triple Representation:

			Operation	Arg1	Arg2
(11)	(0)	(0)	*	a	b
(12)	(1)	(1)	uminus	(11)	
(13)	(2)	(2)	+	a	b
(14)	(3)	(3)	uminus	(13)	
(15)	(4)	(4)	+	c	d
(16)	(5)	(5)	+	(12)	(13)
(17)	(6)	(6)	+	(16)	(14)
(18)	(7)	(7)	=	x	(18)

P.T.O.

Grammar for based numbers

S (head symbol)

based_num → num basechar

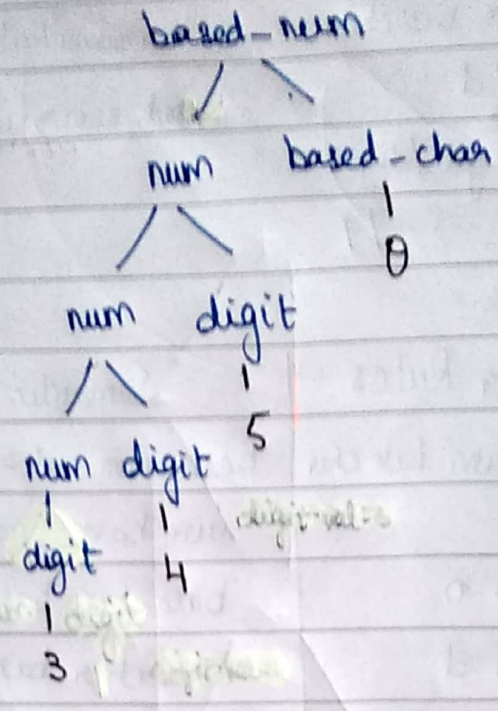
base char → o / d

num → num digit / digit

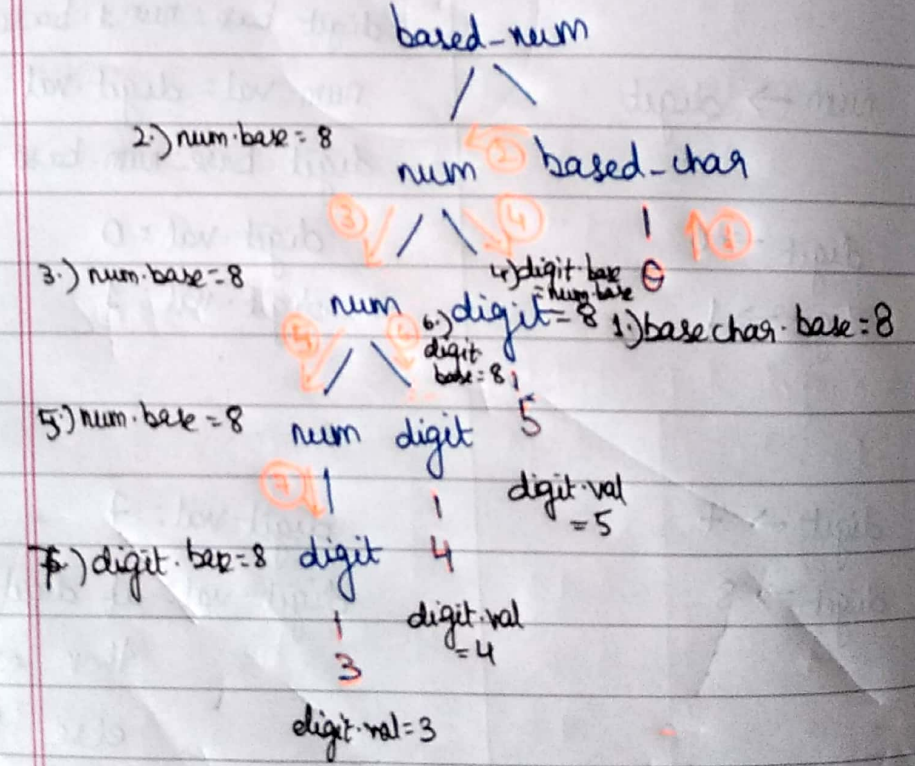
digit → 0 / 1 / 2 / ..... / 9

Grammar Rules	Semantic Rules
based_num → num base char	base_num.val = num.val
	num.base = basechar.base
base char → o	basechar.base = 8
base char → d	basechar.base = 10
num1 → num2 digit	num2.val * num1.base + digit.val
	num.base = num1.base
	digit.base = num1.base
num → digit	num.val = digit.val
	digit.base = num.base
digit → 0	digit.val = 0
digit → 1	digit.val = 1
⋮	⋮
digit → 7	digit.val = 7
digit → 8	digit.val = if digit.base = 8 then error else 8
digit → 9	digit.val = if digit.base = 8 then error else 9

eg. 345 O ⇒ (345)$_8$

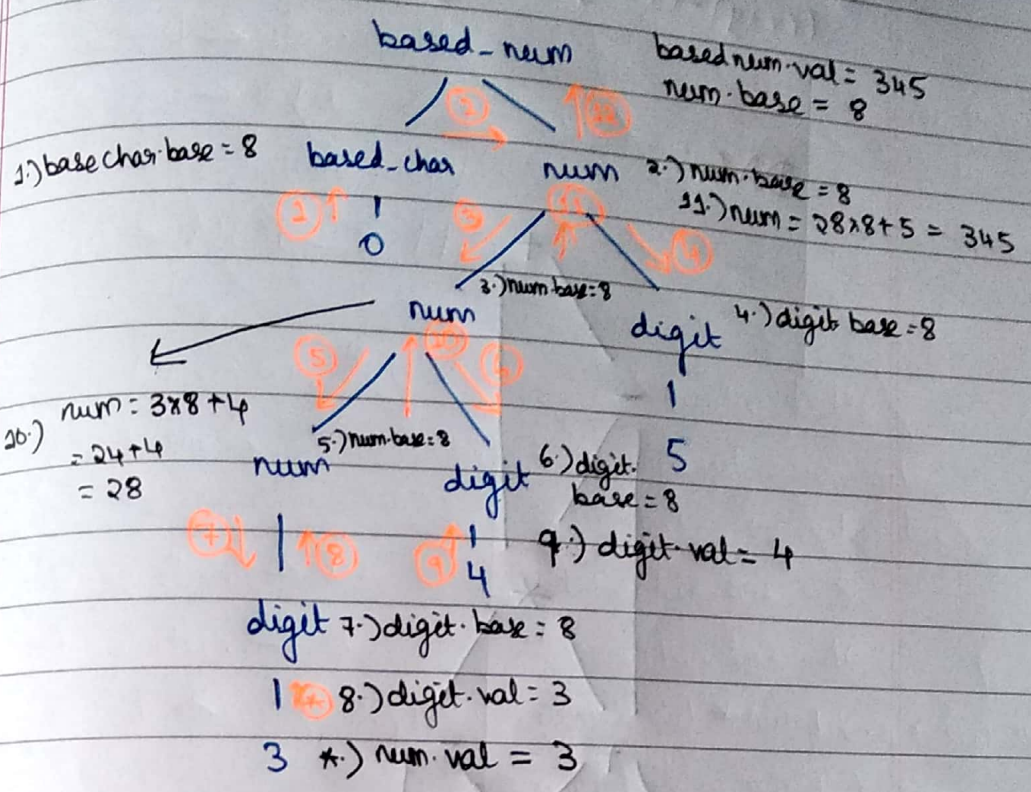


⇒ Syntax tree

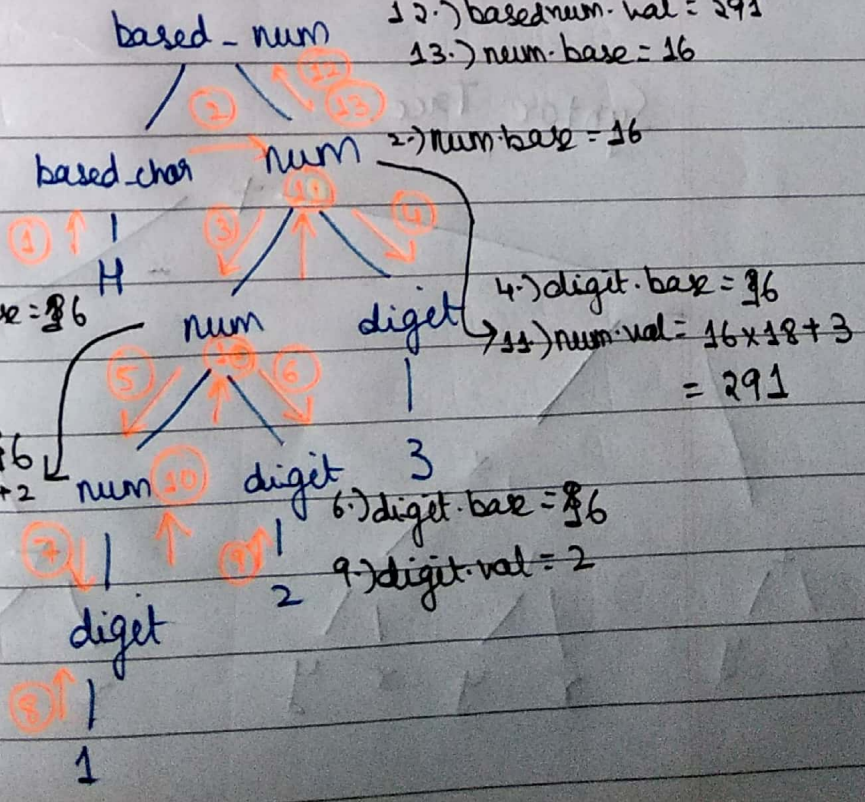


* Value attribute is synthesised attribute

based_num → basedchar num



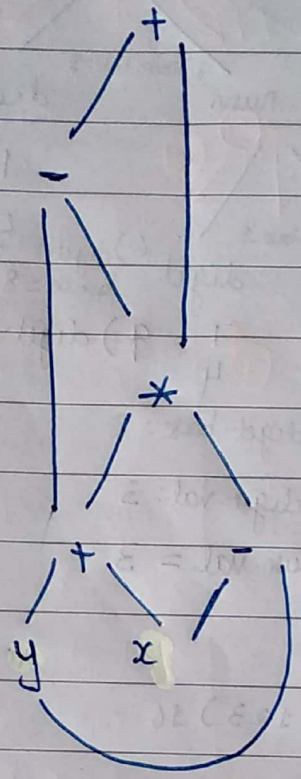
Eg. H 123 => (123)16



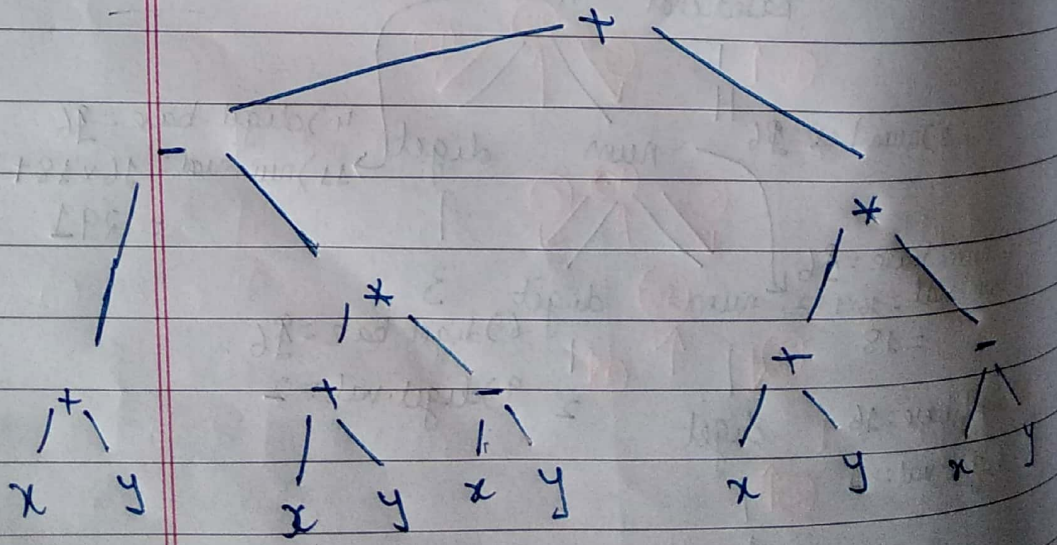
Eg. Construct DAG for:

$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$

DAG:

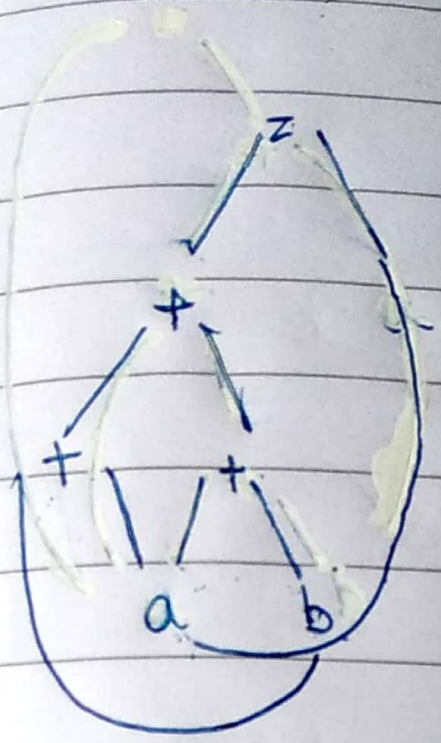


Syntax Tree:

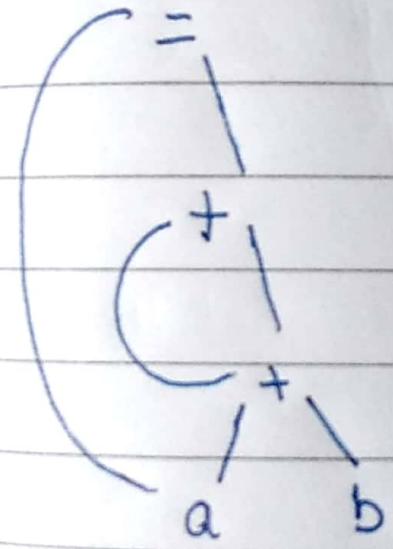


Eg. Value Number Method for: a = a+b+(a+b)

a = a+b+(a+b)



1	a		
2	b		
3	+	1	2
4	+	1	3
5	+	4	5
6	=	1	5



1	a		
2	b		
3	+	1	2
4	+	1	2
5	+	3	4
6	=	1	5