31/8/18

Semantic Analysis

- attributes are attached to the grammar symbols.
- CFG + semantic rules = syntax directed definition

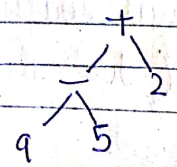
→ Semantic phase performs following tasks :-

1. Flow-of-control check.
2. Type check.
3. Scope resolution.
4. Array out of bound check.

* Symbol table is constructed using the analysis phase of the compiler (ie, first 3 phase of compiler).

Abstract Syntax Tree (AST)

9-5+2



Lexical errors :- If you give input but for that regular expression doesn't exist.

→ It hides the implementation part & just gives the overview struct.

Semantic errors :-

1. Type mismatch.
2. Accessing an out of scope variable
3. Formal & Actual parameter mismatch
4. Undeclared variable.
5. Multiple declaration of a variable in a scope.

Synthesized Attributes :

S → ABC

→ The node can take value only by its children, or node.

attribute/

Inherited attributes :

→ node can take the value from parent, its siblings and the node itself.

S → ABCS.

Eg; $E \rightarrow E+T$

We write attribute as :-

$E.val = E_1.val + T.val$.

* Here .val → represent the attribute attached with that node.

Here we have written E_1 just to differentiate that E can take value from someone else (which is E itself).

S-attributed SDT.

① uses only synthesized attributes.
② bottom-up parsing.
③ semantic rules are present at the end.

↓

E.val = E.val || T.val || +

↓

Semantic rules written at the end.

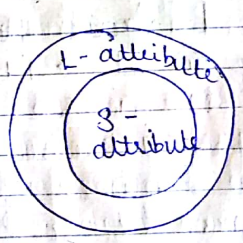
L-attributed SDT.

① uses synthesized attributes.
② uses inherited attributes but with the restriction that node cannot take value from its right sibling.
③ Depth-first order and left-to-right parsing

$S \rightarrow ABC$

A can only take value from S
B can only take value from S & A
C " " " " S, A and B

→ S can take value from ABC



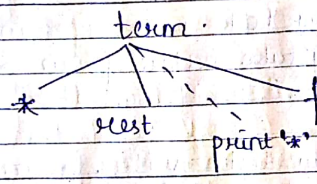
SDD → Syntax directed definition.

SDT → " " translation.

In SDT, semantic rules can come in b/w but in SDD it always come in last.

→ In SDT we have to attach a different node for semantic rule as follow:-

term → * rest { print('*') } factor



→* Till now we have been writing as E → T+E but now in SDD and SDT we are writing attributes along with the flow of information in the parse tree.

Type checking

E → E1 + E2 { if (E1.type == E2.type) && (E1.type = int) then

E.type = int
else
error (" ");

E → (E1) { E.type = E1.type }

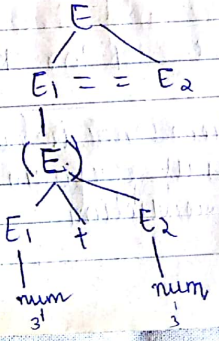
E → E1 == E2 { if (E1.type == E2.type) && (E1.type == int | bool) then E.type = boolean else error (" "); }

E → num { E.type = num.type }

E → true { E.type = boolean }

E → false { E.type = boolean }.

Eg:- (3+2) == 6



(Something like this will be drawn, not exactly. → Info. will flow from bottom to up

3/8/18 Tutorial

Ambiguous Grammar.

$E \to E+E \mid E*E \mid a$

(Operator grammar)

Resolve Ambiguity.

There will come a state,

$E \to E+E\bullet$
$E \to E\bullet+E$
$E \to E\bullet*E$

In Normal, there will be S/R conflict for S/R.

but for first Prod → Reduce
2nd Prod → Shift. — R/S conflict

As + is left associative so we will perform reduce.

Similarly there is a S/R for *.

As * precedence is higher. So shift will perform first and then reduction.

Similary

$E \to E*E\bullet$
$E \to E\bullet+E$
$E \to E\bullet*E$

Here also for + we have a S/R conflict As * precedence is higher than +, we will perform reduce.

For * there is also a conflict, and * left associative so we will perform reduce first.

→ So generally for S/R we have S/R conflict but if we associate operator operations then this can be resolved.

→ When we cannot tell from the grammar its associativity, so we will use assumption (BODMAS).

Q $S \to iSeS \mid iS \mid a$.

$S \to iS\bullet eS$
$S \to iS\bullet$

Here also we have a S/R conflict.

* We generally give shift more importance than reduce, so we will do shift here.

→ This is not an operator grammar, ie, dangling if statement so here shift is that higher precedence than reduce.

∴ Because our objective is to take the higher length input so we perform shift operation.

* There are different methods to cope up errors

① panic error → as soon as it come leave everything

② Statement error → If at a particular input operator error comes then we leave some input character till the error is gone.

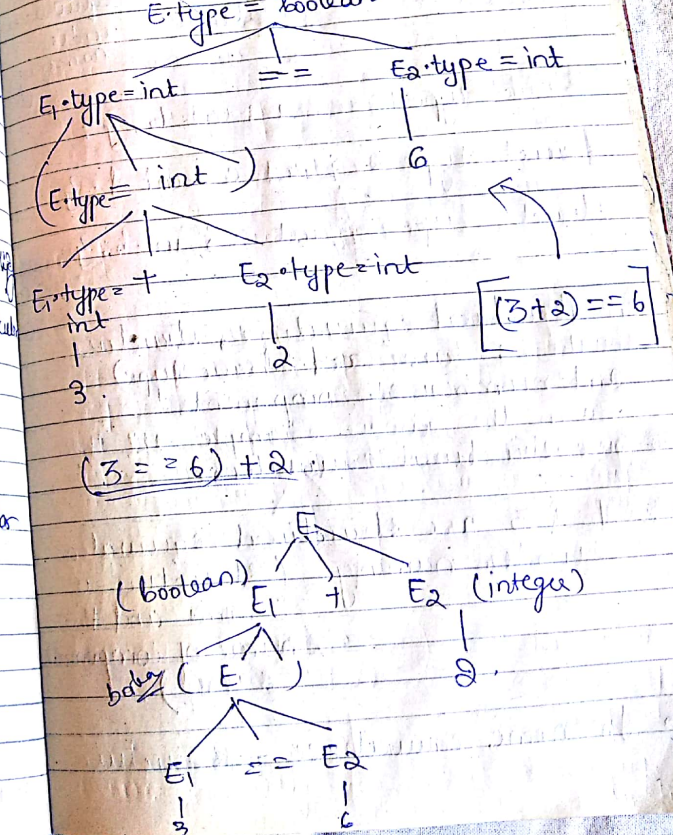
③ Pop every element from the stack and then start from that particular I/P character where there is no error.

We will study this part later (Not for exam)

4/8/18

Annotated Parse Tree

A parse tree showing attribute value at each node



Type Equivalence :

Structural equivalence | Name equivalence | Structural equivalence under naming

```
struct type 1 { int a ; int b; };
struct type 2 { int c ; int d; };
struct type 3 { int a ; int b; };
```

→ In structural equivalence, structure should be same (basic type) and number of components.
So in the above example all three are structural equivalence.

→ 1 & 3 are structural equivalence under naming (structure is same and the name of components same).

→ In name equivalence → structure can be different but name should be same.

Eg:- struct type 1 {int a ; int b};
array type 1 — name same.

Type Conversion :

Type coercion ← Implicit | Explicit

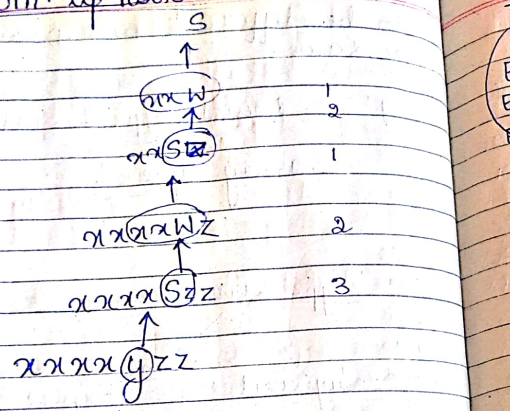
Implicit (Eg:- upgrading int to float and then perform operations).
(changing char to int, there will be no loss of information).

Explicit Eg:- changing int to char (there will be loss of information).

Eg:-
S → x x W { print ('1') }
S → y { print ('3') }
W → S z { print ('2') }.

I/P = x x x x y z z .

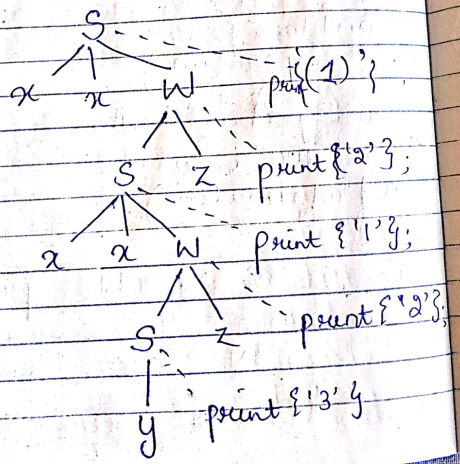
Bottom-Up Parser



O/P-

32121

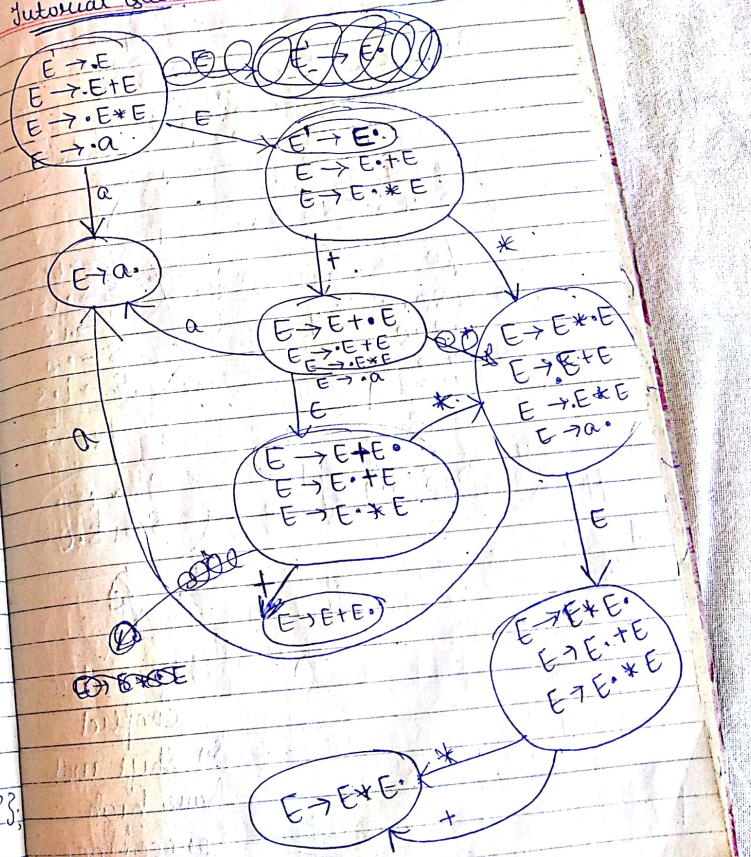
Top-Down

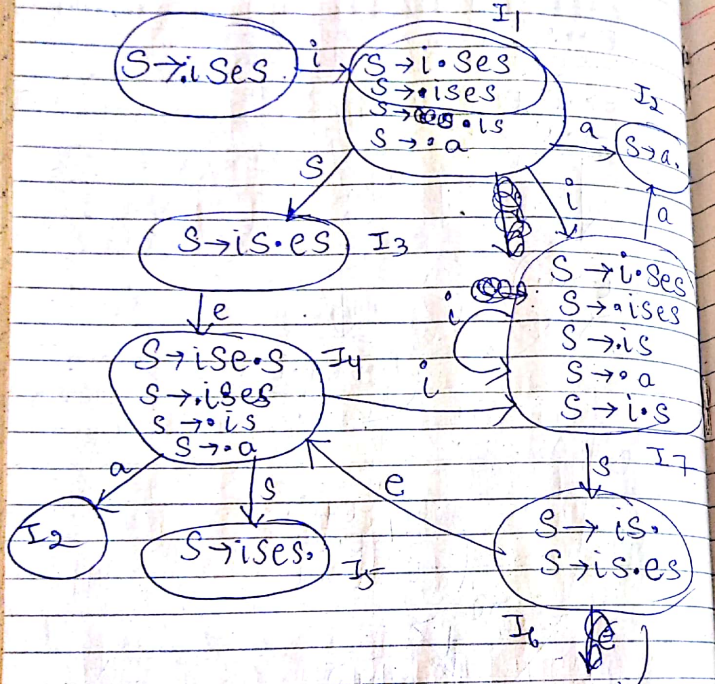


O/P

32121

Tutorial Ques.



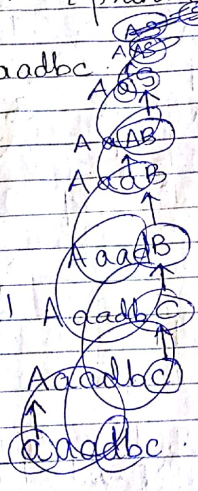


6/9/18

Q
S → AS { print ("1") }
S → AB { print ("2") }
A → a { print ("3") }
B → bC { print ("4") }
B → dB { print ("5") }
C → c { print ("6") }

I/P aaadbc

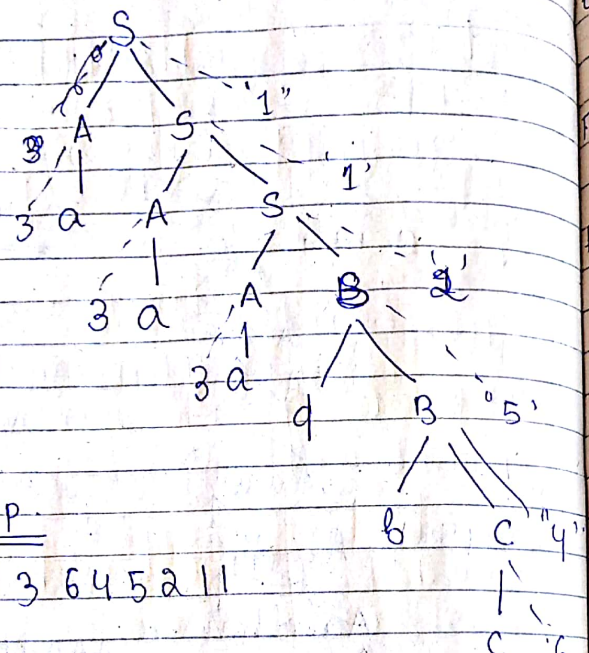
O/P = 33364521



Bottom-Up

S
A S 1
A AS 1
A AB 2
A aB 5
A aaB 4
A aadbC 6
A aaadbC 3
a aadbc 3
aaadbc 3

Top down:



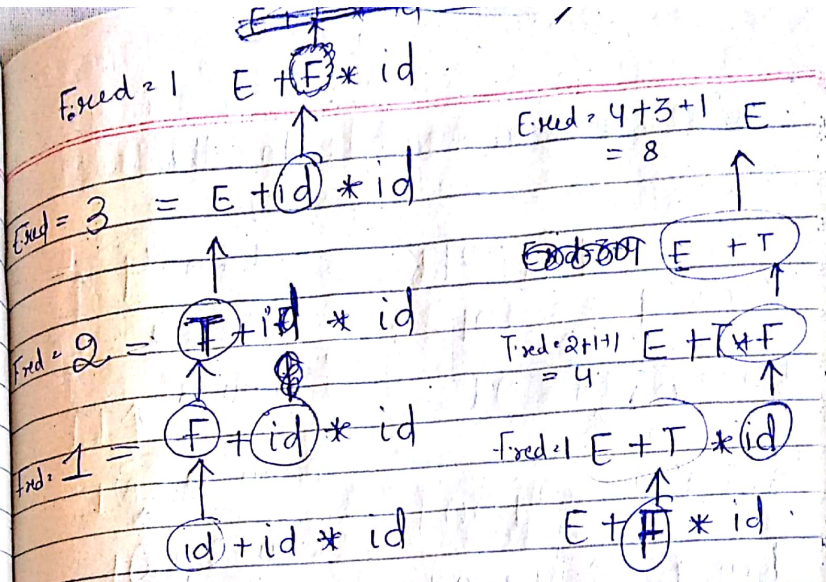
O/P:

3 3 3 6 4 5 2 1 1

Q

$E \to E+T$	$\{E.red = E.red + T.red+1\}$
$E \to T$	$\{E.red = T.red+1\}$
$T \to T*F$	$\{T.red = T.red+F.red+1\}$
$T \to F$	$\{T.red = F.red+1\}$
$F \to id$	$\{F.red = 1\}$

Q Calculate the no. of reductions used in the parsing of "id+id*id".



We have used 8 reductions.

Now we will perform reduction using semantic rules. → we are getting 8 reductions.

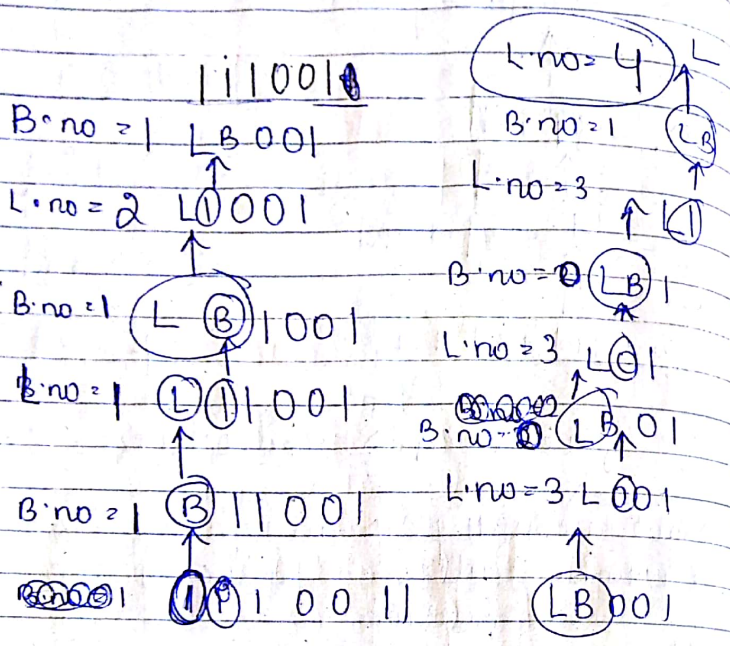
∴ This is S-attributed SDT.

Q Draw SDT to count the no. of 1's in the grammar:-

$S \to L$	$\{B.no = L.no\}$
$L \to LB$	$\{L.no = L.no + B.no\}$
$L \to B$	$\{B.no = B.no + 1\}$
$B \to 0$	$\{B.no = 0\}$
$B \to 1$	$\{B.no = 1\}$

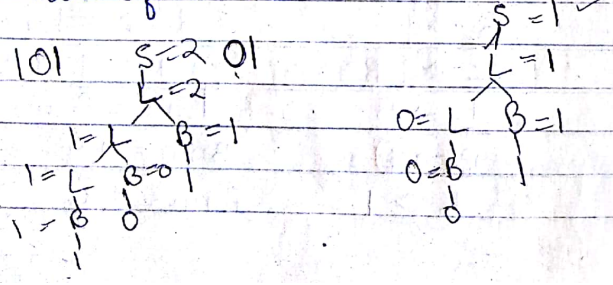
b → B

Also give an example for the same.



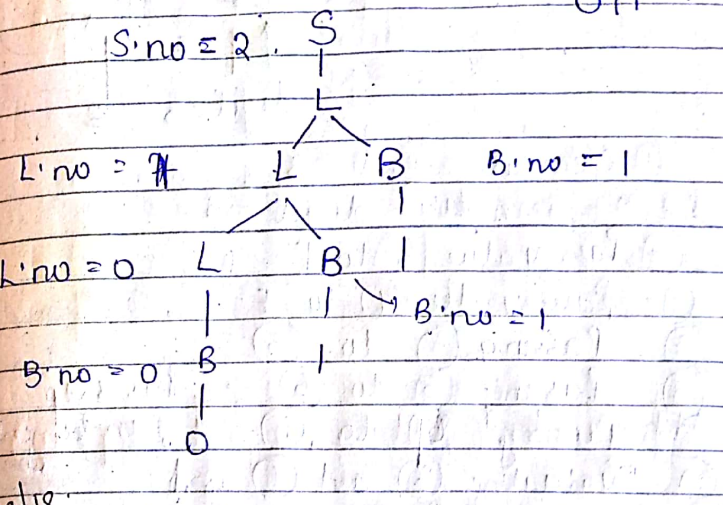
Total number of 1's = 4.

According to annoted parse tree no of 1's = 4.



Annoted Parse tree can be bottom-up & top-down. But preferably draw top-down.

011



18/9/18

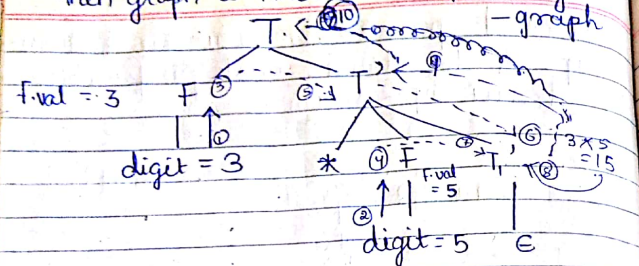
L-attributed

Inherited | synthesized

T → F T'	→	{ T'.inh = F.val ; T.val = T'.syn }
T' → * F T1'	→	{ T1'.inh = T'.inh × F.val ; T'.syn = T1'.syn }
T' → ∈	→	{ T'.syn = T'.inh }
F → digit	→	{ F.val = digit.lexval }

Annoted Parse tree for 3 * 5.

* Whenever we show flow of information then graph is knowns as dependency graph



① Generate digit = 3
② Generate digit = 5
③ Pass value ② to F
④ Pass value ① to F
⑤ Passing ③ to ⑤
⑥ Passing ⑤ to ⑥ } This step can be interchanged
⑦ Passing ④ to ⑦
⑧ Evaluating ⑥ and ⑦ and Pass the resultant to T_1 ⇒ ⑧
⑨
⑩ Pass ⑧ to T' ⇒ ⑩
⑪ Pass ⑨ to T

* Tail Recursion.

```
void fun()
{
    a = a * b;
    fun();
}
```

{ No other statements beyond fun() (recursive) }

24/9/18

Postfix SDT. (An SDT in which all the semantic action are written after the production body)

Eg: $E \rightarrow T$ { }
$T \rightarrow F * T_1$ { }

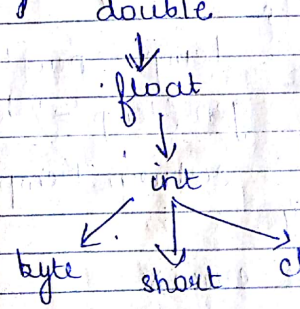
$E \rightarrow$ { print('*') } $E * T$ (Preorder traversal)

$E \rightarrow E +$ { print('+') } T (In-order traversal)

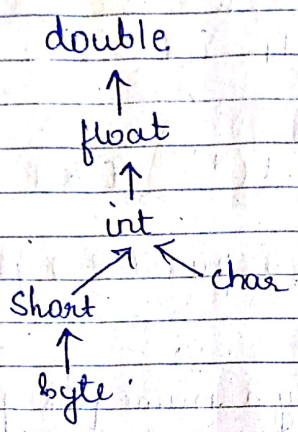
* there is only a name for postfix SDT, rest other are just SDT (Preorder or In order).

Narrowing Conversion. (Not imp).

Widening



Widening conversion:




```
Addr widen (Addr a, Type t, Type w)
  {
    if (t=w) return a;
    else if ((t=integer) && (w=float))
      {
        temp = new temp();
        gen (temp '=' '(float)' a);
        return temp;
      }
    else error
  }
```

$E \rightarrow E_1 + E_2$ { E.type = max(E1.type, E2.type);

a_1 = widen(E1.addr, E1.type, E.type);

a_2 = widen(E2.addr, E2.type, E.type);

E.addr = new temp();

gen (E.addr '=' a_1 '+' a_2);

}

Intermediate Code:

DAG (directed acyclic graph)

Three-address code.

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

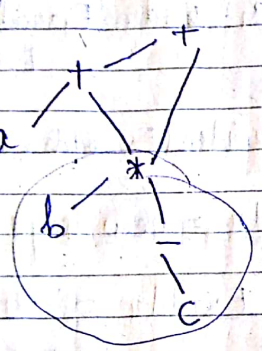
In DAG b * - c is once shown and whenever used later it will be directed to the same node.

But in 3-address code, b * - c is,

shown, the number of the times it will occur.

For 3 addcode

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = a + t_2$
$t_4 = -c$
$t_5 = b * t_4$
$t_6 = t_3 + t_5$



(DAG)

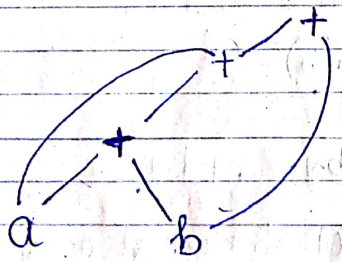
Common expression denoted once only.

Eg:- ① $a+b+a+b$.

② $a+b+(a+b)$.

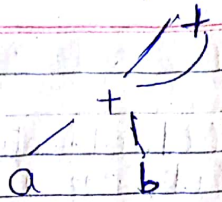
Draw DAG.

①



Assignment → 3rd Oct → Symbol Table Implementation
Data structure to implement symbol Table (Adv. & Disadv).

②



25/9/18

Address.

name (pointer) — temp. generated variables by compilers — constant.

extra knowledge

x = 5 ; → r-value

↓

l-value.

l-value = location.

r-value = denotes the answer.

x = &y → we are storing location of y in the l-value of x.

Representation of DAG. (Read it from book)

- Array of records
- Hash table

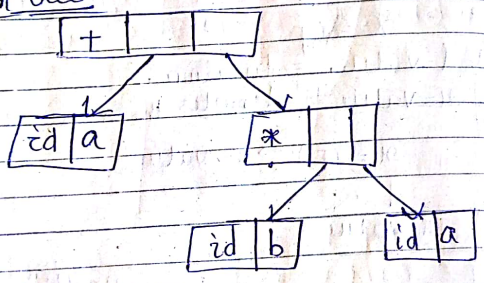
a + b * c

p_1 = newleaf (id, id.entry) (for a)
p_2 = " (id, ") (for b)
p_3 = " (" , ") (for c)

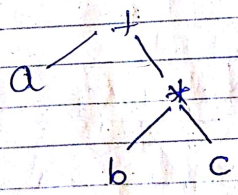
p_4 = newNode (*, p_2, p_3)

p_5 = " (+, p_1, p_4)

Syntax tree



DAG



Array of records for storing DAG

0	id	b
1	id	c
2	*	0 1
3	id	a
4	+	3 2

3- Address Code

Quadruples Triples Indirect triples.

3-addres code by a+b*c

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

Quadruples

	op	arg_1	arg_2	t arg_3
0	*	b	c	t_1
1	+	a	t_1	t_2

Triples:

Advantage:- (Occupy less space than Quad
Drawback:- If we change the address of of instruction, we have to change the address everywhere).

	op	arg1	arg2
0	*	b	c
1	+	a	0

Indirect triples

(index)
We are storing the address of instruction at somewhere else address.

Counter

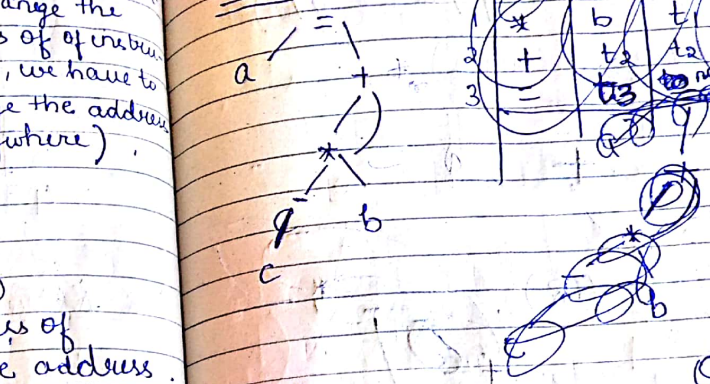
5	0
6	1

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

3 address codes

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

DAG



Quadruples

	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

Triples

add.	op	arg1	arg2
0	-	c	-
1	*	b	0
2	+	1	1
3	=	a	2

Triples

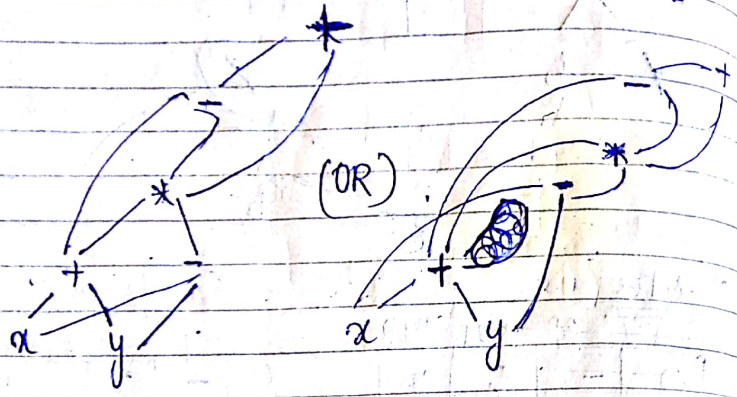
0	-	c	-
1	*	b	0
2	-	c	
3	*	b	2
4	+	1	3
5	=	a	4

Indirect triples

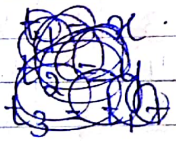
Index	
0	7
1	8
2	9
3	10
4	11
5	12

HW

Q ((x+y)-((x+y)*(x-y)))+
((x+y)*(x-y))



3-address code:



$t_1 = x + y;$
$t_2 = x + y;$
$t_3 = x - y;$
$t_4 = t_2 * t_3$
$t_5 = t_1 - t_4$
$t_6 = x + y$
$t_7 = x - y$
$t_8 = t_6 * t_7$
$t_9 = t_5 + t_8$

27/9/18

Tutorial

SDT for 3-address code for Assignment statement:

$S \rightarrow id = E$ { p = lookup(id.entry); If (p != nil) then emit(p '=' E.place); else error; }

$E \rightarrow E_1 + E_2$ { E.place = newtemp(); emit(E.place '=' E_1.place '+' E_2.place); }

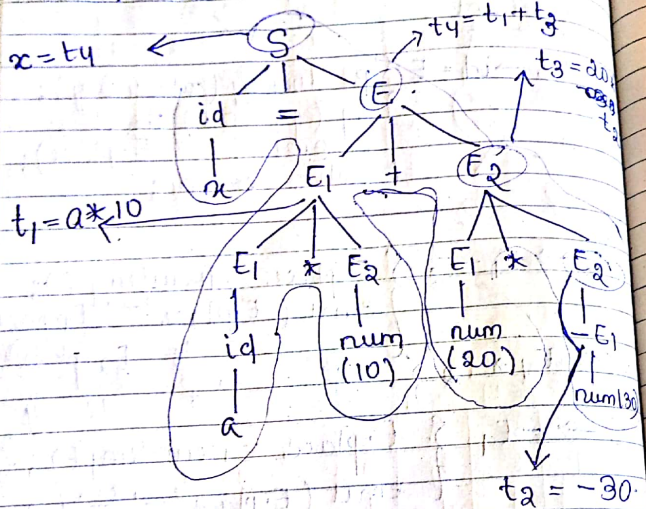
$E \rightarrow E_1 * E_2$ { " " '*' " " }

$E \rightarrow -E_1$ { E.place = new-temp(); emit(E.place '=' '-' E_1.place; }

$E \rightarrow id$ { p = lookup(id.entry); if (p != nil) then E.place = p; else error; }

$E \rightarrow num$ { E.place = num.place; }

$x = a * 10 + 20 * -30$.



emit → perform the result & store the result.

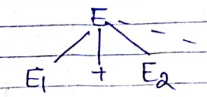
lookup → looking up variable in the symbol table.

new temp → generating new temp. variable.

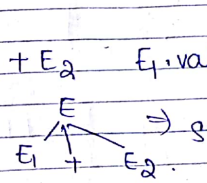
1. Translation of array references.
2. Short circuit code (CSE 4 year).

SDD ⇒ $E \rightarrow E_1 + E_2$ { print '+' }

SDT ⇒



$E \rightarrow E_1 + E_2$ $E_1.val$ || $E_2.val$ || + . (|| → concatenation operator)



The only difference is we attach a temp. leaf node in SDT.

Short circuit code (we represent every (if else & relational op) as labels).

If (a < b || (c < d && e < f)).

E0 : If a < b goto E4 (or directly we can write emit "true") else goto E1.

E1 : If c < d goto E2. emit "false".

E2 : If e < f goto E4 emit "false".

E4 : emit "True".