Unit 2

Question Bank

Q.1] Explain compiler construction tools with example ?

→

- The compiler construction can use some specialized tools that helps to implement various phases of compiler.

1) Parser Generator =>

It produces syntax analyzer from the input based on grammatical description of the programming language or CFG.

e.g: PIC , EQN

2) Scanner Generator =>

- It produces lexical analyzer from the input consist of regular expression description based on token of a language.

- It generatees finite automation to recognize the ~~regeug~~ regular expression.

e.g: LEX

3) Syntax directed translation engines =>

- It generates intermediate code with three address format from the input that consist of a parse tree.

- These engines have routines to traverse the parse tree & produces the intermediate code.

- In this each node of the parse tree is associated with one or more translation.

4) Automatic Code Generators =>

- It generates the machine language for a target machine.
- each operation of intermediate code is translated using collection of rules & taken as a I/p by code generator.

5) Data flow analysis engines =>

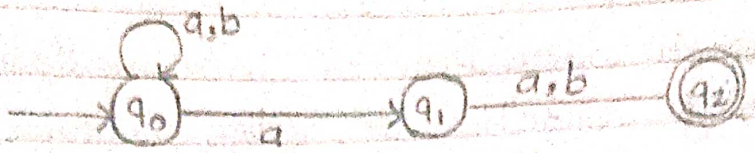
- It facilitates describes the gathering of information about how values are transmitted from one part of the program to other part.
- It is used in code optimization.

6) Compiler Construction toolkits =>

- It provides integrated set of routines for constructing various phases of a compiler.

Q.4] Construct Minimum State DFA for the regular expression (a|b)*a(a|b).

→

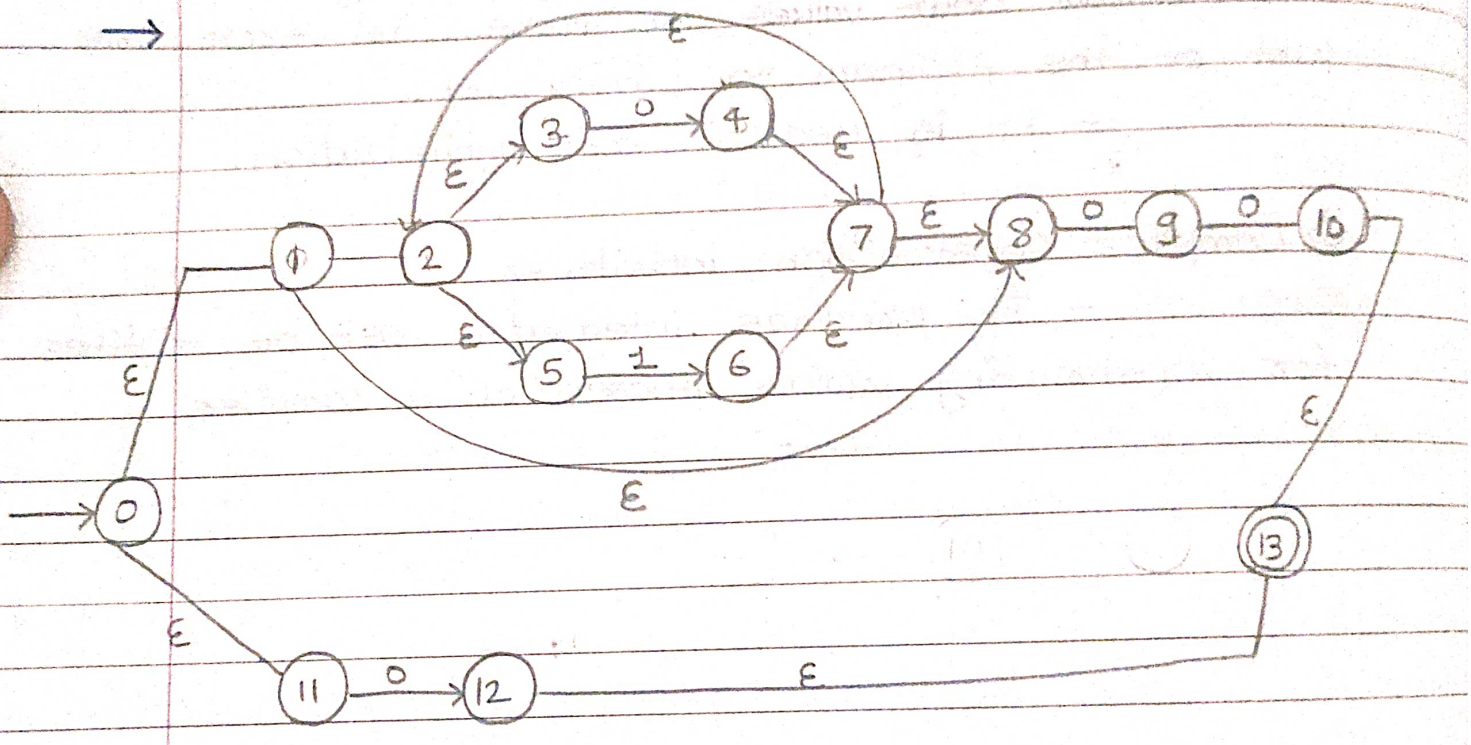


	a	b
q_0	q_1, q_0	q_0
q_1	q_2	q_2
q_2	-	-

Q.5] Construct NFA for the regular expression -

((0|1)* 00) 10)

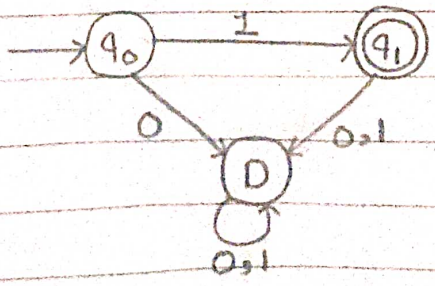
→



Q.6] Construct DFA for accepting the following language over an alphabet {0, 1}

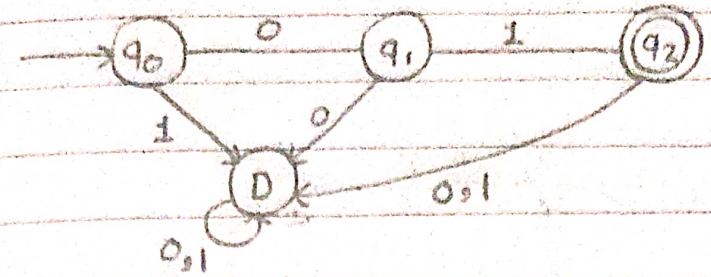
a) Accept only 1 as a string.

→



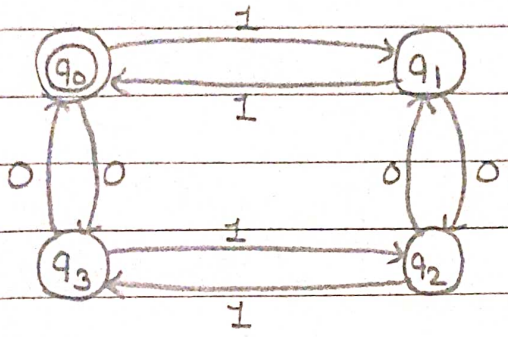
b) Accept String as 01.

→



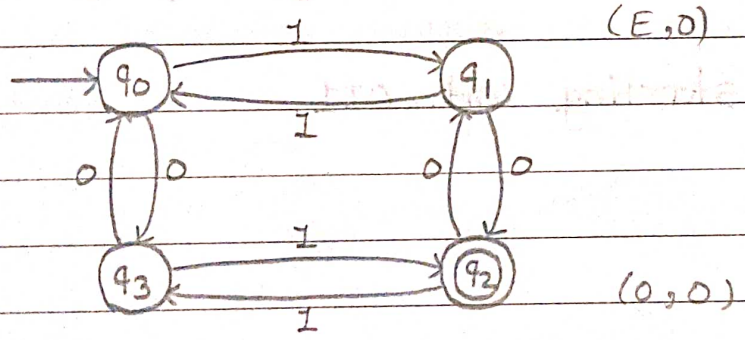
c) Number of 1's is even & number of 0's is even.

→



d) Number of 1's is odd & number of 0's is odd.

→ (EE)



(E,O)

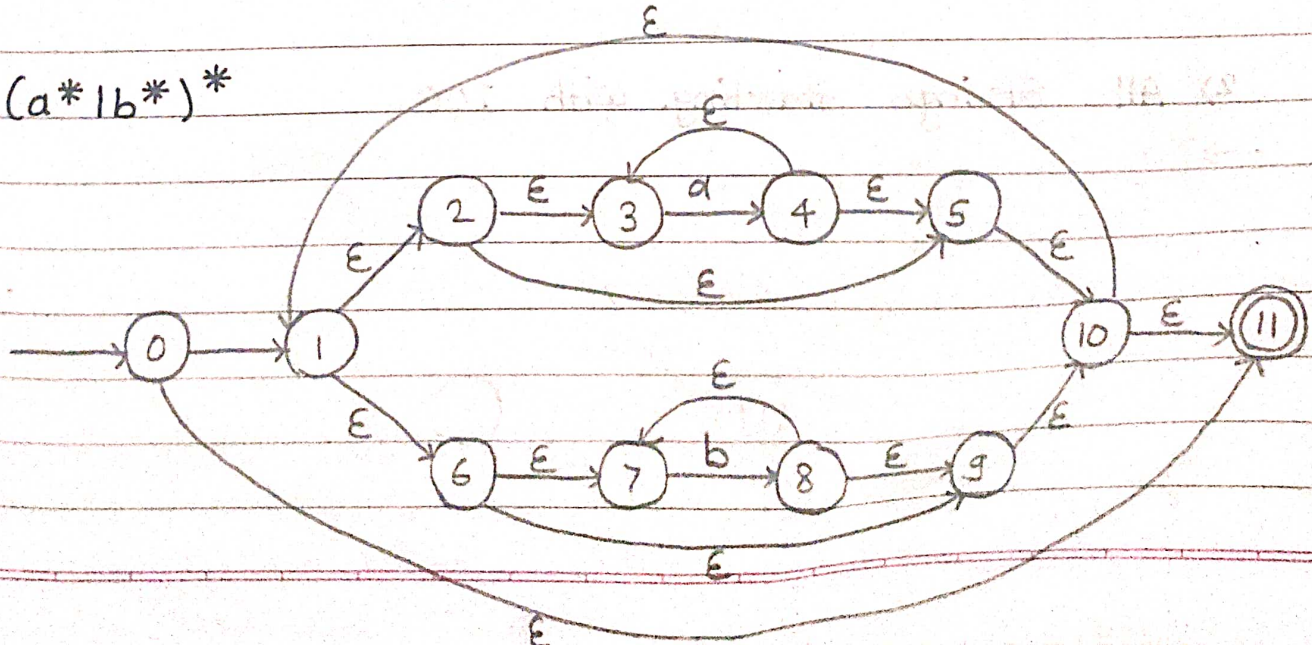
(O,E)

(O,O)

Q.7] Construct NFA using Thompsons Rule.

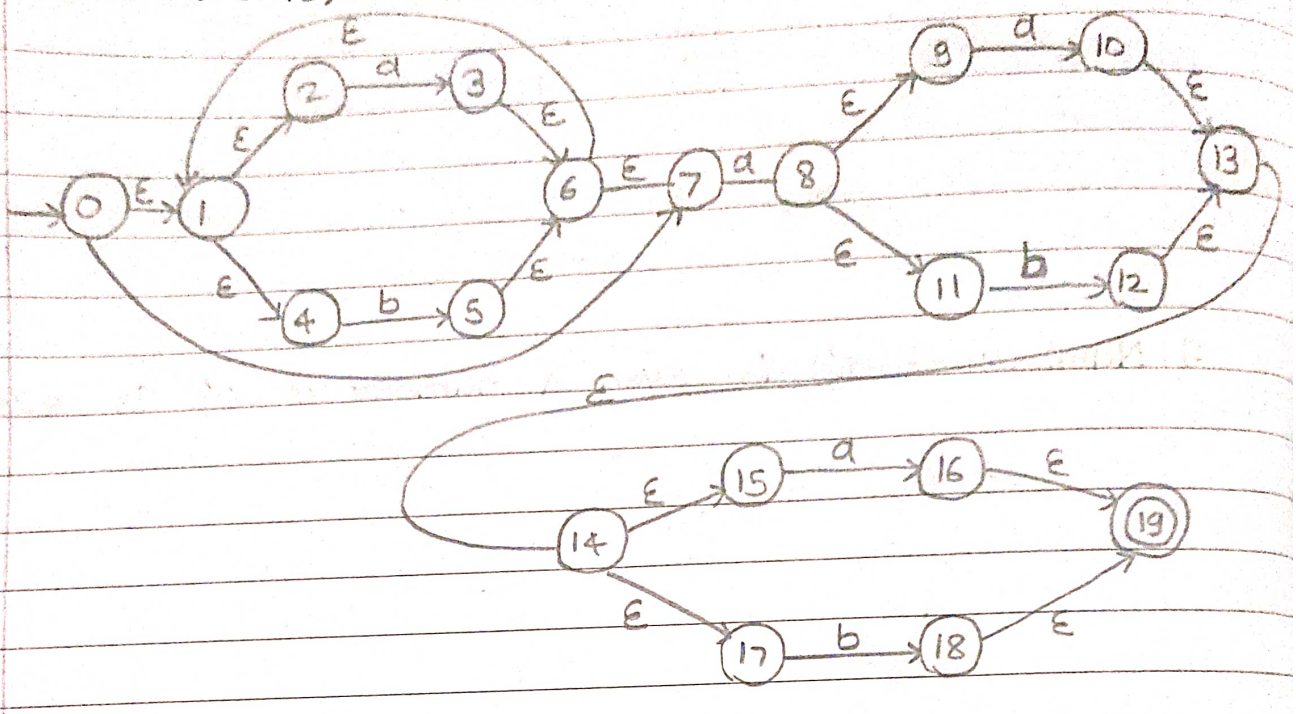
1) $(a^*|b^*)^*$

→



2) (a|b)* a (a|b) (a|b)

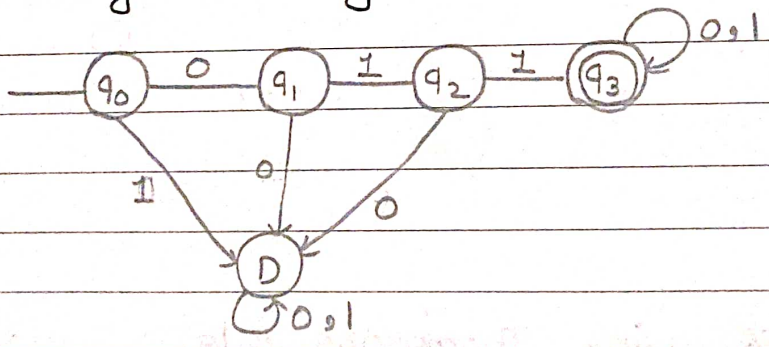
→



Q.8] Construct DFA for the following language:

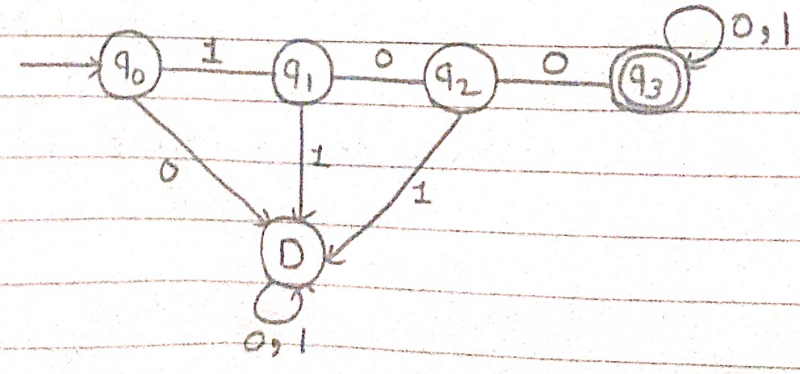
1) All strings starting with 011.

→



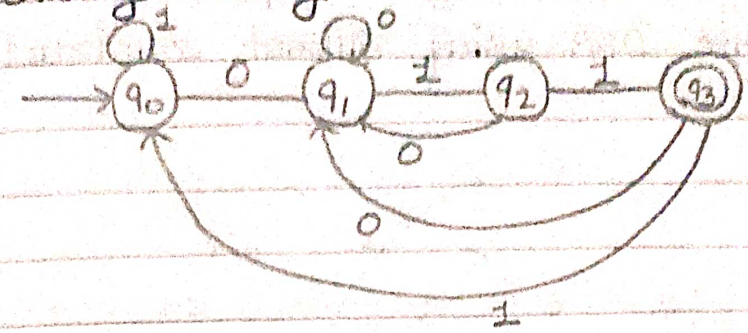
2) All strings starting with 100.

→



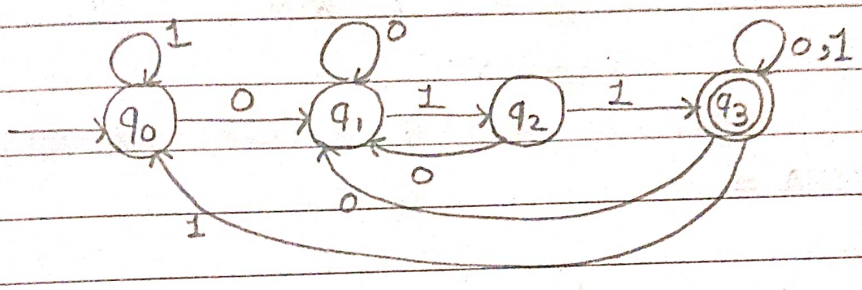
3) All string ending with 011.

→



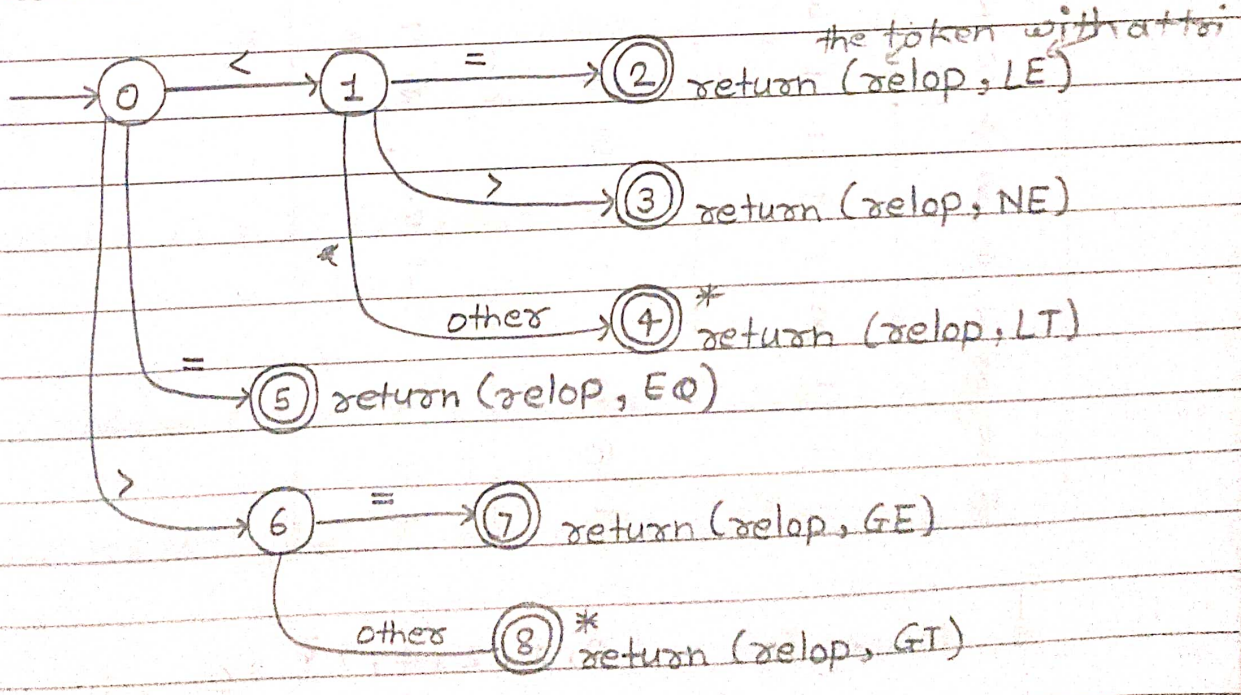
4) All string with a as substring 011 anywhere in the string.

→



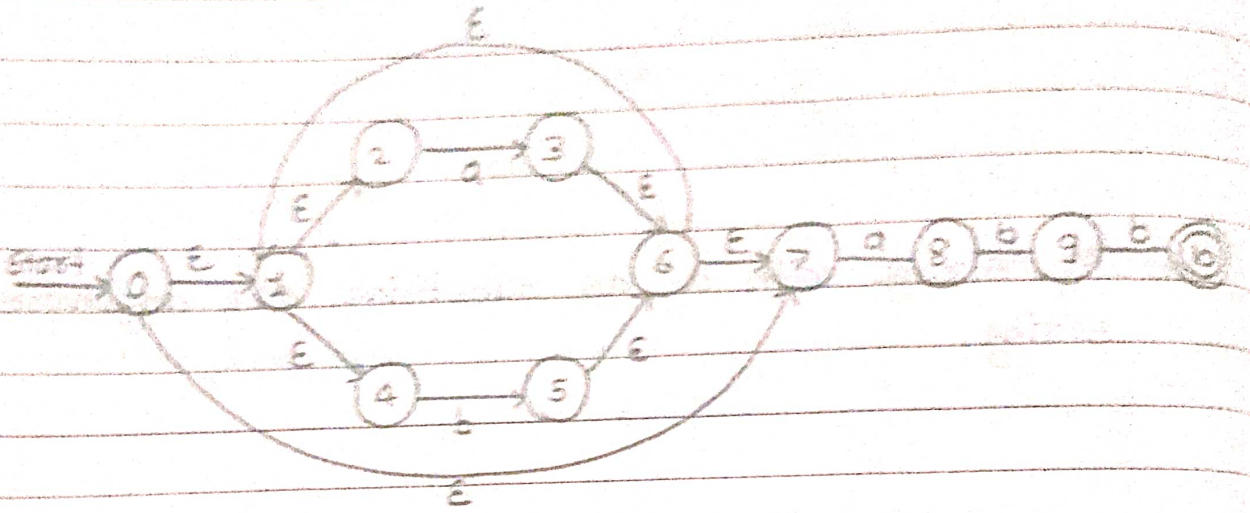
Q.10] Draw a transition diagram to represent relational operators.

→



Q.11] Construct NFA from a ~~regular~~ expression $(a|b)^*abb$. Convert it into DFA using subset construction method.

→



T For NFA =

	a	b	ε
0	–	–	1,7
1	–	–	2,4
2	3	–	–
3	–	–	6
4	–	5	–
5	–	–	6
6	–	–	7
7	8	–	–
8	–	9	–
9	–	10	–
10	–	–	–

ε - closure (0) = {0, 1, 2, 4, 7} = A

Move (A, a) = {(0, 1, 2, 4, 7), a}
= {3, 8}

ε-closure (A,a) = { ε-closure {3,8}
= {1,2,3,4,5,6,7,8} = B

Move (A,b) = {5}

ε-closure (A,b) = ε-closure {5}
= {5,6,7,1,2,4} = C

Move (B,a) = {(1,2,3,4,5,6,7,8),a}
= {3,8} = B

ε-closure (B,a) = ε-closure {3,8}
= B

Move (B,b) = {(1,2,3,4,6,7,8),b}
= {5,9}

ε-closure (B,b) = ε-closure {5,9}
= {5,6,7,1,2,4,9} = D

move (C,a) = {(1,2,4,5,6,7),a}
= {3,8}

ε-closure (C,a) = ε-closure ({3,8}) = B

ε-closure (C,b) = ε-closure ({1,2,4,5,6,7},b)
= {5} = C

Move (D,a) = B

Move (D,b) = {5,10} = {1,2,3,6,7, *10} = E
↑
final state

Move (E, a) = {3, 8} = B

Move (E, b) = {5} = C

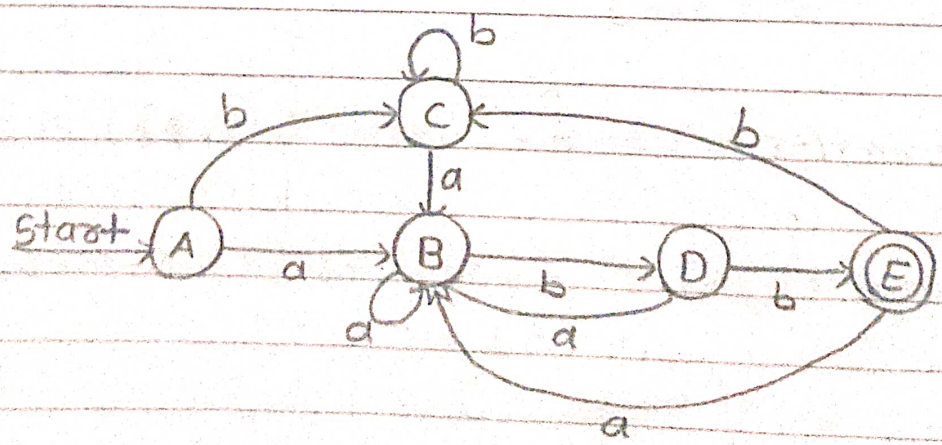
∴ 5 different sets of states we construct are :

1) A = {0, 1, 2, 4, 7}
2) B = {1, 2, 3, 4, 6, 7, 8}
3) C = {1, 2, 4, 5, 6, 7}
4) D = {1, 2, 4, 5, 6, 7, 9}
5) E = {1, 2, 4, 5, 6, 7, 10}

T- For DFA =

State	I/P Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

DFA =



Q.2] What is the role of lexical analysis ? Explain input buffering with example ?

→

Lexical Analysis =

- As the first phase of the compiler the main task of Lexical Analyzer is to read the input character from the source program, group them into lexemes and produce the o/p as sequence of tokens for each lexeme in the source program.

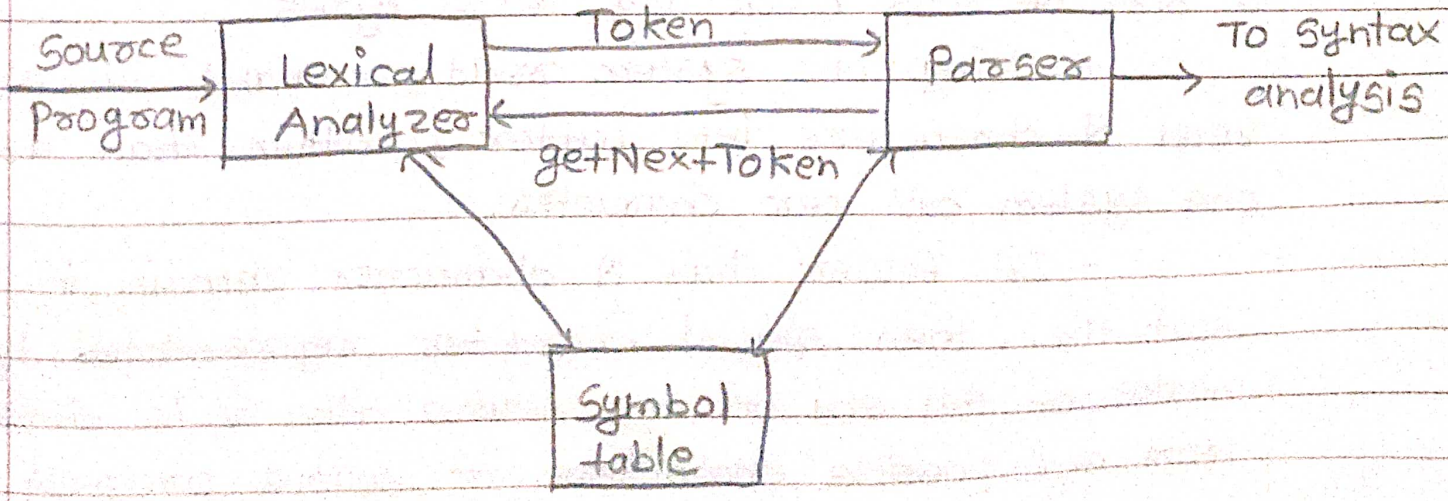
- These sequence of tokens sent to the parser for syntax analysis.

- It is common for lexical analyzer to interact with symbol table as well.

- When the lexical analyzer identifies or discovers lexeme with identifier then it needs to be enter into symbol table.

- In some cases interaction regarding the identifiers may be read from symbol table by lexical analyzer.

- These interactions are shown in figure bellow :-



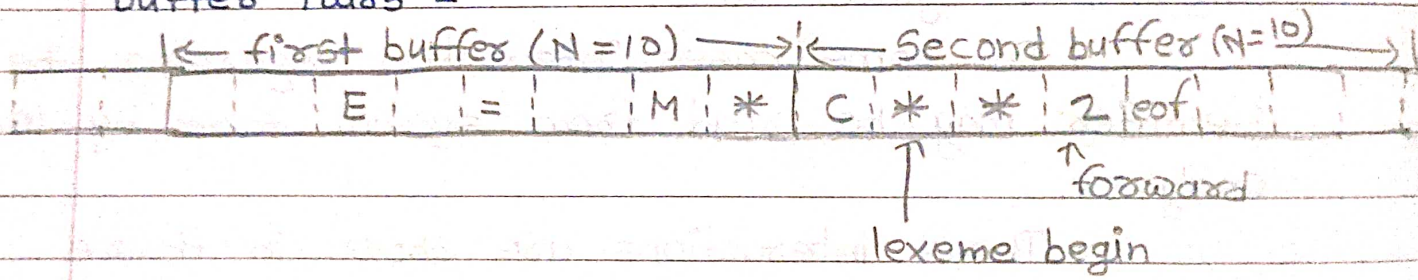
- Lexical analyzer can also perform other tasks :-

1) Stripping out comments & white spaces.
2) Correlating error messages generated by the compiler with the source program.

Input Buffering =>

- Sometimes the lexical analyzer needs to look ahead some symbols to decide about the token to return.
- In C lang : we need to look after -, =, < to decide what token to return.
- So we need to introduce a two buffer scheme to handle large look-aheads safely.

Buffer Pairs =



- Each buffer is of same size N & N is a size of disk block e.g. 4096 bytes.
- Using one system read command we can read N characters into buffer, rather than using one system call per character.
- If fewer than N character remain in the input file, then special character represented by eof, marks of the end of the source file & is different from any possible characters of source program.

- Two pointers to the I/p are mentioned :-

1) Pointer lexemeBegin - Marks the beginning of the current lexeme.

2) Pointer forward - scans ahead until a pattern match is found.

Sentinals =

- It is special character that cannot be part of source program & natural choice is the character eof.

- It is added at end of each buffers, plays the role of entire input end.

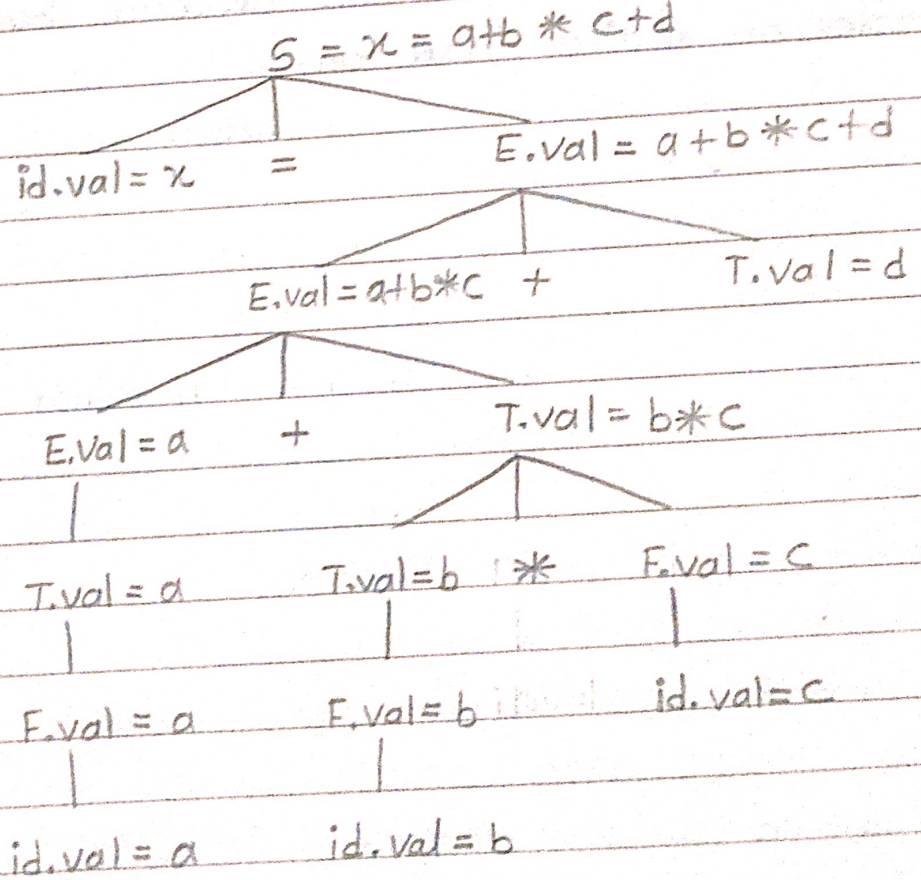
Unit 4

Question Bank

Q.1] Write grammer & SDD for x := a+b * c+d. Construct annotated parse tree.

→ Grammer =

$S \rightarrow id = E$	{ gen (id.val = E.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 }
$F \rightarrow id$	{ F.val = id.~~name~~ val }



Q.2] What is S-attributed and L-attributed definition ? Implement bottom up evaluation of S-attributed for the input 3*5 + 4n.

→

S-attributed ⇒

- S-attributed means synthesized attribute.
- An attribute is said to be synthesized attribute if it's value at a parse tree node is determined from attributes values at the children of the node.
- In this semantic actions placed at right hand side. e.g : A → BC { }
- Attributes are evaluted by bottom up parsers.

L-attributed ⇒

- L-attributed means inherited attribute.
- An inherited attributes is one whose value at parse tree node is determined in terms of attributes at the parent and sibling of that node.
- In this semantic actions placed anywhere. e.g : A → BC { } , A → { } BC , A → A { } BC
- Attributes are evaluted by depth first traversal in bottom up parsers.

Grammer =

L → En

E → E+T

E → T

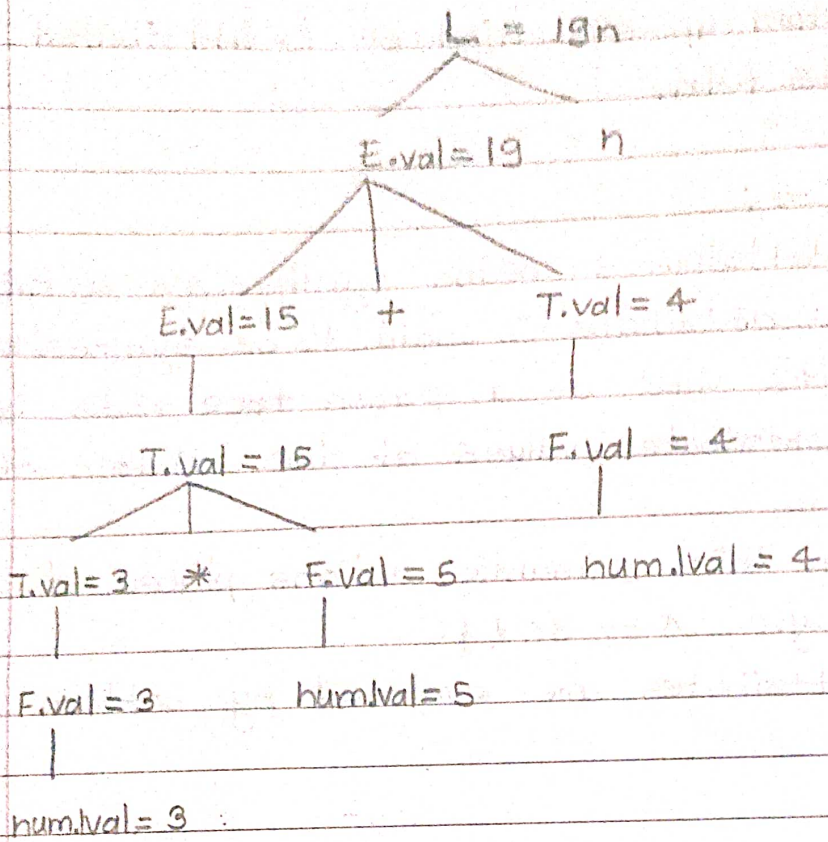
T → T * F

T → F

F → Num

Parse Tree =

3 * 5 + 4n.



Stack Implementation =>

Stack	I/P	Action
$	3 * 5 + 4n $	shift
$ 3	* 5 + 4n $	reduce ~~num~~ ~~num~~ F → num
$ 3	* 5 + 4n $	reduce T → ~~num~~ F
$ 3 *	5 + 4n $	shift
$ 3 *	5 + 4n $	reduce F → num
$ 3 * 5	+ 4n $	shift
$ 3 * 5	+ 4n $	reduce T → T * F
$ 15	+ 4n $	reduce E → T
$ 15 +	4n $	shift
$ 15 + 4	n $	reduce F → num
$ 15 + 4	n $	reduce T → F
$ 19	n $	reduce E → E + T
$ 19n	$	shift

∴ Point 19n

Q.3] Write grammer & SDD for converting infix expression into postfix.

→

Grammer =

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

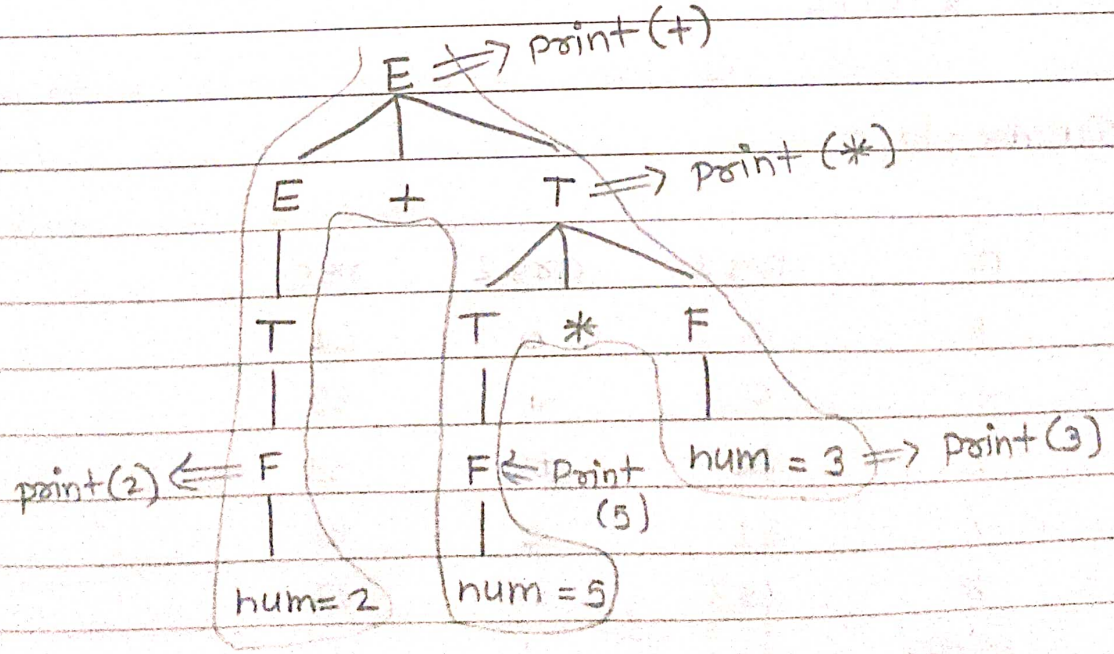
$F \rightarrow num$

SDD =

Production	Semantic Rules
$E \rightarrow E+T$	{ print ("+"); }
$E \rightarrow T$	{ }
$T \rightarrow T*F$	{ print ("*"); }
$T \rightarrow F$	{ }
$F \rightarrow num$	{ print ("num.lexval"); }

Example =

Infix Expression : 2 + 5 * 3



∴ Postfix ⇒ 253 * +

Q.4] What is three address code ? Translate the expression (a+b) / (c+d) * (a+b/c) - d into quadraples, triples & indirect triples.

→

- Three address code is an intermediate code used by optimizing compilers in the implementation of code-improving transformations.
- Each TAC instruction has at most three operands & it is combination of assignment and a binary operator.
- There is atmost one operator on the left side.

TAC for (a+b) / (c+d) * (a+b/c) - d

1) t1 = a+b
2) t2 = c+d
3) t3 = t1 / t2
4) t4 = t1 / c
5) t5 = t3 * t4
6) t6 = t5 - d
7) x = t6

1) Quadraple =

op	arg 1	arg 2	res
+	a	b	t1
+	c	d	t2
/	t1	t2	t3
/	t1	c	t4
*	t53	t4	t5
minus	t5	d	t6

2) Triple =

	OP	arg 1	arg 2
0	+	a	b
1	+	c	d
2	/	(0)	(1)
3	/	(0)	c
4	*	(2)	(3)
5	minus	(4)	d

3) Indirect Triple =

	OP	OP	arg 1	arg 2
35	(0)	+	a	b
36	(1)	+	c	d
37	(2)	/	(0)	(1)
38	(3)	/	(0)	c
39	(4)	*	(2)	(3)
40	(5)	minus	(4)	d

Q.5] Generate three address code for following boolean expression (a<b) or (c<d) and (e<f) using translation scheme of Boolean.

→

Q.6] What is three address code ? What are the types of three address code ? Write SDD to generate TAC for assignment ?

→

- TAC is an intermediate code used by optimizing compilers in the implementation of code-improving transformations.

- Each TAC instruction has at most three operands and it is combination of assignment and a binary operator.

- There is atmost one operator on the right side.

- Following are the types of TAC :-

1) Assignment statement
2) Copy statement - $x := y$
3) Unconditional jump - goto L
4) Conditional jump - if x relop y goto L
5) Procedural call - Param x call p
   return y

SDD to generate TAC for assignment ⇒

Assignment stmt. can be in following 2 forms:

1) $X := op\ y$
2) $X := y\ op\ z$

Q.7] Explain back patching with suitable example.

→

- The codes for boolean expression insert symbolic labels for jump.
- Therefore we need seperate pass to set them to appropriate addresses.
- So, we use technique called backpatching to avoid this.
- BackPatching is a process to free the unknown labels.
- In this we use 3 functions as follows :-

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

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

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

e.g:
$a \overset{<}{>} b$ or b<d and e<f = input expression

⇒

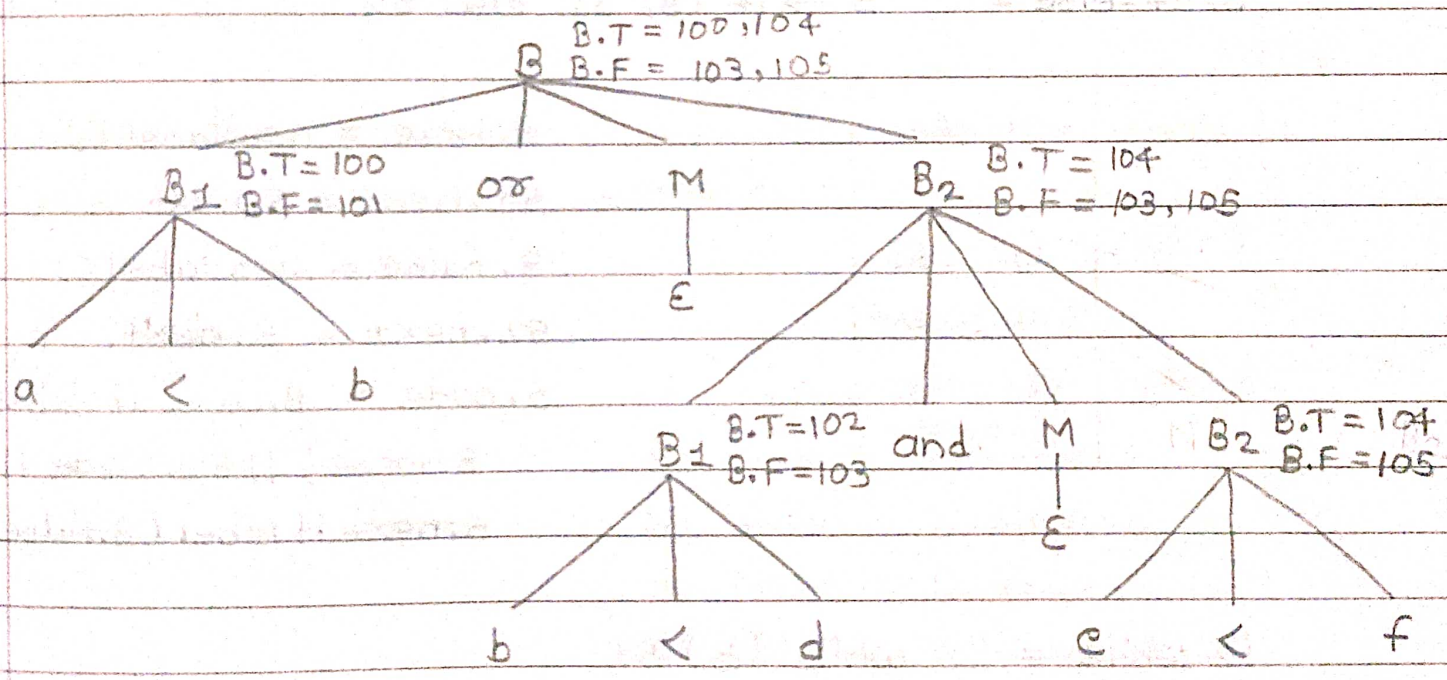
Rules required for constructing this expression ⇒

1) $B \rightarrow B_1$ || M B_2 { backpatch (B1.falselist , M.instr);
B.truelist = merge (B1.truelist + B2.truelist);
B.falselist = ~~m~~ B2.falselist); }

2) $B \rightarrow B_1$ && M B_2 { backpatch (B_1.truelist, M.instr);
B.truelist = B_2.truelist;
B.falselist = merge (B_1.falselist + B_2.falselist); }

3) $B \rightarrow E_1$ rel E_2 { B.truelist = makelist (nextinstr);
B.falselist = makelist (nextinstr + 1);
emit ('if' E_1.addr rel.op E_2.addr 'goto _');
emit ('goto _'); }

Annotated Parse tree =

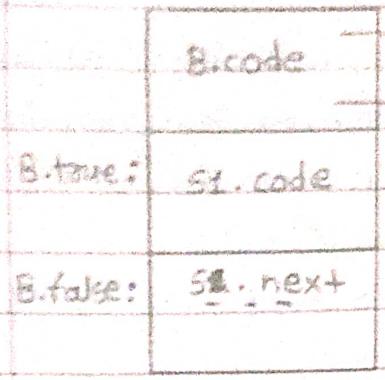


100 : if a < b goto 106
101 : goto 102
102 : if b < d goto 104
103 : goto 107
104 : if e < f goto 106
105 : goto 107
106 : True
107 : false

Q.8] How would you generate three address code for flow of control statements.

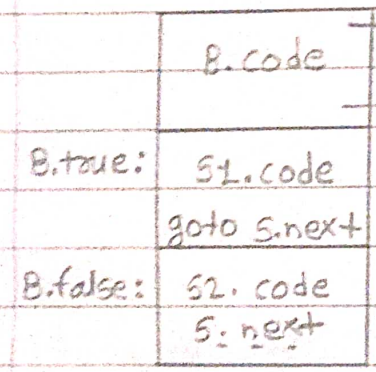
→

a) if = $S \rightarrow if\ (B)\ S1$



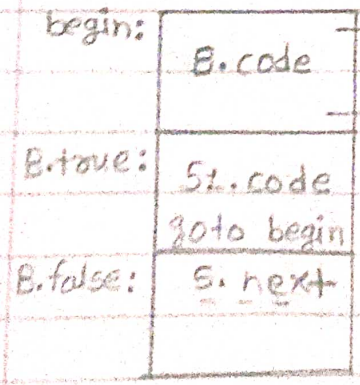
B.true = newlabel()
B.false = S1.next = S.next
S.code = B.code || label(B.true) || S1.code

b) if-else = $S \rightarrow if\ (B)\ S1\ else\ S2$



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

c) while = $S \rightarrow while\ (B)\ S1$



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)

Example =

if (x < 100 || x > 200 && x != y) then x = 0 ;

=> x = 0 ;

TAC =

```
    if x < 100 goto L3
    goto L1
L1 : if x > 200 goto L2
    goto L4
L2 : if x != y goto L3
    goto L4
L3 : x = 0
L4 :
```

[10] Write the grammer & semantic rule for boolean expression (AND, OR, NOT)

→

1) AND =>

	Semantic rule
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$
	$B_1.false = B.false$
	$B_2.true = B.true$
	$B_2.false = B.false$
	$B.code = B_1.code \;\|\|\; label(B_1.true) \;\|\|\; B_2.code$

2) OR =>

$B \rightarrow B_1 \;\|\|\; B_2$	$B_1.true = B.true$
	$B_1.false = newlabel()$
	$B_2.true = B.true$
	$B_2.false = B.false$
	$B.code = B_2.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$

3) NOT ⇒

Grammer

$B \rightarrow !\, B1$

Semantic rule

$B1.true = B.false$

$B1.false = B.true$

$B.code = B1.code$

Unit 6

Tutorial No. 6

Q.1] Explain following machine independent transformation techniques :

a) Common sub expression and dead code elimination =>

i) Common Sub Expression =

- The expression that has been already computed before and appears again in the code for computation is called as Common sub-expression.
- In this technique, As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.
- Types of common sub-expression elimination =>
  1) Local common sub-expression elimination
  2) Global common sub-expression elimination.

E.g :

Before elimination :

```
a = 10;
b = a + 1 * 2;
c = a + 1 * 2;
d = c + a;
```

After elimination :

```
a = 10;
b = a + 1 * 2;
d = b + a;
```

ii) Dead Code Elimination =

- Code that is unreachable or that does not affect the program can be eliminated.
- In this technique, As the name suggests, it involves eliminating the dead code.

- The statement of the code which either never executes or are unreachable or their output is never used are eliminated.

E.g:

Before elimination:

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

After elimination:

```
i = 0;
```

b) Copy propagation and constant folding =>

i) Copy Propagation =

- Copy propagation is the process of replacing the occurrences of targets of direct assignments with their values.
- Given an assignment $x = y$, replace later uses of x with uses of y, provided there are no assignments to x or y.
- $f := g$ are called copy statements.
- Use of g for f, whenever possible after copy statement.

E.g:

```
x[i] = a;          =>    x[i] = a;
sum = x[i] + a;          sum = a + a;
```

ii) Constant Folding =

- In this technique, As the name suggests, it involves folding the constants.
- The expression that contain the operands having constant values at compile time are evaluated.
- Those expressions are then replaced with their respective results.

E.g:

$x := y \ op \cdot z \quad \Rightarrow \quad x := op \ y$

If y & z are constants, then the value can be computed at compilation time.

Q.2] Explain how code motion and frequency reduction used for loop optimizations?

→

Code Motion =

- Code motion is used to decrease the amount of code in loop.
- It takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

e.g:

```
while (i<=limit-2)
// after code motion result:
a = limit-2;
while (i<=a)
```

Frequency reduction =

- It is a type in loop optimization process which is machine independent.
- In frequency reduction code inside a loop is optimized to improve the running time of program.
- It is used to decrease the amount of code in a loop.
- The objective is to reduce the evaluation frequency of expression.
- To bring loop invariant statements out of the loop.

e.g:

```
int a=2, b=3, c;
while (i<5)
{
  c = pow(a,b) +
      pow(b,a);
  cout << c;
  i++;
}
```

⇒

```
int a=2, b=3, c;
c = pow(a,b) + pow(b,a);
while (i<5)
{
  cout << c;
  i++;
}
```

Q.4 Explain different code generation issues with example.

→

Following issues arises during the code generation phase:

1) Input to code generator
2) Target program
3) Memory management
4) Instruction selection
5) Register allocation
6) Evaluation order

1) Input to code generator =>

- The input to the code generation consists of the intermediate representation of the source program produced by front end, together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in IR.

2) Target program =>

- The output of the code generator is the target program.
- The output may be:

a) Absolute machine language =

- It can be placed in a fixed memory location and can be executed immediately.

b) Relocatable machine language =

- It allows subprograms to be compiled separately.

c) Assembly language =

- Code generation is made easier.

3) Memory management =>

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol table entry for the name.
- Labels in three-address stmt have to be converted to addresses of instructions.

4) Instruction Selection =>

- The instructions of target machine should be complete and uniform.

- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of code is determined by it's speed and size.

5) Register allocation ⇒
The set of variables that will reside in registers at a point in the program is selected.

6) Evaluation order ⇒
The order in which the computations are performed can affect the efficiency of the target code.

Q.5] Explain labeling algorithm with example?

→

- Labeling algorithm is used by compiler during code generation phase.
- Basically, this algorithm is used to find out how many registers will be required by a program to complete it's execution.
- Labeling algorithm works is bottom up fashion.
- We will start labeling firstly child nodes & then interior nodes.
- Rules of labeling algorithm are:

1) If 'n' is a leaf node -
   a) If 'n' is left child then it's value is 1.
   b) If 'n' is right " " is 0.

2) If 'n' is an interior node -

Lets assume L1 and L2 are left and right child of interior node respectively.

a) If L1 == L2 then value of 'n' is L1 + 1 or L2 + 1.

b) If L1 != L2 then value of 'n' is MAX(L1, L2)

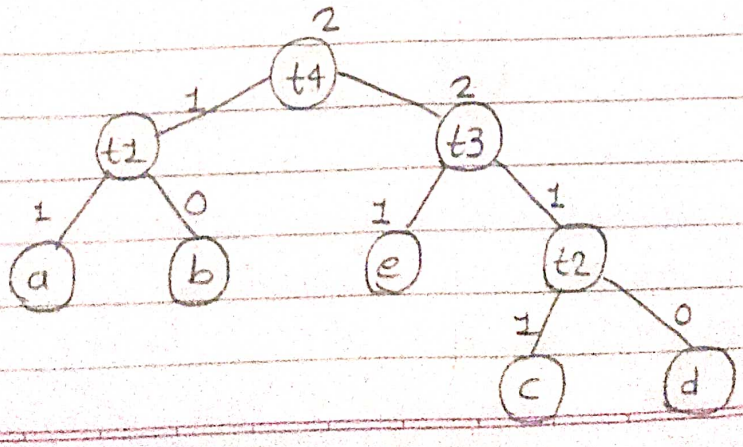
The labelling Algorithm =>

```
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 =

For binary interior nodes:

$$label(n) = \begin{cases} max(l1, l2), & \text{if } l1 \neq l2 \\ l1 + 1, & \text{if } l1 = l2 \end{cases}$$

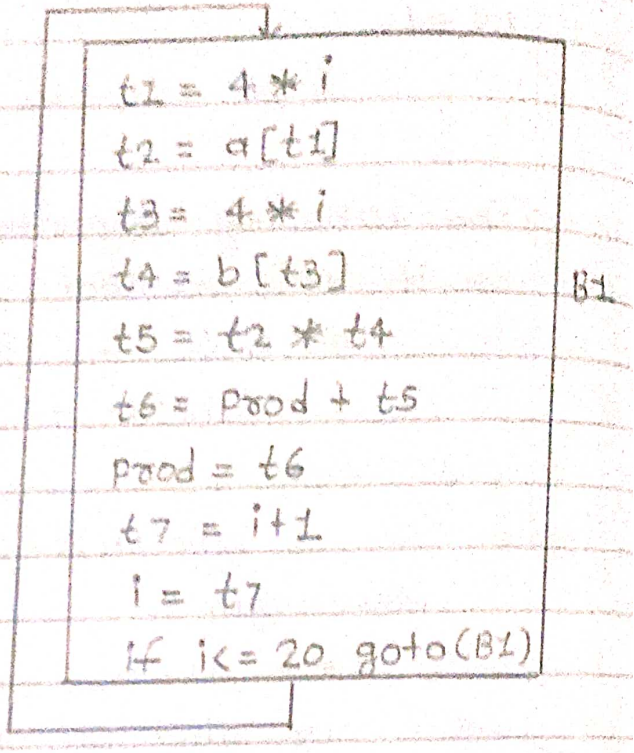


Q.6] For the following three address statements shown below, construct flow graph & DAG :-

1) t1 = 4 * i
2) t2 = a[t1]
3) t3 = 4 * i
4) t4 = b[t3]
5) t5 = t2 * t4
6) t6 = prod + t5
7) prod = t6
8) t7 = i+1
9) i = t7
10) if i<=20 goto (1)

→

CFA =



DAG =