Date
Page 1

70 - Sakshi S. Ghonge. Dec 2022.

Q.1. Differentiate between Application software and System software.

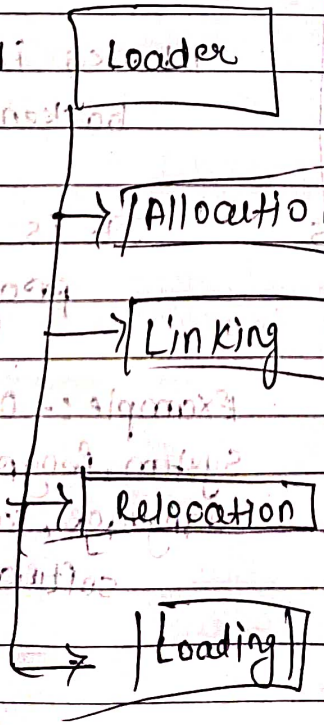
	Application software	System Software.
1.	It is designed and built for specific task	Gives path to software application to run
2.	It's specific software	It's a general purpose software
3.	Programming is not complex like system software	Programming is complex compared to Application Software
4.	Doesn't take hardware into consideration, as it doesn't interact directly	Interacts with the hardware directly.
5.	Installed by the users when they needed	Installed by the manufacturers.
6.	Works in the frontend	Works in the backend.
7.	Can be said to be resources sharing client servers	It is a packeged program
8.	Example : Word doc, Spreadsheet, Database Internet explorers.	Example :- Operating System, Programming language, communication software.

Q.2

B) What are the functions of Loader? Enlist the loader scheme.

→ • A loader is a system program, which takes the object code of a program as a input and prepares it for execution
• Programmers usually define the program to be loaded at some predefined location in the memory
• But this loading address given by the programmer is not be coordinated with OS
• The loader does the job of co-ordinating with the OS to get initial loading address for the .EXE file and load it into memory.

• Loader performs the following functions.
1) Allocation 2) Linking 3) Relocation 4) Loading.



1] Allocation :-

- Allocates the space memory where the object program would be loaded for execution.
- It allocates the space for program in the memory by calculating the size of the program. This activity is called as Allocation.
- In absolute loader allocation is done by the programmer and hence it is the duty of the programmer to ensure that the programs do not get overlap.
- In reloadable loader allocation is done by the loader hence the assembler must supply the loader the size of the program.

2] Linking :-

- It links two or more object codes and provides the information needed to allow references between them.
- It resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses. This activity is called linking.
- In relocatable loader linking is done by the loader and hence the assembler must supply to the loader, the locations at which the loading is to be done.

3) Relocation :-

- It modifies the object program by changing it the certain instructions so that it can be loaded at different address from location originally specified.
- There are some address dependent locations in the program, such address constants must be adjusted according to allocated spaces, such activity is done by loader is called relocation.
- In absolute loader relocation is done by the assembler is aware of the starting address of the program.

4] Loading :-

- It brings the object program into the memory for execution.
- Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution. This activity is called loading.
- In both loaders (absolute, relocatable) loading is done by the loader and hence the assembler must supply to the loader the object program.

Q.9

c. Explain Macro and Macro Expansion with example

→ A macro is a preprocessor directives that defines a code template, which can be used to generate code during compilation process. Macros are typically used for code generation tasks or for defining constants, variables or functions that are used throughout a codebase. Macros are defined using a special syntax, and they are processed by preprocessor which is a step that occurs before the actual compilation of source code.

Macro expansion is the process of replacing macro invocations with their corresponding code templates during the compilation process. When a macro expansion process replaces the macro invocation with actual code defined in the macro which is then compiled along with the rest of source code.

Example :-

```
# define SQUARE (x) ((x)*(x))
int main() {
int a = 5;
int b = square(a);
return 0;
}
```

In this example, the ~~squar~~ 'SQUARE' macro is defined to calculate the square of a given number. The macro is invoked with the argument 'a', and along the macro expansion process, the macro

invocation 'SQUARE' (a)' is replaced with the corresponding code template '((a) * (a))', so, the code after macro expansion would look like this:

```
int main () {
    int a = 5
    int b = ((a) * (a));
    return 0;
}
```

It is important to note that macros are expanded by preprocessor, which operates on the source code before the actual compilation process. This means that macro expansion happens before the code is compiled into machine code, and the resulting expanded code is then compiled along with the rest of the source code to generate the final executable program.

Date
Page

D] Compare Bottom-up and Top-Down Parsers

	Top-down	Bottom-up
1.	Construct tree from root to leaves	Construct tree from leaves to root
2.	"Guess" which RHS to substitute for non-terminal	"Guess" which rule to reduce terminals
3.	Produces left-most derivation	Produces reverse right-most derivation
4.	Recursive descent; LL Parsers	Shift-reduce, LR LALR e.t.c.
5.	Easy for humans	Hard for humans.
6.	Picks a production and tries to match the input	Start in a stated valid for legal first tokens
7.	May require backtrac	as input is consumed, change state to encode possibilities.
8.	Some grammers are backteack-free	Use stack to store both state and sentential forms.

Q.2. Explain with the flowchart design of two pass assemblee.

→ ~~As a language needed, I am not able to generate~~

A two pass assembler is a software program that converts assembly language code into machine code. It typically consists of two passes or phases, the first pass and the second pass. Each pass serves a specific purpose in analyzing and processing the assembly language code.

following steps are involved in two pass assemblee.

1. Input :- The assemblee takes the assembly language code as input. This code is usually written by an programmee using mnemonics and symbolic names to represent machine instructions and memory locations

2. First Pass :-

a) Lexical Analysis :- The assemblee scans the input code and performs lexical analysis, which involves breaking the code into smallee units called tokens, such as instructions, labels operators and comments.

b) Symbol Table Generation :- The assemblee builds a symbol table during the first pass. The symbol table keeps track of all lables and their corresponding memory address is used in the code.

3. Second Pass :

a. Syntax Analysis :- The assemblee performs syntax analysis during the second pass. It checks the syntax of the assembly language instructions and operands based on the rules of the target machine's instruction set.

b) Machine Code generation :- The assembler generates machine code during the second pass. It uses the information from the symbol table and the intermediate code to replace symbolic references with actual memory addresses and generate the corresponding machine instructions.

c) Output :- The assembler generates the machine code as output which can be loaded into the memory addresses and generate the corresponding machine instructions.

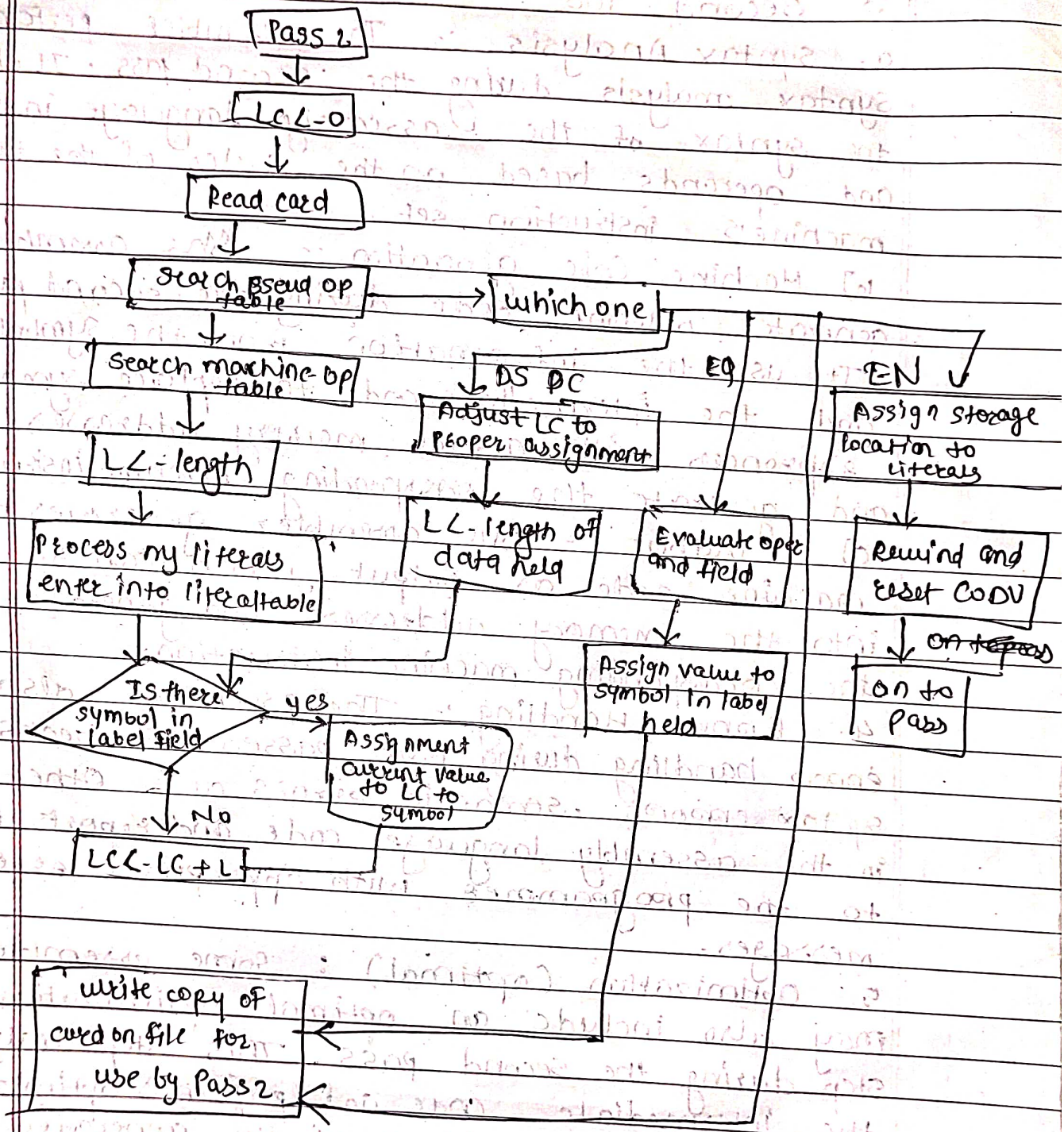
4. Error Handling :- The assemblee also performs error handling during both passes. It checks for syntax errors, semantic errors and other issues in the assembly language code and reports them to the programmee with appropriate error messages.

5. Optimization (optimal) : Some assemblees may also include an optimal optimization step during the second pass. This step analyzes the intermediate code and performs optimizations to improve the efficiency of the generated machine code, such as removing redundant instructions or reordering instructions for better performance.

dec 2022

Date
Page

Flowchart :-



Date
Page 11

Q.2.B Construct the three address code for the following program.

```
For(i = 0 ; i<10; i++)
  {
  If (i<5)
  a = b+c * 5
  else
  x = y+z ;
  }
```

1) i = 0 ;
2) if i<10 goto (4)
3) goto
4) t1 = i+1
5) i = t1
6) goto (2)
7) t2 = y+2
8) x = t2
9) goto 4.

Q.3. Explain different features of macros with suitable example.

→ There are mainly 4 important features of macro they are :-

1] Macro Instruction Argument :-

- To overcome lack of flexibility problem, we use macro instruction arguments where these arguments appear in macro call.
- The correspondingly macro dummy arguments appear in macro definition.

Example :-

Source

```
Macro
INCR &arg1, &arg2, &arg3, &lab1
A1, &arg1
D2, &arg2
A3, &Arg3
MEND
:
Loop1 INCR data1, data2, data3
:
Loop2 INCR data3, data2, data1.
:
data 1 . DC F'5'
data 2 DC F'6'
data3 DC F'7'
```

Expanded Source Code

```
:
Loop 1, A1, data 1
A2, data 2
A3, data 1
:
Loop 2 A1 data 3
A2, data 2
A3, data 1
:
data 1, DC F'5'
data 2 DC F'6'
data 3 DC F'7'
```

Q.3. Explain different features of macros with suitable example.

→ There are mainly 4 important features of macro they are :-

1] Macro Instruction Argument :-

- To overcome lack of flexibility problem, we use macro instruction arguments where these arguments appear in macro call.
- The correspondingly macro dummy arguments appear in macro definition.

Example :-

Source Macro

```
INCR &arg1, &arg2, &arg3, &lab1.
A1, &arg1
D2, &arg2
A3, &Arg3
MEND
:
Loop1 INCR data1, data2, data3
:
Loop2 INCR data3, data2, data1.
:
data1 DC F'5'
data2 DC F'6'
data3 DC F'7'
```

Expanded Source Code

```
:
Loop1, A1, data1
A2, data2
A3, data1
:
Loop2 A1 data3
A2, data2
A3, data1
:
data1, DC F'5'
data2 DC F'6'
data3 DC F'7'
```

2) Conditional Macro Expansion

- This feature helps to defining a group of similar that appear in expansion of macro call.
- There are two important macro-processor pseudo operation, they are → AIF → AGO.

AIF → is a conditional branching pseudo operation, the are format of AIF is
AIF < expression < sequencing label.

AGO → is unconditional branching pseudo operation format of AGO is
AGO < sequencing model.

FINI → is a macro label
→ it dirts macro processor to skip to statements labelled FINI if parameters corresponding to & labelled count is 1, otherwise macroprocessor continues with statement following AIF pseudo op.

Example :- Source Code

```
MACRO
ADD1, &arg.
L1 ' &arg.
A1 = F'10'
ST, &arg
MEND
Loop 1, INCR 3, data 1, data 2, data 3
Loop 2, INCR 2, data 3, data 2
Loop 3, INCR 1, data 1
data 1 DC '5'
data 2 DC '6'
data 3 DC '7'
```

Expanded code.

```
loop, A1, data 1
A2, data 2
A3, data 3.
loop 2, A1, data 3
A2, data 2.
loop 3 A, data 3
data 1, DC 5
data 2, DC '6'
data 3, DC '7'
```

Macro calls within macro :-

- macro calls can be called only after being defined. sometime one macro will call another macro such as concept called macro calls within macros.
- The defination of called macro should appear before the macro defination which containes the macro call.

Source Code

```
MACRO
ADD1, &arg
L1, &arg
A1, = F'10'
ST, &arg
MEND
MACRO
ADDS, &arg 1
&arg 2, &arg3
ADD1, &arg1
ADD1, &arg2
ADD1, &arg3
MEND
:
ADDS data1, data2, data3
:
data1 DC F '5'
data2 DC F '6'
data3 DC F '7'
END.
```

Level 1

```
Expansion of ADDS
:
ADD1, data1
ADD1, data2
ADD1, data3
```

Level 2.

```
L1, data1
A1 = F,,10"
ST1, data1

L1, data2
A1 = F,,10''
ST, data2

L1, data3
A1 = F,,10''
ST, data3
data1 DC F '5'
data2, DC F '6'
data3, DC F '7'
END
```

4) Macro definations within macro defination.

- This feature helps to defining a group of similar macros using a single macro instructions.
- It is possible to have macro defination with the body of macro. The inner macro definition with the body of macro. The inner macro defination is not defined until the outer macro has been called.

Example:-

```
Define & sub
MACRO
& sub & y
CNOP 0,4
BAL 1, *+8
DC A(& y)
L 15 =X(& SUB)
BLR 14,15
MEND
```

Defination of macro and sub

Defination of macro define.

B) Design LL(1) parsing table for the given grammer

$S \rightarrow Ad$

$A \rightarrow aB \mid BC$

$B \rightarrow b$

$C \rightarrow e \mid \varepsilon$

Also state that whether the given grammer is LL(1) or not.

→ To determine if the grammer is LL(1) or not we need to check for the following conditions :-

1. No two productions of the same non-terminal have a common prefives
2. There should be no ε (epsilon) productions in the first set of any non-terminal.
3. If non-terminal has a production that derives ε (epsilon), then the FIRST set of that non-terminal should not intersect with the follow set of the same non-terminal.

Let construct the LL(1) parsing table for the given grammer.

Non-terminal	First	Follow	Productions
S	{a}	{$}	S -> Ad.
A	{a, b}	{$, e}	A -> aB | A -> BC
B	{b}	{a, $, e}	B -> b
C	{e, ε}	{a, $, e}	C -> e | C -> E

From the above table, we can say that all entries in the parsing table are unique and there are no conflicts. Therefore, the given grammer is LL(1) because it satisfies all the condition mentioned above.

Q.4. Explain the working of a single-Pass macro processor with neat flowchart.

→ It process all macro definitions for each macro name defined.

1) Macro Name Table (MNT) : It is used to enter macro name.

2) Macro defination Table :- It is used to store the entries of macro defination.

3) Auxiliary information is added to MNT It indicates where the definition of macro can be found in MDT

4. All the statements are processed in the assembly source program to detect all macro calls.

5. for each macro call defined in macro definations :-

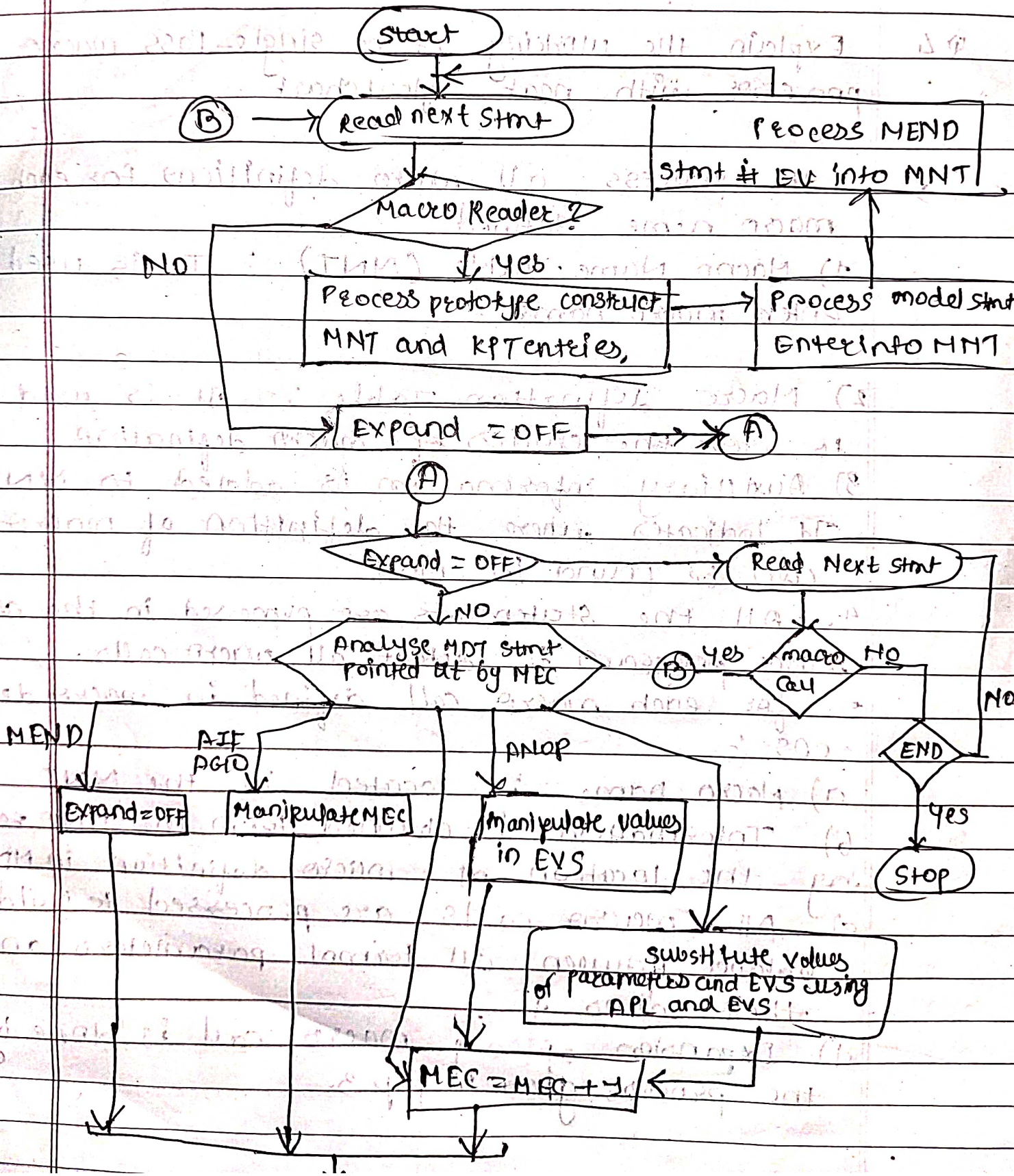
a) Macro name is located in the MNT

b) Information is obtained from the MNT regarding the location of macro definition in MDT

c) All macro calls are processed to build balance between all formal parameters and their values.

d) Expansion of all macro call is done by the process given in Step 3.

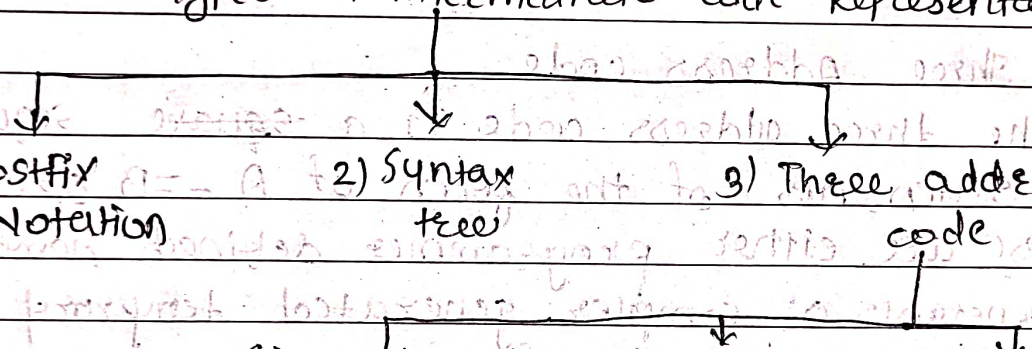
process the statements in the macro definition as found in MDT in their expansion time order until the MEND state is encountered. AIF and AGO will make changes in the certain expansion time relations between values of formal parameters and expansion time variables



Q.4.

b) What are the different ways of representing intermediate code ? Explain with suitable examples.

→ The translation of the source code into the object code for the target machine, a compiler can produce a middle-level language code, which is referred to as intermediate code or intermediate code representation.



1] Postfix Notation :- In postfix notation the operator comes after an operand i.e the operator follows an operand.

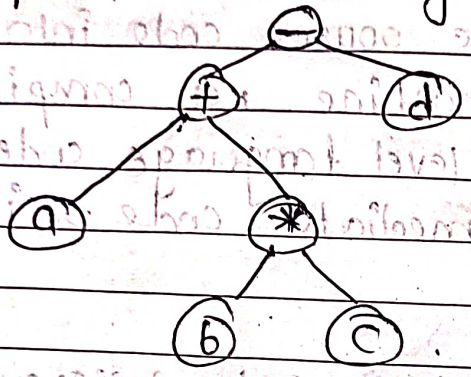
Example :-

a) Postfix notation for the expression (a+b)*(c+d) is → ab+cd+*

b) Postfix notation for the expression (a*b) - (c+d) is → ab+cd+ -

2) Syntax tree :- A tree in which each leaf node describes an operand and each interior node an operator. The syntax tree is shortened form of the parse tree.

Example :- Draw syntax tree for the string a+b*c-d



3) Three Address code :-

The three address code is a ~~sequence~~ sequence of statements of the form of A := B op C, where ABC are either programmer defined names, constants or compiler generated temporary names the op represents for an operator that can be fixed or floating point arithmatic operator or Boolean valued data or logical operator There are three types of three address code as follows :-

i) Quadruples Representation :- Records with the field for the operators and operands can be define three address statements. It is possible to use a record structure with field, first hold the operator 'op' next two hold operators 1 and 2 respectively and the last one holds the result. This representation of three addresses is called as Quadruples Representation

ii) Triples representation :- The contents of operand 1 operand 2 and result field are generally pointer to symbol records for the names described by these fields. Therefore, it is important to introduce temporary names into the symbol table as they are generated. This can be prevented by using the positions of statements defines temporary values.

iii) Indirect Triples Representation :- The indirect triple representation uses an extra array to list the pointer to the triple representation in the desired sequence. The triple representation for the statement x := (a+b)* -c is as follows

statement	statement	location	operator	operand 1	operand 2
(0)	(1)	(1)	+	a	b
(1)	(2)	(2)	-	c	
(2)	(3)	(3)	*	(0)	(1)
(3)	(4)	(4)	/	(2)	d
(4)	(5)	(5)	:=	(3)	

Q.5. Explain different issues in code generation phase of compiler.

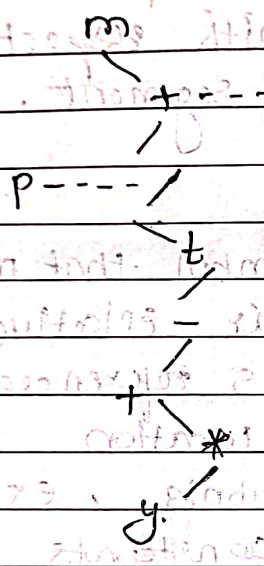
A) - Whether to use an existing Intermediate representation
- If target machine architecture is similar.
- If the new language is similar
- Whether the IR is appropriate for the kind of optimizations to be performed
- Eg:- Speculation and prediction.
- Some transformation may take much longer than they would be a different IR.

- Designing new IR needs to consider:
- Level (How machine dependent it is)
- Structure
- Expressiveness
- Appropriateness for code generation.
- Whether multiple IRs should be used.

The code generation phase of compiler involves several complex tasks, including register allocation, instruction selection, code optimization, handling complex language features, target specific code generation and error handling. Addressing these issues effectively is critical for generating efficient, correct and optimized machine code from the intermediate representation.

Q5.B: construct DAG for the following expression.
$X = m + p/q - t + p/q * y$.

→ we can break down the expression into smaller sub-expression and represented them as nodes in the DAG. each node represents an operation or a variable, and the edges represent the dependencies between them. Here is how the DAG would look like :-



In this DAG, :-
- The nodes "m", "q", "p", "t", "x" and "y" represent the variables in the expression. The nodes "+" and "-" represent the addition and substraction operations, respectively.
- The node "*" represents the multiplication operation.
- The edges represent the dependencies between the nodes. for example :- the node "+" depends on the nodes "m" and "q" as they are operands of the addition operation, similarly, the node "-" depends on the nodes "p/q" and "t" and so on.
- ~~Tuno~~ "p/q" is represented as single node in DAG.

Q 6. Explain Direct linking loader in detail.

→ A direct linking loader is the most popular loading scheme used. It allows the programmer to use the multiple procedure and data segments giving user a complete choice in referencing data or instruction enclosed in other parts of segments. It provides flexible intersegment referencing accessing ability. Assembler provides following information to loader with respect to each procedure or data segment.

- Segment length
- Records of all the symbol that may be referenced by other symbols and their relative location.
- Records of all the symbols referenced by other symbols and their relative location.
- Records of all the symbols, explanation about the address constants, their location in the segment and information about their revise values.
- Information about the source codes machine code translation and the relative addresses assigned.

design of direct linking loader :-

direct linking loader uses four types of records object file they are as ESD, TXT, RLD, END.

1) External Symbol Directory (ESD)
It combines information about all the symbols that are mentioned in the program but that may be referenced elsewhere

2) Text Records (TXT)
It ~~combine~~ contains the information about the actual object code translated version of the source program.

3) Relocation and linkage directory (RLD)
RLD cards are used to store those location and addresses on which the program content is dependent

4) END Record :—
It specifies the end of the object file and starting address for execution if the assembled routine is in the main program

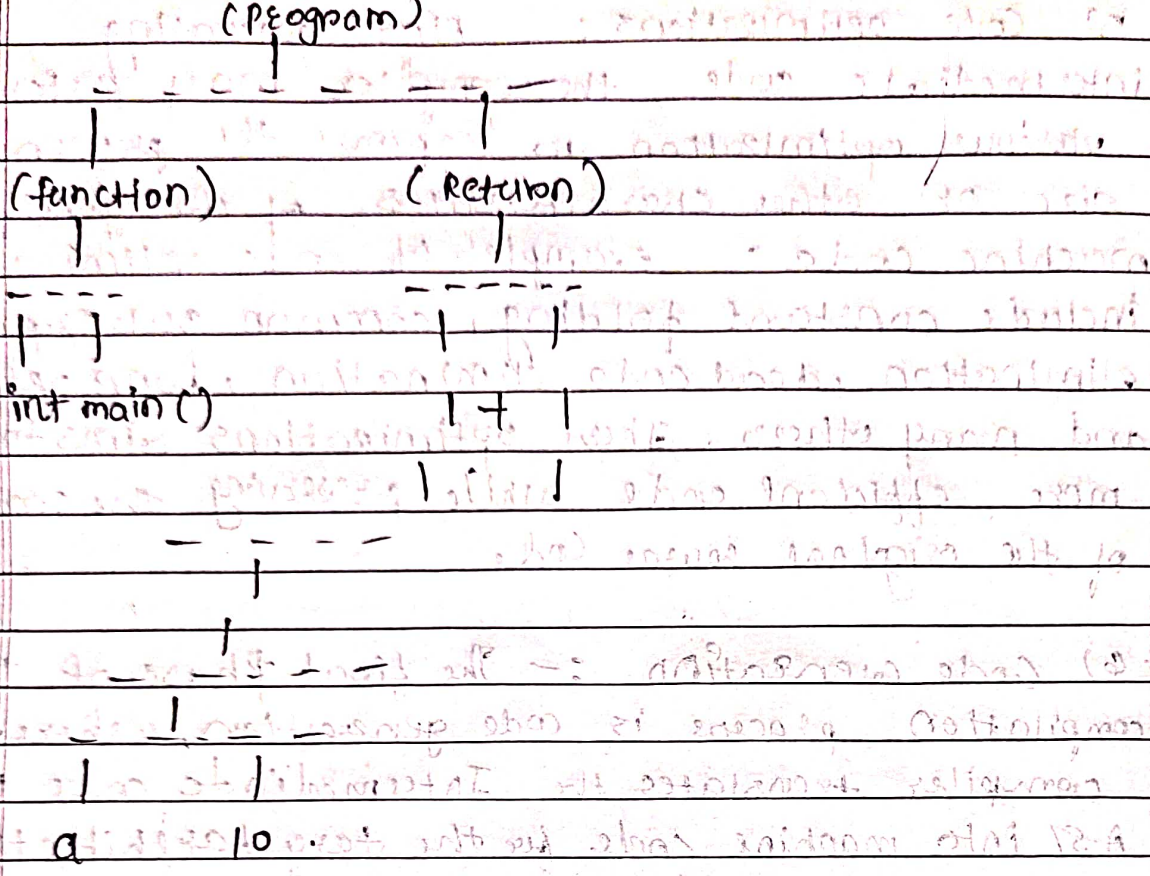
Q. 5. Explain the different phases of compiler with suitable example.

→ A compiler is a software program that translates the source code written in a high level programming language into machine code that can be executed by computers processor.

1) Lexical Analysis :-
The first phase of a compiler is lexical analysis, also known as scanning. It involves breaking the source code into a sequence of tokens, which are the smallest meaningful units in a programming language. Tokens can be keywords, identifiers, literals, operators, or special symbols. For example.

```
int main () {
int a = 5;
int b = 10;
int sum = a + b;
return sum;
}
```

2) Syntax Analysis :-
The next phase is syntax analysis, also known as parsing. It involves analyzing the sequence of tokens generated in the lexical analysis phase and building an abstract syntax tree (AST), which represents tree (AST), which represents the syntactic structure of the source code. The AST is a hierarchical representation that captures the relationships between different language constructs for example :-



3) Semantic Analysis :- After the syntax analysis phase, the compiler performs semantic analysis, which involves checking the correctness of the source code in terms of the language's semantics. It verifies that the code adheres to the language's code rules. such as checking for undeclared variables, type compatability, and other semantic constraints. For semantic analysis phase would detect this error and report it.

4. Intermediate code generation :- Once the syntax and semantic analysis are complete the compiler may generate an intermediate representation (IR) code as an intermediate step before generating the final machine code. The IR is a lower-level represents that is closer to the target machine architecture but still abstract enough to allow for further optimizations.

5) Code optimizations :- After generating intermediate code, the compiler may perform various optimization to improve the performance, size or other characteristics of the generated machine code. examples of code optimization include constant folding, common subexpression elimination, dead code elimination, loop optimizatio and many others. These optimizations aims to genera more efficient code while preserving the semantics of the original source code.

6) Code Generation :- The final phase of the compilation process is code generation, where the compiler translates the Intermediate code or the AST into machine code for the target architecture. This involves selecting appropriate machine instructions, allocating registers, managing memory and generating other necessary machine specific code.