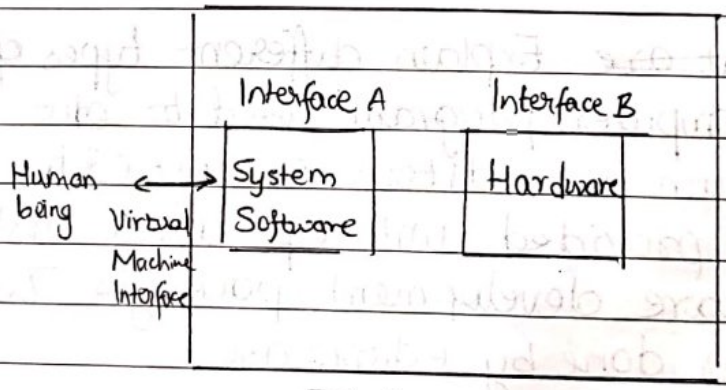
Date :
Page No. :

Q1)(a) What is system software and application software?

Ans) System software :- It consists of a variety of programs that supports the operation of a computer. System software or system program is a computer program that provides the infrastructure over which programs can generate. operate. This software makes it possible for the users to focus on an application or other problems to be solved, without needing to know the details of how machine works internally. System software is the combination of the following:

- Device Drivers
- Operating Systems
- Utilities
- Windowing System
- Servers



The Role of System Software.

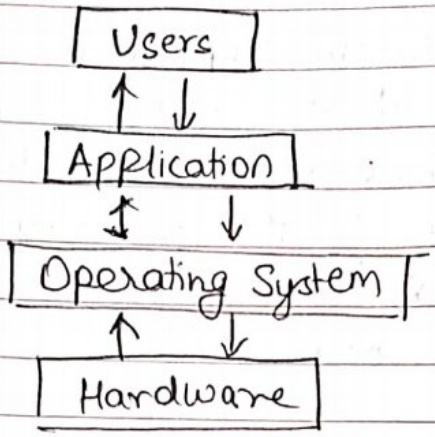
Application Software: It is a tool that functions and is operated by means of a computer and allow end users to accomplish one or more specific (not directly computer development related) tasks. Typically application software includes :-

- Industrial

Scanned with CamScanner

Date :
Page No. :

- Office Suites (eg Microsoft office)
- Business Software's
- Computer Games.
- Databases
- Medical Software's
- Media Players etc.



~~Q2)~~

b) ~~What are~~ Explain different types of text editors.

Ans) A computer program used to edit the text is known as editors or text editors. These programs are provided with operating systems and software development packages. The various tasks done by editors are:-

- Editing configuration files
- Editing document files
- Editing programming language source code.

Types of Text Editors:-

- Microsoft Windows:- Notepad, WordPad
- Unix: Pico, Vi and Emacs editors
- Mac OS: SimpleText and TextEdit.
- World Wide Web: offers a variety of HTML editors for creating web pages.

Scanned with CamScanner

- Source Code Editors for writing the source code of various languages.
- Tex or LaTex used by various mathematicians, physicists, Scientists etc.

(c) Explain left recursion with example.

Ans) A grammar is said to be left recursive if it has a non-terminal, say A, such that there is a derivation $A \Rightarrow A\alpha$, for some string α. Presence of left recursion creates difficulties while designing the parsers. Left recursion may be of two types:

1. Immediate left recursion.
2. General left recursion.

- Top-down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left-recursion is needed. It means driving the input in one or more steps. Here A is non terminal and α denotes some input string. This expansion of A causes further expansion of A only and due to generation of $A, A\alpha, A\alpha\alpha, A\alpha\alpha\alpha$ the input pointer will not be advanced. This causes major problem in top down parsing and therefore elimination of left recursion is a must.

Consider the following left-recursive grammar for arithmetic expressions.

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid id$$

Elimination of immediate left recursion

Scanned with CamScanner

from the rules modifies the grammar as

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' / \varepsilon$$
$$T \rightarrow FT'$$
$$T \rightarrow *FT' / \varepsilon$$
$$F \rightarrow (F) / id.$$

(d) Write a note on: Input buffering scheme of lexical analyser.

Ans) To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.

- Hence a two-buffer scheme is introduced to handle large look ahead ~~safely~~ safely.
- Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer ~~ends~~ end have been adopted. There are three general approaches for the implementation of a lexical analyzer:-

(i) By using a lexical analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.

(ii) By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the

Scanned with CamScanner

Date :

Page No. :

input.

(iii) By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Scanned with CamScanner

SPCC
May 2017

Q.2 (a) With reference to assembler, explain the following tables with suitable example.

(i) POT

Pseudo Opcode Table (POT)

POT is fixed length table i.e the contents of these tables are not filled in or altered during the assembly process.

← 8-bytes per entry →

Pseudo-Opcode (5-bytes) (characters)	Address of routine to process Pseudo-Opcode (3bytes = 24 bit address)
"DROPbb"	P1DROP
"ENDbb"	P1END
"EQUbb"	P1EQU
"START"	P1START
"USING"	P1USING

↑ These are presumably P1→PASS1 lables of routines in pass 1; the table will actually contains the physical addresses

(ii) MOT Machine Opcode Table (MOT)

MOT is a fixed length table, i.e, the contents of these tables are not filled in or altered during the assembly process.

Machine – Op Table (MOT) for pass 1 and pass 2

← 6bytes per entry →

Mnemonic op-code (4-bytes) (characters)	Binary op-code (1-byte) (hexadecimal)	Instruction length (2-bits) (binary)	Instruction format (3-bits) (binary)	Not used in this design (3-bits)
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
"ALbb"	5E	10	001	
"ALRb"	1E	01	001	
"ARbb"	1A	01	001	
....	....	....	.....	
"MVCb"	D2	11	100	

b ~ represents the character "blank"

Codes

Instruction length

01 = 1 Half-Words = 2bytes
10 = 2 Half-Words = 4 bytes
11 = 3 Half-Words = 6 bytes

Instruction format

000 = RR – Register to Register
001 = RX – Register and Indexed Storage Operation
010 = RS – Register and Storage Operation
011 = SI – Storage and Immediate Operation
100 = SS – Operation using all implied operand and storage

Scanned with CamScanner

SPCC
May 2017

(c) Symbol Table (ST)

Symbol (8-bytes) (characters)	Value (4-bytes) (hexadecimal)	Length (1-bytes) (hexadecimal)	Relocation (1-byte) (character)
"JOHNbbbb"	0000	01	"R"
"FOURbbbb"	000C	04	"R"
"FIVEbbbb"	0010	04	"R"
"TEMPbbbb"	0014	04	"R"

Symbol Table includes for each entry:
(i) Name of the symbol
(ii) Symbols assembly time value
(iii) Length (in bytes) and
(iv) Relative Location Indicator (R/A)
Operand can be either immediate (typically one byte values coded in the instruction itself) or the addresses of the data located elsewhere.
(Relative / Absolute value)

(d) Literal Table (LT)

Name of Literal (1-byte)	Value of Literal (4-byte)	Length of Literal (1-byte)	Relative Location Indicator (1-byte)

Literal equivalent to define a constant explicitly and assign an address label for it literal table is often organized as a hash table, using literal name or value as the key.

Literal Table includes for each entry

(i) Name of the literal
(ii) Literal assembly time value
(iii) Length(in bytes) and
(iv) Relative Location Indicator (R/A)

Name of Literal (1-byte)	Value of Literal (4-byte)	Length of Literal (1-byte)	Relative Location Indicator (1-byte

(b) Explain the different code optimization techniques in compiler

⇒ The different code optimization techniques in compiler design. design are as follows:—

A) Compile Time Evaluation

In this technique, expressions whose values can be pre-computed at the compilation time and used, perhaps repeatedly, in the execution of the program are identified. There are two ways:

1) Constant Folding

Evaluation of an expression with constant operands to replace the expression with single value.

Example: area := (22.0/7.0) * r ** 2 is replaced with

area := 3.14286 * r ** 2

SPCC
May 2017

2) Constant Propagation

Replace a variable with constant which has been assigned to it it earlier.

Example :

```
pi := 3.14286     area = pi * r ** 2
        ⟶         area = 3.14286 * r ** 2
```

B) Common Sub-Expression Elimination

Two operations are common, if they produce the same result. In such case, it is likely more efficient to compute the result once and reference it to the second time rather than re-evaluate it. An expression is alive if the operands used to compute the expression have not been changed. An expression that is no longer alive is dead.

Example :

```
a := b * c                    temp := b * c
...                           a := temp
...          ⟹               ...
x := b * c + 5                x := temp + 5
```

c) Dead Code Elimination

Dead Code is a portion of the program which will not be executed in any path of the program. If an instruction's result is never used in the program. If an instruction's result is never used in the program, the instruction is considered to be "dead" and can be removed from the instruction stream.

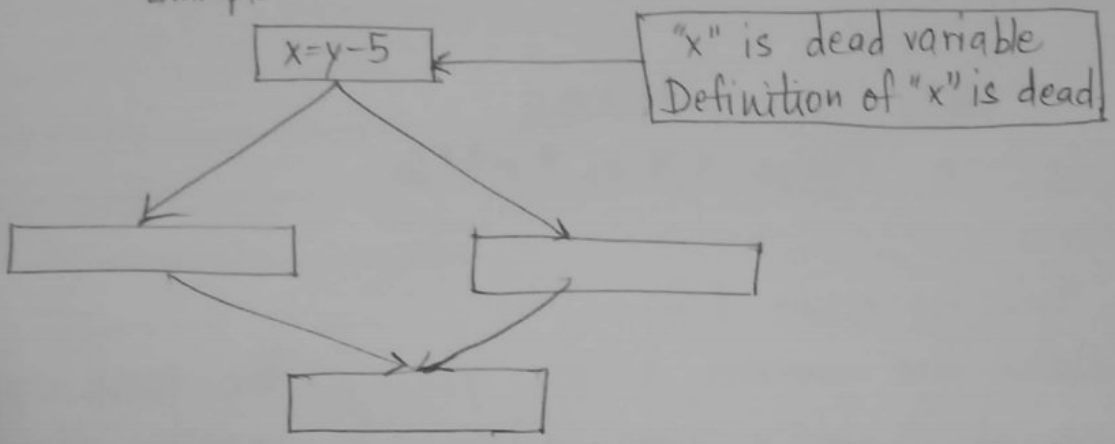
Scanned with CamScanner

So if we have

tmp1 = tmp2 + tmp3

If tmp1 is never used again, we can eliminate this instruction altogether. However, we need to be careful and check whether tmp1 holds the result of a function call: tmp1 = swap();

Example:



Always take care of side effects in code during dead code elimination.

D) Copy Propagation

→ This optimization is similar to constant propagation, but generalized to non-constant values. Copy Propagation is the process of removing them and replacing the copies generate by the compilers in an intermediate form. It often reveals dead-code.

Example :

```
tmp2 = tmp1;
tmp3 = tmp2 * tmp1;
y = x;
z = 3 + y
```

⇩

```
tmp3 = tmp1 * tmp1
z = 3 + z
```

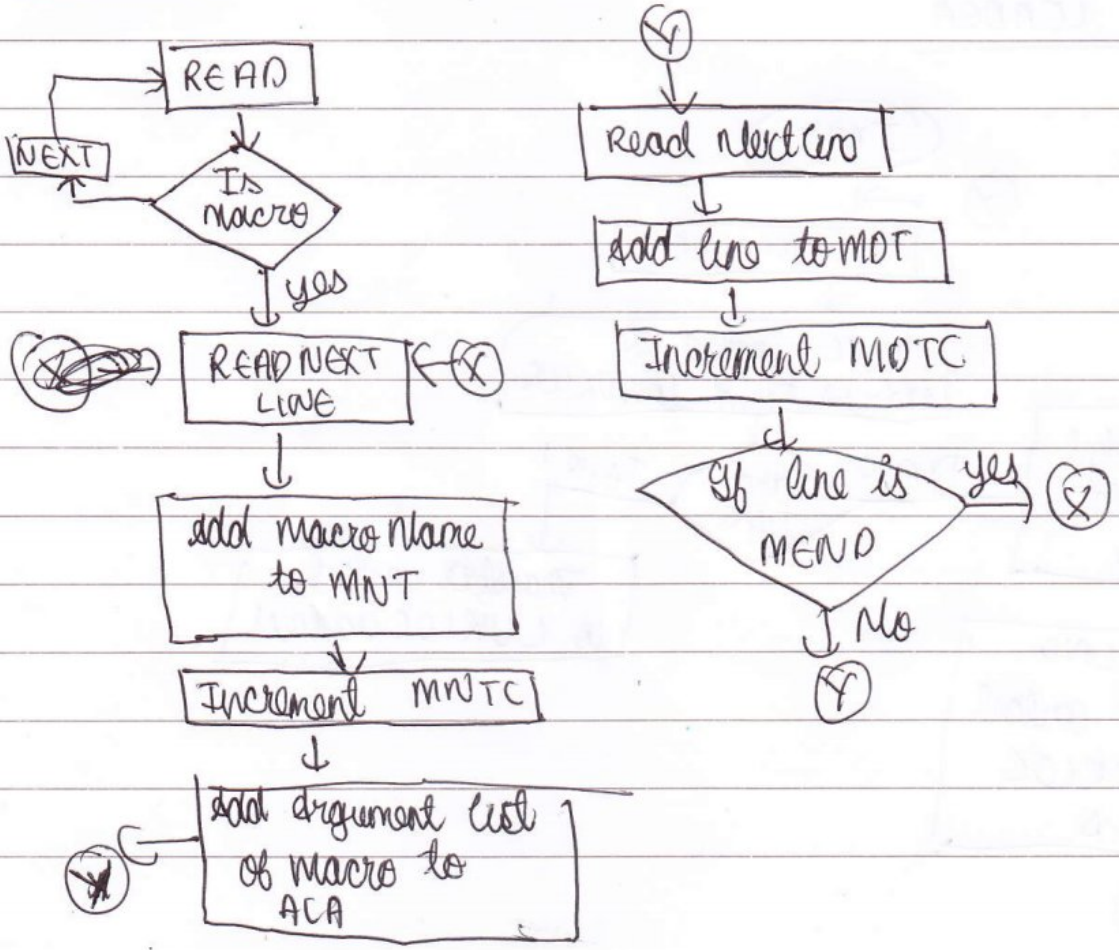

Scanned with CamScanner

Q3 PASS I of MACRO PROCESSOR

1. DATABASE

* MNT :- Stores the Macro name
* MDT :- Used to perform Macro expansion
* MDTC :- used to point index of MDT. starting
* ALA :- Used to replace index argument with it's actual value
* INPUT :- The file containing the input source code

FLOWCHART



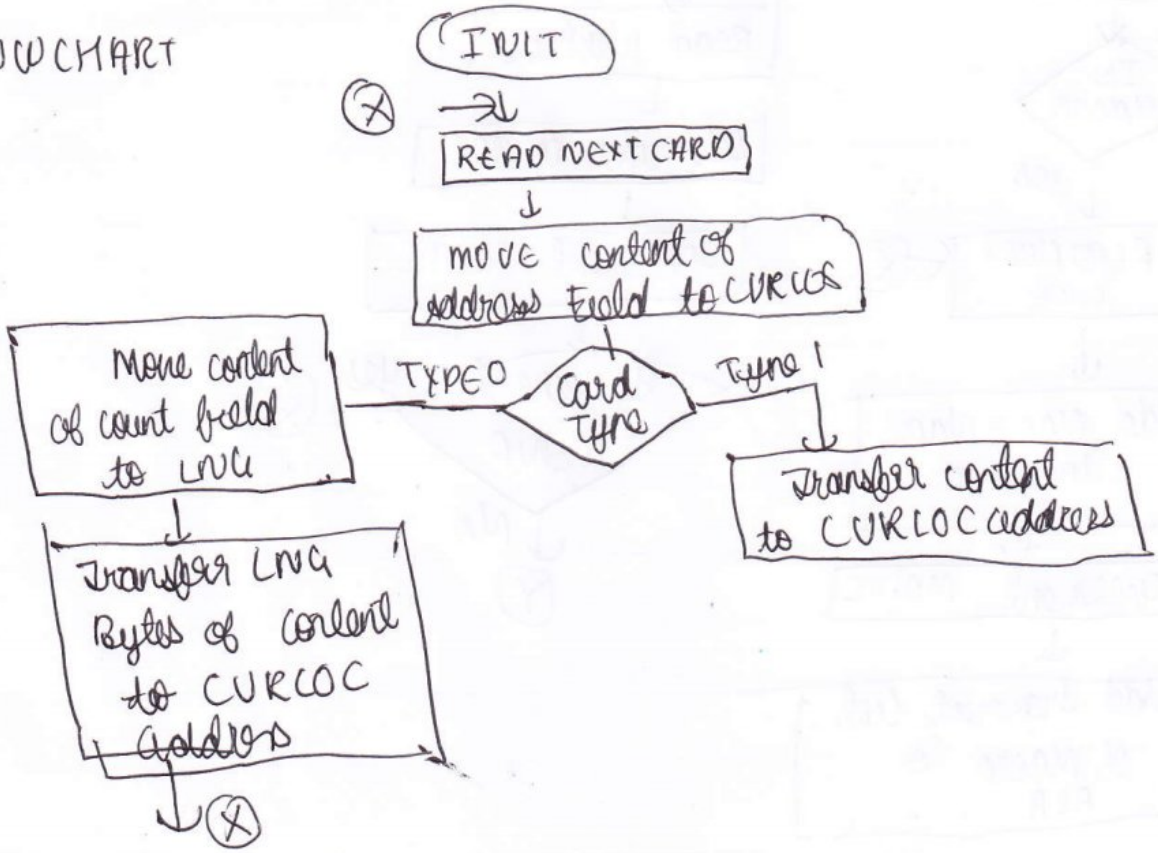
Q3 (b)

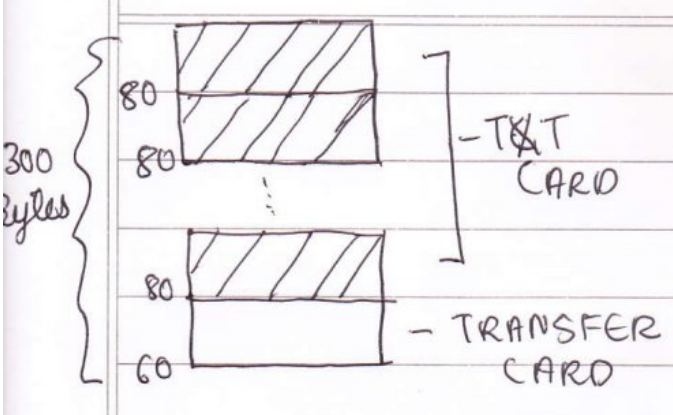
FUNCTIONS of LOADER

* ALLOCATION
  allocate space in memory where object is loaded for execution using size of program
* LINKING
  link multiple object codes if read & provide info required to allow references between them by assigning user & library subroutine address
* RELOCATION
  modifies program to ensure that a program can be loaded from an address different than original
* LOADING
  physically load instructions to memory to start execution

ABSOLUTE LOADER

FLOWCHART





This is O/P generated by assembler as i/p for loader

DESIGN

It accepts machine code & places it in main memory specified. places it into core, initiates executions by transfering control to start of the program with data stored on cards instead of main memory. It's loaded at specific memory locations. At the end, loader jumps to specified location to begin execution

SINGLE PASS OPERATIONS :-

- Header Record (H) :- Verifies correct program present for loading
- TEXT RECORD (T) :- Move object code into address indicated
- END RECORD (E) :- Jump specified address to begin execution

Q.4

a) compare LR(0), LR(1) and LALR parser.

LR(0)

An LR(0) item is a production G with dot at some position on the right side of the production LR(0) items is useful to indicate that how much of the inputs has been scanned up to a given point in the process of parsing. In the LR(0), we place the reduce node in the entire row. An LR(0) parser is a shift reduce parser that uses zero tokens of lookhead to determine what action to take.

LR(1)

The parsing table for the canonical LR(1) parser is constructed in a similar way as with LR(0) parsers except the items that the items in the item set also has a follow symbol. A LR(1) item is a production with a marker together with a terminal. such rules states that how much of a production we have already processed (x, P) what we expect next (Q, y)

Scanned with CamScanner

The only difference between LR(0) and SLR(1) is this extra ability to help decide what action to take when there are conflicts.

LALR

LALR parser are same as CLR parser with one difference. In CLR parser if two states differ only in lookhead then we combine those states in LALR parser. After minimization if the passing table has no conflict that the grammar is LALR also. It is a LookAhead LR

To construct we use canonical collection of LR(1) items.

Scanned with CamScanner

b] Explain different ways to represent three address code.

→ Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addreses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

General representation –

$a = b \text{ op } c$

where a, b, or c represents operands like names, constants or compiler generated temporaries and op represents the operator.

example converting expression $a * -(b+c)$ into three address code.

$t_1 = b + c$

$t_2 = \text{uminus } t_1$

$t_3 = a * t_2$

Scanned with CamScanner

Implementation of Three addres code

There are 3 representations of three address code

1) Quadruple.

It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes operator and arg1 and arg2 denotes the two operator and result is used to store the result of the expression.

2) Triples -

This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triples value is needed, a pointer to that triple is used. So it consist of only three fields namely op, arg1 and arg2.

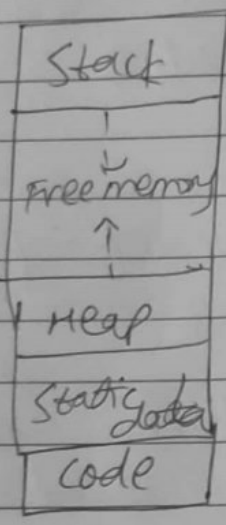
3) Indirect Triples.

This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to reaarrange code.

Scanned with CamScanner

q.5 Explain run time storage organization in detail

→ An executable program generated by a compiler will have following organization in memory on a typical architecture



This is the layout in memory of an executable program.
The machine code of program is typically located at the lowest part of the layout. Then, after the code there is a section to keep all the fixed size static data in the program.
The dynamically allocated data (i.e the data created using malloc in C) as well

Scanned with CamScanner

Dynamic Allocation:-

The allocation can be varied during execution. It makes use of recursive function.

In a dynamic storage allocation strategy, the data area requirements for a program are not known entirely at compilation time.

Stack Allocation:-

Procedure calls and their activation are managed by means of stack memory allocation.

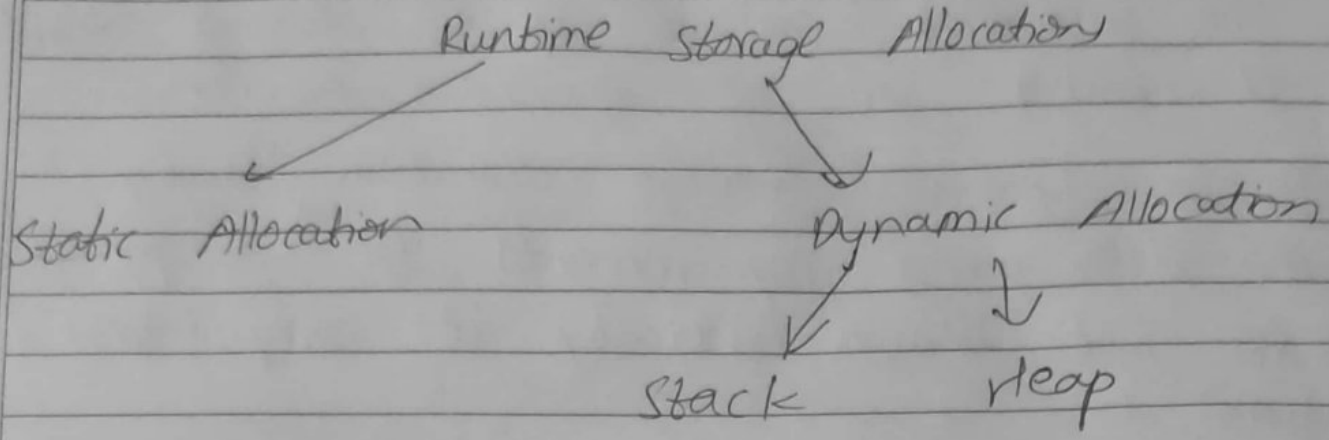
It works in LIFO (Last in First out) and this allocation is very useful for recursive procedure calls.

Heap Allocation:-

Variables local to a procedure are allocated and ~~dlt~~ deallocated only at runtime.

Heap allocation is used to dynamically allocate memory to the variables and claim it back when variables are no more required.

Scanned with CamScanner



Static :-

In this allocation scheme, the compilation data is bound to a fixed location in memory and it does not change when 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.

In static, it is necessary to decide at compile time exactly where each data object will reside at run time. Two factors

1. The size of each object must be known at compile time.
2. only one occurrence of each object is allowable at a given moment during program execution.

Scanned with CamScanner

as the static data without a fixed size are created and kept in the heap.

The focus of this section is the stack in the memory layout. It is called the run-time stack.

Activate Record:
- It is used to store the current record
- It contains return value.

Paramater:
It specifies the no. of parameters used in functions

Local Data:-
The data that is been used inside the function is called a local address.

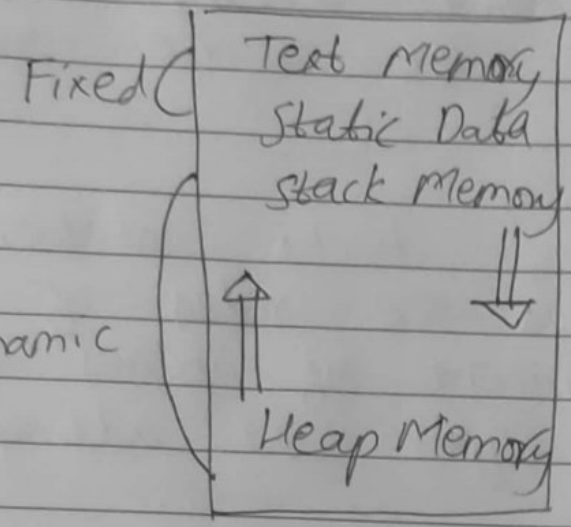
Temporary Data:-
It is used to store the data in temporary variables.

Links:-
It specifies the additional links that are required by the program.

Status:-
It specifies the status of program that is the flag used.

Scanned with CamScanner

Except static, both stack and heap can grow and shrink dynamically.
Therefore, they cannot be provided with a fixed amount of memory in the system.



Q.5 Explain different phases of compiler.

The different phases are:-

=> Lexical analysis:-

1. The first phase of scanner works as a text scanner.
2. This phase scans the source code as a stream of characters and converts it into meaningful lexemes.
3. Lexical analyzer represents these lexemes in form of tokens.
4. Example tokens are identifiers, constants, etc.

Scanned with CamScanner

Syntax Analysis:-

1. The next phase is called the syntax analysis or parsing.
2. It takes the token produced by lexical analysis as input and generates a parse tree.
3. In this phase, token arrangements are checked against the source code grammar.

Semantic Analysis:-

1. semantic analysis checks whether the parse tree follows the rules of language.
2. Fore eg) assignment of values is betn compatible data types and adding string to an integer.

Intermediate code Generation:-

1. It represents a program for some abstract machine.
2. It is betn the high level and machine language.
3. The intermediate code should be generated in such a way that makes it easier to translated into ~~the~~ target machine code

Scanned with CamScanner

Code Optimization:

1. The next phase does code optimization of intermediate code.
2. Optimization can be assumed as something that removes unnecessary code lines and arranges the sequence of statements in order to speed up the program execution.

Code Generation:

1. The code generator takes the optimized representation of intermediate code and maps it to target machine language.
2. The code generator translates the code into a sequence of relocatable machine code.

Symbol table:-

1. It is a data structure maintained throughout all phases of a compiler.
2. All the identifier's names along with their types are stored here.
3. It is also used for scope management.

Error handling:-

It handles all errors during compilation.

Scanned with CamScanner

86)(a) Top Down Parsing	Bottom up parsing
i) 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 rules of grammar.
ii) 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.
iii) Here we start from Symbol of Presentee to the leaf node of Parse tree	Here we Start from leaf node of Parse tree to the start Symbol of Parse tree.
iv) Left most Derivation used	Right most Derivation used.
v) main decision is what Production rule to use in order to construct the String	main decision is select when to use a Production rule to reduce the String to get Start Symbol.

Recursive Descent Parser:-

It is a kind of Top-Down Parser. A Top-down Parser builds the Parse tree from top to down, starting with the start non-terminal. A Predective Parser is a special case of Recursive Decent Parser, where no Backtracking is required.

By carefully writing a grammar means eliminating left recursion and left factoring from it, the left recursion and left a factoring from resulting grammar will be a grammar that can be parsed by recursive descent parser.

Eg:-

Before removing left recursion	after removing left recursion.
$E \to E+T \mid T$	$E \to T\,E'$
$T \to T*F \mid F$	$E' \to +T\,E' \mid \epsilon$
$F \to (E) \mid id$	$T \to F\,T'$
	$T' \to *F\,T' \mid \epsilon$
	$F \to \epsilon (E) \mid id$

Q6 8]). The basic block is a sequence of consecutive statements which are always executed in sequence without halt or possibility of branching.
- The basic blocks doesn't have any jump statements among them.
- 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.

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

Three address code -

$t_1 = b + c$

$t_2 = t_1 + e$

$a = t_2$

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

Rules :-
- The basic blocks are the nodes to the flow graph
- The block whose leader is the first statement is called initial block.
- There is a direct edge from block B_1 to block B_2 if B_2 immediately follows B_1 in the given sequence. we can say that B_1 is a predecessor of B_2

Q6 B) ii) Java compiler Environment :-

A Java virtual machine (JVM) is an abstract computing machine that enables a computer to run Java program. There are three notions of the JVM:

i) Specification
ii) implementation
iii) instance.

Java compiler :-

Javac compiles Java programs :-

Java C [options] filename.java ...
Javac -g [options] filename.java ...

The javac command compiles java source code into java bytecode. you then use java interpreter - the java command - to interpret the java bytecode. Java source code must be contained in files whose filenames end with the .java extension. The compiler looks in the class file, recompiling the source if it is newer. Set the property javac.Pike output to true to send output messages to System.out New portable java compile-Time Environment.