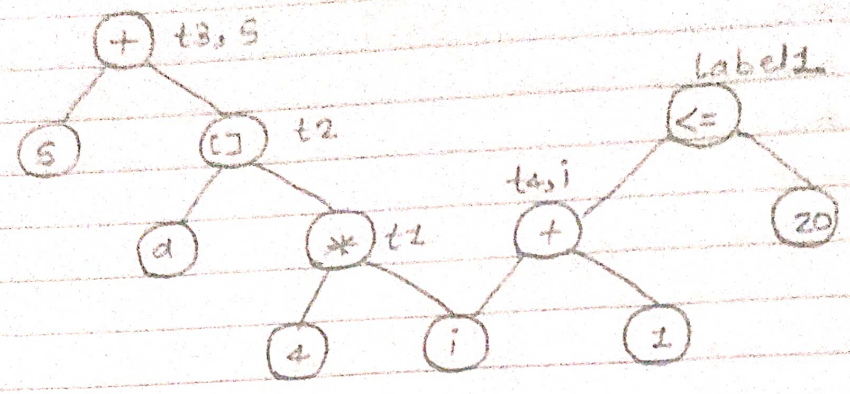
DAG =



Q.12] Generate DAG representation of following code & list out the application of DAG.

```
I = 1 ;
while ( I<=10)
Do
Sum ! = a[i]
```

→

Applications =>

1) DAG (directed acyclic graph) is a useful data structure for implementing transformations on basic blocks.

2) DAG is used in -

i) Determining the common subexpression.

ii) Determining which names are used inside the block & computed outside the block.

iii) Determining which statements of the block could have their computed value outside the block.
iv) Simplifying the list of quadraples by eliminating the common subexpression.

Q.9) Explain register allocation and assignment with the suitable example?

→

- Efficient utilization of limited set of registers is important to generate good code.
- Registers are assigned by -

1) Register allocation
2) Register assignment

- Register allocation is to select the set of variables that will reside in registers at a point in the code.
- Register assignment is to pick the specific register that a variable will reside in.
- This approach has the advantage that it simplifies the design of a compiler.
- Disadvantage is that, applied too strictly, it uses registers inefficiently.
- There are various strategies used in register and assignment that are -

1) Global Register Allocation
2) Usage Counts
3) Register Assignment for outer loops
4) Register Allocation by Graph coloring.

Example =

```
t := a * b
t := t + a
t := t / d
```

⇓ {R1 = t}

```
MOV a, R1
MUL b, R1
ADD a, R1
DIV d, R1
MOV R1, t
```

```
t := a * b
t := t + a
t := t / d
```

⇓ {R0 = a, R1 = t}

```
MOV a, R0
MOV R0, R1
MUL b, R1
ADD R0, R1
DIV d, R1
MOV R1, t
```

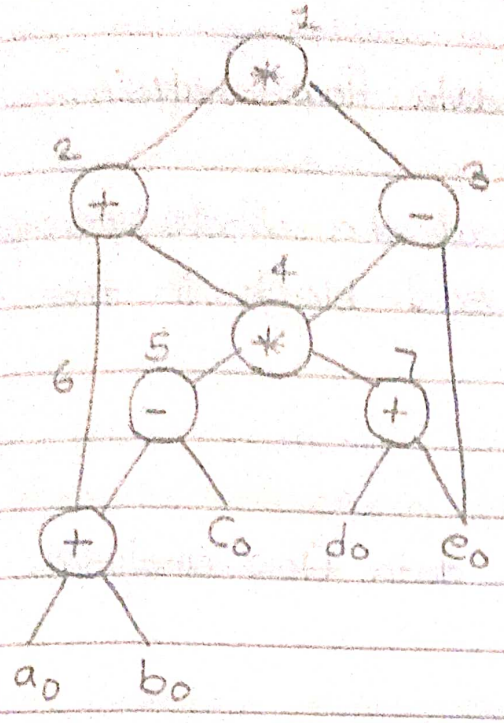
Q.10] Explain node listing and labeling algorithm with example.

→

Node Listing Algorithm =>

```
While unlisted interior nodes remain do begin
    select an unlisted node n, all of whose
    parents have been listed;
    list n;
    while the leftmost child m of n has no unlisted
    parents and is not a leaf do begin
        list m;
        n := m;
    end
end
```

Exp :-

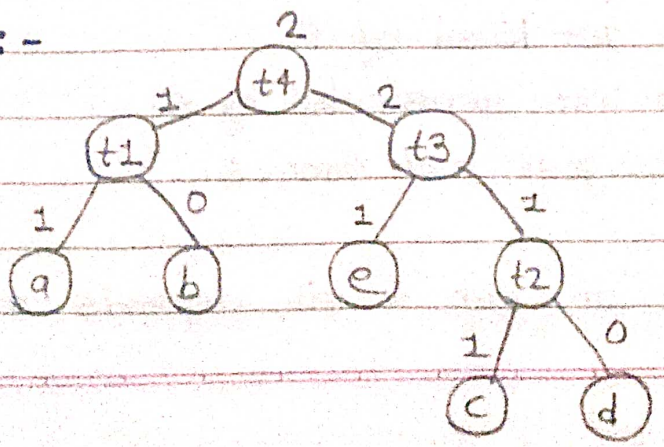


$t7 := d + e$
$t6 := a + b$
$t5 := t6 - c$
$t4 := t5 - t7$
$t3 := t4 - e$
$t2 := t6 + t4$
$t1 := t2 * t3$

The labeling algorithm =>

```
if n is a leaf then
    if n is the leftmost child of it's parent then
        label(n) := 1
    else
        label(n) := 0
else begin
    let n1, n2, --- nk be the children of n ordered by
        label so that label(n1) >= label(n2) >= --- label(nk);
    label(n) := max 1≤i≤k (label(ni) + i - 1)
end
```

Exp:-



Q. 3] Explain global data flow analysis using data flow equa...

OR

Q. 14] Discuss in detail about global data flow analysis.

→

- Collect information about the whole program & distribute the information to each block in the flow graph.

Data flow information :-

- Information collected by data flow analysis.

Data flow equations :-

- A set of equations solved by DFA to gather data flow information.

- Component of data flow equations :-

Sets containing collected information =

1) in set = info. coming into BB from outside
2) gen set = info collected within BB
3) kill set = info collected within BB, will affect.
4) out set = info leaving the BB.

- A typical data flow equation :

out [S] = gen [S] ∪ (in [S] - kill [S])

where,

S : statement
in [S] : info goes into S
kill [S] : info get killed by S
gen [S] : New info. generated by S
out [S] : Info. goes out from S.

- The notation of gen & kill depends on the desired information.

- In some cases, in may be defined in terms of out - equation is solved as analysis traverses in the backward direction.

- Data flow analysis follows control flow graph. Equations are set at the level of basic blocks, or even for a statement.

Q.8] Explain structure preserving transformation techniques with example.

→

a) Common subexpression elimination ⇒

```
a := b+c                a := b+c
b := a-d        ⟹      b := a-d
c := b+c                c := b+c
d := a-d                d := b
```

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

b) Dead - code elimination ⇒

Suppose x is dead, that is never subequenthy used, at the point where the statement x := y+z appears in basic block.

```
i = 0;
if (i == 1)        ⟹    i = 0;
  a = x+5;
```

c) Renaming temporary variables ⇒

A statement $t := b+c$ can be changed to $u := b+c$ and all uses of this instance of t can be changed to u without changing the value of basic block. Such a block is called a normal-form block.

d) Interchange of statements ⇒

- Suppose a block has the following two adjacent:

Suppose,

Statements : $t1 := b+c$

$t2 := x+y$

- We can interchange the two statements without affecting the value of the block if & only if neither x nor y is t and neither b nor c is t.

Q.13] Explain principles sources of optimization.

→

1) Common Sub expression elimination
2) Copy propagation
3) Dead-Code elimination
4) Constant folding.

Refer Tut 6, Q.1]

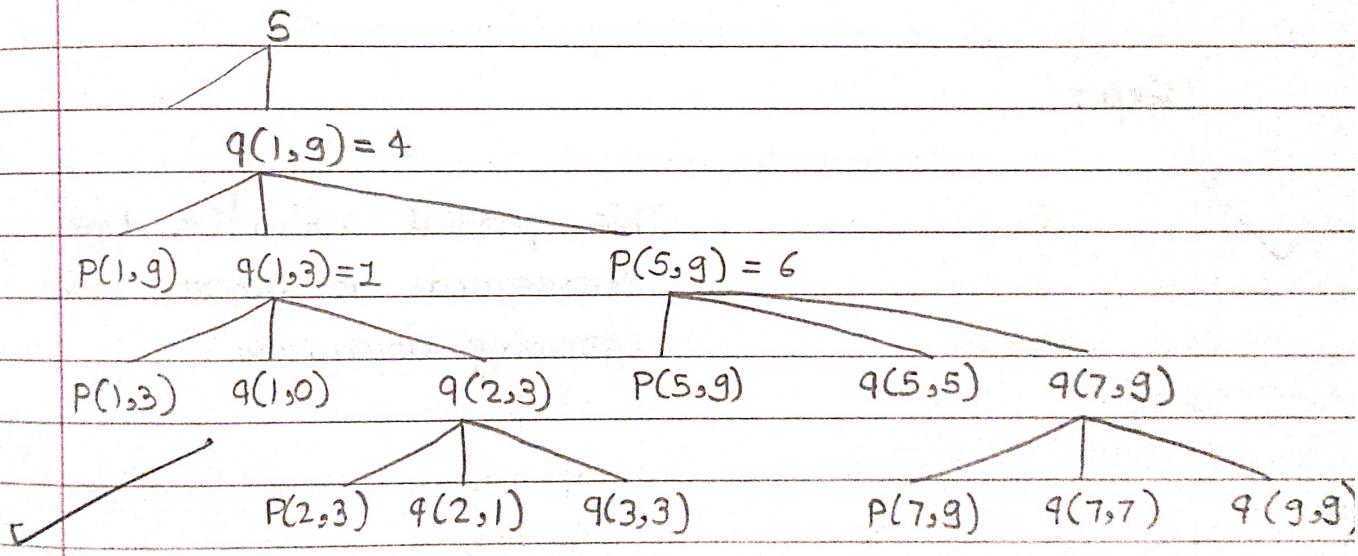
Tutorial No. 8

Q.1] Explain activation tree and control stack with example?

→

- Every execution of a procedure is called an activation.
- The lifetime of an activation of procedure P is the sequence of steps between the first and last step of P's body, including any procedures called while P is running.
- If the procedure is recursive, several of it's activation may be alive at the same time.
- Therefore, each execution of procedure body is referes to as activation of procedure.

Exp:



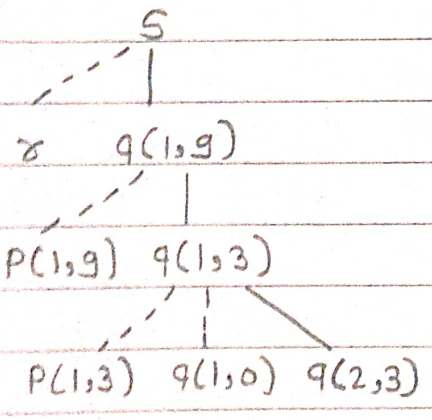
Activation tree

- Each node represents the activation of a procedure.
- Each root represents activation of a main program.
- Any node from A to B, if and only if control flows from A to B.

- If node A is parent node, if and only if control flows from A to B.
- If A is left child node of B, if lifetime of A occurs before lifetime of B.

Control Stacks =
- We can use a stack to keep track of currently-active activations.
- We push a record onto the stack when a procedure is called and pop that record off the stack when the procedure returns.
- At any point in time, the control stack represents a path from the root of activation tree to one of the nodes.

Exp:



This partial activation tree corresponds to control stack (growing downward)

s
q(1,9)
q(1,3)
q(2,3)

Q.2) Explain stack allocation with example?

→

- Stack keep track of procedure activations
- Manage runtime allocation with a stack storage.
- Local data are allocated on the stack.

- stack allocation is a procedure in which stack is used to organize the storage.

- The stack used in stack allocation is known as control stack.

- In this type of allocation, creation of data objects is performed dynamically.

- In stack allocation, activation records are created for the allocation of memory.

- These activation records are pushed onto the stack using Last in first out (LIFO) method.

- Locals are stored in activation records at run time and memory addressing is done by using pointers and registers.

- In this the allocation / deallocation is automatic

- It is less expensive & space for allocation is limited.

- It cannot support recursion.

Q.3] Explain what is activation record? Explain the contents of activation record?

→

- Any information needed for a single activation of a procedure is stored in ACTIVATION RECORD, sometimes called the STACK FRAME.
- A pointer to the current activation record is ~~mained~~ maintained in a register.

Actual Parameters
Returned value
Control link
Access link
Saved machine status
Local data
Temporaries

1) Temporaries =
evaluation of expression

2) Local data =
field for local data

3) Saved machine status =
Information about the machine state before the procedure call.
- Return address (value of program counter)
- Register contents

4) Access link =
access nonlocal data

5) Control link =

Points to the activation record of the caller

6) Returned values =

Space for the value to be returned

7) Actual parameters =

Space for actual parameters.

Q.4] Explain dynamic storage allocation strategies ?

→

Dynamic storage allocation strategies =>

1) Stack allocation

2) Heap allocation

1) Stack allocation =

- The stack allocation is a runtime storage management technique. The activation records are pushed and popped as activations begin and end respectively.

- Storage for the locals in each call of the procedure is contained in the activation record for that call.

- Thus, locals are bound to fresh storage in each activation, because a new activation record is pushed onto stack when the call is made.

- It can be determined the size of the variables at a run time & hence local variables can have different storage locations & different values during various activations.

2) Heap Storage Allocation =

- It enables the allocation of memory in a Non-nested design. Storage can be allocated & freed arbitrarily from an area known as heap.
- Heap allocation is helpful for executing data whose size varies as the program is running.
- Heap is maintained as a list of free space called free space list.

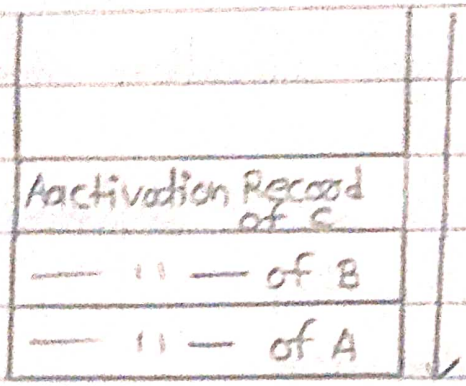
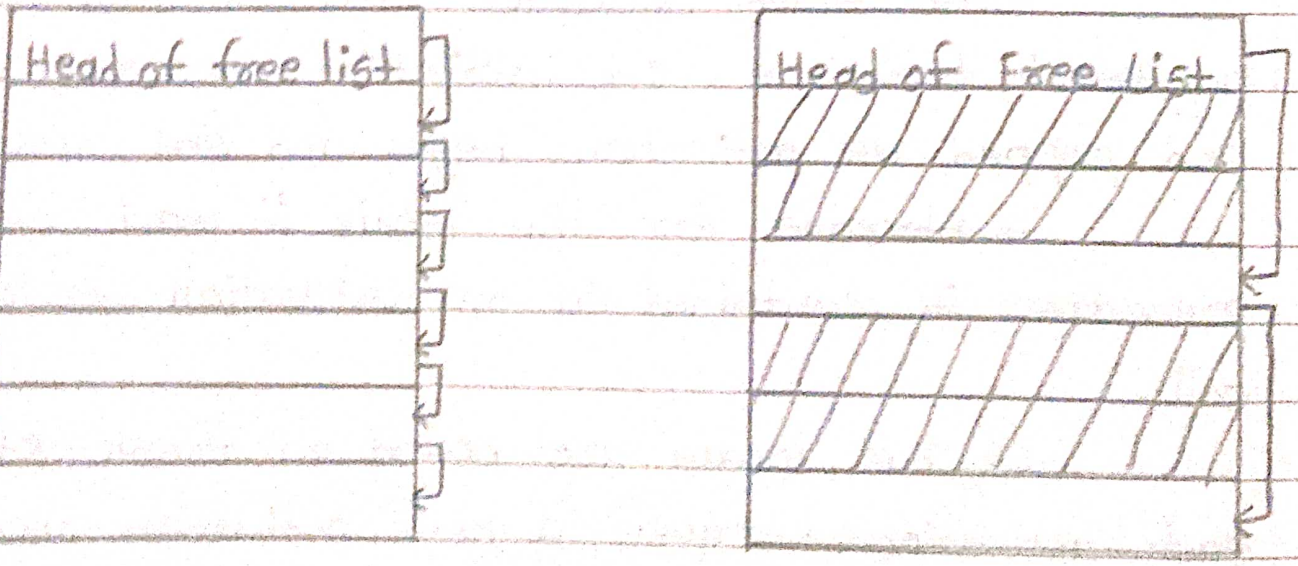


fig. Stack Allocation



Initial Free Space list Free Space After Execution

fig. Heap Storage Allocation

Q.5] Explain following parameter passing methods:

a) Call by Value =

- The actual parameters are evaluated & their r-values are passed to called procedure.
- The formal parameters do not changes the operation on value of actual parameter.
- The arguments are evaluated at the time of call and the value parameters are copied and either behave as -

a) behave as constant values during executing of procedure.

b) are viewed as initialized local variables.

Exp:

```
CallByValue (int y)
{
    y = y+1;
    print (y);
}

main ()
{
    int x = 42;
    print (x);
    callByValue (x);
    print (x);
}
```

Output:

```
x = 42
y = 43
x = 42
```

← x's value does not changes when y's value is changed.

b) Call by value result =

- The arguments are evaluated at call time and the value parameters are copied and used as a local variables.

- The final values of these variables are copied back to the location of the arguments when the procedure exits.

Exp :

```
callByValueResult (int y, int z)
{
    y = y+1; z = z+1;
    print (y);
    print (z);
}

main ()
{
    int x = 42;
    print (x);
    callByValueResult (x, x);
    print (x);
}
```

Output :

```
x = 42
y = 43
z = 43
x = 43
```

Note that x's value is different from both using call-by-value and call-by-Reference.

c) Call by name =

- The procedure body is substituted for call in caller with actual parameters substituted for formals.

- The actual parameters can be surrounded by parenthesis to preserve their integrity.

- The locals name of called procedure and names of calling procedure are distinct.

Exp :

```
callByName (int closure y)
{
    print (y);
    print (y);
}

main ()
{
    int x = 42;
    print (x);
    callByName ([[x = x+1]]);
    print (x);
}
```

Output :

```
x = 42
y = 43
y = 44
x = 44
```

↑
x's value changes
when y is evaluated

d) Call by reference =

- The arguments must have allocated memory locations.
- The compiler passes the address of the variable, and the parameter becomes an alias for the argument.
- Local accesses to the parameter are turned into indirect accesses.

Exp :

```
callByRef (int &y)
{
    y = y + 1;
    print (y);
}

main ()
{
    int x = 42;
    print (x);
    callByRef (x);
    print (x);
}
```

Output :

```
x = 42
y = 43
x = 43
```

↑ x's value changes when y's value is changed.

Unit 4

Tutorial No. 5

SDT & TAC

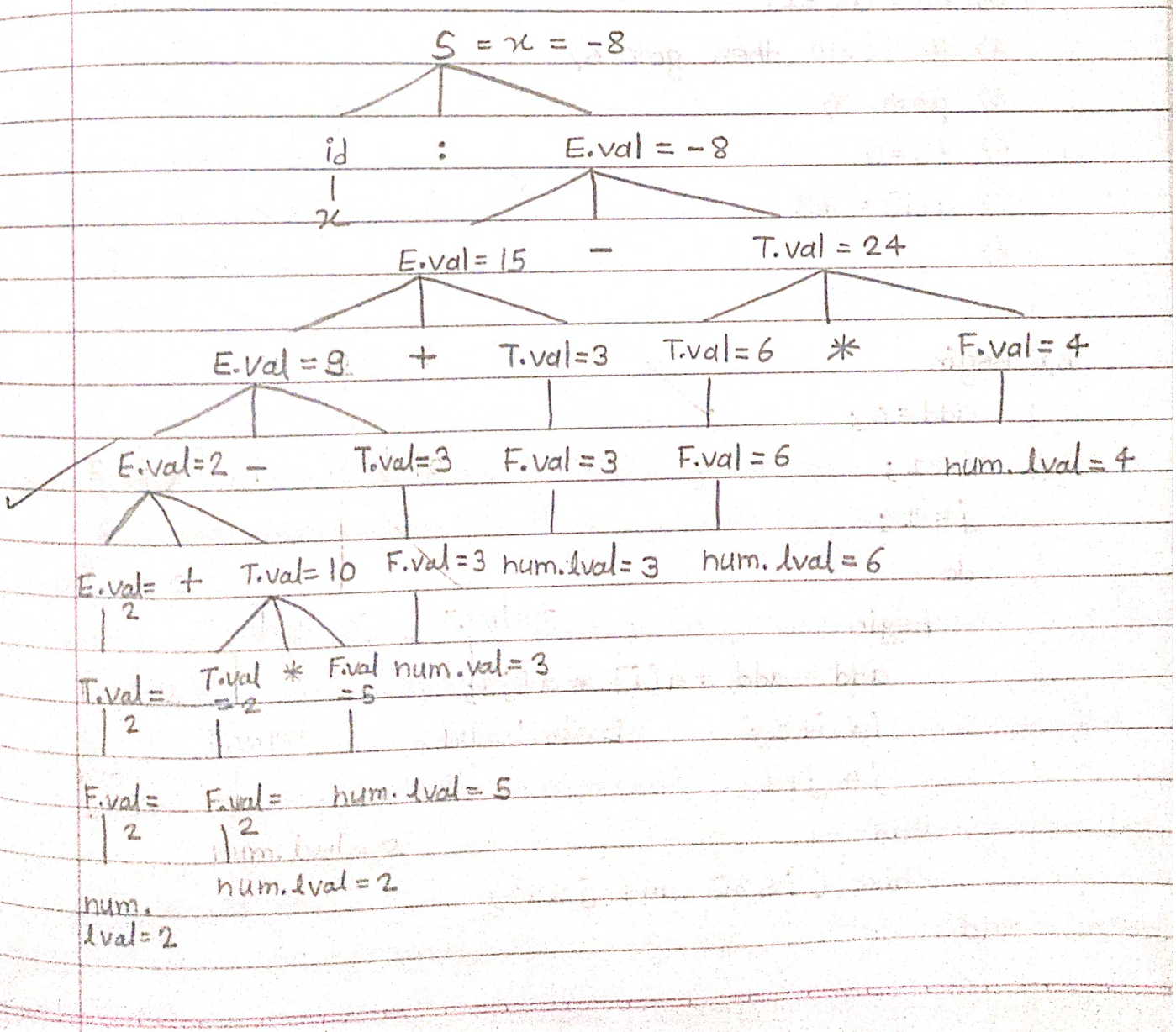
Q.1] $x = a + a * (b-c) + (b-c) * d$

Write the grammer and SDD to evaluate the below expression & construct syntax tree.

$x := 2 + 2 * (5-3) + (7-6) * 4$

→

$S \rightarrow id : E$	{ gen (id.name) = E.place }.
$E \rightarrow E + T$	{ E.val = E.val + T.val
$E \rightarrow E - T$	{ E.val = E.val − T.val
$E \rightarrow T$	{ E.val = T.val
$T \rightarrow T * F$	{ T.val = T.val * F.val
$T \rightarrow F$	{ T.val = F.val
$F \rightarrow num$	{ F.val = num.lval }



Q.2] Generate the 3 address code for following :

a)
```
main()
{
  int i=1;
  int a[10];
  while (i<=10)
    a[i]=0;
}
```

→

1) i=1
2) t1= 10 * i
3) t2 = a[t1]
4) if i<=10 then goto 6)
5) goto 3)
6) t3=0
7) a[i]= t3
8)

b)
```
begin
  add=0;
  i= 1;
  j= 1;
  do
    begin
      add = add + a[i] * a[j];
      i= i+1;
      j= j+1;
    end
  while (i≤20 and j≤20)
end
```

→ 1) add = 0
2) i = 1
3) j = 1
4) t1 = 4 * i
5) t2 = 4 * j
6) t3 = a [t1]
7) t4 = a [t2]
8) t5 = t3 * t4
9) t6 = add + t5
10) add = t6
11) t7 = i+1
12) i = t7
13) t8 = j+1
14) j = t8
15) if i ≤ 20 goto 17)
16) goto 19)
17) if j ≤ 20
18) goto 19)
19)

Q.3] What is Backpatching ? Write grammer & SDD for following expression a<b and c<d or e<f.

→

- The code for boolean expression inserts symbolic labels for jumps.
- It therefore needs a separate pass to get them to appropriate addresses.
- So, we use a technique named backpatching.

To ~~avoid this~~ -
- Backpatching is a process to free the unknown labels.

- In this we use 3 f^n as follows :

1) makelist (i) =
By using this f^n we can create a new list.

2) Merge (P1, P2) =
Concatenating two lists pointed by P1 & P2.

3) Backpatch (p, i) =
Insert i as target label for each of the instruction on list pointed by P.

Backpatching for a<b and b<d or e<f =>

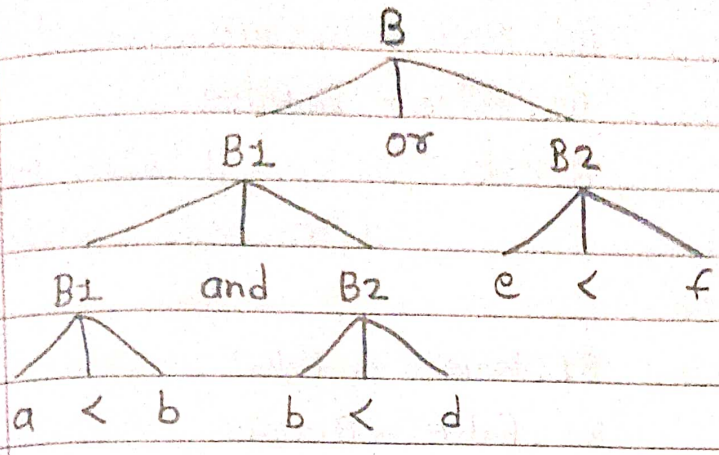
1) B → B1 || MB2 { Backpatch (B1.falselist, M.instr);
B.truelist = Merge (B1.truelist,
B2.truelist);
B.falselist = B2.falselist ; }

2) B → B1 && MB2 { Backpatch (B1.truelist , M.instr);
B.truelist = B2.truelist ;
B.falselist = Merge (B1.falselist,
B2.falselist); }

3) B → True { B.truelist = makelist (next instr);
emit ('goto_'); }

4) B → False { B.falselist = makelist (next instr);
emit ('goto_'); }

a<b and b<d or e<f



100 : if a<b then goto 102
101 : goto 104
102 : if b<d then goto 106
103 : goto 104
104 : if e<f then goto 106
105 : goto 107
106 : True
107 : False

Q.4] Write grammer and SDD for following boolean expression.

B → B1 || B2
B → B1 && B2
B → ! B1
B → B1 relop B2

→

1) B → B1 || B2

B1.true = B.true
B1.false = newlabel()
B2.true = B.true
B2.false = B.false
B.code = B1.code ||
label (B1, false) || B2.code

2) B → B1 && B2

B1.true = newlabel()
B1.false = B.false
B2.true = B.true
B2.false = B.false
B.code = B1.code || label(B1.true) || B2.code

3) B → ! B1

B1.true = B.false
B1.false = B.true
B.code = B1.code

4) B → E1 rel E2

B.code = E1.code || E2.code || gen('if' E1.addr rel.op E2.addr 'goto' B.true) || gen('goto' B.false)

5) B → True

B.code = gen('goto' B.true)

6) B → false

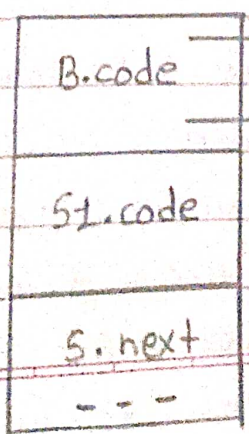
B.code = gen('goto' B.false)

Q.5] Write SDD for flow of control statements like if-then, if-then-else & while.

→

a) if =>

S → if (B) S1



B.true = newlabel()
B.false = S1.next = S.next
S.code = B.code || label(B.true) || S1.code

b) if-else =>

S → if (B) S1 else S2

	B.code	→ B.true → B.false
B.true:	S1.code goto S.next	
B.false:	S2.code S.next	

B.true = newlabel()
S1.next = S.next
B.false = newlabel()
S2.next = S.next
S.code = B.code || label(B.true) || S1.code || Label(B.false) || S2.code

c) while =>

S → while (B) S1

	B.code	→ B.true → B.false
B.true:	S1.code goto begin	
B.false:	S.next ----	

begin = newlabel()
B.true = newlabel()
B.false = S.next()
S1.next = begin
S.code = label(begin) || B.code || label(B.true) || S1.code || gen('goto' begin)

Tutorial No. 7

Q.1] Generate TAC for following:

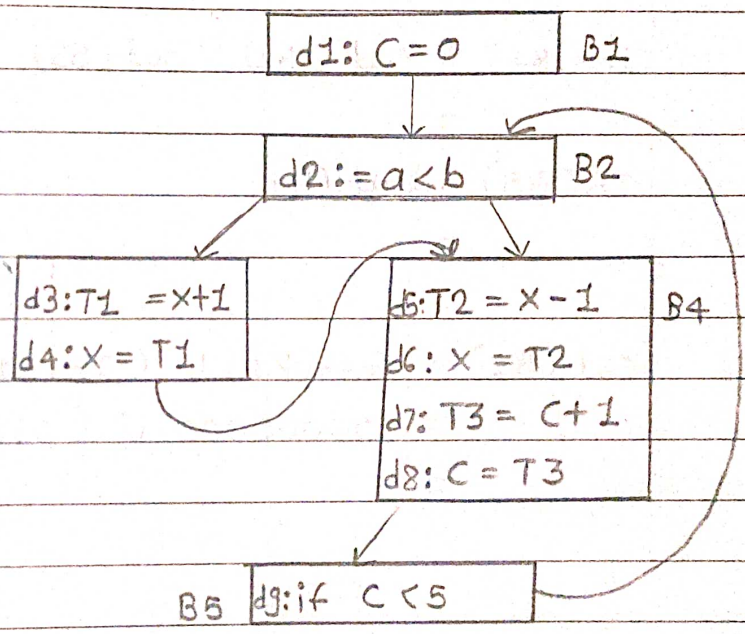
```
c = 0
do
{
   if (a<b) then
      x++;
   else
      x--;
      c++;
} while (c<5)
```

→

TAC =

1) c = 0
2) if (a<b) goto (4)
3) goto (7)
4) T1 = x+1
5) x = T1
6) goto (9)
7) T2 = x-1
8) x = T2
9) T3 = c+1
10) c = T3
11) if (c<5) goto (2)
12)

CFG =



Data Flow Analysis =

gen (B1) = {d1} = {100000000}
gen (B2) = {d2} = {010000000}
gen (B3) = {d3, d4} = {001100000}
gen (B4) = {d5, d6, d7, d8} = {000011110}
gen (B5) = {d9} = {000000001}

kill (B1) = {d7, d8, d9} = {000000111}

kill (B2) = ϕ

kill (B3) = {d5, d6} = {000011000}

kill (B4) = {d1, d3, d4, d9} = {101100001}

kill (B5) = {d1, d7, d8} = {100000110}

In (B1) = Gen (B2)

= {d2} = {010000000}

In (B2) = out (B1)

=

In (B3) = out (B2)

=

In (B4) = out (B2) U out (B3)

=

In (B5) = out (B4)

=

Out (B1) = Gen (B1) U (In (B1) - kill (B1))

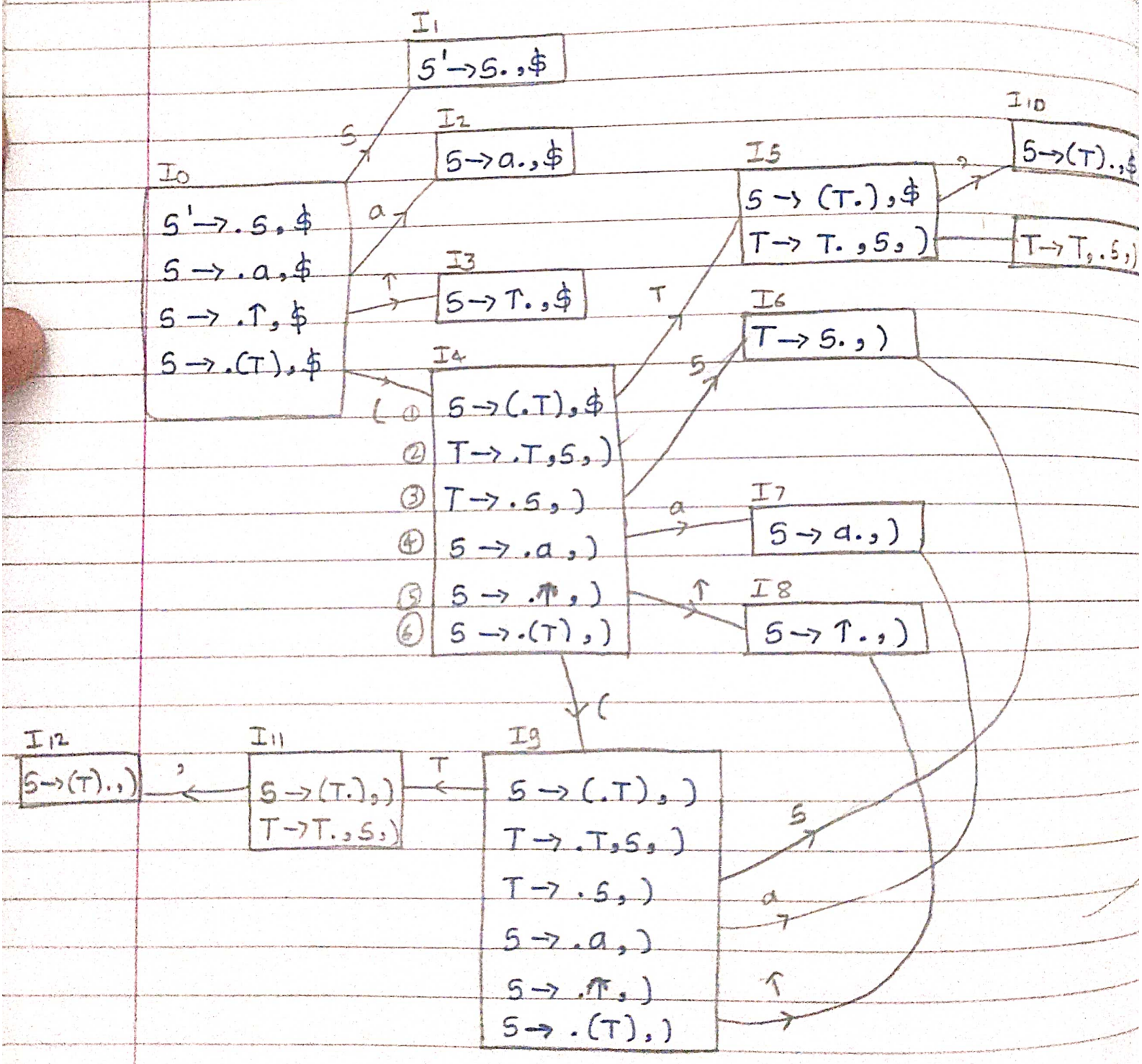
010000000 = 100000000 U (010000000 - 000000111)

-000000111 =

* Find LR(0) items & construct parsing table: (a, a) $

1) S → a // CLR
   S → ↑
   S → (T)
   T → T, S
   T → S

→



* Parsing table for SLR *

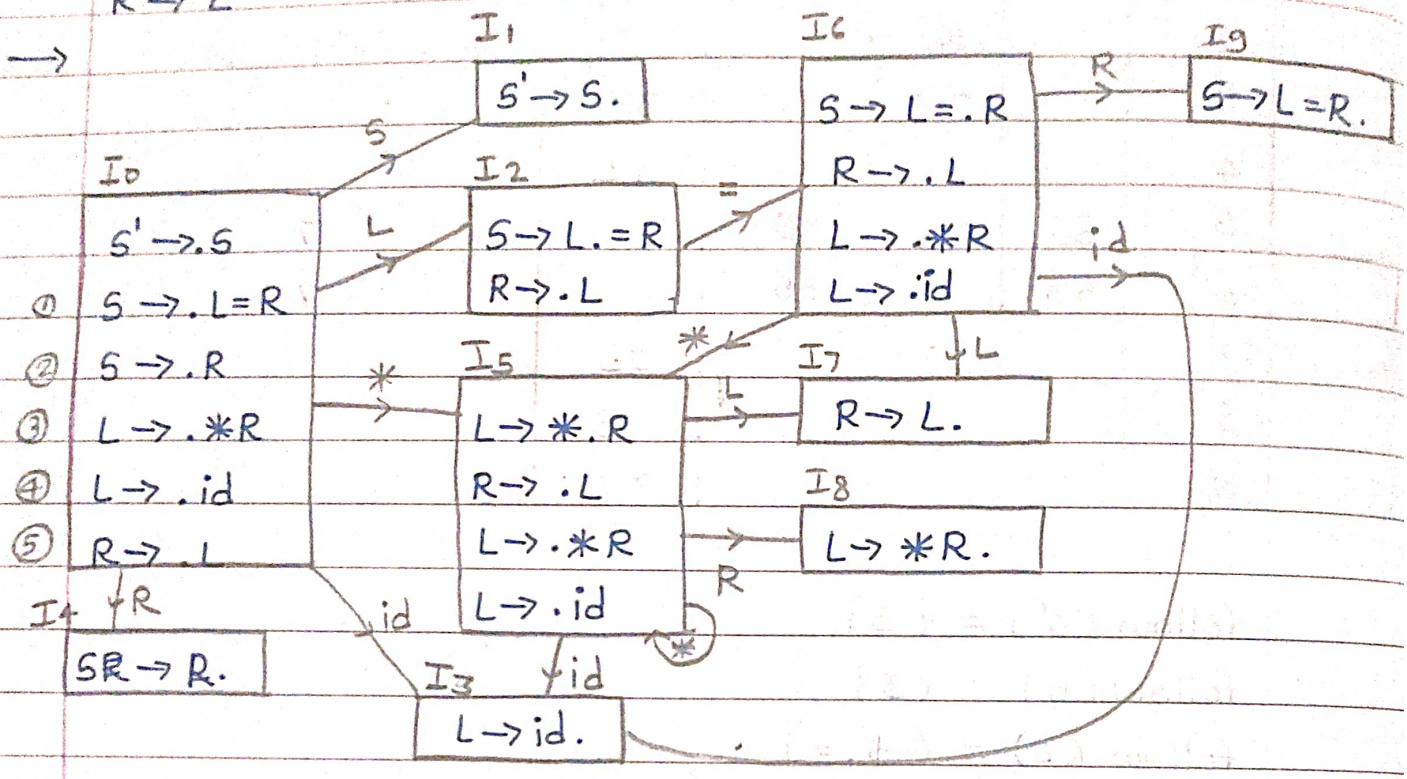
State	Action				Goto		
	id	=	*	$	S	L	R
0	S3		S5		1	2	4
1				accept			
2		S6					
3		r4		r4			
4				r2			
5	S3		S5			7	8
6	S3		S5			7	9
7		r5		r5			
8		r3		r3			
9				r1			

follow (S') = {$}
follow (S) = {$}
follow (L) = {$, =}
follow (R) = {$, =}

	Action					Goto	
	)	a	↑	(	$	S	T
0		S2	S3	S4		1	
1					accept		
2							
3							
4		S7	S8			6	5
5	S10						
6					r3		
7							
8							
9							
10							
11							
12							

ii) $S' \to S$
$S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

// LR(0) & SLR



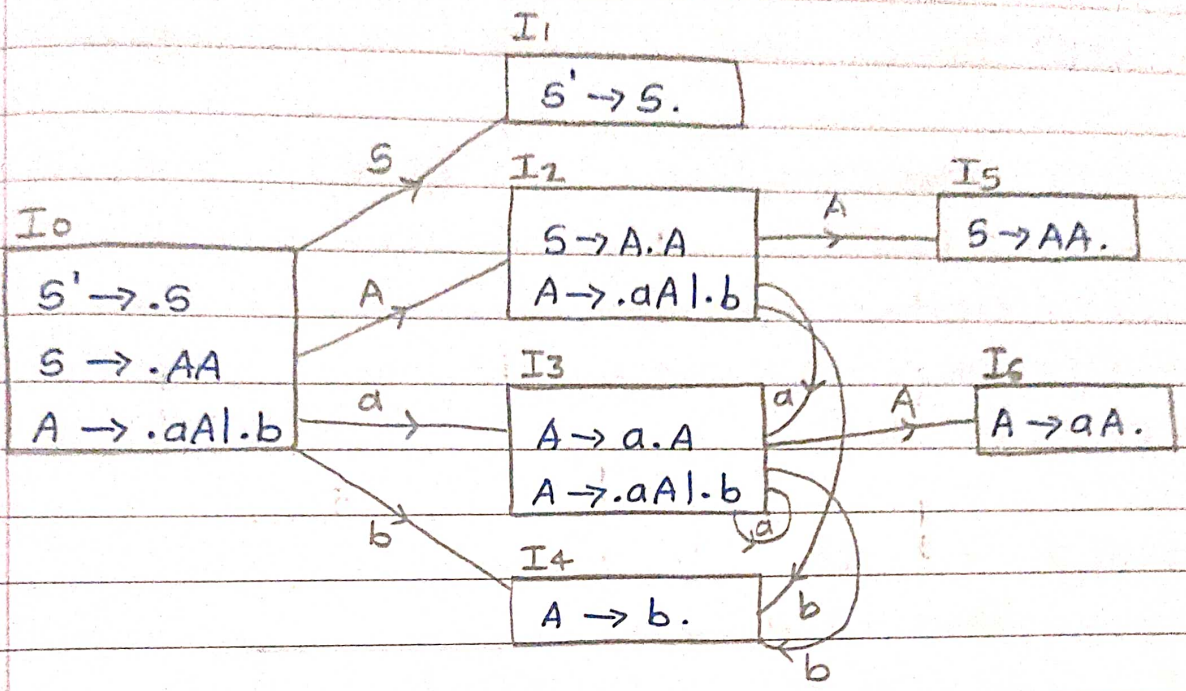
* Parsing table for LR(0) =

	Action				Goto		
	id	=	*	$	S	L	R
0	S3		S5		1	2	4
1				accept			
2		S6					
3	r4	r4	r4	r4			
4	r2	r2	r2	r2			
5	S3		S5			7	8
6	S3		S5			7	9
7	r5	r5	r5	r5			
8	r3	r3	r3	r3			
9	r1	r1	r1	r1			

1) LR(0) Parser =>

e.g. i) S → AA
A → aA | b

→



Parsing table =

State No.	Action			Goto	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
$4	γ3	γ3	γ3		
$5	γ1	γ1	γ1		
6	γ2	γ2	γ2		

5) $E \to TE'$

$E' \to +TE' \mid \varepsilon$

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id$

$\longrightarrow$

$First(E) = first(T) = first(F) = \{ (, id \}$

$F(E') = \{ +, \varepsilon \}$

$F(T) = first(F) = \{ (, id \}$

$F(T') = \{ *, \varepsilon \}$

$F(F) = \{ (, id \}$

$Follow(E) = \{ \$,) \}$

$F(E') = follow(E) = \{ \$,) \}$

$F(T) = [(first(E') - \varepsilon) \cup follow(E) \cup follow(E')] = \{ +, \$,) \}$

$F(T') = follow(T) = \{ +, \$,) \}$

$F(F) = (first(T') - \varepsilon \cup follow(T) \cup follow(T')) = \{ *, +, \$,) \}$

6) $S \to ACB \mid CbB \mid Ba$

$A \to da \mid BC$

$B \to g \mid \varepsilon$

$C \to h \mid \varepsilon$

$\longrightarrow$

$first(S) = \{ d, g, h, \varepsilon, b, a \}$

$first(A) = first(d) \cup first(B) - \varepsilon \cup first(C) = \{ d, g, h, \varepsilon \}$

$f(B) = \{ g, \varepsilon \}$

$f(C) = \{ h, \varepsilon \}$

$Follow(S) = \{ \$ \}$

$f(A) = \{ first(C) - \varepsilon \cup first(B) - \varepsilon \cup follow(S) \} = \{ h, g, \$ \}$

$f(B) = \{ follow(S) \cup first(a) \cup first(C) - \varepsilon \cup follow(A) = \{ \$, a, h, g \}$

$f(C) = \{ first(B) - \varepsilon \cup follow(S) \cup first(b) \cup follow(A) =$

$\{ g, \$, b, h \}$

First

$S \to A$ first(S) = first(A) = {a}

$A \to aBA'$ f(A) = {a}

$A' \to dA' \mid \varepsilon$ f(A') = {d, ε}

$B \to b$ f(B) = {b}

$c \to g$ f(c) = {g}

Follow(S) = {$}

Follow(A) = follow(S) = {$}

f(A') = follow(A) = {$}

f(B) = first(A') ∪ follow(A) = {d, $}

f(c) = NA

3)

	First	Follow
$S \to (L) \mid a$	{(, a}	f(S) = {$, ', ,)} - (first(L') - ε) ∪ follow(L) ∪ follow(L)
$L \to SL'$	{(, a} - f(S)	follow(L) = {)}
$L' \to ,SL' \mid \varepsilon$	{, , ε}	f(L') = follow(L) = {)}

4) $S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

→

first(S) = {a, b} follow(S) = {$}

first(A) = {ε} follow(A) = first(a) ∪ first(b) = {a, b}

first(B) = {ε} follow(B) = {a, b}

* Compute First & follow ⇒

1) S → aBDh
B → cC
C → bC | ε
D → EF
E → g | ε
F → f | ε

→

First(S) = {a}
First(B) = {c}
First(C) = {b, ε}
First(D) = (First(E) - ε) ∪ First(F) = {g, f, ε}
First(E) = {g, ε}
First(F) = {f, ε}

Follow(S) = {$}
Follow(B) = (first(D) - ε) ∪ first(h) = {g, f, h}
Follow(C) = (follow(B)) = {g, f, h}
Follow(D) = (first(h)) = {h}
Follow(E) = ((first(F) - ε) ∪ follow(D)) = {f, h}
Follow(F) = (follow(D)) = {h}

2)

S → A
A → aB | Ad
B → b
C → g

→

Eliminate left recursion.