End Term Examination
2017

Q1 Attempt any five:-

(a) Explain the process of bootstraping in compiler design with example.

(a) Bootstrapping is the technique for producing a self-compiling compiler i.e. compiler written in source program language that it intends to compile.

Example:- Let say you have to write a new programming language named "XYZ". The first step is to write a compiler for this language. Since, this new language doesn't ~~exist~~ exist yet, you write the compiler in (say java). We call it 'jxyz'.

Now, we have a java program (jxyz) that takes an XYZ source file and produces an executable. It is then possible to undertake writing a compiler for XYZ in XYZ that is compiled by jxyz.

At this point, we have a compiler for XYZ that was compiled with jxyz. Let's call this 'xyzFromJ'.

'xyzFromJ' should be able to take itself as input and compile itself completely removing anything created by jxyz.

(b) Differentiate between SDD & SDT.

Syntax Directed Defination (SDD)

A syntax directed defination generalizes a context-free grammer by associating a set of attributes with each node in a parse tree. Each attribute gives some information about the node.

for example:- $X \rightarrow YZ$

Suppose X, Y, Z have associated attributes X.a, Y.a & Z.a respectively. If the semantic rule $\{X.a = Y.a + Z.a\}$ is associated with $X \rightarrow YZ$ then parser adds 'a' to X, Y & Z.

Syntax Directed Translation (SDT)

It is a notation in which each CFG is associated with some set of semantic rules.

For example:-

$S \rightarrow AB$ {print ("1")}
$A \rightarrow b$ {print ("2")}
$B \rightarrow c$ {print ("3")}

CFG + Semantic Rules = SDT.

It is used to add attributes with grammer symbol and semantic rules with productions to make the translation of constructs easy.

(c) What is Left Recursion & Left Factoring?

Left Recursion is the property of grammer when the left most non terminal in a production of a non terminal is the non terminal itself. Example:-

$A \rightarrow Aq$ is left recursion.

to remove left recursion what we do is :-

$A \rightarrow Bq$
$B \rightarrow Ar$

left Recursion should be removed during top down parsing

left factoring is a grammar translation technique. It is removing the common left factor that appears in two productions of same non terminal.

Ex:- $A \to qB / qC$

Solu^{n}:- $A \to qD$
$D \to B/C$

(d) What is backpatching?

When we generate the code for Boolean expressions and flow-of-control statement in a single pass then we face the problem that, we may not know the labels that control must go to at the time the jump statements are made. We can overcome this problem by generating a series of branching statements with the targets of the jumps temporarily left unspecified. Each such statement will be put on a list of goto statements whose label will be filled in when the proper label can be determined. This is called BACKPATCHING.

There are 3 functions that are used:-

① Makelist(i):- creates new list containing only i.

② Merge(q_1, q_2):- takes the list pointed to by q_1 & q_2 concatenates them into one list, and returns pointer to the list.

(e) What is the process of identifying basic blocks in code optimization?

Optimization of basic blocks is done by:-

(1) Common subexpression elimination:- Common subexpressions are not computed again & are eliminated.

Ex:- $A = B + C + D$
$E = B + C + M$

$\longrightarrow$

$T_1 = B + C$
$A = T_1 + D$
$E = T_1 + M$

(2) Deadlock Elimination :- Deadcode is useless code which can be removed from the program.

Ex:-
```
a = 1;                    a = 1;
if (a == 2)      ⟶        :
{  a = a+2;               :
}
:
```

(3) Constant folding :- Any constant that is stored as variable a is being used in a statement that produces static result is removed. Example:

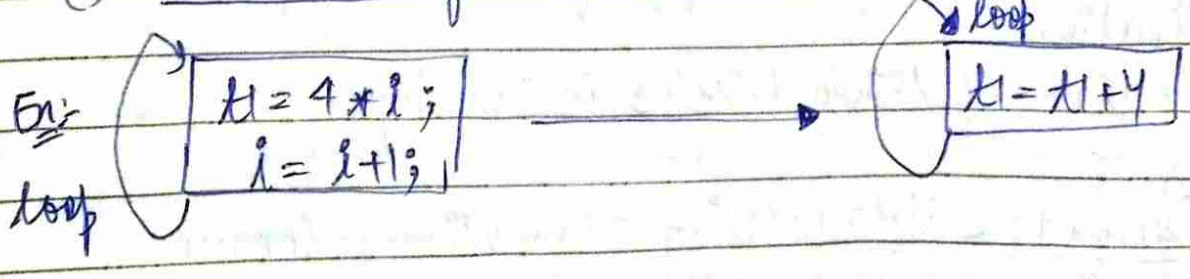
```
const a = 5;        ⟶    b = 7;
b = a+2;
```

(4) Code Motion :- Moving the code out of loop

Ex:-
```
a=1; b=2;                      a=1; b=2;
for (i=1; i<=10; i++)          x = a*b;
{   x = a*b;          ⟶       for (i=1; i<=10; i++)
    sum = sum+i;               {  sum = sum + i;
}                              }
```

(5) Elimination of Induction Variable :-

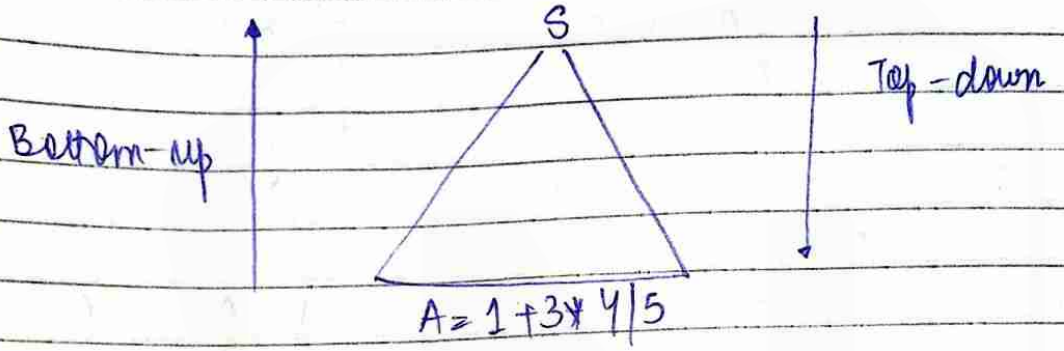
Ex:-



(f) Differentiate between Top Down & Bottom up parsing

- Top down parsers starts constructing the parse tree at the top (root) of the parse tree and move down towards the leaves.

They are easy to implement by hand, but work with restricted grammer. Example: Predictive Parsers i.e. LL(k) parsers.



- Bottom up Parsers build the nodes on the bottom of the parse tree first. It is suitable for automatic parser generation.

It handles a larger class of grammer. Example: Shift Reduce Parser

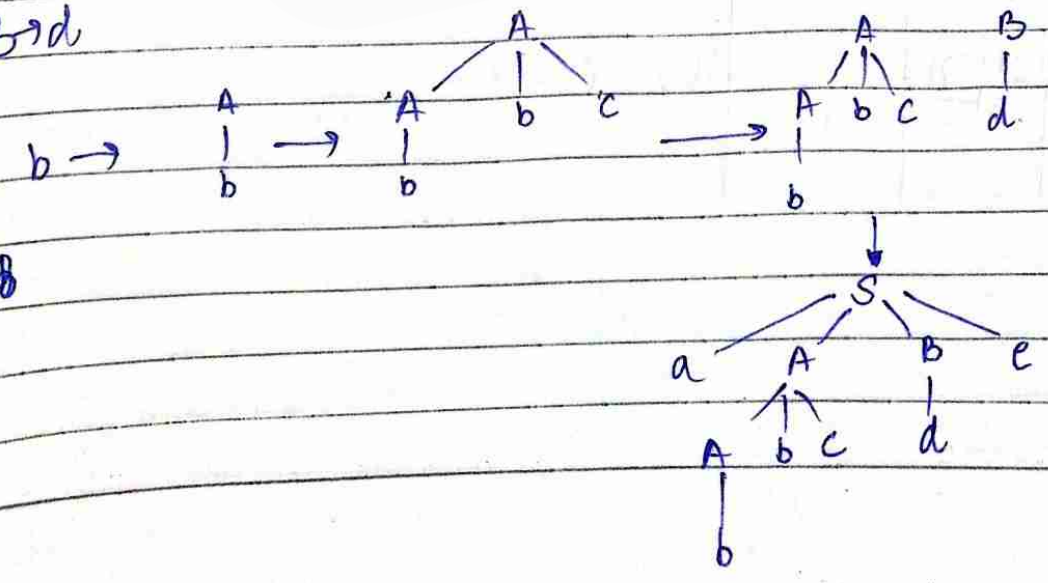
Ex:- Bottom up Parsing

S → aABe

A → Abc

A → b

B → d



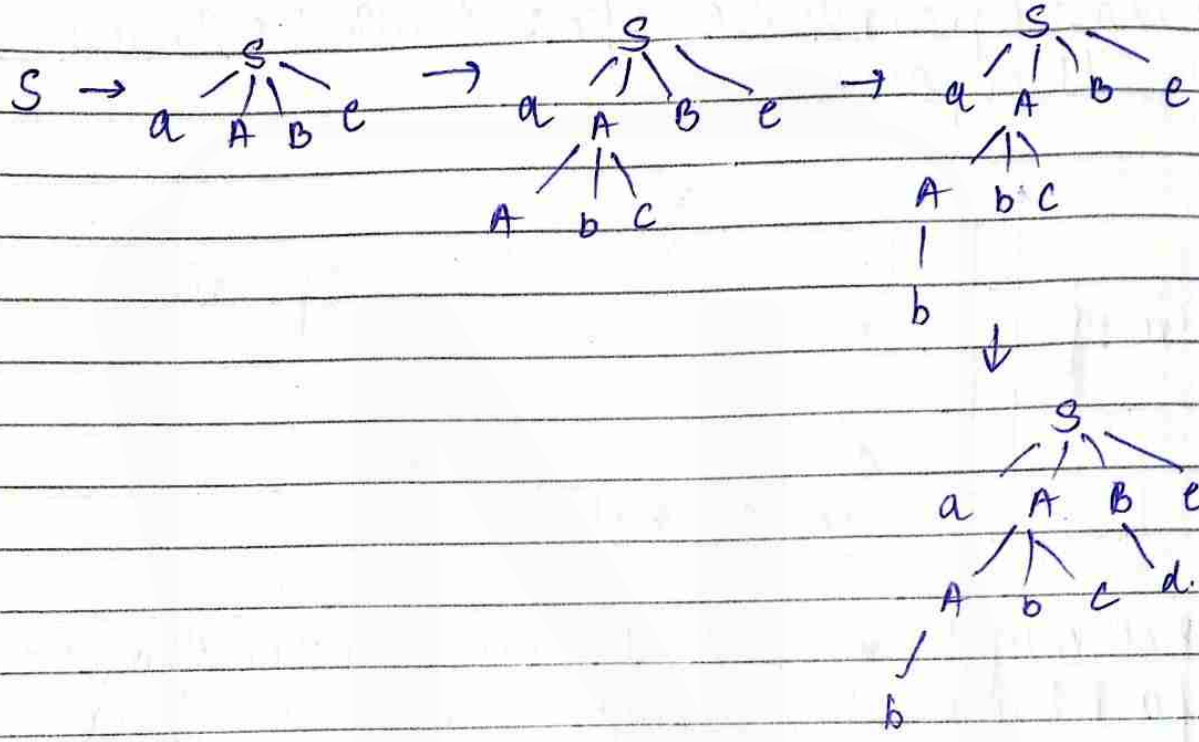
Top Down Parsing Example:-

$S \rightarrow aABe$

$A \rightarrow Abc$

$A \rightarrow b$

$B \rightarrow d$



(g) Write a SDT for converting infix to postfix expression with suitable example.

SDT helps in converting infix to postfix expressions. It increases efficiency and arithmetic expressions are solved easily.

Example:- $a = (b * \ominus c) + d$

Postfix:- $a\ b\ c\ \ominus\ *\ d\ +\ =$

Q2 (a) Construct LL(1) parsing table for

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) / id$

(1) Calculating First & Follow Functions

	First	Follow
E	(, id	$,)
E'	+, ε	$,)
T	(, id	+, $,)
T'	*, ε	+, $,)
F	(, id	+, *, $,)

LL(1)

(2) Passing Table :

Non Terminal	id	+	*	(	)	$
E	$E \to TE'$		~~$E \to TE'$~~	$E \to TE'$		
E'		$E' \to +TE'$		~~$E' \to \varepsilon$~~	$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

Q2

(b) Check whether grammer is LL(1) or not

1) $S \to A/a$
$A \to a$

2) $S \to aSA/\varepsilon$
$A \to c/\varepsilon$

(1) Calculating First & Follow functions

$F_i(S) = a$ $\quad F_o(S) = \$$

$F_i(A) = a$ $\quad F_o(A) = \$$

2) Parsing Table

	a	$
S	$S \to a$	
A	$A \to a$	

Since, there is no multiple entries in LL(1) parsing table. Grammar is LL(1)

(2) $S \to aSA/\varepsilon$
$A \to c/\varepsilon$

	First	Follow
S	a, ε	$, c, ε
A	c, ε	$, c, ε

Constructing Parsing Table,

	a	c	$
S	$S \to aSA$	$S \to \varepsilon$	$S \to \varepsilon$
A		$A \to c$ $A \to \varepsilon$	$A \to \varepsilon$

∴ There are two production $A \to c$, $A \to \varepsilon$ in one box, grammer is not LL(1).

Q3

(a) What do you mean by handle? Check whether grammar

$E \to E+T \mid T$

$T \to a$ or id

is LR(0) or not.

A "handle" of a string is a substring that matches RHS of a production and whose reduction to non-terminal (or LHS of production) represents one step along the reverse of a rightmost derivation ~~tow~~ towards reducing to the start symbol.

Example

If $S \to^* aAw \to^* a\beta w$

then $A \to \beta$ is a handle of $a\beta w$

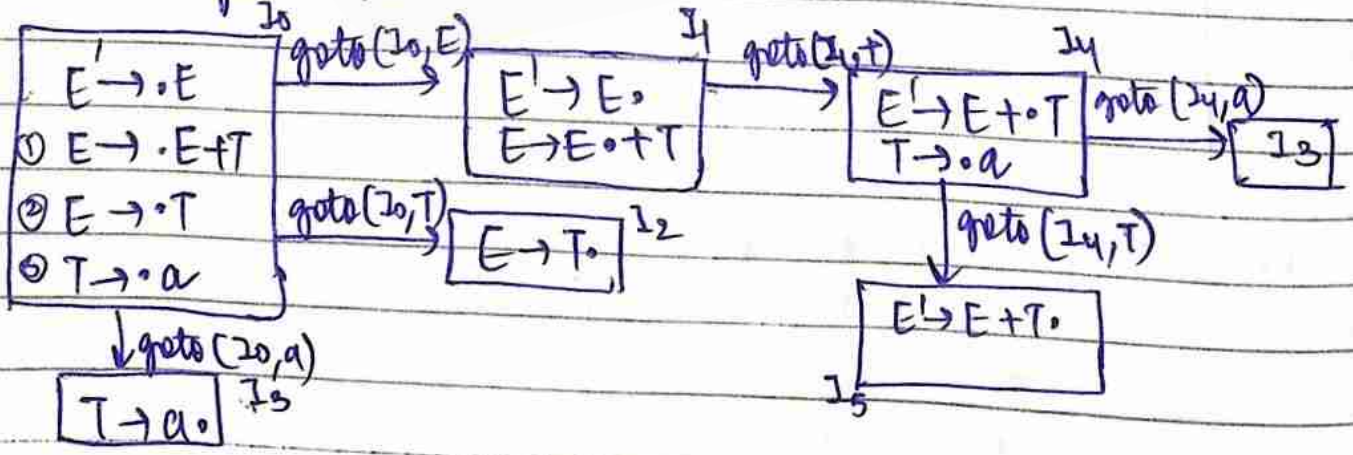
- Augmented Grammar:-

$E' \to E$

$E \to E+T \mid T$

$T \to a$

- Constructing LR(0) items.



Q3 Constructing parsing table

States	Actions			Goto	
	+	a	$	E	T
I0		S3		1	2
I1	S4		Accept		
I2	r2	r2	r2		
I3	r3	r3	r3		
I4	r3	S3/r3	r3		5
I5	r1	r1	r1		

There is SR conflict
∴ It is not LR(0)

Unit-2

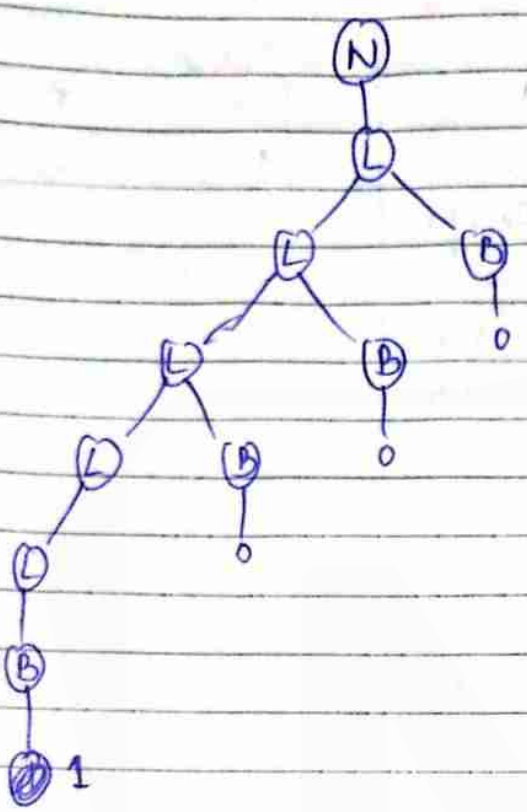
Q4 (a) Write an SDT to count the number of binary digits in a binary number.

Defining grammer:-
Binary no. 'B' can be 1 or 0.
Number can be a list of binaries.
List can recursively define bits 'B'

$N \rightarrow L$
$L \rightarrow LB$
$L \rightarrow B$
$B \rightarrow 0$
$B \rightarrow 1$

Let Binary string be 1000.

Parse Tree for 1000



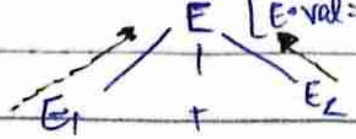
$B \to 0$	$\{B.val = 0\}$
$B \to 1$	$\{B.val = 1\}$
$N \to L$	$\{N.val = L.val\}$
$L \to LB$	$\{L.val = L.val + B.val\}$
$L \to B$	$\{L.val = B.val\}$

Q4 (b) Differentiate between S-attributed & L-attributed SDTs. Write the steps to create SDT for any problem & write SDT for converting any number from binary to decimal.

Synthesized Attributes

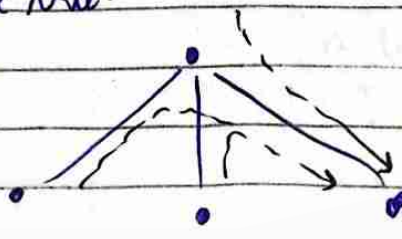
An attribute at a node is synthesized if its value is computed from the attributed values of the children of that node in the parse tree.

Example:- $E \to E_1 + E_2$ $\{E.val = E_1.val + E_2.val\}$



Inherited Attributes :-

An attribute at a node is inherited if its value is computed from attribute values at the siblings and/or parent of that node in parse tree.



~~Inherited~~ Inherited Attributes at this node.

Steps to create SDT :-

1. Write every production available.
2. Create annotated Parse Tree
3. Add semantic attributes to the production.

SDT for converting binary to decimal

Grammer :- "B" can be 0 or 1, ~~Number can~~ $B \to 0$

$B \to 1$

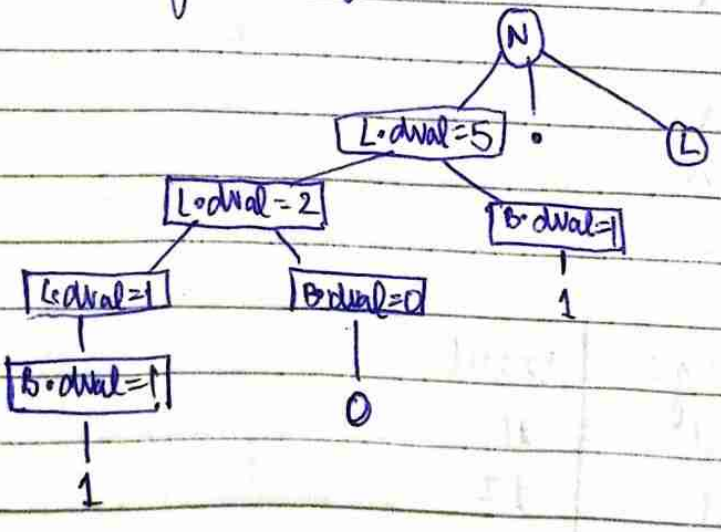
Number can be list of binary digits, $N \to L$

list can recursively define bits, $L \to LB$, $L \to B$

A binary number can be with or without decimal, $N \to L_1 \cdot L_2$

Lets take a small string & build parse tree.

Eg :- Number without decimal :- 101.101 taking 101 i.e. left subtree



$N \to L_1 . L_2$	{ }
$N \to L$	{N.dval = L.dval}
$L \to LB$	{L.dval = L.dval * 2 + B.dval}
$L \to B$	{L.dval = B.dval}
$B \to 0$	{B.dval = 0}
$B \to 1$	{B.dval = 1}

Q5

(a) What do you mean by 3 address code? Explain how the 3 address code is represented in quadruples, triples & Indirect triples with examples.

Three address code is the representation of intermediate code generation step in the stage of compilation.

Atmost 3 addresses are used to represent any statement

ex:- X = Y operator Z

Result, operator 1, operator 2

i.e. all these variables are stored in memory has an associated addresses along with it.

There are three types of representations:-

① Quadruples:

Quadruple is a record structure with four fields.

example:- a = b + c * d

In 3 address code,

t1 = c * d

t2 = t1 + b

a = t2

	Operator	Arg 1	Arg 2	Result
(0)	*	c	d	t1
(1)	+	b	t1	t2
(2)	=	t2		a

② Triples:-
It contains 3 address fields
Here, temporary values are computed by position in statement
Eg:- a = b + c * d.

	operator	Arg 1	Arg 2
(0)	*	c	d
(1)	+	b	(0)
(2)	=	a	(1)

③ Indirect Triples:-
Here, rather than listing triples, these pointers are listed.

	Statement
(0)	(14)
(1)	(12)
(2)	(13)

	operator	arg 1	arg 2
(14)	*	c	d
(12)	+	b	(14)
(13)	=	a	(12)

Q5
(b) Write 3 address code for
i) while (a<5) do a = b + 2
ii) - a(a+b) * (c+d) + (a+b+c)

i) ① if a<5
② t1 = b+2
③ a = T1

ii) t1 = a+b
t2 = uminus (a)
t3 = c+d
t4 = t6 * t3
t5 = t1 + c
t6 = t2 - t1
t7 = t4 + t5

Q:

(a) What do you mean by symbol table? Write an example to show how different phases of compiler interact

Specific keywords / commands like 'ls', 'cat' or 'exit' are stored by system.

For a program to process such input, it must have some way to associate such symbolic names with their abstract values. Mechanisms which maintain such associations are are called symbol table.

Various phases of compiler:-

① Lexical Analysis:- Create new table entries in table.

② Syntax Analysis:- Adds information regarding attribute type, scope, dimension etc.

③ Semantic Analysis:- Uses available information in table to check for semantics i.e. verify that expressions & assignments are semantically correct (type checking)

④ Intermediate Code Generation:- Refers symbol table ~~to check for semantics~~ for knowing how much & what type of run-time is allocated & table helps in adding temporary value info.

⑤ Code Optimization:- Uses information present in symbol table for machine dependent optimization.

⑥ Target code generation:- Generates code by using address information of identifier in table.

Q6

(b) How is data stored in symbol Table for block & non block languages?

① List:

- In this method, an array is used to store names & associated information.
- A pointer "available" is maintained at end of all stored records and new names are added in order as they arrive.
- To search for a name, we start from beginning of list till available pointer and if not found we get an error "use of undeclared name".
- While inserting a new ~~from~~ name we must ensure that it is not present otherwise error occurs i.e. "multiplied defined name"
- Advantage is that it takes minimum amount of space.

② Linked List:

- A link field is added to each record.
- Searching of names is done in order pointed by of link field
- A pointer "First" is maintained to point to first record of symbol table.
- Insert in fast O(1)

③ Hash Table:-

- In hashing scheme two tables are maintained - a hash table and symbol table and is the most commonly used method to implement symbol tables
- A hash table is an array with index range: 0 to tablesize - 1. These entries are pointer pointing to names of symbol table
- To search for a name we use hash function that will result in any integer between 0 to tablesize - 1.

④ Binary Search Tree-

- Another approach to implement symbol table is used to binary search tree i.e. we add two link fields i.e. left & right child.
- All names are created as child of root node that always follow the property of binary tree.
- Insertion and lookup are $O(\log_2 n)$ on average.

Q7

(a) What are different types of errors that occur during lexical, syntactic & semantic phase.

Generally, errors in programs are detected at different levels:

(1) At lexical analysis:
Unrecognized group of characters like & rabcetc which cannot be an identifier nor keyword

(2) At syntax analysis:
missing operator / operands in expressions.

(3) At semantic analysis:
Incompatible types of operands to an operator.

④ Logical errors:
Infinite loop.

To correct such errors 4 different recovery strategies are used.

① Panic mode: On an error, parser has to skip input symbol

one at a time until one of the designated synchronizing tokens is found.

(3) Phase Level Recovery:-
On detecting an error, the passer may perform some local corrections on the remaining input. It could be replacing an incorrect character with correct one or swapping two adjacent characters such that passer can continue the process.

(4) Error Productions:-
If the compiler designer has a good idea about possible errors, then he can add rules with the grammer of programming language to handle erroneous constructs.

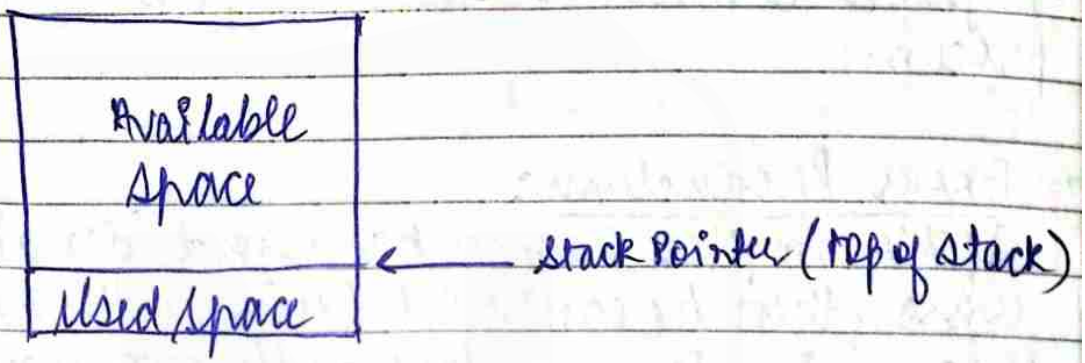
(5) Global Correction:-
There are algorithms to obtain a globally least cost corrections that help in choosing a minimal sequence of changes.

Q7
(b) What are the storage allocation strategies in the runtime environment of compiler.
Storage allocation strategies depends upon the programming language but can be classified in 3 ways:-

(1) Static allocation:-
* Storage space allocated once was never released.
* Early language like Fortran had static allocation
* It is efficient because no time or space is expended for storage management during execution.
* Incompatible with recursive program calls.

(2) Stack Based Storage Management

* Begins at one end.
* Storage must be freed in reverse order of allocation so block allocated space recently must free the space first.
* Only a singe stack pointer is all that is needed to control storage management



(3) Heap Storage Management:-

* Heap is used for storage of values which may require to be accessible from the time the storage is allocated until program terminates.
* It can shrink & grow dynamically.

Unit-4

Q8

(a) What do you mean by code optimization? What is term leader? Write algorithm to identify out the basic blocks.

Code Optimization aims mainly at the arranging the computation in away in a program so as to gain the advantage of execution speed, without changing the meaning of a program.

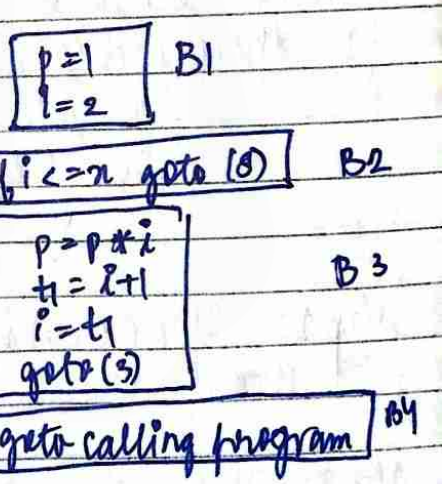
Algorithm to find basic blocks:

Step 1:- Determine the set of leaders, that is:
(a) First instruction of function is header
(b) Target of conditional or unconditional goto.
(c) Conditional or unconditional goto.

Step 2:- For each header, its basic block consists of itself & all instruction upto, but not including next header.

Eg:-
```
exp(x)
{
  int p=1;
  for(i=2; i<=x; i++)
    p=p*i
  return(p);
}
```

(1) p = 1
(2) i = 2
(3) if i <= x goto (8)
(4) p = p * i
(5) t1 = i + 1
(6) i = t1
(7) goto (3)
(8) goto caller program.



Leaders are:-
i) (1), because its first instruction
ii) (3), since its target is goto
iii) (4), since it follows goto
iv) (8), since it targets a goto.

Q:
(b) Identify basic blocks & construct ~~graph~~ DAG

```
main()
{
    int i=0, n=10;
    int a[n];
    while (i<=(n-1))
    {
        a[i]=i*4
        i=i+1;
    }
    return;
}
```

DAG Construction:-
function node(id) returns most recently created node associated with its id.
for three address statement, steps are:-

$x = y \; OP \; z$
$x = OP \; y$
$x = y$

Step 1:- If node(y) is undefined, create a leaf ~~node~~ labelled y, ~~and~~ if node(z) is undefined, create a leaf labelled z.

Step 2:- If a node exists i.e. labelled OP whose left child is node(y) and whose right child is node(z), then return this node. Otherwise create a node & return it. If the statement is of form $x = OP \; y$, then check if a node exist that is labelled OP whose only child is node(y). Return this node.

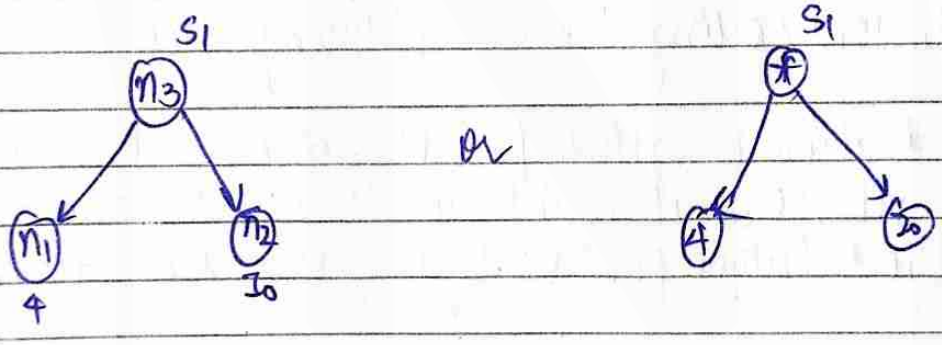
Step 3:- Append x to the list of identifiers for the node n returned in Step(2). Delete 'x' from the list of attached

Identifiers for node(n), and set node(n) to be node n.

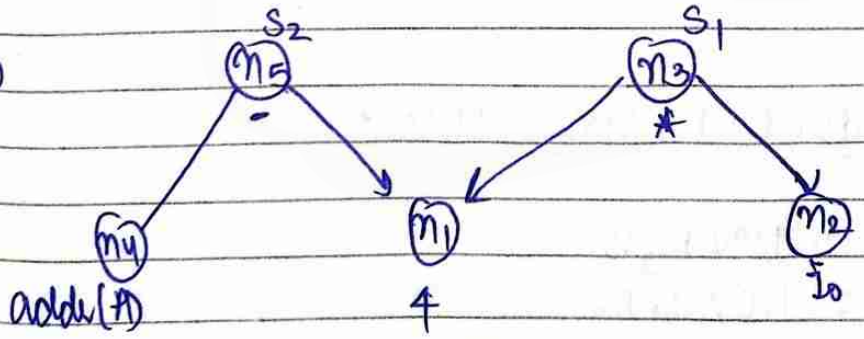
Example:-

(1)
$S_1 = 4 * i$
$S_2 = addr(A) - 4$
$S_3 = S_2[S_1]$
$S_4 = 4 * i$
$S_5 = addr(B) - 4$
$S_6 = S_5[S_4]$
$S_7 = S_3 * S_6$
$S_8 = prod + S_7$
$prod = S_8$
$S_9 = i + 1$
$i = S_9$
if $i <= 20$ goto (1)

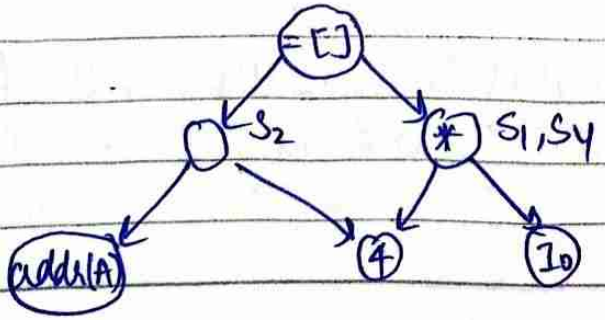
(1)



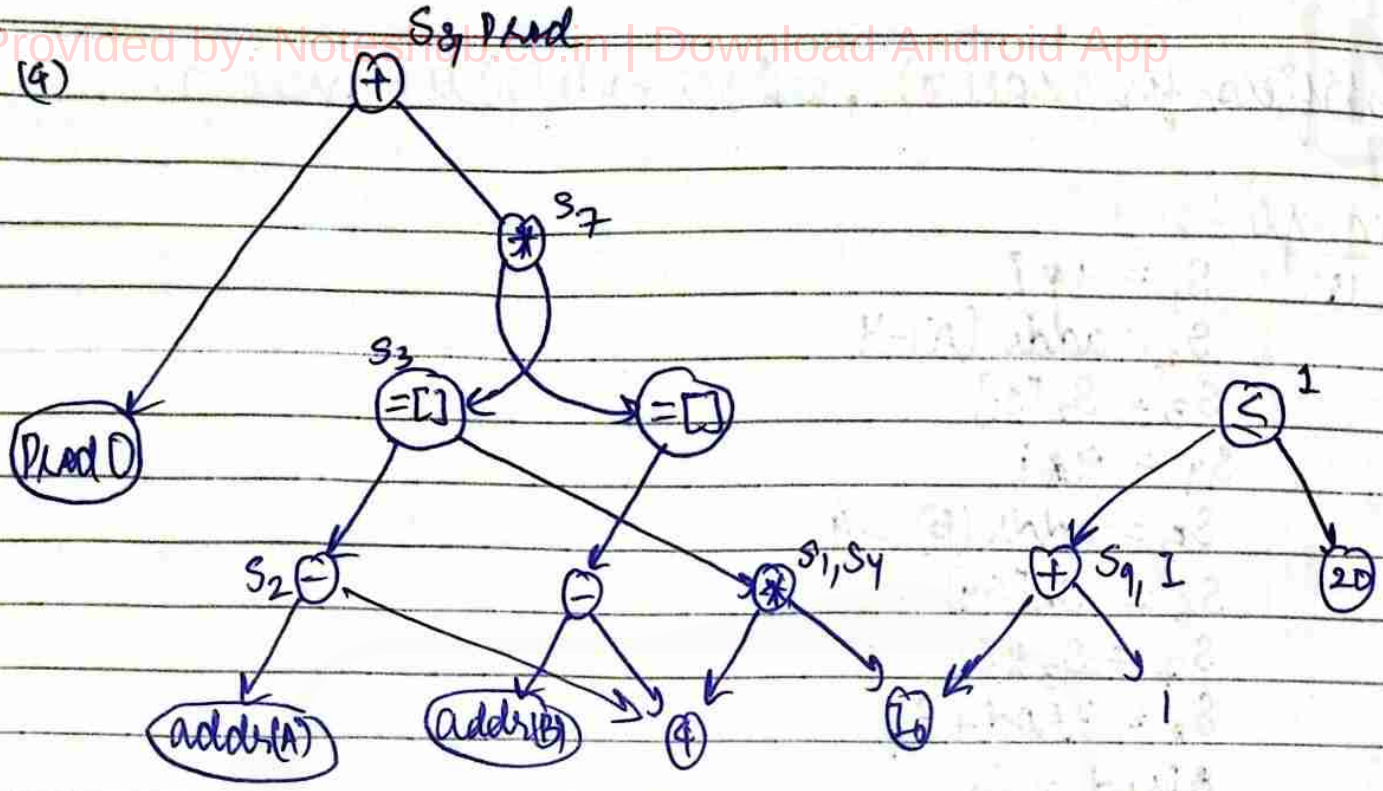
(2)



(3)



(a)



Q9 (a) What do you mean by peehole optimization.

Peehole optimization technique is used by many compilers to optimize their intermediate or object code.
This helps in controlling the errors during SDT.

This optimization is called peehole since in it we look for a small segment of intermediate code at time. It is an effective technique for locally improving the target code.

This is done as:-

(i) Eliminating Redundant loads & Stores:

Instruct^n sequence:- 1. MOV R, x
2. MOV x, R.

We can delete 2nd record if it is unlabelled as, the first instruct^n ensures that value of x is in Register R already.

(ii) Unreachable Code Elimination:-
an unlabelled instruction that immediately follows an unconditional jump can be removed.

(iii) Eliminating Multiple Jumps:-
If we have jump to other jumps, then unnecessary jumps can be eliminated

Ex:- goto L_1 ... L_1: goto L_2 → goto L_2 ... L_1: goto L_2.

Q9 (b) What are the major issues during code generation process?

(1) Memory management:-
During code generation process references to symbol table entries have to be mapped to actual addresses and levels of instructions.
Mapping names in the source program to address of data is done in pass 1 (Frontend) and pass 2 (Code Generator).

(2) Instruction Selection:-
The nature of instruction set of the target machine determines selection.
Peculiarities in target instruction set have to be taken into account.
If we don't care about efficiency of target program, selection is straight forward

3) Register Allocation

Register can be accessed faster than memory words. Frequently accessed variables should reside in registers. Register assignment is picking a specific register for each such variable.

4) Storage Allocation:

Delay decision about storage allocation until final code generation, when the width of each type is known :-

- Initalize location field of each identifier/temporary to special value UNDEF.

- Maintain a counter that gives the displacement of next available slot in stack frame.

- During code generation, if there is a reference to a symbol table entry whose location is = UNDEF, then set the value using the counter before generating any code.

Q3
(b) construct LR(1) parsing table for

$S \to Aa$
$S \to bAc$
$S \to Bc$
$S \to bBa$
$A \to d$
$B \to d$

Step 1:- Augmented Grammer

$S' \to S$
$S \to Aa$
$S \to bAc$
$S \to Bc$
$S \to bBa$
$A \to d$
$B \to d$

Step 2:- Find Closure & goto

I_0
$S' \to \cdot S, \$$
$S \to \cdot Aa, \$$
$S \to \cdot bAc, \$$
$S \to \cdot Bc, \$$
$S \to \cdot bBa, \$$
$A \to \cdot d, a$
$B \to \cdot d, c$

I_1
goto (I_0, S)
$S' \to S \cdot, \$$

I_2
goto (I_0, A)
$S \to A \cdot a, \$$

I_3
goto (I_0, b)
$S \to b \cdot Ac, \$$
$S \to b \cdot Ba, \$$
$A \to \cdot d, c$
$B \to \cdot d, c$

I_4
goto (I_0, B)
$S \to B \cdot c, \$$

I_5
goto (I_0, d)
$A \to d\cdot, a$
$B \to d\cdot, c$

I_6
goto (I_2, a)
$S \to Aa\cdot, \$$

I_7
goto (I_3, A)
$S \to bA\cdot C, \$$

I_8
goto (I_3, B)
$S \to bB\cdot a, \$$

I_9
goto (I_3, d)
$A \to d\cdot, a$
$B \to d\cdot, c$

I_{10}
goto (I_4, C)
$S \to Bc\cdot, \$$

I_{11}
goto (I_7, C)
$S \to bAc\cdot, \$$

I_{12}
goto (I_8, a)
$S \to bBa\cdot, \$$

LR(1) Parsing Table

State	action					goto		
	a	b	c	d	$	S	A	B
I_0		S3		S5		1	2	4
I_1				~~Accept~~	Accept			
I_2	S6							
I_3				S9			7	8
I_4			S10					
I_5	r5		r6					
I_6					r1			
I_7			S11					
I_8	S12							
I_9	r6		r5					
I_{10}					r3			
I_{11}					r2			
I_{12}					r4			

The Grammer is LR(1).