PROBABLE QUESTION COMPILER DESIGN

Module-1

Short Questions:

1.Q.What is a compiler?

Ans: A compiler is a program that reads a program written in one language –the source language and translates it into an equivalent program in another language-the target language. The compiler reports to its user the presence of errors in the source program.

2.Q.What are the two parts of a compilation? Explain briefly

Ans: Analysis and Synthesis are the two parts of compilation.

The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

The synthesis part constructs the desired target program from the intermediate representation.

3.Q.List the subparts or phases of analysis part.

Ans: Analysis consists of three phases:

- Linear Analysis.
- Hierarchical Analysis.
- Semantic Analysis.

4.Q. **Depict diagrammatically how a language is processed.**

Ans: Skeletal source program

↓

Preprocessor

↓

Source program

↓

Compiler

↓

Target assembly program

↓

Assembler

↓

Relocatable machine code

↓

Loader/ link editor ←library, relocatable object files

↓

Absolute machine code

5.Q. What is linear analysis?

Linear analysis is one in which the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of characters having a collective meaning.

Also called lexical analysis or scanning.

6.Q. **List the various phases of a compiler.**

Ans: The following are the various phases of a compiler:

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate code generator
- Code optimizer
- Code generator

7.Q. **What are the classifications of a compiler?**

Ans: Compilers are classified as:

- · Single- pass
- · Multi-pass
- · Load-and-go
- · Debugging or optimizing

8.Q. **What is a symbol table?**

Ans: A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer.

9.Q. **Mention some of the cousins of a compiler.**

Ans: Cousins of the compiler are:

- · Preprocessors
- · Assemblers
- · Loaders and Link-Editors

10.Q. **List the phases that constitute the front end of a compiler.**

Ans: The front end consists of those phases or parts of phases that depend primarily on the source language and are largely independent of the target machine. These include

- · Lexical and Syntactic analysis
- · The creation of symbol table
- · Semantic analysis
- · Generation of intermediate code

A certain amount of code optimization can be done by the front end as well. Also includes error handling that goes along with each of these phases.

11.Q. **Mention the back-end phases of a compiler.**

Ans: The back end of compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language. These include

- · Code optimization
- · Code generation, along with error handling and symbol- table operations.

12.Q. **Define compiler-compiler.**

Ans: Systems to help with the compiler-writing process are often been referred to as compiler-compilers, compiler-generators or translator-writing systems.

Largely they are oriented around a particular model of languages , and they are suitable for generating compilers of languages similar model.

13.Q. **List the various compiler construction tools.**

Ans: The following is a list of some compiler construction tools:

- · Parser generators
- · Scanner generators
- · Syntax-directed translation engines
- · Automatic code generators
- · Data-flow engines

14.Q. **Differentiate tokens, patterns, lexeme.**

- **Ans:** Tokens- Sequence of characters that have a collective meaning.
- · Patterns- There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token
- · Lexeme- A sequence of characters in the source program that is matched by the pattern for a token.

15.Q. **List the operations on languages.**

- **Ans: Union –** L U M ={s | s is in L or s is in M}
- · **Concatenation** – LM ={st | s is in L and t is in M}
- · **Kleene Closure –** L* (zero or more concatenations of L)
- · **Positive Closure –** L+ (one or more concatenations of L)

16.Q. **Write a regular expression for an identifier.**

Ans: An identifier is defined as a letter followed by zero or more letters or digits.

The regular expression for an identifier is given as

letter (letter | digit)*

17.Q. **List the various error recovery strategies for a lexical analysis.**

Ans: Possible error recovery actions are:

- · Panic mode recovery
- · Deleting an extraneous character
- · Inserting a missing character
- · Replacing an incorrect character by a correct character
- · Transposing two adjacent characters

18.Q. **Define parser.**

Ans: Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning. Also termed as Parsing.

19.Q. **Mention the basic issues in parsing.**

Ans: There are two important issues in parsing.

- · Specification of syntax
- · Representation of input after parsing.

20.Q. **Define ambiguous grammar.**

Ans: A grammar G is said to be ambiguous if it generates more than one parse tree for some sentence of language L(G).

i.e. both leftmost and rightmost derivations are same for the given sentence.

21.Q. **List the properties of LR parser.**

Ans: 1. LR parsers can be constructed to recognize most of the programming languages for which the context free grammar can be written.

2. The class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.

3. LR parsers work using non backtracking shift reduce technique yet it is efficient one.

22.Q. **Mention the types of LR parser.**

- Ans: SLR parser- simple LR parser
- · LALR parser- lookahead LR parser
- · Canonical LR parser

23.Q. **What are the problems with top down parsing?**

Ans: The following are the problems associated with top down parsing:

- · Backtracking
- · Left recursion
- · Left factoring
- · Ambiguity

24.Q. **Write the algorithm for FIRST and FOLLOW.**

Ans: **FIRST**

1. If X is terminal, then FIRST(X) IS {X}.

2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).

3. If X is non terminal and $X \rightarrow Y1,Y2..Yk$ is a production, then place a in FIRST(X) if for some i , a is in FIRST(Yi) , and ε is in all of FIRST(Y1),...FIRST(Yi-1);

FOLLOW

1. Place $ in FOLLOW(S),where S is the start symbol and $ is the input right endmarker.

2. If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except for ε is placed in FOLLOW(B).

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε , then everything in FOLLOW(A) is in FOLLOW(B).

25.Q. **What is meant by handle pruning?**

Ans: A rightmost derivation in reverse can be obtained by handle pruning.

If w is a sentence of the grammar at hand, then w = γn, where γn is the nth right-sentential form of some as yet unknown rightmost derivation

S = γ0 => γ1...=> γn-1 => γn = w

26.Q. **Define LR(0) items.**

Ans: An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. Thus, production A → XYZ yields the four items

A→**.**XYZ

A→X**.**YZ

A→XY**.**Z

A→XYZ**.**

27. **Define handle.**

Ans: A handle of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation.

A handle of a right – sentential form γ is a production A→β and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ. That is , if S =>αAw =>αβw,then A→β in the position following α is a handle of αβw.

28.Q. **What is phrase level error recovery?**

Ans: Phrase level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

Module-2:

29.Q. **What are the benefits of intermediate code generation?**

- **Ans:** A Compiler for different machines can be created by attaching different back end to the existing front ends of each machine.
- A Compiler for different source languages can be created by proving different front ends for corresponding source languages t existing back end.
- A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

30.Q. **What are the various types of intermediate code representation?**

There are mainly three types of intermediate code representations.

- · Syntax tree
- · Postfix
- · Three address code

31.Q. **Define backpatching.**

Ans: Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.In the semantic actions the functions used are mklist(i),merge_list(p1,p2) and backpatch(p,i).

32.Q. **What is the intermediate code representation for the expression a or b and not c?**

Ans: The intermediate code representation for the expression a or b and not c is the three address sequence

t1 := not c

t2 := b and t1

t3 := a or t2

33.Q. **What are the various methods of implementing three address statements?**

Ans: The three address statements can be implemented using the following methods.

- · Quadruple : a structure with atmost four fields such as operator(OP),arg1,arg2,result.
- · Triples : the use of temporary variables is avoided by referring the pointers in the symbol table.
- · Indirect triples : the listing of triples has been done and listing pointers are used instead of using statements.

34.Q. **What is a DAG? Mention its applications.**

Ans: Directed acyclic graph(DAG) is a useful data structure for implementing transformations on basic blocks.

DAG is used in

- · Determining the common sub-expressions.
- · Determining which names are used inside the block and computed outside the block.
- · Determining which statements of the block could have their computed value outside the block.
- · Simplifying the list of quadruples by eliminating the common su-expressions and not performing the assignment of the form x := y unless and until it is a must.

Module-3

35.Q. **Mention the properties that a code generator should possess.**

- Ans: The code generator should produce the correct and high quality code. In other words, the code generated should be such that it should make effective use of the resources of the target machine.
- · Code generator should run efficiently.
  - · **Define and use –** the three address statement a:=b+c is said to define a and to use b and c.
  - · **Live and dead –** the name in the basic block is said to be live at a given point if its value is used after that point in the program. And the name in the basic block is said to be dead at a given point if its value is never used after that point in the program.

36.Q. **What is a flow graph?**

Ans: A flow graph is a directed graph in which the flow control information is added to the basic blocks.

- · The nodes to the flow graph are represented by basic blocks
- · The block whose leader is the first statement is called initial block.
- · There is a directed edge from block B1 to block B2 if B2 immediately follows B1 in the given sequence. We can say that B1 is a predecessor of B2.

37.Q. **Define peephole optimization.**

Ans: Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence.

38.Q. **List the characteristics of peephole optimization.**

- **Ans:** Redundant instruction elimination
- · Flow of control optimization
- · Algebraic simplification
- · Use of machine idioms

39.Q. **What is a basic block?**

Ans: A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching.

Eg. t1:=a*5

t2:=t1+7

t3:=t2-5

t4:=t1+t3

t5:=t2+b

1. 40.Q. **Mention the issues to be considered while applying the techniques for code optimization.**

- Ans: The semantic equivalence of the source program must not be changed.
- · The improvement over the program efficiency must be achieved without changing the algorithm of the program.
  - · The machine dependent optimization is based on the characteristics of the target machine for the instruction set used and addressing modes used for the instructions to produce the efficient target code.
  - · The machine independent optimization is based on the characteristics of the programming languages for appropriate programming structure and usage of efficient arithmetic properties in order to reduce the execution time.
    - · Available expressions
    - · Reaching definitions
    - · Live variables
    - · Busy variables

41.Q. **What are the basic goals of code movement?**

- **Ans:** To reduce the size of the code i.e. to obtain the space complexity.
- To reduce the frequency of execution of code i.e. to obtain the time complexity.
- **Module-4**

6. 42.Q. **What are the contents of activation record?**

   Ans: The activation record is a block of memory used for managing the information needed by a single execution of a procedure. Various fields f activation record are:

- · Temporary variables
- · Local variables
- · Saved machine registers
- · Control link
- · Access link
- · Actual parameters
- · Return values

7. 43.Q. **What is dynamic scoping?**

   Ans: In dynamic scoping a use of non-local variable refers to the non-local data declared in most recently called and still active procedure. Therefore each time new findings are set up for local names called procedure. In dynamic scoping symbol tables can be required at run time.

8. 44.Q. **Define symbol table.**

   Ans: Symbol table is a data structure used by the compiler to keep track of semantics of the variables. It stores information about scope and binding information about names.

   45.Q.Difference between S-attributed SDT grammar and L-attributed SDT grammar

   Ans: **S-attributed SDT :**

- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

L-attributed SDT:

- If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.

46.Q.Define dependency graph

Ans: a **dependency graph** is a directed **graph** representing **dependencies** of several objects towards each other. It is possible to derive an evaluation order or the absence of an evaluation order that respects the given **dependencies** from the **dependency graph**.

47.Q.What is annotated parse tree.

Ans: AN **ANNOTATED PARSE TREE** is a **parse tree** showing the values of the attributes at each node. The process of computing the attribute values at the nodes is called annotating or decorating the **parse tree**. Node in such a **tree** represent procedures which have run; childen represent procedures called by their parent.

48.Q.Define SDT

Ans: **Syntax-directed translation** refers to a method of **compiler** implementation where the source language translation is completely driven by the parser. Thus, parsing a string of the grammar produces a sequence of rule applications. **SDT** provides a simple way to attach semantics to any such syntax.

49.Q.What is loop unrolling?

Ans: **Loop unrolling**, also known as **loop unwinding**, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size, which is an approach known as space–time tradeoff. The transformation can be undertaken manually by the programmer or by an optimizing compiler. On modern processors, loop unrolling is often counterproductive, as the increased code size can cause more cache misses

Normal loop	After loop unrolling

`int x;`
`for (x = 0; x < 100; x++)`
`{`
`  delete(x);`
`}`
	`int x;`
	`for (x = 0; x < 100; x += 5 )`
	`{`
	`  delete(x);`
	`  delete(x + 1);`
	`  delete(x + 2);`
	`  delete(x + 3);`
	`  delete(x + 4);`

	}

50. Q.What is common sub expression elimination?

Ans: **common subexpression elimination** (**CSE**) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value.

```
a = b * c + g;
d = b * c * e;
```

it may be worth transforming the code to:

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

if the cost of storing and retrieving `tmp` is less than the cost of calculating `b * c` an extra time.

Long Questions:

Module-1:

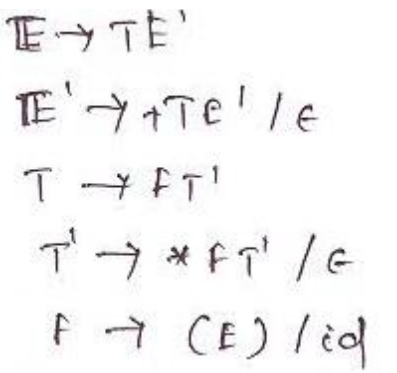
1.Q.Differentiate between top down parsing and bottom up parsing.

Ans:

S.No	Top Down Parsing	Bottom Up Parsing
1.	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
2.	Top-down parsing attempts to find the left most derivations for an input string.	Bottom-up parsing can be defined as an attempts to reduce the input string to start symbol of a grammar.
3.	In this parsing technique we start parsing from top (start symbol of parse tree) to down (the leaf node of	In this parsing technique we start parsing from bottom (leaf node of parse tree) to up (the start

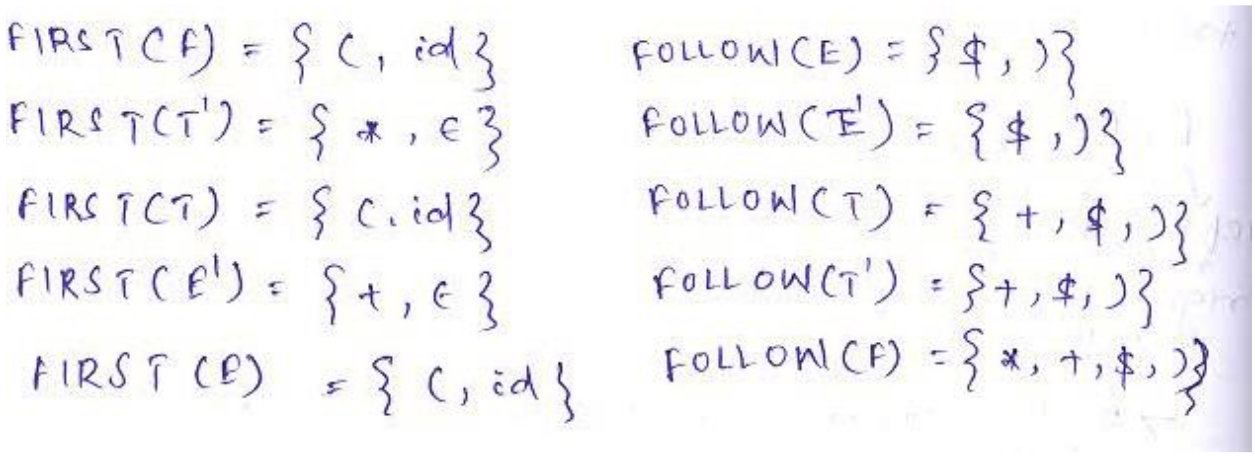
	parse tree) in top-down manner.	symbol of parse tree) in bottom-up manner.
4.	This parsing technique uses Left Most Derivation.	This parsing technique uses Right Most Derivation.
5.	It's main decision is to select what production rule to use in order to construct the string.	It's main decision is to select when to use a production rule to reduce the string to get the starting symbol.

2.Q. What is a LL(1) grammar?



Check the grammar is LL(1) or not.

Ans: **In formal language theory, an** LL grammar **is a context-free grammar that can be parsed by an LL parser, which parses the input from** L**eft to right, and constructs a** L**eftmost derivation of the sentence (hence LL, compared with LR parser that constructs a rightmost derivation).**



M-table / parsing table

	+	*	(	)	id	$
E			$F \to (E)$		$F \to id$	
E'	$E' \to +TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T			$F \to (E)$		$F \to id$	
T'	$T' \to \epsilon$	$T \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F			$F \to (E)$		$F \to id$	

Hence, the parsing table doesn't entry multiple production rule in any ~~etc~~ cell then it is known as LL(1) grammer.

3.Q.Write the predictive parsing algorithm?

D → type list ;
list → id tlist ;
tlist → , id tlist / ∈
type → int / float
w = int id , id ;
parse the input string using predictive parsing algorithm.

Ans:

Predictive Parsing Algorithm :—

→ Set the input pointer at the first symbol of input string w$.

→ Let 'x' be the stack top symbol and 'a' is the first symbol of input string.

→ If x = a = $, then parser halts and announces the successful completion of parsing.

→ If x = a ≠ $, then parser pop x from the top of the stack and advance the input pointer to the next symbol of input string.

→ If x is a non-terminal, then check the M-table for entry M[x, a], if there is a production $x \rightarrow Y_1 Y_2 Y_3 \cdots Y_k$ then pop x from the stack and push $Y_k, Y_{k-1}, \cdots Y_2$ to the stack.

FIRST(type) → {int, float}

FIRST(tlist) → {, , ∈}

FIRST(list) → {id}

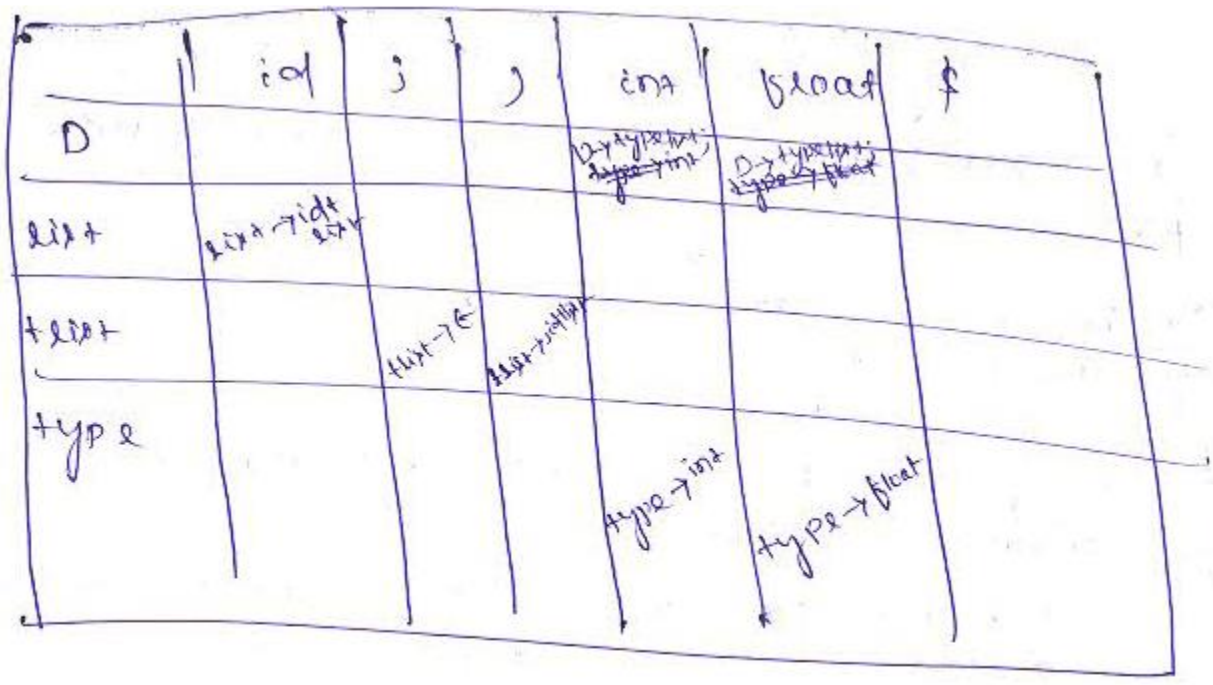
FIRST(D) → {~~type~~ int, float}

FOLLOW(D) → {$}

FOLLOW(list) → {;}

FOLLOW(tlist) → {;}

FOLLOW(type) → {id}



Stack	Input	Action
$ D	int id, id; $	D → type list; ~~type → int~~
$ ~~int~~ ; list type	int id, id; $	~~pop int~~ type → int
$; list int	int id, id; $	pop int
$; list	id, id; $	list → id tlist
$; tlist id	id, id; $	pop id
$; tlist	, id; $	tlist → , id tlist
$; tlist id ,	, id; $	pop ,
$; tlist id	id; $	pop id
$; ε list	; $	tlist → ε
$; ε	; $	pop ;
$	$	Accept

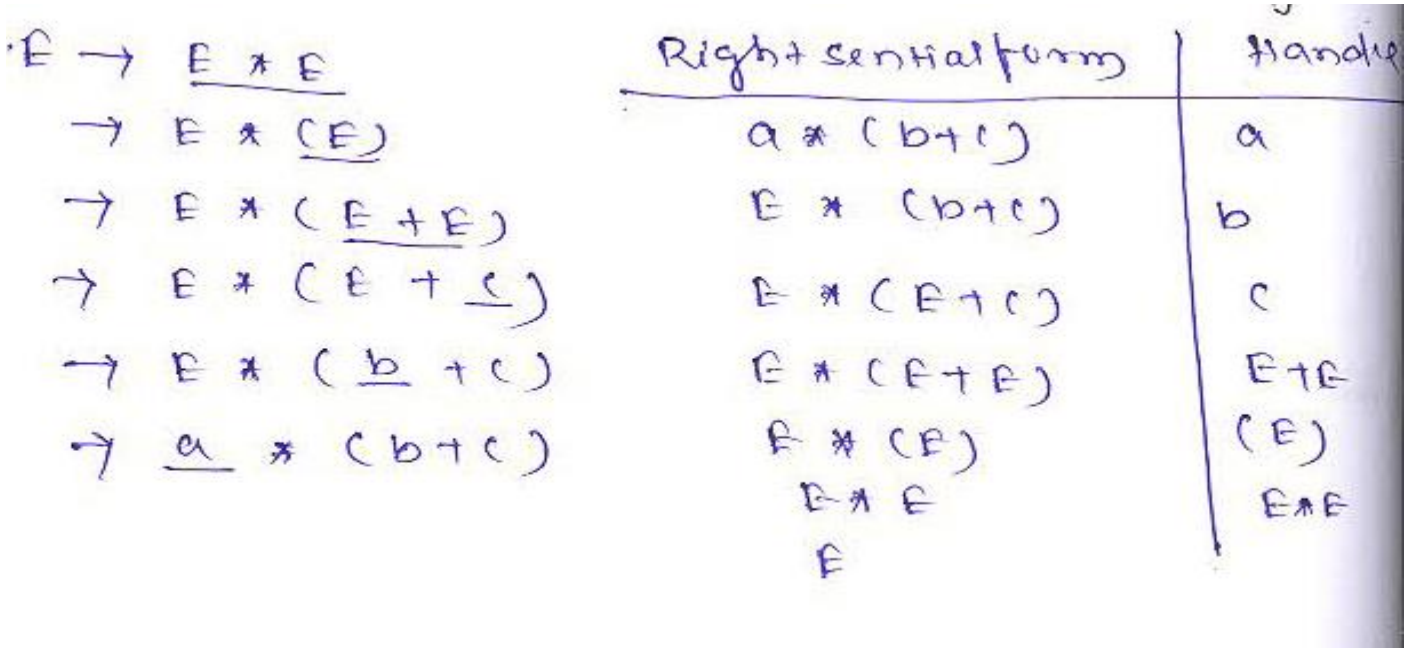
4.Q.What is shift reduce parsing.

E → E + E / E * E / (E) / a / b / c

w = a * (b+c)

Parse the string using Shift Reduce parsing Algorithm

Ans:



E → E * E
→ E * (E)
→ E * (E + E)
→ E * (E + c)
→ E * (b + c)
→ a * (b + c)

Right sentential form	Handle
a * (b+c)	a
E * (b+c)	b
E * (E+c)	c
E * (E+E)	E+E
E * (E)	(E)
E * E	E*E
E	

Stack	input	Action
a	a*(b+c)$	shift a
E	a*(b+c)$	Reduce E→a
E*	*(b+c)$	shift *
E*(	(b+c)$	shift (
E*(b	b+c)$	Reduce E→b
E*(E	+c)$	shift +
E*(E+	c)$	shift (
E*(E+(	)$	Reduce E→(
E*(E+E	)$	Reduce E+E→E
E*(E	)$	shift)
E*(E)	$	Reduce (E)→E
E*E	$	Reduce E→E*E
E		Accepted

5.Q.

$S \to AA$

$A \to aA \mid b$ construct the LALR parsing table by finding the LR(1) items.

Augmented grammer

$S' \to .S$,

$S \to AA$

$A \to aA \mid b$

closure ($S' \to .S$)

$S' \to .S, \$$

$S \to .AA, \$$

$A \to .aA, a/b$

$A \to .b, a/b$] I_0

GOTO (I_0, S)

$S' \to S., \$$] I_1

GOTO (I_0, A)

$S \to A.A, \$$

$A \to .aA, \$$

$A \to .b, \$$] I_2

GOTO (I_0, a)

$A \to a.A, \$/a/b$

$A \to .aA, a/b$

$A \to .b, a/b$] I_3

GOTO (I_0, b)

$A \to b., a/b$] I_4

GOTO (I_2, A)

$S \to AA., \$$] I_5

GOTO (I_2, a)

$A \to a.A, \$$

$A \to .aA, \$$

$A \to .b, \$$] I_6

GO TO (I_2, b)

$A \to b\cdot, \$]\ I_7$

GO TO (I_3, A)

$A \to aA\cdot, a/b]\ I_8$

GO TO (I_2, a)

$$\left.\begin{array}{l} A \to a\cdot A, a/b \\ A \to \cdot aA, a/b \\ A \to \cdot b, a/b \end{array}\right] I_3$$

GO TO (I_3, b)

$A \to b\cdot, a/b]\ I_4$

GO TO (I_6, A)

$A \to aA\cdot, \$]\ I_9$

GO TO (I_6, a)

$$\left.\begin{array}{l} A \to a\cdot A, \$ \\ A \to \cdot aA, \$ \\ A \to \cdot b, \$ \end{array}\right] I_6$$

GO TO (I_6, b)

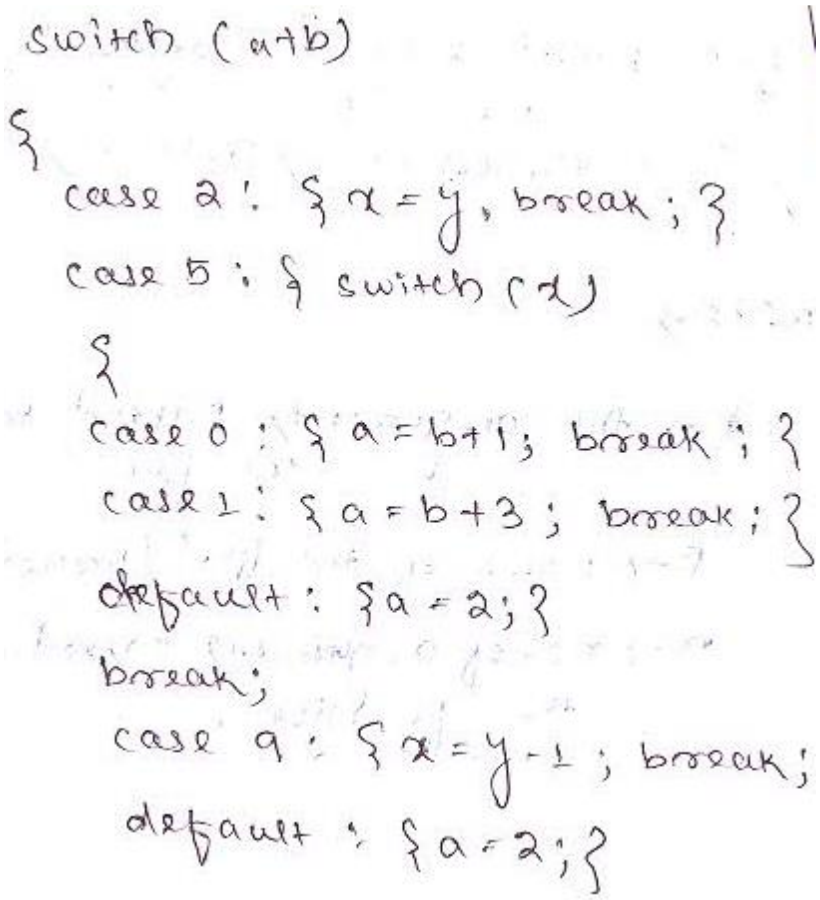
$A \to b\cdot, \$]\ I_7$

State	Action			goto	
	a	b	$	S	A
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		

$C = \{I_0, I_1, \ldots, I_9\}$

Module-2:

1.Q.Find the three address code of the following Code



Ans:Refer to the class note for answer

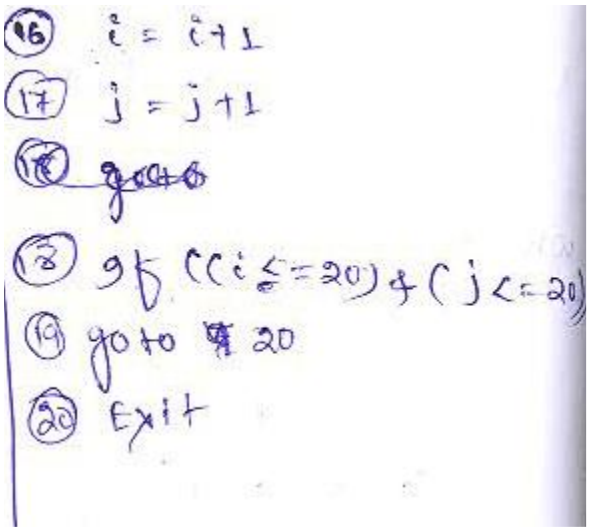
2.Q. Find the three address code of the following Code

```
main()
{
int a[20][20], b[20][20], add=0, i=1, j=1;
  do
  {
   add = add + a[i][j] * b[j][i];
       i++;
       j++;
  } while ((i<=20) & (j<=20));
```

word size of each element is 4.

(w = 4)

① add = 0
② i = 1
③ j = 1
④ $t_1 = i * 20$
⑤ $t_1 = t_1 + j$
⑥ $t_4 = t_1 * 4$
⑦ $t_2 = addr(a) - 84$
⑧ $t_3 = t_2[t_1]$
⑨ $t_4 = j * 20$
⑩ $t_4 = t_4 + i$
⑪ $t_4 = t_4 * 4$
⑫ $t_5 = addr(b) - 84$
⑬ $t_6 = t_5[t_4]$
⑭ $t_7 = t_3 * t_6$
⑮ add = add + t_7



3.Q.write and explain implementation of three address code.

Ans:

Quadruples-

In quadruples representation, each instruction is splitted into the following 4 different fields-

op, arg1, arg2, result

Here-

- The op field is used for storing the internal code of the operator.
- The arg1 and arg2 fields are used for storing the two operands used.
- The result field is used for storing the result of the expression.

Triples-

In triples representation,

- References to the instructions are made.
- Temporary variables are not used.

Indirect Triples-

- This representation is an enhancement over triples representation.
- It uses an additional instruction array to list the pointers to the triples in the desired order.
- Thus, instead of position, pointers are used to store the results.
- It allows the optimizers to easily re-position the sub-expression for producing the optimized code.

4.Q. Translate the following expression to quadruple, triple and indirect triple-

a + b x c / e ↑ f + b x c

Three Address Code for the given expression is-

T1 = e ↑ f

T2 = b x c

T3 = T2 / T1

T4 = b x a

T5 = a + T3

T6 = T5 + T4

Quadruple Representation-

Location	Op	Arg1	Arg2	Result
(0)	↑	e	f	T1
(1)	x	b	c	T2
(2)	/	T2	T1	T3
(3)	x	b	a	T4
(4)	+	a	T3	T5
(5)	+	T5	T4	T6

Triple Representation-

Location	Op	Arg1	Arg2
(0)	↑	e	f
(1)	x	b	c
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Indirect Triple Representation-

Statement	
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Location	Op	Arg1	Arg2
(0)	↑	e	f
(1)	x	b	e
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

5.Q. Translate the following expression to quadruple, triple and indirect triple-

a = b x – c + b x – c

Ans:Try by yourself according to above questions.

Module-3:

1.Q. What is peephole optimization .Explain

Ans: **Peephole optimization** is an optimization technique performed on a small set of compiler-generated instructions; the small set is known as the peephole or window.

Peephole optimization involves changing the small set of instructions to an equivalent set that has better performance.

For example:

- instead of pushing register A onto the stack and then immediately popping the value back into register A, a peephole optimization would remove both instructions;
- instead of adding A to A, a peephole optimization might do an arithmetic shift left;
- instead of multiplying a floating point register by 8, a peephole optimization might scale the floating point register's exponent by 3; and
- instead of multiplying an index by 4, adding the result to a base address to get a pointer value, and then dereferencing the pointer, a peephole optimization might use a hardware addressing mode that accomplishes the same result with one instruction.

```
...
aload 1
aload 1
mul
...
```

can be replaced by

```
...
aload 1
dup
mul
...
```

This kind of optimization, like most peephole optimizations, makes certain assumptions about the efficiency of instructions. For instance, in this case, it is assumed that the `dup` operation (which duplicates and pushes the top of the stack) is more efficient than the `aload X` operation (which loads a local variable identified as `X` and pushes it on the stack).

2.Q. **What are the Factors involved in code generation. Explain briefly.**

Ans: Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.
The following issue arises during the code generation phase:

1. Input to code generator – The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

2. Target program – The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language. • Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
• Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
• Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

3. Memory Management – Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

4. Instruction selection – Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, the respective three-address statements would be translated into the latter code sequence as shown below:

```
P:=Q+R
S:=P+T
MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

5. Register allocation issues – Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:
1. During Register allocation – we select only those set of variables that will reside in the registers at each point in the program.

2. During a subsequent Register assignment phase, the specific register is picked to access the variable.

As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example
`M a, b`
These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

6. Evaluation order – The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

7. Approaches to code generation issues: Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are: • Correct
• Easily maintainable
• Testable
• Efficient

3.Q.What is a basic block .How control flow graph can be drawn from basic block. Explain with example.

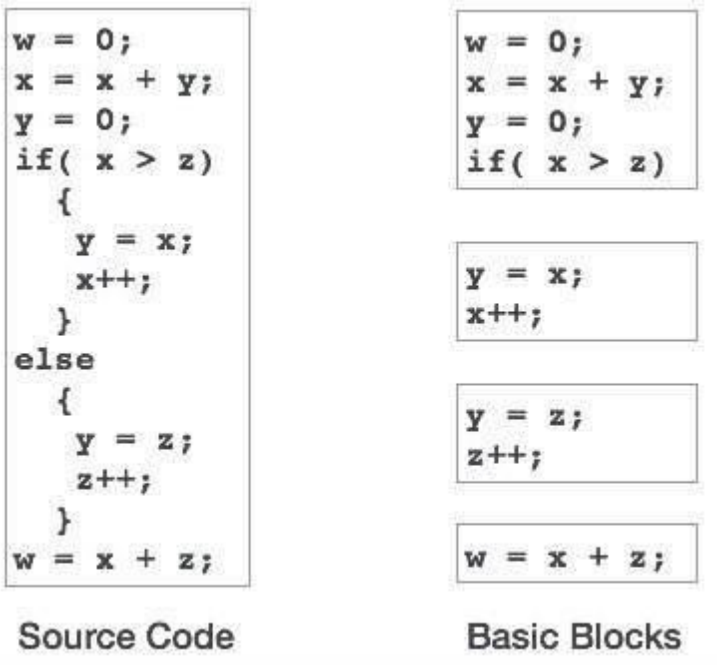
Ans: **Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.**
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.
Basic block identification
We may use the following algorithm to find the basic blocks in a program:

- **Search header statements of all the basic blocks from where a basic block starts:**

```
      o First statement of a program.
o Statements that are target of any branch
(conditional/unconditional).
o Statements that follow any branch statement.
```

- **Header statements and the statements following them form a basic block.**

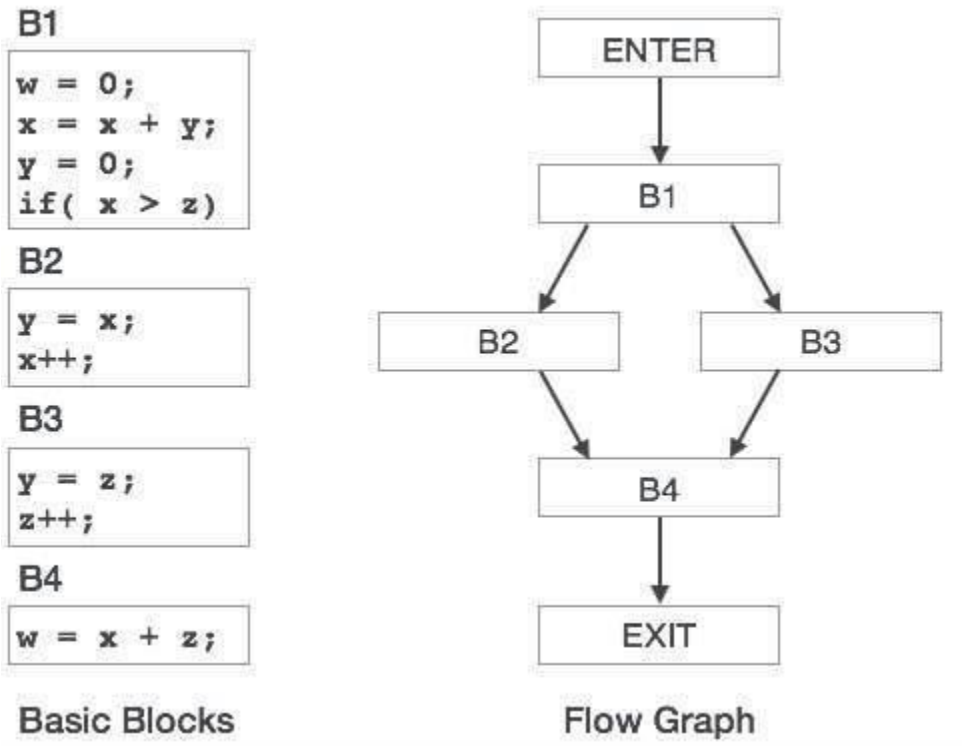
• A basic block does not include any header statement of any other basic block.
Basic blocks are important concepts from both code generation and optimization point of view.



Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



4.Q.Explain different types of code optimization technique with suitable example.

Ans:Refer to class notes.

Module-4

1.Q.Explain static and dynamic runtime environment .

Ans: A program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.

Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

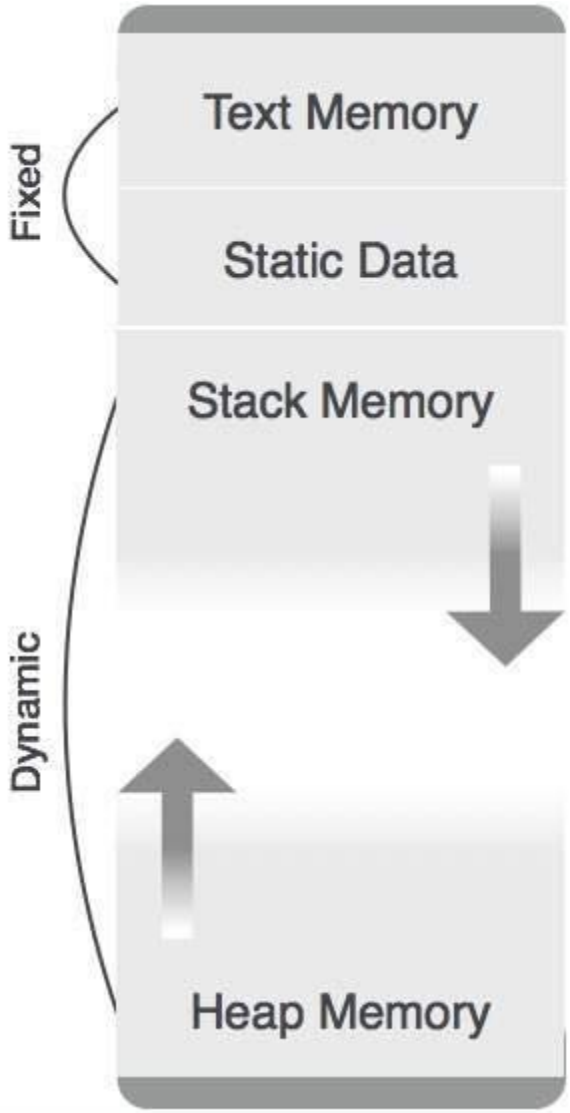
Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

- **Code** : It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.

- **Procedures** : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- **Variables** : Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Static Allocation
In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.
Stack Allocation
Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.
Heap Allocation
Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.


Text Memory
Static Data
Stack Memory
Heap Memory
Fixed
Dynamic