Answer All the Section

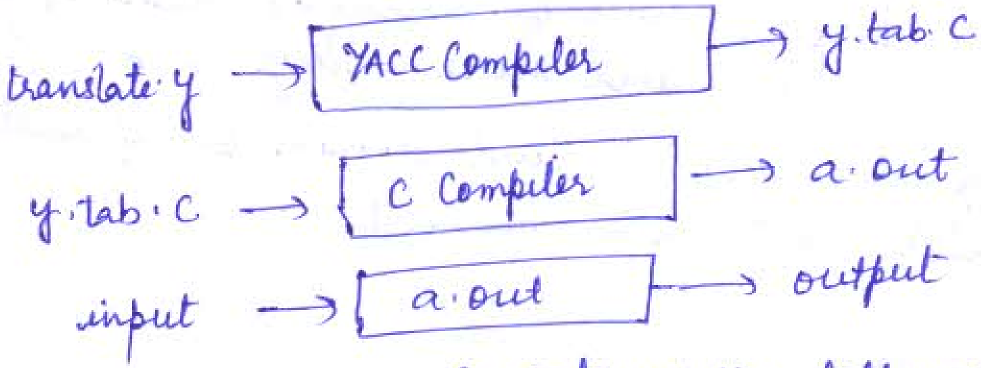
Section A

Q.1 Attempt all parts. All parts carry equal Marks. Write answer of each part in short (2×10 = 20)

(a) How YACC can be used to generate parser?

Answer → YACC stands for "Yet another compiler-compiler". It is an LALR parser generator which is basically available as a command on UNIX system. It is a tool that compiles a source program and translates it into a C program that implements the parser.

An input/output translator constructed using YACC is given as.



A YACC source program consists of the following three parts:

declaration // ordinary C declarations delimited by %{ and %}
declaration of grammar tokens

%% transition rules // includes the grammar productions along with their semantic actions

%% supporting C routines // includes the lexical Analyzer yylex() that produces pair consisting of tokens name & their associated values

(b) Discubs the challenges in compiler design.

Answer:- Challenges in Compiler Design -

(1) They are often easier to learn, read and undesstand. If a feature is hard to compile, it may well be hard to understand.

(2) They will have quality compilers on a wide variety of machines. This fact is often crucial to a language's success.

(3) Often, better code will occur. If a compiler was does not fully understand a language, how can they produce a sound compiler?

(4) The compiler will be smaller, cheaper, faster more reliable and more widely used.

(5) Compiler diagonistics and program development features will often be better.

(c) Discuss various compiler writing Tools.

Answer:-

Parser Generators :- These produce syntax analyzers, normally from input that is based on a content free grammar. In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.

Many parser generators utilize powerful parsing algorithms that are too complex to be carried out by hand.

Scanner Generators → These automatically generate lexical analyzers, normally from a specification based on regular expressions.

Syntax-Directed translation engines - These produce collections of routines that walk the parse tree.

Automatic Code generators - Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.

Data-flow engines - Much of the information needed to perform good code optimization involves "data-flow analysis", the gathering of information about how values are transmitted from one part of a program to each other part.

(d) Discuss the role of Macros in programming languages?

Answer:-

Many assembly (and programming) languages provide a "macro" facility whereby a macro statement will translate into a sequence of assembly language statements & perhaps other macro statements before being translated into machine code. Thus, a macro facility is a text replacement capability.

Macro Definition

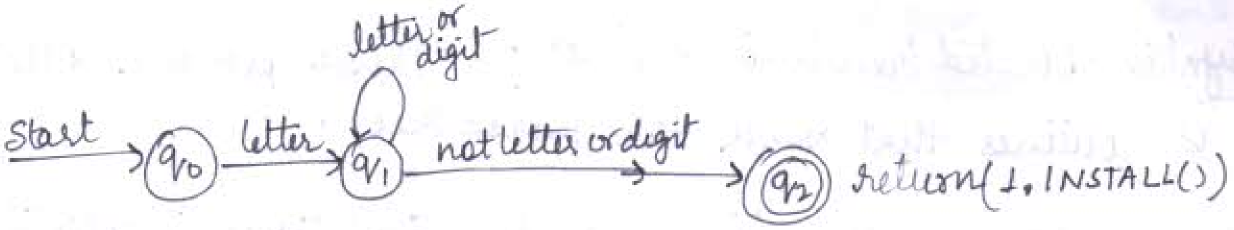
```
MACRO   ADD2   X,Y
        LOAD   Y
        ADD    X
        STORE  Y

END MACRO
```

There are two aspects ~~of~~ to Macros → definitions and use.

(e) Draw the transition diagram for identifiers?

Answer →



(f) Differentiate between dynamic loaders and linkers.

Answer →

Loader → Loader is a program that performs the function of loading and linkage editing. The process of loading consists of taking re-locatable machine code, altering the re-locatable address and placing the altered instructions and data in memory at the proper location.

Linker → Linker is a program that combines (multiple) objects files to make an executable. It converts name of variables and functions to members (machine addresses).

(g) Describe languages denoted by the following regular expressions: $(1+0)^*$.

Answer:- terminals → 1, 0

nonterminal → S

productions → S → 1S

S → 0S

S → 1/0

S → ε

It's CFL is $(1+0)^*$

(h) Write the prefix and postfix expression for $A = (20+(-5)*6+12)$

<u>Answer:-</u>

$$A = (20+(-5)*6+12)$$

Prifix expression → ++ 20 * -5 6 12

Postfix expression → 20 5- 6 * + 12 +

(i) What are the code optimization Techniques ?

<u>Answer:-</u> Code optimization techniques are generally applied after syntax analysis usually both before and during code generation. The techniques consist of detecting patterns in the program and replacing these patterns by equivalent but more efficient constructs. These patterns may be local or global, and the replacement strategy may be machine- dependent or machine- independent.

(j) Show that the following grammar is unambiguous:

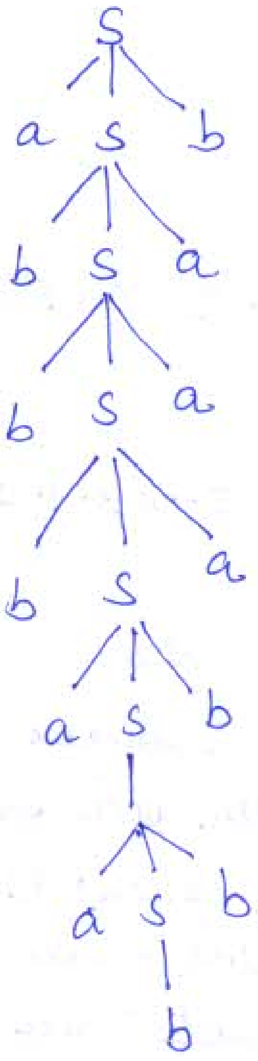
S → aSb | bSa | b

For a string abbbaabbbaaab draw a parse tree.

<u>Answer→</u> For a given grammar

S→ aSb / bSa / b

there is only one parse tree is generated for given string, so this grammar is unambiguous for given string.



(Parse tree)

ottag

SECTION - B

Q:-2 What are different ways to write three address code? Write the three address code. for the following code segment.

```
        switch a+b
      { Case 1: x=x+1
        Case 2: y=y+2
        Case 3: z=z+3
        default: c=c-1
      }
```

Ans:- The three address codes are of different types -

1. Assignment statement of the form
   (a) A = B op c (b) A = op B and (c) A = B
2. The unconditional jump gotoL means instruction to execute Lth three address statement.
3. Conditional jump such as if A<B gotoL. if the condition is true then execute Lth statement else next sequencial statement must be executed.
4. Param A, and call P,n. used for writing three address code for procedure call. like

```
param $A_1$
param $A_2$
   ,
   ,
   ,
param An
 Call P,n
```

where A_1 - - A_n is arguments.

5. Indexed assignment of the form A:=B[I] and A[I]:=B. The first of these sets A to value in the location I memory units beyond location B. Second sets the location I units beyond A to the value of B'

6. Address and pointer assignments of the form A:=addr B, A:=*B and *A=B.

The three address code for the problem

```
switch a+b
{ case1: x=x+1
  case 2: y=y+2
  case 3: z=z+3
  default: c=c-1
}
```

is given as:

```
① T1=a+b
② goto 15
③ T2=x+1
④ x=T2
⑤ goto 19
⑥ T2=y+2
⑦ y=T2
⑧ goto 19
⑨ T2=z+3
⑩ z=T2
⑪ goto 19
⑫ T2=c-1
⑬ c=T2
⑭ goto 19
⑮ if T=1 goto 3
⑯ if T=2 goto 6
⑰ if T=3 goto 9
⑱ goto 12
⑲ exit.
```

(3) Given the algorithm for computing precedence function Consider the following operator precedence matrix draw precedence graph and compute the precedence function:-

	a	(	)	;	\$
a			>	>	>
(	<	<	=	<	
)			>	>	>
;	<	<	>	>	
\$	<·	<			

Answer:→ Algorithm for computing precedence function :→

Input :→ An operator precedence Matrix

output → Precedence functions representing the input matrix, or an indication that none exist.

Method :→

1 - create symbols f_a and g_a for each a that is a terminal or \$.

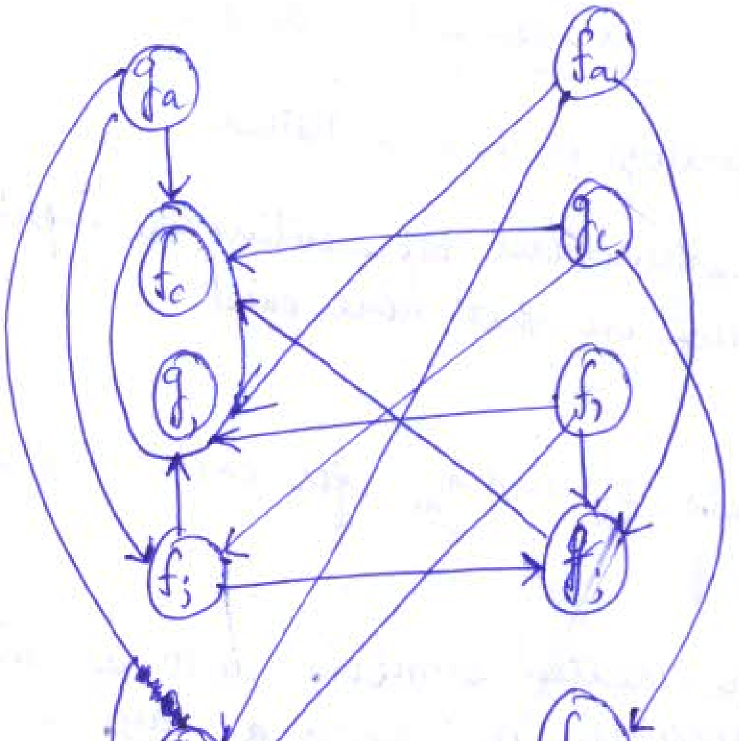
2 - partition the created symbols into as many groups as possible, in such a way if $a \doteq b$, then f_a and g_b are in same group.

3. Create a directed graph whose nodes are the groups found in (2). for any a and b, if $a <\cdot b$, place an edge from the group of g_b to the group of f_a.

If $a > b$, place an edge from the group of f_a to tha of g_b.

(4) If the graph constructed in (3) has a cycle, then no precedence functions exists. If there are no cycles, let $f(a)$ be the length of longest path beginning at the ~~grop~~ group of f_a; let $g(a)$ be the length of the longest path from ~~beg~~ the group of g_a.

graph->



	a	c	)	;	$
f	0	0	0	0	0
g	1	1	0	0	0

(4) Define backpatching and semantic rules for Boolean expression. Derive the three address code for the following expression.

P < Q or R < S and T < U.

Answer →

Backpatching – To generate the three address code, two passes are necessary.

~~In first pass, labels are properly filled, is called backpatching~~

In first pass, labels are not specified. These statements are placed in a list. In second pass, these labels are properly filled, is called backpatching.

following are the three functions used for backpatching

(a) Makelist(i) → creats a new list containing only i, an index into the array of quadruples being generated Makelist returns a pointer to the list it has made.

(b) Merge(p1, p2) → takes the lists pointed by p1 and p2, concatenates them into one list, and returns a pointer to the concatenated list.

(c) Backpatch(p, i) → makes each of the quadruples on the list pointed to by p take quadruple i as a target.

Semantic Rules for Boolean Expression

The Grammar is

(1) $E \rightarrow E^{(1)}$ or $M E^{(2)}$
$\quad | E^{(1)}$ and $M E^{(2)}$
$\quad |$ not $E^{(1)}$
$\quad | (E^{(1)})$
$\quad |$ id
$\quad | id^{(1)}$ relop $id^{(2)}$.

$M \rightarrow \epsilon$

The syntax Directed Translation scheme is as follows:-

(1) $E \rightarrow E^{(1)}$ or $M E^{(2)}$
{ Backpatch (E^1. false, M.Quad);
E.True = Merge(E^1.True, E^2.True);
E.false = ϵ^2.false }

(2) $E \rightarrow E^{(1)}$ and $ME^{(2)}$ { Backpatch (E^1.True, M.Quad);
E.True = E^2.True;
E.false = Merge (ϵ^1.false, ϵ^2false) }

(3) $E \rightarrow$ not$E^{(1)}$ { E.True = ϵ^1.false;
E.false = ϵ^1.True }

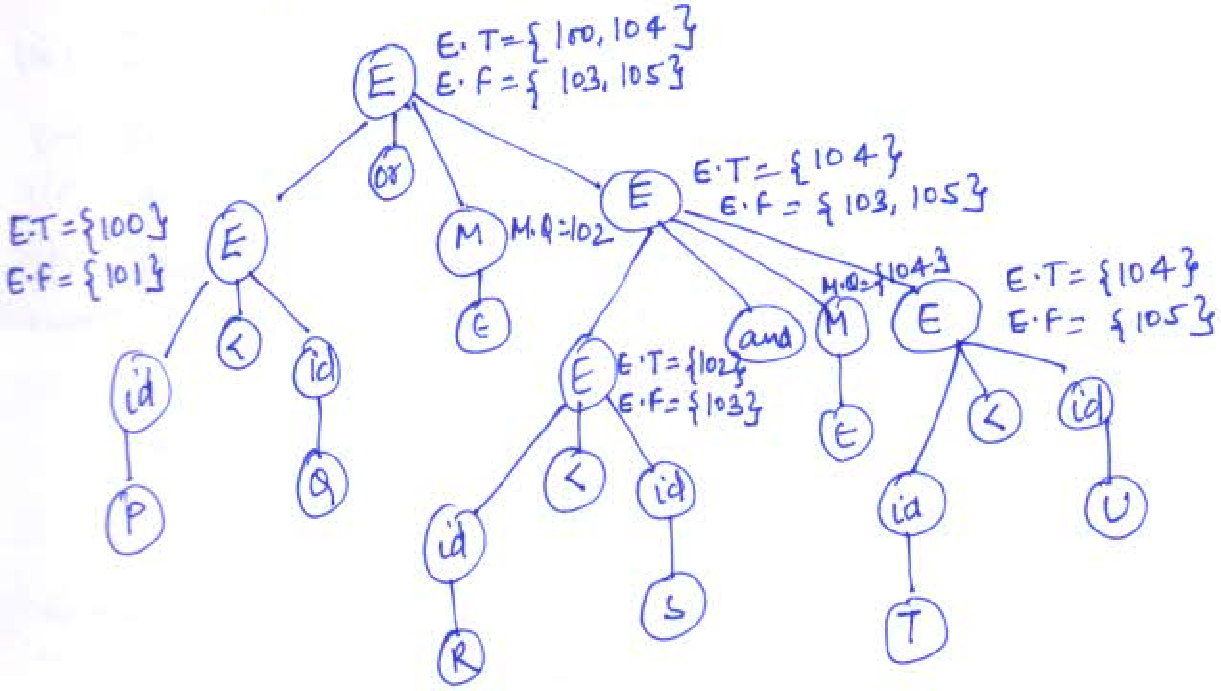
(4) $E \rightarrow (E^{(1)})$ { E.True = $E^{(1)}$.True;
E.false = $E^{(1)}$.false }

(5) $E \rightarrow$ id = { E.True = Makelist (NextQuad);
E.false = Makelist (NextQuad+1);
Gen (if id.place goto_);
Gen (goto_) }

(6) $E \rightarrow id^{(1)}$ relop $id^{(2)}$ { E. True = Makelist (nextQuad);
E.false = Makelist (nextQuad+1);
Gen(if $id^{(1)}$. place relop $id^{(2)}$. place goto_)
Gen (goto_) }

(7) $M \rightarrow \epsilon$ { M. Quad := nextQuad }

Parse Tree for P< Q or R< S and T< U



100: if P< Q goto -
101: goto <u>102</u>
102: if R< S goto 104
103: goto -
104: if T< U goto -
105: goto -

Q.5 What are the lexical phase errors, syntactic phase errors and semantic phase errors, explain with suitable examples.

Answer →

Lexical phase errors → These are not many errors that can be caught at the lexical level; those you should be looking for are:-

(i) Characters that cannot appear in any token in our source language, such as @ or #.

(ii) Integers constants out of bounds (range is 0 to 32767).

(iii) Identifier names that are too long. (maximum length is 32 characters)

(iv) Text strings that are two long. (max^m length is 256 characters)

(v) Text strings that span more than one line.

(vi) Certain other errors, such as malformed identifiers, could be caught here, or by the parser. The only one of these errors you are responsible for at this stage is the following: Unmatched right comment delimiters (*/).

Syntax Errors →

- A syntax Errors occurs when stream of tokens is an invalid string.

→ In LL(k) or LR(k) parsing tables, blank entries refer to syntax error.

How should syntax error be handled?

(i) Repor error, terminate compilation ⇒ not user friendly.

(ii) Report error, recover from error, and search for more errors ⇒ better.

Semantic Errors →

The semantic errors :→
(i) Undeclared identifiers
(ii) Unreachable statements
(iii) Identifiers used in the wrong content.
(iv) Methods called with the wrong number of parameters or with parameters of the wrong type.

(v) Type Mismatch

(vi) Undeclared variable

(vii) Reserved identifier misuse.

(viii) Multiple declaration of variable in a scope.

(ix) Accessing an out of scope variable

(x) Actual & formal parameter mismatch.

Q.6 What is the role of symbol table? Discuss different data structures used for symbol table.

Answer → A symbol table is a compile time data structure that is used by the compiler to collect and use information about the source program constructs, such as variables, contants, functions etc.

The symbol table helps the compiler in determining and verifying the semantics of given source program.

Data Structures used for symbol Table →

The various data structures used for implementing the symbol table are linear list, self-organizing list, hash table & search tree. The organization of symbol table depends on the selection of the data structure scheme used to implement the symbol table. The data structure schemes are evaluated on the basis of access time, simplicity, storage and performance.

Linear list → A linear list of records is the simplest data structure and its easiest - to - implement data structure as compared to other data structures for organizing a symbol-table. A single array or collection of several arrays is used to store names and their associated information. It uses a simple linear linked list to arrange the names sequentially in the memory.

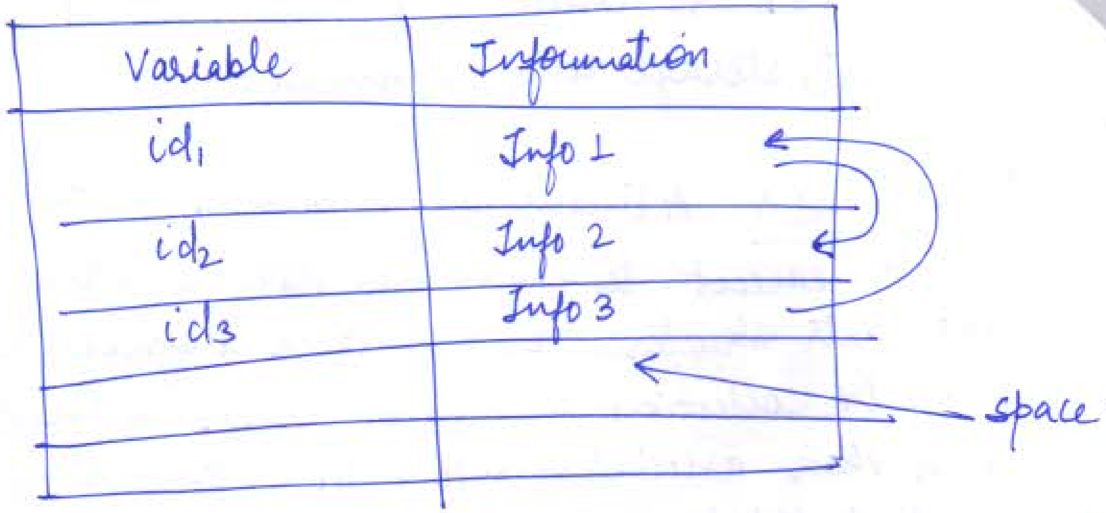
The new names are added to the table in the order of their arrival. Whenever a new name is added, the whole table is searched linearly or sequentially to check whether the name is already present in the symbol table or not.

If not, then a record for new name is created and added to the linear list at a location pointed to by the space pointer, & the pointer is incremented to point to the next empty location.

Variable	Information (type)	space (byte)
a	int	2
b	char	1
c	float	4
d	long	4

space.

Self-organizing list → We can reduce the time of searching the symbol table at the cost of a little extra space by adding an additional LINK field to each record or to each array index. Now, we search the list in the order indicating by links.



Variable	Information
id_1	Info 1
id_2	Info 2
id_3	Info 3

The main reason for using the self-organizing list is that if a small set of names is heavily used in a section of program, then these names can be placed at the top while that section is being processed by the compiler; However, if references are random, then the self-organizing list will cost more time & space.

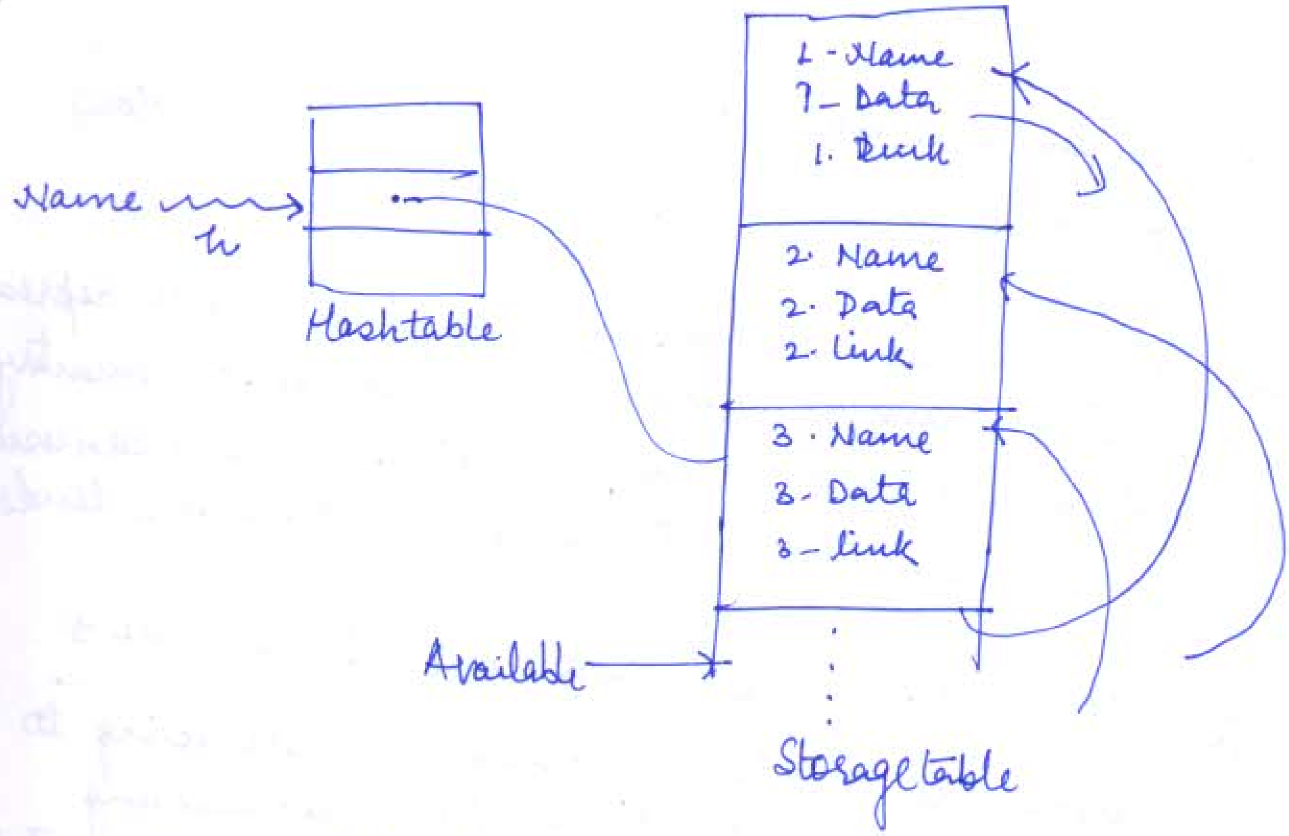
Demerits of self-organizing list are as follows:-

- It is difficult to maintain the list if a large set of names is frequently used.
- It occupies more memory as it has link field for each record.
- As self-organizing list organizes it itself, so it may cause problems in pointer movements.

Hash Table → A hash table is a data structure that associates keys with values. The basic hashing schemes has two parts:-

→ A hash table consisting of a fixed array of k pointers to table entries.

→ A storage table with the table entries organized into k seperate linked lists & each record in the symbol table appears on exactly one of these lists.



Search Tree → Search tree is an approach to organize symbol table by adding two link fields, Left and Right, to each record. These two fields are used to link the records into a binary search tree. All names are created as child nodes of root node that always follow the properties of a binary search tree.

→ The name in each node is a key value, that is, no two nodes can have identical names.

→ The names in the nodes of left sub tree, if exists, is Smaller than the value is the root node.

→ The names in the nodes of right sub tree, if exists, is greater than the value in the root node.

→ The left & right subtrees, if exists, arealso binary Searchtrees.

Q.7: Why run-time storage Management is required? How simple stack implementation is implemented?

Answer:→

A compiler must allocate resources of the target machine to represent the data objects manipulated by the source program. Elementary data types such as integers, real, & logical variables can usually be represented by equivalent data objects at the m/c levels.

Implementation of A simple stack - Allocation Scheme →

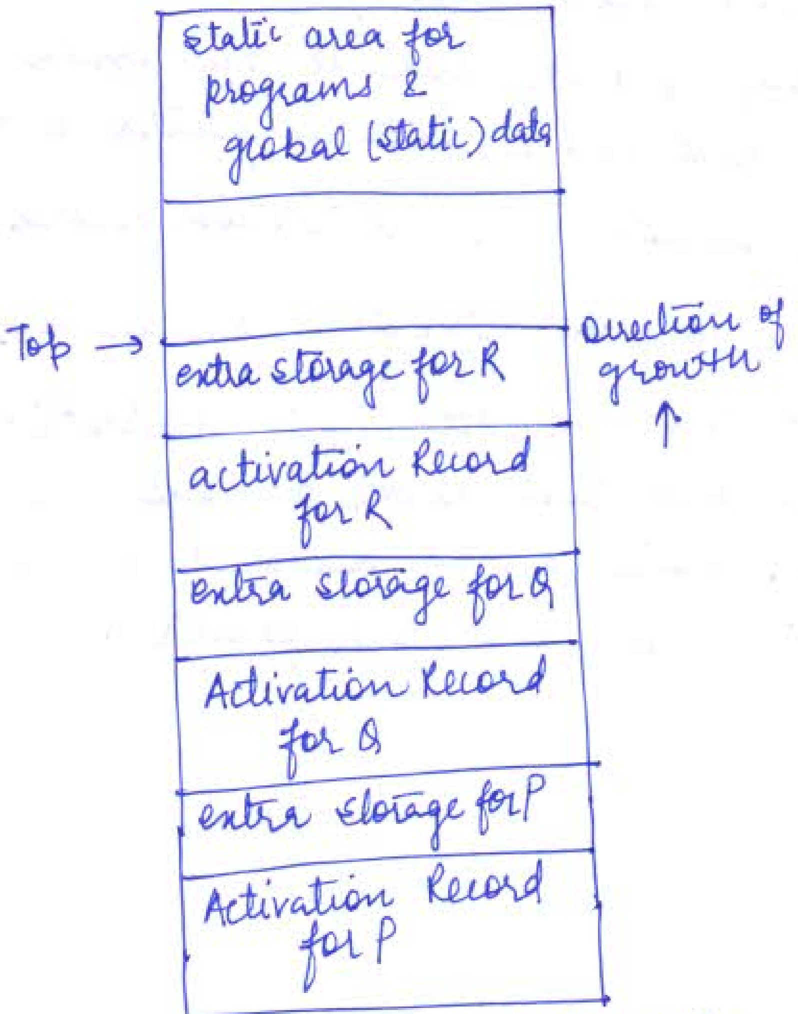
As an indroduction to stack allocation, we are going to consider an implementation of the UNIX programming language C, which allows somewhat simpler implementation than more other stack-oriented languages like ALGOL.

Data in C can be global, meaning it is allocated static storage and available to any procedure, or local meaning it can be accessed only by the procedure in which it is declared.

A program consists of a list of global data declarations and procedures; there is no block structure or nesting of procedures. However, recursion is permitted, so local names must be allocated space on stack.

Stack allocation of storage each procedure has an activation record on the stack, in which the values of local names are kept.

The low-numbered memory locations contains the code for the various procedures & space for the global data.



(Memory orgⁿ. for C program).

starting from the highest numbered available memory location is the runn-time stack.

We show two pointers to the stack, which are actually permanently allocated registers.

One, called the stack pointer (SP), always points to particular position in the activation record for the currently active procedure.

The second, called TOP, always points to the top of the stack to the TOP points to In C, it is often the case that TOP points the TOP of the ~~stack~~ top activation record. However, ~~temporaries~~ temporaries used for expression evaluation are allocated storage above the top activation record.

In a language like, which permits adjustable arrays, it is customary allocate space to the array above the fixed-size data and to store a pointer to this space in a fixed position of the activation record.

For uniformity, fixed-length ~~always~~ arrays are treated this way too. This ~~activation~~ arrangement allows each activation record to have a fixed size, and all local data can be accessed by going ~~to~~ a known distance from the stack pointer.

Q8:- Construct SLR(1) parsing table for the following grammar-

S → A)

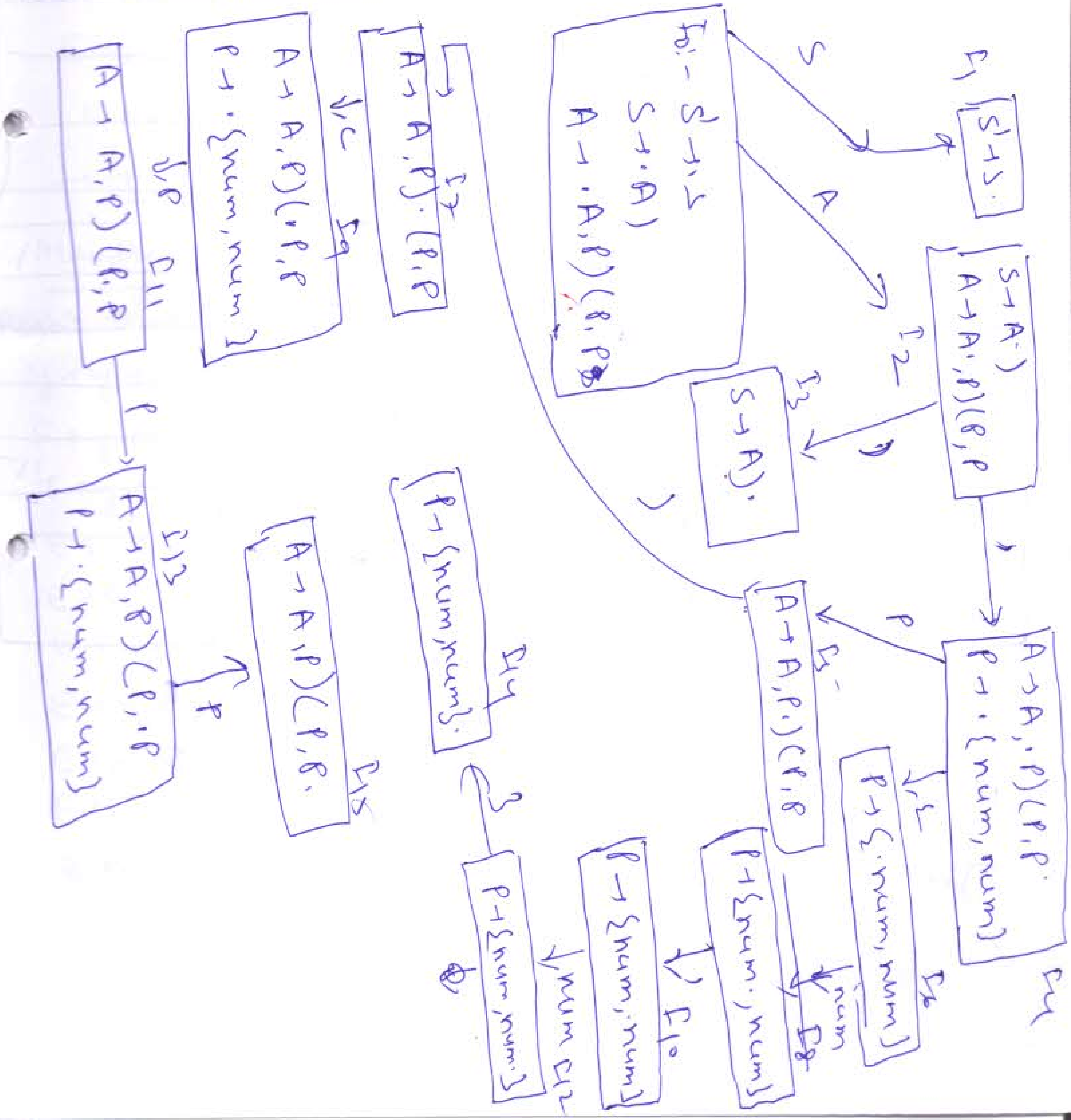
A → A,P) (P,P

P → {num, num}

Ans:- Initially the grammar made augmented

S → S' ; (1)S → A) ; (2)A → A,P) (P,P ; P → (3){num, num}

Constructing LR(0) itemsets:-



SLR(1) Parsing table

State	ACTION							GOTO		
	,	(	)	{	}	num	$	S	A	P
0								1	2	
1							acc			
2	s4		s3							
3							r1			8
4				s6						5
5		s7								
6						s8				
7		s9								
8	s10									
9				s6						11
10						s12				
11										
12					s14					
13				s6						15
14	r3		r3							
15	r2		r2							

Follow(S)! = { $ }
Follow(A) = {), , }
Follow(P) = {), , }

The above table is SLR(1) table.

eg:- Consider the following grammar -

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid y$

Show shift reduce parser action for the string $y+y+y*y$.

Ans:- Derive the string using RMD and find the handles.

$E \to \underline{E+T} \to E+\underline{T*F} \to E+T*\underline{y} \to E+\underline{F}*y \to$
$E+\underline{y}*y \to \underline{E+T}+y*y \to E+\underline{F}+y*y \to E+y+y*y$
$\to \underline{T}+y+y*y \to \underline{F}+y+y*y \to y+y+y*y$

Handle pruning process

Right sentential form	handle	used production
$y+y+y*y$	y	$F \to y$
$F+y+y*y$	F	$T \to F$
$T+y+y*y$	T	$E \to T$
$E+y+y*y$	y	$F \to y$
$E+F+y*y$	F	$T \to F$
$E+T+y*y$	$E+T$	$E \to E+T$
$E+y*y$	y	$F \to y$
$E+F*y$	F	$T \to F$ $F \to y$
$E+T*y$	~~E+T~~ y	~~E→E+T~~
~~E+T~~ $E+T*F$	$T*F$	$T \to T*F$
$E+T$	$E+T$	$E \to E+T$
E	—	—

Stack Implementation

~~input~~ Stack	input	Action
$	y+y+y*y$	shift y
$y	+y+y*y$	reduce F→y
$F	+y+y*y$	reduce T→F
$T	+y+y*y$	reduce E→T
$E	+y+y*y$	shift +
$E+	y+y*y$	shift y
$E+y	+y*y$	reduce F→y
$E+F	+y*y$	reduce T→F
$E+T	+y*y$	reduce E→E+T
$E	+y*y$	shift +
$E+	y*y$	shift y
$E+y	*y$	reduce F→y
$E+F	*y$	reduce ~~T~~T→F
$E+T	*y$	shift *
$E+T*	y$	shift y
$E+T*y	$	reduce F→y
$E+T*F	$	reduce T→T*F
$E+T	$	reduce E→E+T
$E	$	accept.

SECTION-C

Q.10 → Consider the following grammar for addressing the array elements. Give the syntax directed translation scheme to convert into three address code.

S → L = E
E → E+E | (E) | L
L → id [Elist]
Elist → E] | E, Elist

Generate three address code for the following expression

A[I,J] = B[I,J] + C[A[K,L]] + D[I,J]

Ans:- The semantic Action for the grammar is as follows.

```
S → L := E    { if L.OFFSET := null then /* L is a simple
                                               id */
                  GEN(L.PLACE := E.PLACE);
                 else GEN(L.PLACE [L.OFFSET] := E.PLACE) }
E → E+E       { T := NEWTEMP();
                 E.PLACE := T;
              GEN(T := E(1).PLACE + E(2).PLACE) }
E → (E(1))     { E.PLACE := E(1).PLACE }
E → L         { if L.OFFSET := null then
                   GEN (E.PLACE := L.PLACE);
                else begin T := NEWTEMP();
                  GEN (T := L.PLACE[L.OFFSET])
                   E.PLACE := T
                   end }
Elist → E]     { L.PLACE := Elist.PLACE;
                  L.NDIM := 1;
                  L.ARRAY := id.PLACE
                  }
```

```
L → id[Elist] { T := NEWTEMP();
                U := NEWTEMP();
                GEN(T := E.ARRAY - C);
                GEN (U := bpw * E.PLACE);
                  L.PLACE := T;
                  L.OFFSET := U;
                  }

Elist → E, Elist(1) { T := NEWTEMP();
                  GEN(T := E.PLACE * LIMIT (E.ARRAY,
                      E.NDIM+1));
                  GEN (T := T+E.PLACE);
                  Elist.ARRAY := E.ARRAY
                  Elist.PLACE := T;
                  Elist.NDIM := E.NDIM+1}
```

The expression is:-

$A[I,J] = B[I,J] + C[A[K,L]] + D[I+J]$

Suppose $d_1 = 10$, $d_2 = 20$, $bpw = 4$

$T_1 = 20*I$

$T_1 = 20*I+J$

$T_2 = 4*T_1$

$T_3 = A-84$

$T_4 = T_3[T_2]$

$T_5 = B-84$

$T_6 = T_5[T_2]$

$T_7 = D-84$ $T_{16} = I+J$

$T_8 = T_7[T_{16}]$

$T_9 = 20*K$

$T_9 = T_9+L$

$T_{10} = 4*T_9$

~~$T_9 = A$~~

$T_{11} = T_3[T_{10}]$

$T_{12} = C-4$

$T_{13} = T_{12}[T_{11}]$

$T_{14} = T_6 + T_{13}$

$T_{15} = T_{14} + T_8$

$T_4 = T_{15}$

Q.17(a) What is DAG? Give algorithm for DAG construction? Also construct the DAG for the following expression:

$(a+b)-(e-(c+d))$

Ans:- DAG (DIRECTED ACYCLIC GRAPH):- A DAG is a directed graph with no cycle which gives a picture how the value computed by each statement in a basic block is used in subsequent statement in the block. A computation

DAG is a directed graph with following lables on node.

I) Leaves are labled by unique identifiers either variable names or constants.

II) Interior nodes are labled by an operator symbol.

III) Nodes are also optionally given an extra set of identifiers for lables.

A DAG computation of basic block is optimized version of triples.

Algo:- Constructing a DAG

INPUT:- A basic block

OUTPUT:- A DAG with the following information:

1) A label for each node. For leaf the label is an identifier (Constant permitted) and for interior nodes an operator.

II) For each node a list of attached identifiers are allowed (constants are not permitted)

Method:- The DAG computation process is to do the following step 1 through step 3 for each statement in block. Initially we assume that there are no nodes and NODE() Is undefined. Suppose the Three address codes are (I) A = B op C (II) A = op B (III) A = B

Rule 1:- If node B is undefined, create a leaf labeled with B. Let NODE(B) be this node. In case I if node C is undefined, create this node.

Rule 2:- In case I determine if there is a node labelled operator whose left child is node B and right child is node C In case II determine whether there is node labelled operator whose lone child is node B. If not create a node. In case III, let n be the node B.

Rule 3: Append A to list of attached identifier for the node found in step 2.

The expression is

(a+b) - (e-(c+d))

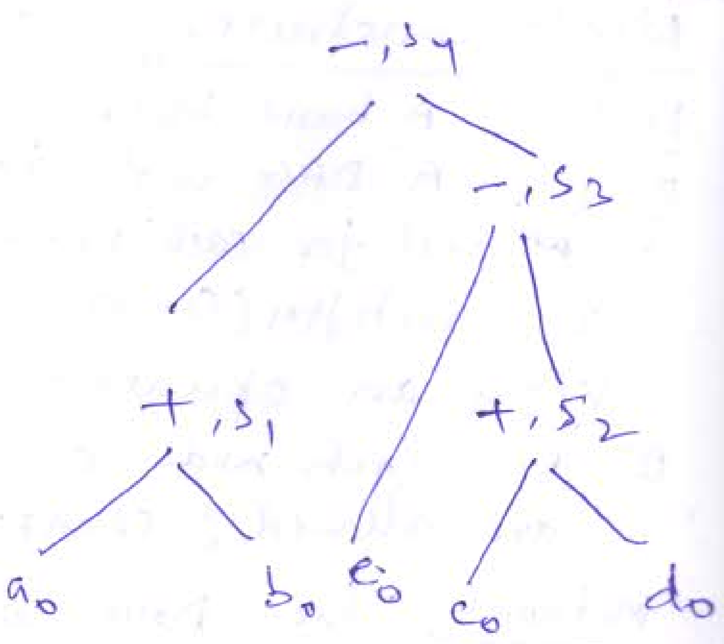
The triples are:-

$S_1 = a+b$

$S_2 = c+d$

$S_3 = e - S_2$

$S_4 = S_1 - S_3$



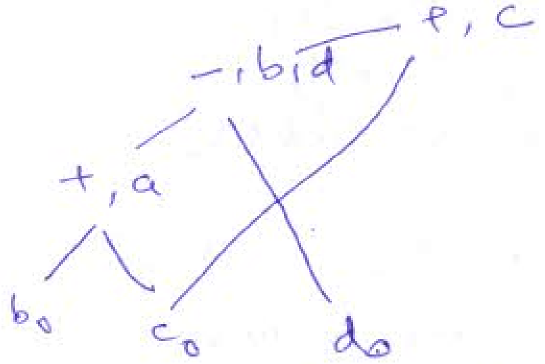
The resulting DAG

11(b)→ What are DAG advantages in context of optimization?

Ans:- DAG (Directed Acyclic Graph) is a graphical representation with no cycle to perform optimization of basic blocks. The advantages of DAG is -

① Elliminating Local common subexpression→

$a = b+c$
$b = a-d$
$c = b+c$
$d = a-d$



here b & d is local common subexpression and after elliminating the expression becomes

$a = b+c$, $d = a-d$, $c = d+c$

Ⅱ Dead Code elimination→

- Delete from a DAG any root (node with no ancestor) that has no live variables attached.
- Repeated applications of this transformation will remove all nodes from the DAG that corresponds Dead Code.

Ⅲ Reordring statement that donot depend on one another.

Ⅳ Use of Algebraic identities→

- eliminate computations

$x+0 = 0+x = x$ $\quad x-0 = x$
$x*1 = 1*x = x$ $\quad x/1 = x$

- Reduction in strength

$x^2 = x \times x$, $2 \times x = x+x$, $x/2 = x \times .05$

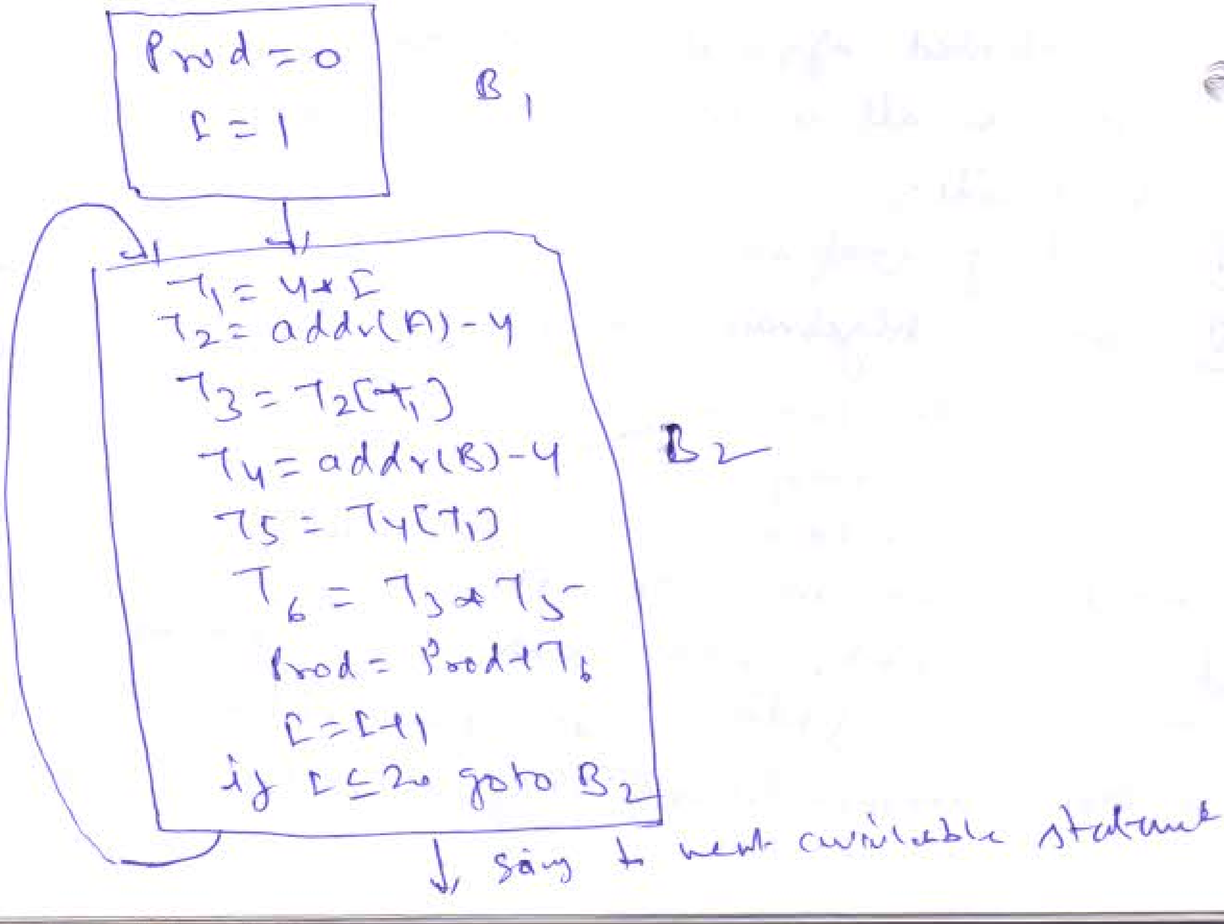
- constant folding $2*3.14 = 6.28$ evaluated at compile time
- Othe transformations $x*y = y*x$, $x > y$ and $x-y > 0$

Q12(a) Consider the following sequence of three addr
code -

```
1. PROD = 0
2. I = 1 → 3. T1 = 4*I
4. T2 = addr(A) - 4
5. T3 = T2[T1]
6. T4 = addr(B) - 4
7. T5 = T4[T1]
8. T6 = T3 * T5
9. Prod = Prod + T6
10. I = I+1
11. if I ≤ 20 goto (3)
```

Perform Loop optimization.

Ans:- The given three address codes are initially converted into basic block for performing loop optimization. The Basic blocks are defined by leader statement. In this problem the statement (1) and statement (3) are leaders. So initially two basic blocks are made.

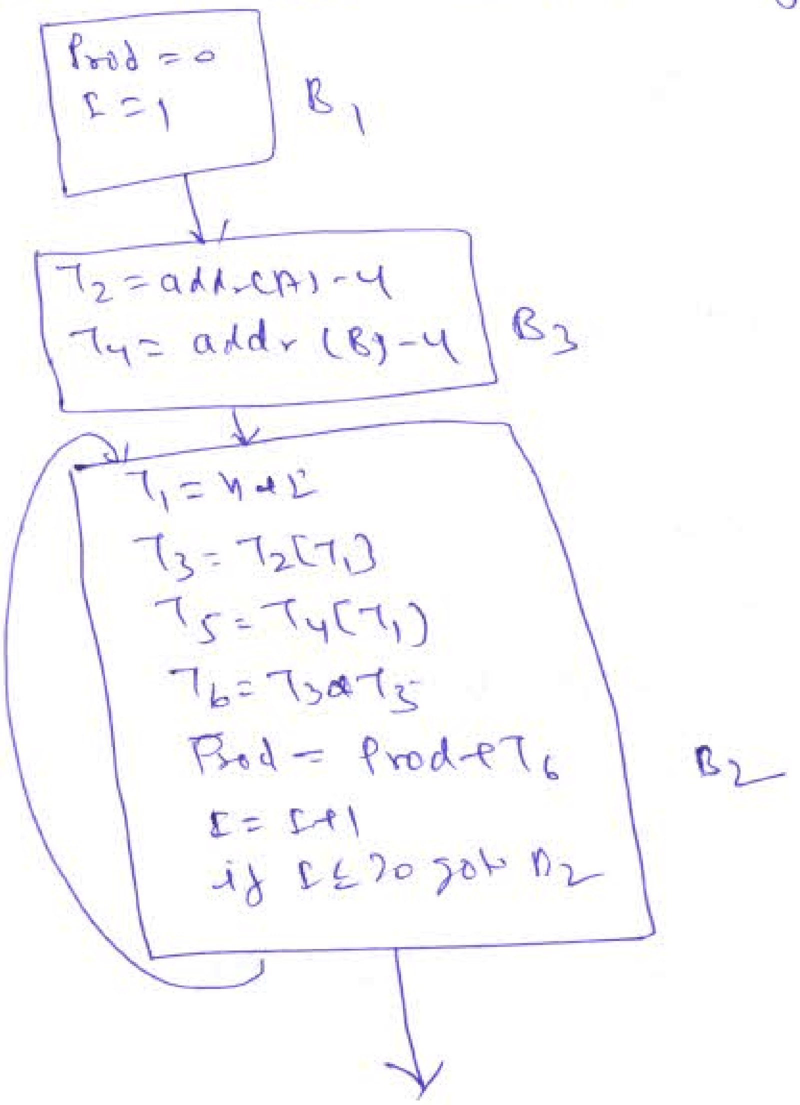


```
Prod = 0
I = 1
```

```
T1 = 4*I
T2 = addr(A) - 4
T3 = T2[T1]
T4 = addr(B) - 4
T5 = T4[T1]
T6 = T3 * T5
Prod = Prod + T6
I = I+1
if I ≤ 20 goto B2
```

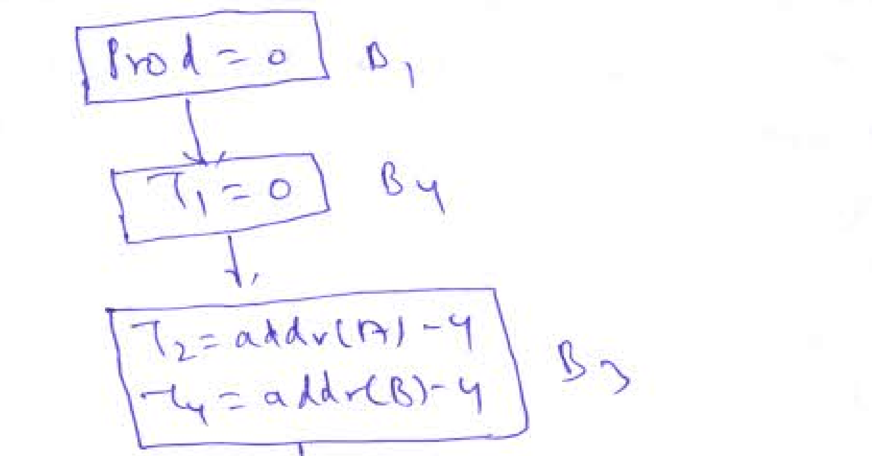
After Designing basic block there are edges connecting each basic block with another. The complete structure is called flow graph. After that we are going to optimize that flow graph using loop optimization.

(1) Code Motion: The running time of a program may be improved if we decrease the length of one of its loops especially an inner loop, even if increase the amount of code outside the loop. This approach is called code motion. In this problem $T_2 = addr(A) - 4$ & $T_4 = addr(B) - 4$ its independent from loop index. So removed from Basic block B_2. The resulting Basic blocks are.



(f) Induction variable elliminations The induction variable I & T_1 is found in block B_2. The value of I is from 1, 2 -- 20 then the value of T_1 is from 4, 8 -- 80. So the both values are generated as arithmatic prograssion and are induction variable. We remove I from the expression by converting $T_1 = 4*I$ into $T_1 = T_1 + 4$ and $T_1 = 0$ initializing. One index variable I is deleted and costly operator * is replaced by cheaper operator + is called strength reduction.

So the optimized blocks are.



```
Prod = 0                 B1

T1 = 0                   B4

T2 = addr(A) - 4         B3
T4 = addr(B) - 4

T1 = T1 + 4              B2
T3 = T2[T1]
T5 = T4[T1]
T6 = T3 * T5
Prod = Prod + T6
if T1 <= 76 goto B2
```

The above is optimized TAC for the problem

12(b) :- Write short notes (on any two) -

(I) Global data flow analysis -

Ans:- It is an informal approach of examining entire program. A new concept U-d-chaining (use-definition -chaining) which answers the question: Given that identifier A is used at point p, at what points could the value of A used at point p have been defined.

By a use of identifier A means any occurrence of A as an operand. By a definition of A means either an assignment to A or reading of value for A. By a point in a program means the position before or after any intermediate statement control reaches the point just before statement execution and after when statement has been executed.

There are data flow equation that used to calculate the reaching definitions in terms of use-definition - chaining. For each variable A there are two sets GEN(B) i.e. is the set of generated definitions and KILL(B) the set of related definitions outside the block. Then IN(B) is used to generate the input set and OUT(B) is generated by following data flow equation -

(I) OUT(B) = IN(B) - KILL(B) + GEN(B)

(II) IN(B) = OUT(P)

where P is a predecessor of B

By using above flow equation the reaching definition is calculated and also the dependency of variables according to their definition and declaration is defined.

(II) <u>Loop</u> <u>Unrolling</u>→

Ans:- The loop unrolling avoids a test at every iteration by recognizing that the number of iterations is constant and replicating the body of loop.

Lets the loop.

```
begin
        I:=1
        while I ≤ 100 do
            begin
                A[I]:=0
                I:=I+1
            end
    end
```

We could do with 50 tests if we converted the code to

```
begin   I:=1
        while I ≤ 100 do
    begin
                A[I]:=0
                I=I+1;
                A[I]=0
                I:=I+1
        end
end.
```

(iii) Loop Jamming

Ans:- The loop Jamming is to merge the bodies of two loops. It is necessary that each loop be executed the same number of times and that the indices be the same.

For ex:-

```
begin
    for I := 1 to 10 do
        for J := 1 to 10 do
            A[I, J] = 0 (* Zero the matrix/
    for I = 1 to 10 do
        A[I, I] = 1
end
```

~~Can be jammed~~

The above loop can be jammed by concatenating the bodies of two loops on I to form

```
for I := 1 to 10 do
    begin
        for J = 1 to 10 do
            A[I, J] := 0
        A[I, I] := 1
    end.
```