SARASWATI Education Society's

SARASWATI College of Engineering

Learn Live Achieve and Contribute

Kharghar, Navi Mumbai - 410 210.

NAAC Accredited

Department of Computer Engineering

Name: Shaikh Adnan Shaukat Ali

Roll No: 76

Subject code: CSL601

Subject Name: System Programming and Compiler Construction Lab

Class/Sem: T.E. / SEM-VI

Year: 2021-2022

SARASWATI Education Society's

SARASWATI College of Engineering

Learn Live Achieve and Contribute

Kharghar, Navi Mumbai - 410 210.

NAAC Accredited

COMPUTER ENGINEERING DEPARTMENT

Subject: System Programming and Compiler Construction Lab. **Class/Sem: TE/VI**

Name of the Laboratory: Project Lab. **Year: 2021-2022**

LIST OF EXPERIMENTS

H/W Requirement	P I and above, RAM 128MB, Printer, Cartridges.
S/W Requirement	Flex Windows (LEX and YACC), C Compiler.

Prof. Shatabdi Bhalerao
Subject In-charge

Prof. Sujata Bhairnallykar
HOD

Experiment No. 1

Aim: Case study on System Programming

Theory:

Simply stated, a compiler is a program that can read a program in one language | the source language | and translate it into an equivalent program in another language | the target language; see Fig. 1.1. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

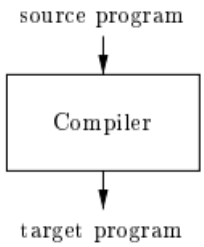


Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs; see Fig. 1.2.

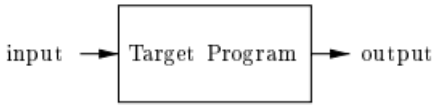


Figure 1.2: Running the target program

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig. 1.3.

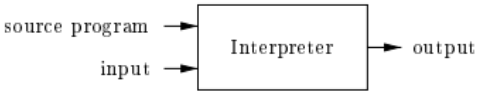


Figure 1.3: An interpreter

Example: Java language processors combine compilation and interpretation, as shown in Fig. 1.4. A Java source program may first be compiled into an intermediate form called bytecodes. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network. In order to achieve faster processing of inputs to outputs, some Java compilers, called just-in-time compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.

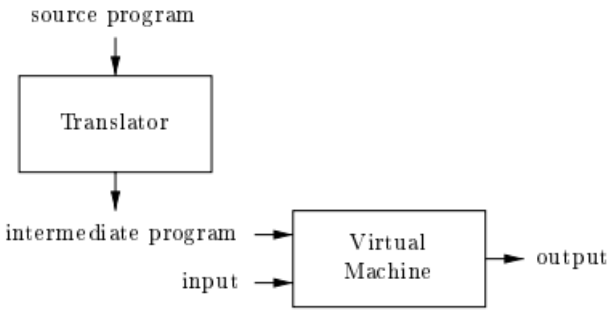


Figure 1.4: A hybrid compiler

In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1.5. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a preprocessor. The preprocessor may also expand shorthands, called macros, into source language statements.

The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output.

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually

runs on the machine. The linker resolves external memory addresses, where the code in one file may refer to a location in another file. The loader then puts together all of the executable object files into memory for execution.

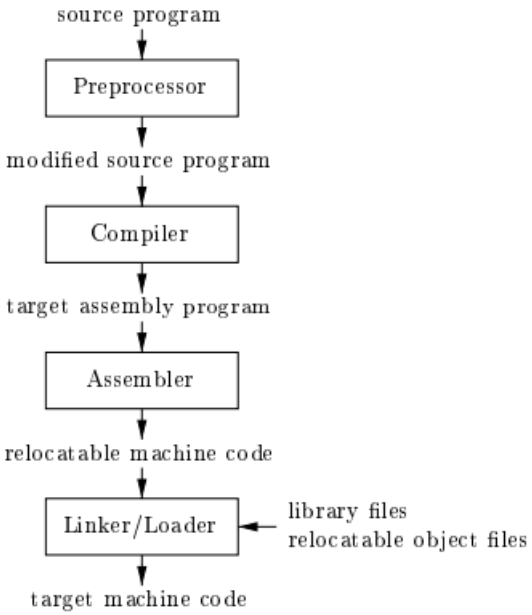


Figure 1.5: A language-processing system

The Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part. The synthesis part constructs the desired target program from the intermediate representation

and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end. If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Fig. 1.6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler. Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the backend can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Fig. 1.6 maybe missing.

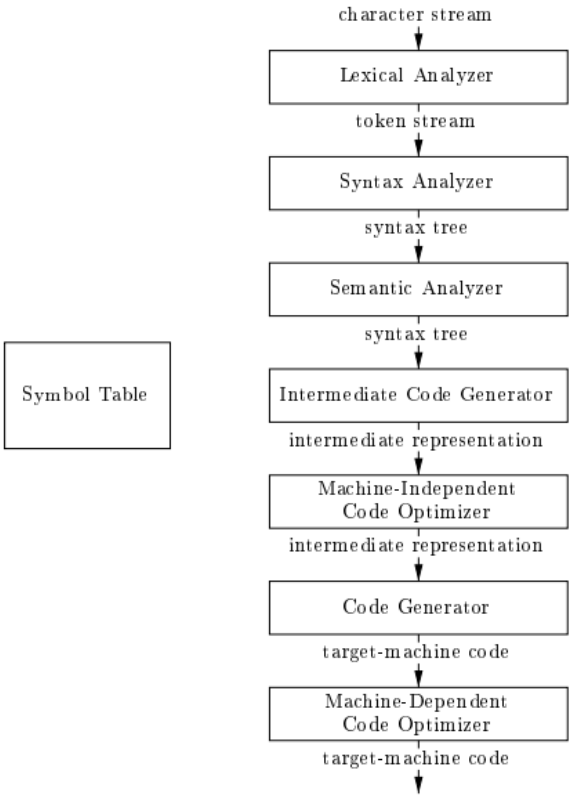


Figure 1.6: Phases of a compiler

Lexical Analysis

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form <token-name, attribute-value> that it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation. For example, suppose a source program contains the assignment statement

$$position = initial + rate * 60 \quad (1.1)$$

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. Position is a lexeme that would be mapped into a token <id, 1>, where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol = is a lexeme that is mapped into the token <=>. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as assign for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. Initial is a lexeme that is mapped into the token <id, 2>, where 2 points to the symbol-table entry for initial.

4. + is a lexeme that is mapped into the token <+>.

5. Rate is a lexeme that is mapped into the token <id, 3>, where 3 points to the symbol-table entry for rate.

6. * is a lexeme that is mapped into the token <*>.

7. 60 is a lexeme that is mapped into the token h60i.1 Blanks separating the lexemes would be discarded by the lexical analyzer. Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

$$< id, 1 > <=> < id, 2 > < +> < id, 3 > < 60 > \quad (1.2).$$

In this representation, the token names =, +, and are abstract symbols for the assignment, addition, and multiplication operators, respectively.

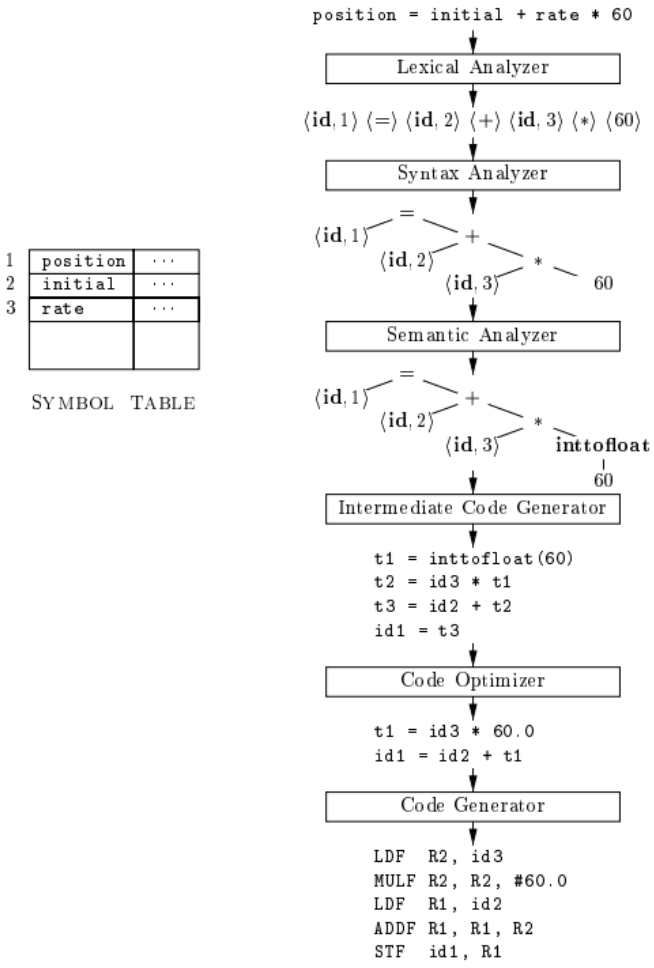


Figure 1.7: Translation of an assignment statement

Syntax Analysis

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation. A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in Fig. 1.7. This tree shows the order in which the operations in the assignment

$$position = initial + rate * 60$$

are to be performed. The tree has an interior node labelled with hid; 3i as its left child and the integer 60 as its right child. The node hid; 3i represents the identifier rate. The node labelled makes it explicit that we must first multiply the value of rate by 60. The node labelled + indicates that we must add the result of this multiplication to the value of initial. The root of the tree, labelled =, indicates that we must store the result of this addition into the location for the identifier position. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition. The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.

Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during

intermediate-code generation. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array. The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Such a coercion appears in Fig. 1.7. Suppose that position, initial, and rate have been declared to be floating-point numbers, and that the lexeme 60by itself forms an integer. The type checker in the semantic analyzer in Fig. 1.7discovers that the operator is applied to a floating-point number rate and n integer 60. In this case, the integer may be converted into a floating-point number. In Fig. 1.7, notice that the output of the semantic analyzer has an extra node for the operator int to float, which explicitly converts its integer argument into a floating-point number.

Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis. After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine. The output of the intermediate code generator in Fig. 1.7 consists of the three-address code sequence

$$t1 = inttofloat(60)$$

$$t2 = id3 * t1$$

$$t3 = id2 + t2$$

$$id1 = t3$$

(1.3)

There are several points worth noting about three-address instructions. First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program (1.1). Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some "three-address instructions" like the first and last in the sequence (1.3), above, have fewer than three operands.

Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation that comes from the semantic analyzer. A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the int to float operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, t3 is used only once to transmit its value to id1 so the optimizer can transform (1.3) into the shorter sequence

$$t1 = id3 * 60.0$$

$$id1 = id2 + t1$$

(1.4)

There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables. For example, using registers R1 and R2, the intermediate code in (1.4) might get translated into the machine code

LDF R2, id3

MULF R2, R2, 60.0

LDF R1, id2

ADDF R1, R1, R2

STF id1, R1

(1.5)

The first operand of each instruction specifies a destination. The F in each instruction tells us that it deals with floating-point numbers. The code in (1.5) loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0. The # signifies that 60.0 is to be treated as an immediate constant. The third instruction moves id2 into register R1 and the fourth adds to it the value previously computed in register R2. Finally, the value in register R1 is stored into the address of id1, so the code correctly implements the assignment statement (1.1).

Symbol Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned. The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly

Compiler -Construction Tools

The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler. These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms. The most successful tools

are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some commonly used compiler-construction tools include

1. Parser generators that automatically produce syntax analyzers from a grammatical description of a programming language.

2. Scanner generators that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. Syntax-directed translation engines that produce collections of routines for walking a parse tree and generating intermediate code.

4. Code-generator generators that produce a code generator from a collection of rules for translating each operation of the intermediate language in to the machine language for a target machine.

5. Data- ow analysis engines that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data- ow analysis is a key part of code optimization.

6. Compiler-construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

Experiment No. 2

Aim: To implement: 1) Identify a given no is even or odd

2) Identify vowels in a string

3) Identify relational operators

4) Copy content of one file to another

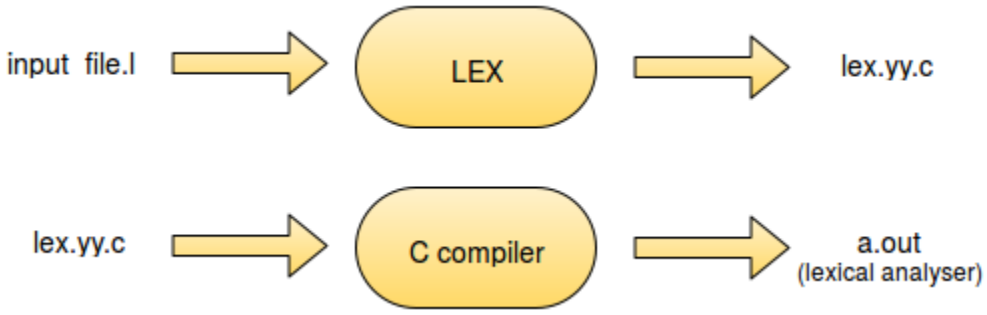
Using Lexical Analyzer tool: LEX

Requirements: Lex tool and C.

Theory:

Introduction

LEX is a tool used to generate a lexical analyzer. This document is a tutorial for the use of LEX for **ExpL Compiler** development. Technically, LEX translates a set of regular expression specifications (given as input in input_file.l) into a C implementation of a corresponding finite state machine (lex.yy.c). This C program, when compiled, yields an executable lexical analyzer.



The source ExpL program is fed as the input to the the lexical analyzer which produces a sequence of tokens as output. (Tokens are explained below). Conceptually, a lexical analyzer scans a given source ExpL program and produces an output of tokens.

Each token is specified by a token name. The token name is an abstract symbol representing the kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. For instance integer, boolean, begin, end, if, while etc. are tokens in ExpL.

“integer” {return ID_TYPE_INTEGER;}

This example demonstrates the specification of a **rule** in LEX. The rule in this example specifies that the lexical analyzer must return the token named ID_TYPE_INTEGER when the pattern “integer” is found in the input file. A rule in a LEX program comprises of a 'pattern' part (specified by a regular expression) and a corresponding (semantic) 'action' part (a sequence of C statements). In the above example, “integer” is the pattern and {return ID_TYPE_INTEGER;} is the corresponding action. The statements in the action part will be executed when the pattern is detected in the input.

Lex was developed by Mike Lesk and Eric Schmidt at Bell labs.

The structure of LEX programs

A LEX program consists of three sections: **Declarations, Rules and Auxiliary functions**

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

1 Declarations

The declarations section consists of two parts, **auxiliary declarations** and **regular definitions**.

The auxiliary declarations are copied as such by LEX to the output *lex.yy.c* file. This C code consists of instructions to the C compiler and are not processed by the LEX tool.The auxiliary declarations (which are optional) are written in C language and are enclosed within ' %{ ' and ' %} ' . It is generally used to declare functions, include header files, or define global variables and constants.

LEX allows the use of short-hands and extensions to regular expressions for the regular definitions. A regular definition in LEX is of the form: **D R** where D is the symbol representing the regular expression R.

2 Rules

Rules in a LEX program consists of two parts:

1. The pattern to be matched
2. The corresponding action to be executed

3 Auxiliary functions

LEX generates C code for the rules specified in the Rules section and places this code into a single function called yylex(). (To be discussed in detail later). In addition to this LEX generated code, the programmer may wish to add his own code to the lex.yy.c file. The auxiliary functions section allows the programmer to achieve this.

The auxiliary declarations and auxiliary functions are copied as such to the lex.yy.c file

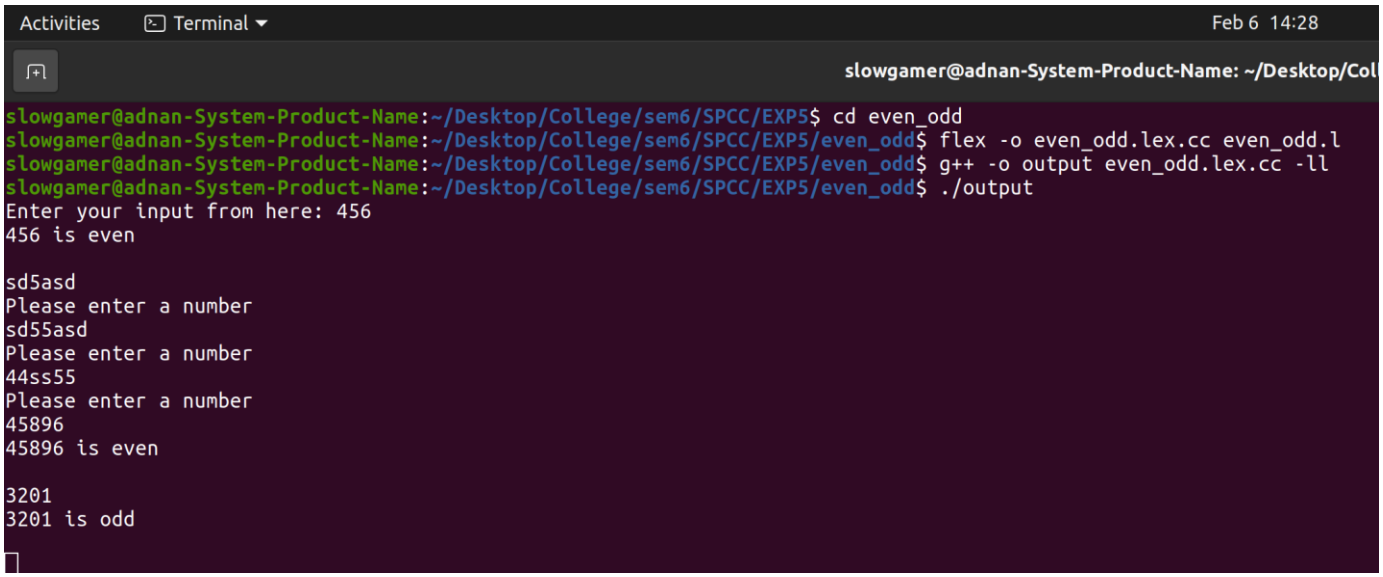
Once the code is written, lex.yy.c maybe generated using the command lex "filename.l" and compiled as gcc lex.yy.c

1) Even Odd

Code:

```
%{
   #include<stdio.h>
%}
%%

^[0-9]+$ {
   if(atoi(yytext)%2) printf("%s is odd\n",yytext);
   else printf("%s is even\n",yytext);
}
[a-zA-Z0-9_]+ {printf("Please enter a number");}

%%

int yywrap(){
   return 0;
}

int main(){
   printf("Enter your input from here: ");
   yylex();
}
```

Output:

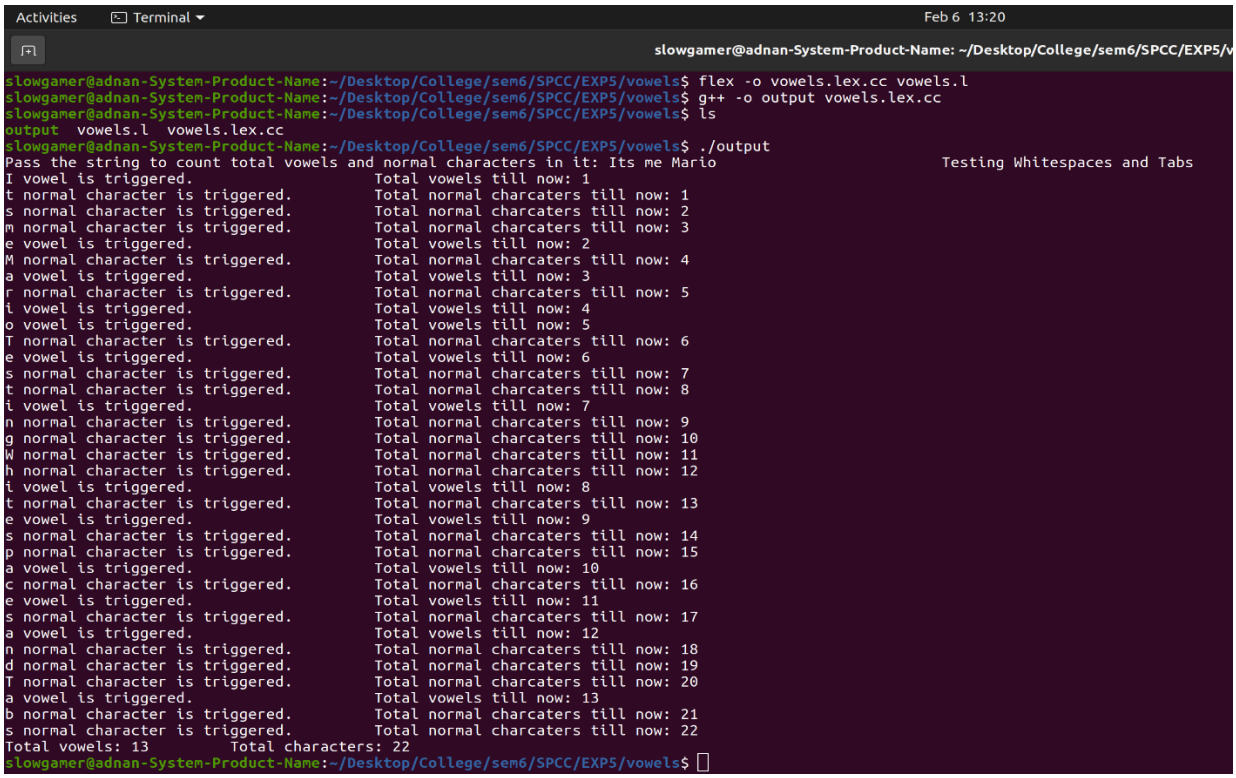


2) Vowels

Code:

```
%{
   #include<string.h>
   int vowels = 0, not_vowels = 0;
%}
%%
[aeiouAEIOU] {vowels++; printf("%s vowel is triggered.\t\t\t Total vowels till now: %d\n",yytext,vowels);}
[\t ] /*Skipping whitespaces and tab*/;
. {not_vowels++; printf("%s normal character is triggered.\t Total normal charcaters till now: %d\n",yytext,not_vowels);}
\n {printf("Total vowels: %d \t Total characters: %d \n",vowels,not_vowels); return 0;}
%%

int yywrap(){
   return 0;
}

int main(){
   printf("Pass the string to count total vowels and normal characters in it: ");
   yylex();

}
```

Output:

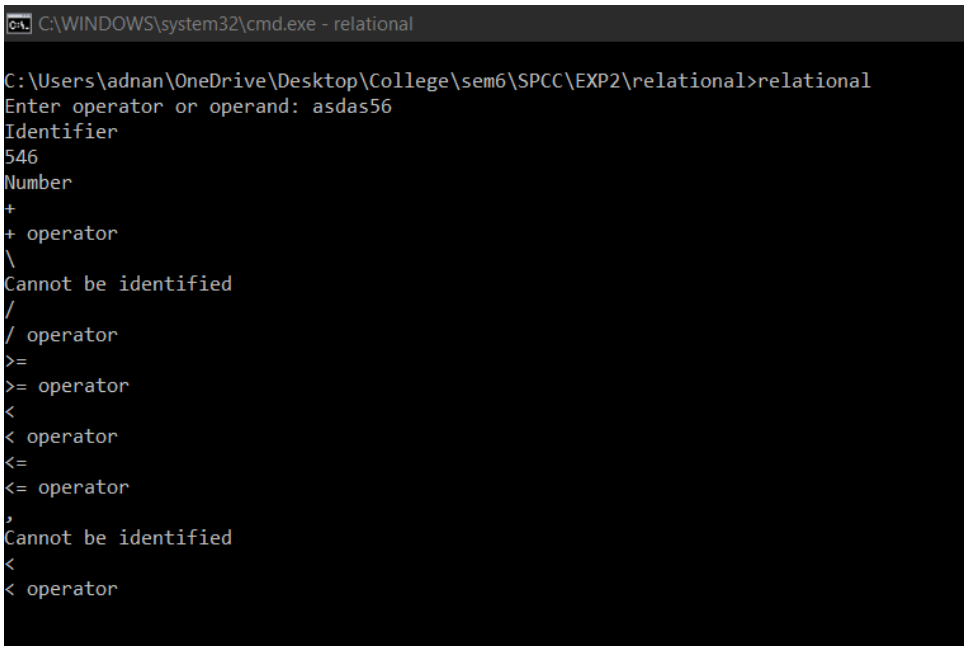


3) Relational Operator

Code:

```
%{
	#include<stdio.h>
%}

%%
- {printf("- operator");}
\/ printf("/ operator");
\+ {printf("+ operator");}
\* {printf("* operator");}
\> {printf("> operator");}
\< {printf("< operator");}
\>= {printf(">= operator");}
\<= {printf("<= operator");}
= {printf("= operator");}
== {printf("== operator");}
[0-9]+ {printf("Number");}
[A-Za-z][A-Za-z0-9_]* {printf("Identifier");}
.* {printf("Cannot be identified");}

%%
int main()
{
	printf("Enter operator or operand: ");
	yylex();
	return 0;
}
```

Output:



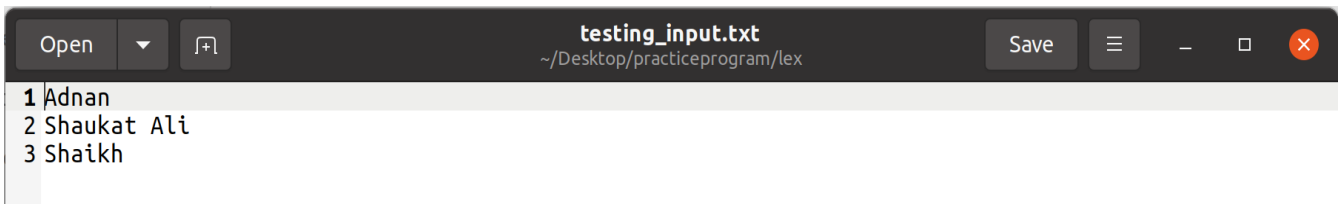
4) Moving Content of a file

Code:

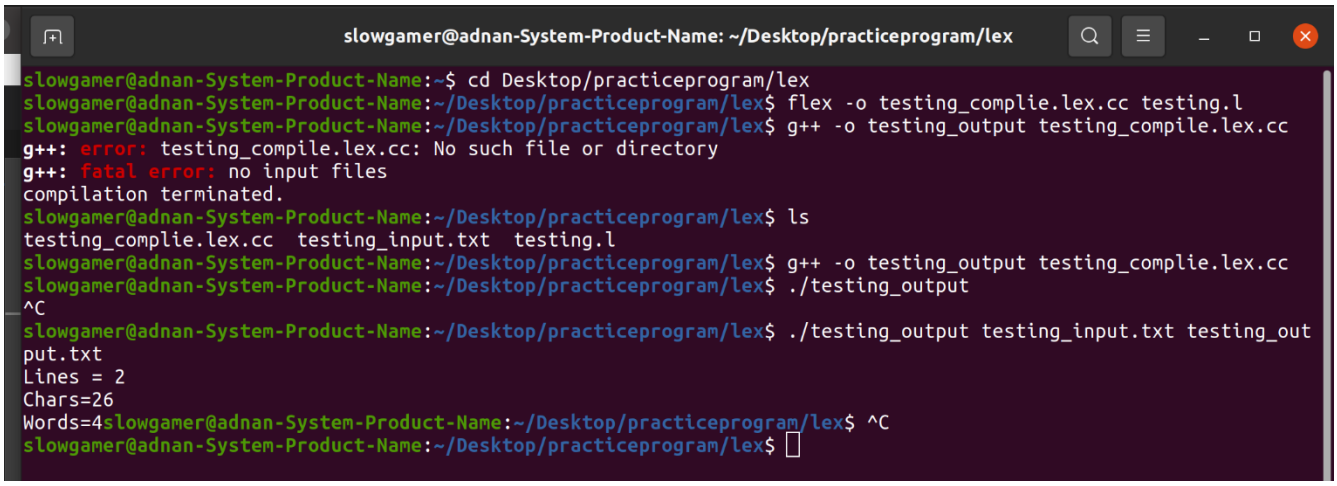
```
%{
int nlines,nwords,nchars;
%}

%%
\n {
        nchars++;nlines++; fprintf(yyout,"%s",yytext);
        }

[^ \n\t]+ {nwords++, nchars=nchars+yyleng; fprintf(yyout,"%s",yytext);}
. {nchars++; fprintf(yyout,"%s",yytext);}
%%
int yywrap(void)
{
 return 1;
}
int main(int argc, char*argv[])
{
        yyin = fopen(argv[1],"r");
    yyout = fopen(argv[2],"w");
 yylex();
 fclose(yyin);
 fclose(yyout);
 printf("Lines = %d\nChars=%d\nWords=%d",nlines,nchars,nwords);

 return 0;
}
```

Output:

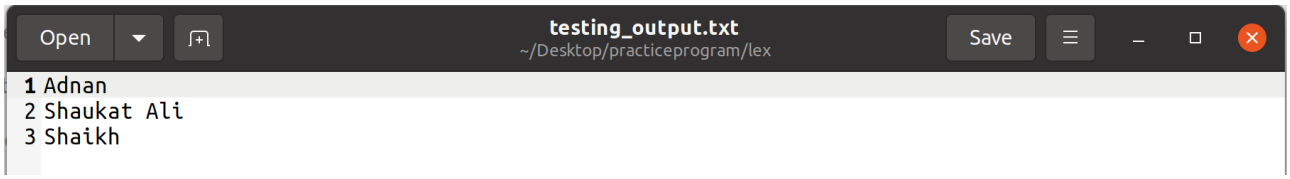
Input.txt



Executing code



Output.txt



Conclusion: We have successfully implemented program to: Identify a given no is even or odd, Identify vowels in a string, Identify relational operators and Copy content of one file to another in LEX.

Experiment No. 3

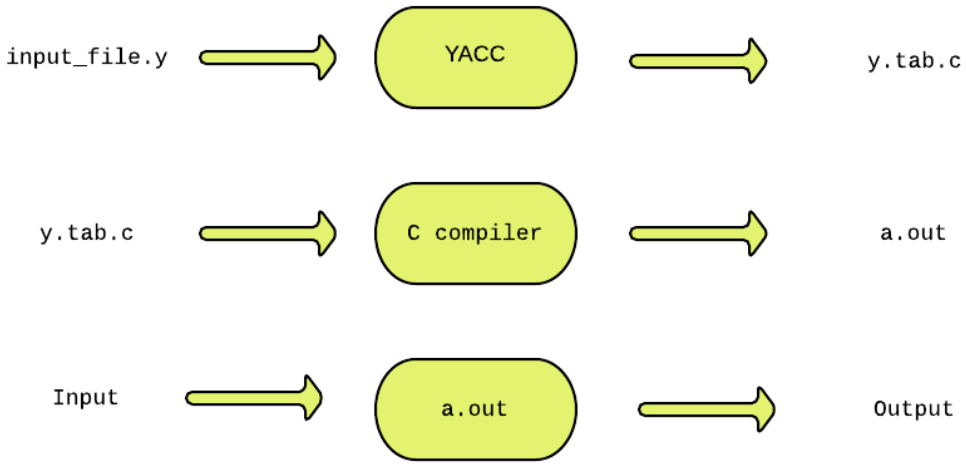
Aim: To perform arithmetic operations using YACC

Requirements: Lex, Yacc and C

Theory:

Introduction to YACC

YACC (Yet Another Compiler Compiler) is a tool used to generate a parser. This document is a tutorial for the use of YACC to generate a parser for ExpL. YACC translates a given Context Free Grammar (CFG) specifications (input in input_file.y) into a C implementation (y.tab.c) of a corresponding push down automaton (i.e., a finite state machine with a stack). This C program when compiled, yields an executable parser.



The source SIL program is fed as the input to the generated parser (a.out). The parser checks whether the program satisfies the syntax specification given in the input_file.y file.

YACC was developed by Stephen C. Johnson at Bell labs.

Parser

A parser is a program that checks whether its input (viewed as a stream of tokens) meets a given grammar specification. The syntax of SIL can be specified using a Context Free Grammar. As mentioned earlier, YACC takes this specification and generates a parser for SIL.

Context Free Grammar (CFG)

A context free grammar is defined by a four tuple (N,T,P,S) - a set N of non-terminals, a set T of terminals (in our project, these are the tokens returned by the lexical analyzer and hence we refer to them as tokens frequently), set P of productions and a start variable S. Each production consists of a non-terminal on the left side (head part) and a sequence of tokens and non-terminals (of zero or more length) on the right side (body part).

The structure of YACC programs

A YACC program consists of three sections: Declarations, Rules and Auxiliary functions. (Note the similarity with the structure of LEX programs).

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

1 Declarations

The declarations section consists of two parts: (i) C declarations and (ii) YACC declarations . The C Declarations are delimited by **%{** and **%}**. This part consists of all the declarations required for the C code written in the *Actions* section and the *Auxiliary functions* section.

YACC copies the contents of this section into the generated y.tab.c file without any modification.

2 Rules

A rule in a YACC program comprises of two parts (i) the production part and (ii) the action part. In this project, the syntax of SIL programming language will be specified in the form of a context free grammar.

2.1 Productions

Each production consists of a production head and a production body.

2.2 Actions

The action part of a rule consists of C statements enclose within a '{' and '}'. These statements are executed when the input is matched with the body of a production and a reduction takes place.

3 Auxiliary Functions

The Auxiliary functions section contains the definitions of three mandatory functions main(), yylex() and yyerror(). You may wish to add your own functions (depending on the the requirement for the application) in the y.tab.c file. Such functions are written in the auxiliary functions section. The main() function must invoke yyparse() to parse the input.

Code:

Lex:

```
%{
#include<stdio.h>
#include "calc.tab.h"
extern int yylval;
%}
```

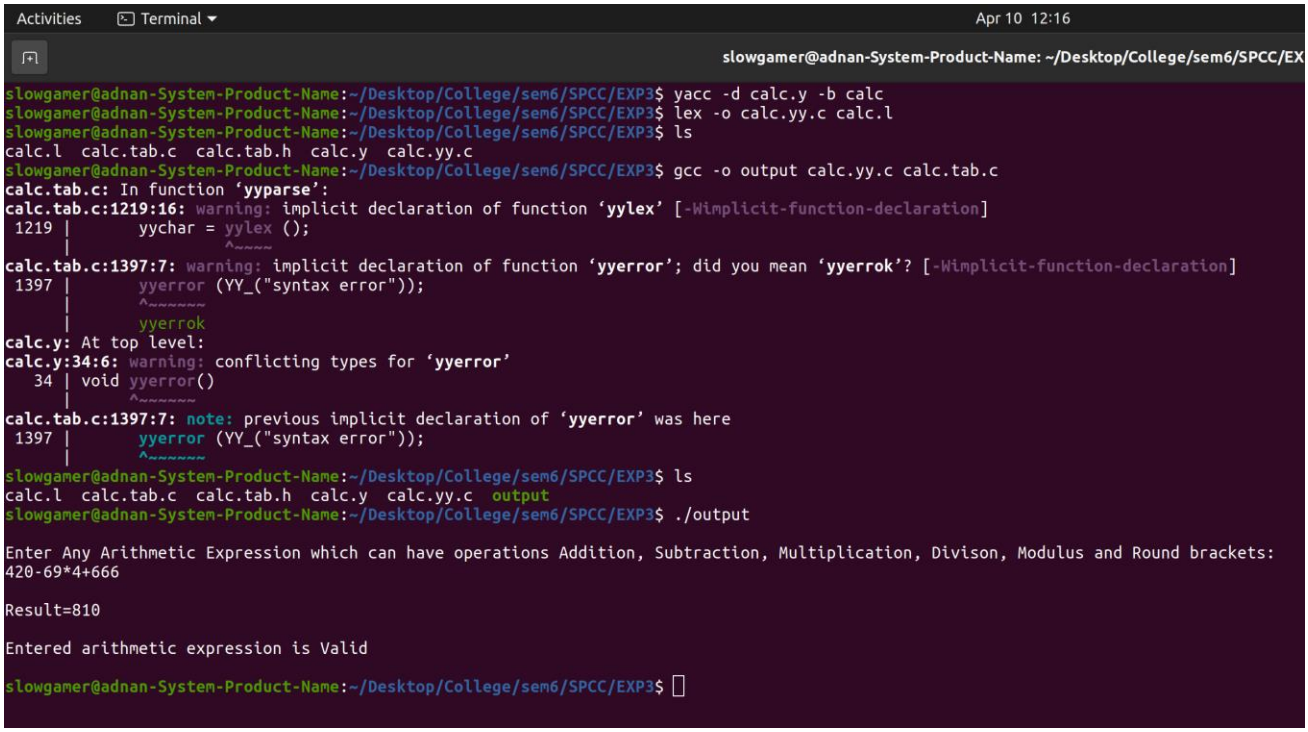
```
%%
[0-9]+ {
        yylval=atoi(yytext);
        return NUMBER;
      }
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

Yacc:

```
%{
   #include<stdio.h>
   int flag=0;

%}
%token NUMBER

%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
ArithmeticExpression: E{
        printf("\nResult=%d\n",$$);
        return 0;
       }
E:E'+'E {$$=$1+$3;}
 |E'-'E {$$=$1-$3;}
 |E'*'E {$$=$1*$3;}
 |E'/'E {$$=$1/$3;}
 |E'%'E {$$=$1%$3;}
 |'('E')' {$$=$2;}
 | NUMBER {$$=$1;}
;
%%

void main()
{
   printf("\nEnter Any Arithmetic Expression which can have operations Addition,
Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");
   yyparse();
  if(flag==0)
   printf("\nEntered arithmetic expression is Valid\n\n");
```

```
}
void yyerror()
{
  printf("\nEntered arithmetic expression is Invalid\n\n");
  flag=1;
}
```

Output:



Conclusion: We have successfully implemented program to perform arithmetic operations using YACC.

Experiment No. 04

Aim- Design an implementation of pass I of two pass assembler

Requirement- Java

Theory - The one-pass assembler cannot resolve forward references of data symbols. It requires all data symbols to be defined prior to being used. A two-pass assembler solves this dilemma by devoting one pass to exclusively resolve all (data/label) forward references and then generate object code with no hassles in the next pass. If a data symbol depends on another and this another depends on yet another, the assembler resolved this recursively.

It generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce machine code. Now, if assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler. Here assembler divide these tasks in two passes:

Pass-1:

- Define symbols and literals and remember them in symbol table and literal table respectively.
- Keep track of location counter
- Process pseudo-operations
- Assign addresses to all statements in the program.
- Save the values (addresses) assigned to all labels (including label and variable names) for use in Pass 2 (deal with forward references).
- Perform some processing of assembler directives (e.g., BYTE, RESW, these can affect address assignment)

The Pseudo Code for Pass 1

Pass 1:

```
begin
  read first input line
  if OPCODE = 'START' then
     begin
        save #[OPERAND] as starting address
        initialize LOCCTR to starting address
        write line to intermediate file
        read next input line
     end {if START}
  else
     initialize LOCCTR to 0
  while OPCODE ≠ 'END' do
     begin
        if this is not a comment line then
           begin
              if there is a symbol in the LABEL field then
                 begin
```

\

Code-

```
import java.util.*;
import java.lang.*;
import java.io.*;

class pass1
{
   public static void main(String []args)
   {
      BufferedReader reader;
      int lc=0,sti=0,di=0,i,j,li=0;
      int[][] symtab = new int[100][3];
      int[][] littab = new int[100][3];
      String[] sym = new String[100];
      String[] data = new String[100];
      try
      {
         reader = new BufferedReader(new FileReader("prg.txt"));
         String line = reader.readLine();
         String[] words = line.split("\\s+");
         //System.out.println(sym[0]+" "+symtab[0][0]);
         while (!words[1].equals("END"))
         {
            if(!words[0].equals(""))
            {
               if(words[1].equals("START"))
               {
                  sym[sti] = words[0];
```

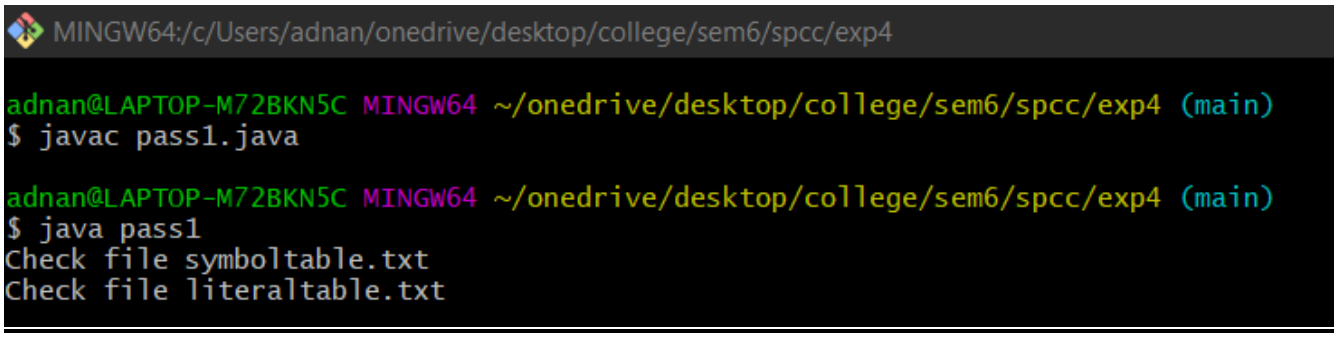
```
            symtab[sti][0] = 0;
            symtab[sti][1] = 1;
            symtab[sti][2] = 0;
            sti++;
            //System.out.println("in1");
        }
        else if(words[1].equals("EQU"))
        {
            sym[sti] = words[0];
            if(words[2].equals("*")) symtab[sti][0] = lc;
            else symtab[sti][0] = Integer.parseInt(words[2]);
            symtab[sti][1] = 1;
            symtab[sti][2] = 1;
            sti++;
            //System.out.println("in2");
        }
        else if(words[0].equals("SAVE"))
        {
            sym[sti] = words[0];
            symtab[sti][0] = lc;
            symtab[sti][1] = 4;
            symtab[sti][2] = 0;
            sti++;
            String[] lcw = words[2].split("F");
            int ds = Integer.parseInt(lcw[0]);
            lc+= (ds*4);
            //System.out.println("in3");
        }
        else
        {
            sym[sti] = words[0];
            symtab[sti][0] = lc;
            symtab[sti][1] = 4;
            symtab[sti][2] = 0;
            sti++;
            if(words[0].equals("LOOP")) lc+=4;
            //System.out.println("in4");
        }
    }
    else if(words[1].equals("USING"))
    {
        line = reader.readLine();
        words = line.split("\\s+");
        //System.out.println("in11");
        //System.out.println(words[0]);
        continue;
    }
    else if(words[1].equals("LTORG"))
    {
        //System.out.println(lc+" "+words[1]);
```

```
                while(lc%8!=0) lc++;
                for(i=0;i<di;i++)
                {
                    littab[li][0] = lc;
                    littab[li][1] = 4;
                    littab[li][2] = 0;
                    lc+=4;
                    li++;
                }
                //System.out.println("in12");
            }
            else
            {
                String[] opr = words[2].split(",");
                //System.out.println(lc+" "+words[1]);
                //System.out.println(opr[0]+" "+opr[1]);
                for(i=0;i<opr.length;i++)
                {
                    if(opr[i].charAt(0) == '=')
                    {
                        data[di] = opr[i];
                        di++;
                    }
                }
                if(words[1].charAt(words[1].length()-1) == 'R') lc+=2;
                else lc+=4;
                //System.out.println("in13");
            }
            //System.out.println(words[1]);
            line = reader.readLine();
            words = line.split("\\s+");
            //System.out.println(sym[sti-1]+" "+symtab[sti-1][0]+" "+symtab[sti-1][1]+"
"+symtab[sti-1][2]);
            //System.out.println(data[di-1]+" "+littab[di-1][0]+" "+littab[di-1][1]+" "+littab[di-
1][2]);
            //System.out.println(words[0]);
        }
        reader.close();
        //System.out.println("-----------------------------------------------------------");
        //System.out.println("Symbol Table");
        //System.out.println("Symbol   Value   Length  Relocation(0-R;1-A)");
        //System.out.println("-----------------------------------------------------------");

        try(OutputStream fw = new FileOutputStream("symboltable.txt"))
        {
            for(i=0;i<sti;i++)
            {
                //BufferedWriter bw = new BufferedWriter(fw);
                String content = sym[i]+" "+symtab[i][0]+" "+symtab[i][1]+"
"+symtab[i][2]+System.getProperty("line.separator");
```

```
                //System.out.println(content);
                fw.write(content.getBytes(),0,content.length());
                //System.out.println(sym[i]+" "+symtab[i][0]+" "+symtab[i][1]+"
"+symtab[i][2]);
            }
        }
        catch (IOException e) { e.printStackTrace(); }
        System.out.println("Check file symboltable.txt");
        //System.out.println("------------------------------------------------------------");
        //System.out.println("Literal Table");
        //System.out.println("Literal   Value   Length  Relocation(0-R;1-A)");
        //System.out.println("------------------------------------------------------------");
        try(OutputStream fw = new FileOutputStream("literaltable.txt"))
        {
           for(i=0;i<di;i++)
           {
              //BufferedWriter bw = new BufferedWriter(fw);
              String content = data[i]+" "+littab[i][0]+" "+littab[i][1]+"
"+littab[i][2]+System.getProperty("line.separator");
              //System.out.println(content);
              fw.write(content.getBytes(),0,content.length());
              //System.out.println(sym[i]+" "+symtab[i][0]+" "+symtab[i][1]+"
"+symtab[i][2]);
           }
        }
        catch (IOException e) { e.printStackTrace(); }
        System.out.println("Check file literaltable.txt");
     }
     catch (IOException e) { e.printStackTrace(); }
  }
}
```

Output-

Execution:



Input File:

prg - Notepad

File Edit View

```
PRGAM2 START 0
         USING *,15
         LA    15,SETUP
         SR    TOTAL,TOTAL
AC       EQU   2
INDEX  EQU   3
TOTAL  EQU   4
DATABASE EQU 13
SETUP  EQU   *
         USING SETUP,15
         L     DATABASE,=A(DATA1)
         USING DATAAREA,DATABASE
         SR    INDEX,INDEX
LOOP   L     AC,DATA1(INDEX)
         AR    TOTAL,AC
         A     AC,=F'6'
         ST    AC,SAVE(INDEX)
         A     INDEX,=F'4'
         C     INDEX,=F'8000'
         BNE   LOOP
         LR    1,TOTAL
         BR    14
         LTORG
SAVE   DS    2000F
DATAAREA EQU *
DATA1  DC    F'01'
       END
```

Symbol Table and Literal Table:

symboltable - Notep...

File Edit View

```
PRGAM2 0 1 0
AC 2 1 1
INDEX 3 1 1
TOTAL 4 1 1
DATABASE 13 1 1
SETUP 6 1 1
LOOP 12 4 0
SAVE 64 4 0
DATAAREA 8064 1 1
DATA1 8064 4 0
```

Ln 1, Col 1 100% Wi

literaltable - Notepad

File Edit View

```
=A(DATA1) 48 4 0
=F'6' 52 4 0
=F'4' 56 4 0
=F'8000' 60 4 0
```

Ln 1, Col 1 100% Wind

Conclusion- We have successfully implemented Pass 1 of two pass Assembler

Experiment No. 05

Aim- Design an implementation of pass II of 2 pass assembler

Requirement- Java and printout pages

Theory-

1. Generate object code by converting symbolic op-code into respective numeric op-code
2. Generate data for literals and look for values of symbols
3. Assemble instructions (generate opcode and look up addresses)
4. Generate data values defined by BYTE, WORD
5. Perform processing of assembler directives not done in Pass 1
6. Write the object program and the assembly listing

Algorithm-

```
begin
 read first input line (from intermediate file)
 if OPCODE ='START' then
 begin
  write listing line
  read next input line
 end {if START}
```

```
write Header record to object program
initialize first Text record
while OPCODE != 'END' do
begin
  if this is not a comment line then
begin
    search OPTAB for OPCODE
    if found then
     begin
      if there is a symbol in OPERAND field then
       begin
        search SYMTAB for OPERAND
        if found then
     store symbol value as operand address
        else
          begin
          store 0 as operand address
          set error flag (undefined symbol)
        end
        end {if symbol}
```

```
      else
       store 0 as operand address
      assemble the object code instruction
 end {if opcode found}
   else if OPCODE ='BYTE' or 'WORD' then
     convert constant to object code
 if object code will not fit into the current Text record then
    begin
     write Text record to object program
     initialize new Text record
    end
 add object code to Text record
   end {if not comment}
 write listing line
 read next input line
 end(while not END)
 write last Text record to object program
 write End record to object program
 write last listing line
end{Pass 2}
```

Code-

```
import java.io.*;
import java.util.*;

class pass2
{
   static int lc=0,sti=0,di=0,i,j,li=0,ri=0,r,lci=0,mci=0,fin=-1,index=0,base=0,ln=0;
   static int[][] symtab = new int[100][3];
   static int[][] littab = new int[100][3];
   static int[][] reg = new int[50][2];
   static String[] sym = new String[100];
   static String[] data = new String[100];
   static String[][] macode = new String[100][4];

   public static int getbr(int n)
   {
      int min=100,pos=-1;
      for(i=0;i<50;i++)
      {
         if(min>Math.abs(reg[i][0]-n) && reg[i][1]==1)
         {
            min=Math.abs(reg[i][0]-n);
            pos = i;
            //System.out.println(min+" "+pos);
         }
      }
      return pos+1;
   }

   public static int getsymlc(String s)
   {
      for(i=0;i<sti;i++)
      {
         if(s.equals(sym[i]))
         {
            return symtab[i][0];
         }
      }
      return -1;
   }

   public static int getlitlc(String s)
   {
      for(i=0;i<di;i++)
      {
         if(s.equals(data[i]))
         {
            return littab[i][0];
```

```
            }
        }
        return -1;
    }

    public static void assmc(String a, String b, String c, String d)
    {
        macode[mci][0] = a;
        macode[mci][1] = b;
        macode[mci][2] = c;
        macode[mci][3] = d;
        //System.out.println(macode[mci][0]+" "+macode[mci][1]+" "+macode[mci][2]);
        mci++;
    }

    public static void main(String []args)
    {
        BufferedReader reader;

        for(i=0;i<50;i++)
        {
            for(j=0;j<2;j++)
            {
                reg[i][j] = 0;
            }
        }

        try
        {
            reader = new BufferedReader(new FileReader("symboltable.txt"));
            String line = reader.readLine();
            while(line!=null)
            {
                String[] words = line.split("\\s+");
                sym[sti] = words[0];
                symtab[sti][0] = Integer.parseInt(words[1]);
                symtab[sti][1] = Integer.parseInt(words[2]);
                symtab[sti][2] = Integer.parseInt(words[3]);
                sti++;
                line = reader.readLine();
            }
            //for(i=0;i<sti;i++) System.out.println(sym[i]+" "+symtab[i][0]+" "+symtab[i][1]+"
"+symtab[i][2]);

            reader = new BufferedReader(new FileReader("literaltable.txt"));
            line = reader.readLine();
            while(line!=null)
            {
                String[] words = line.split("\\s+");
                data[di] = words[0];
```

```
            littab[di][0] = Integer.parseInt(words[1]);
            littab[di][1] = Integer.parseInt(words[2]);
            littab[di][2] = Integer.parseInt(words[3]);
            di++;
            line = reader.readLine();
         }
         //for(i=0;i<di;i++) System.out.println(data[i]+" "+littab[i][0]+" "+littab[i][1]+"
"+littab[i][2]);
         //System.out.println(getlitlc("=F'4'"));
      }
      catch (IOException e) { e.printStackTrace(); }
      // 1 - Base 2 - Index
      try
      {
         reader = new BufferedReader(new FileReader("prg.txt"));
         String line = reader.readLine();
         String[] words = line.split("\\s+");
         ln++;
         //System.out.println(sym[0]+" "+symtab[0][0]);
         while (!words[1].equals("END"))
         {
            if(words[1].equals("USING"))
            {
               String[] opr = words[2].split(",");
               if(opr[0].equals("*"))
               {
                  r = Integer.parseInt(opr[1]);
                  reg[r-1][0] = lc;
                  reg[r-1][1] = 1;
                  //System.out.println(r+" "+reg[r-1][0]+" "+reg[r-1][1]);
               }
               else if(opr[1].matches("[0-9]+"))
               {
                  r = Integer.parseInt(opr[1]);
                  reg[r-1][0] = getsymlc(opr[0]);
                  reg[r-1][1] = 1;
                  //System.out.println(r+" "+reg[r-1][0]+" "+reg[r-1][1]);
               }
               else
               {
                  r = getsymlc(opr[1]);
                  reg[r-1][0] = getsymlc(opr[0]);
                  reg[r-1][1] = 1;
                  //System.out.println(r+" "+reg[r-1][0]+" "+reg[r-1][1]);
               }
            }
            else if(words[1].equals("LA"))
            {
               String[] opr = words[2].split(",");
               r = Integer.parseInt(opr[0]);
```

```
                lci = Math.abs(getsymlc(opr[1])-reg[r-1][0]);
                if(fin==-1) index=0;
                assmc(Integer.toString(ln),Integer.toString(lc),words[1],opr[0]+","+Integer.toStri
ng(lci)+"("+Integer.toString(index)+","+Integer.toString(getbr(lci))+")");
                lc+=4;
            }
            else if(words[1].length()>1 && words[1].charAt(1) == 'R' &&
(words[1].equals("SR") || words[1].equals("AR") ||words[1].equals("LR")))
            {
                String[] opr = words[2].split(",");
                if(opr[0].matches("[0-9]+"))
assmc(Integer.toString(ln),Integer.toString(lc),words[1],opr[0]+","+Integer.toString(getsymlc
(opr[1])));
                else if(opr[1].matches("[0-9]+"))
assmc(Integer.toString(ln),Integer.toString(lc),words[1],Integer.toString(getsymlc(opr[0]))+",
"+opr[1]);
                else
assmc(Integer.toString(ln),Integer.toString(lc),words[1],Integer.toString(getsymlc(opr[0]))+",
"+Integer.toString(getsymlc(opr[1])));
                lc+=2;
                if(opr[1].equals("INDEX")) index=getsymlc(opr[0]);
            }
            else if(words[1].equals("L") || words[1].equals("ST") )
            {
                String[] opr = words[2].split(",");
                if(opr[1].charAt(0) == '=')
                {
                    r = getsymlc(opr[0]);
                    lci = Math.abs(getlitlc(opr[1])-6);
                    assmc(Integer.toString(ln),Integer.toString(lc),words[1],Integer.toString(r)+","
+Integer.toString(lci)+"("+Integer.toString(index)+","+Integer.toString(getbr(r))+")");
                }
                else
                {
                    String[] opr21 = opr[1].split("\\(");
                    String op21 = opr21[0];
                    //System.out.println(getsymlc(op21));
                    opr21 = opr21[1].split("\\)");
                    String op22 = opr21[0];
                    //System.out.println(op22);
                    if(op22.equals("INDEX") && op21.equals("DATA1"))
assmc(Integer.toString(ln),Integer.toString(lc),words[1],Integer.toString(getsymlc(opr[0]))+",
0("+Integer.toString(index)+","+Integer.toString(getbr(getsymlc(op21)))+")");
                    else
assmc(Integer.toString(ln),Integer.toString(lc),words[1],Integer.toString(getsymlc(opr[0]))+",
"+Integer.toString(getsymlc(op21)-
6)+"("+Integer.toString(index)+","+Integer.toString(getbr(getsymlc(op21)))+")");
                }
                lc+=4;
            }
```

```
            else if(words[1].equals("A") || words[1].equals("C"))
            {
               String[] opr = words[2].split(",");
               lci = getlitlc(opr[1]);
               assmc(Integer.toString(ln),Integer.toString(lc),words[1],Integer.toString(getsyml
c(opr[0]))+","+Integer.toString(lci-6)+"(0,"+Integer.toString(getbr(lci-6))+")");
               lc+=4;
            }
            else if(words[1].equals("BNE"))
            {
               lci = getsymlc(words[2]) - reg[getbr(7)-1][0];
               assmc(Integer.toString(ln),Integer.toString(lc),"BC","7,"+Integer.toString(lci)+"(
0,"+Integer.toString(getbr(7))+")");
               lc+=4;
            }
            else if(words[1].equals("BR"))
            {
               assmc(Integer.toString(ln),Integer.toString(lc),"BCR",Integer.toString(getbr(Inte
ger.parseInt(words[2])))+","+words[2]);
               lc+=4;
            }
            else if(words[1].equals("LTORG"))
            {
               //System.out.println(di);
               i=1;
               //System.out.println(data[i]);
               String[] opr21 = data[0].split("\\(");
               String op21 = opr21[0];
               //System.out.println(getsymlc(opr21[1]));
               opr21 = opr21[1].split("\\)");
               String op22 = opr21[0];
               assmc(Integer.toString(ln),Integer.toString(littab[0][0]),Integer.toString(getsyml
c(op22)),"");
               //System.out.println("in");
               opr21 = data[1].split("\\'");
               //System.out.println(opr21[0]);
               assmc(Integer.toString(ln),Integer.toString(littab[1][0]),opr21[1],"");
               opr21 = data[2].split("\\'");
               assmc(Integer.toString(ln),Integer.toString(littab[2][0]),opr21[1],"");
               opr21 = data[3].split("\\'");
               assmc(Integer.toString(ln),Integer.toString(littab[3][0]),opr21[1],"");
            }
            else if(words[1].equals("DS"))
            {
               assmc(Integer.toString(ln),Integer.toString(getsymlc(words[0])),"",".");
            }
            else if(words[1].equals("DC"))
            {
               String[] opr21 = words[2].split("\\'");
```

```
                assmc(Integer.toString(ln),Integer.toString(getsymlc(words[0])),"",opr21[1]);
            }
            line = reader.readLine();
            words = line.split("\\s+");
            ln++;
        }
        reader.close();
    }
    catch (IOException e) { e.printStackTrace(); }
    try(OutputStream fw = new FileOutputStream("machinecode.txt"))
        {
            String content = "LN LC Instruction/datum"+System.getProperty("line.separator");
            fw.write(content.getBytes(),0,content.length());
            for(i=0;i<mci;i++)
            {
                //BufferedWriter bw = new BufferedWriter(fw);
                content = macode[i][0]+"    "+macode[i][1]+"    "+macode[i][2]+"
"+macode[i][3]+System.getProperty("line.separator");
                //System.out.println(content);
                fw.write(content.getBytes(),0,content.length());
                //System.out.println(data[i]+" "+littab[i][0]+" "+littab[i][1]+" "+littab[i][2]);
            }
        }
        catch (IOException e) { e.printStackTrace(); }
        System.out.println("Check file machinecode.txt");
    }
}
```

Output-

Execution:

Input File:

prg - Notepad

File Edit View

```
PRGAM2 START 0
         USING *,15
         LA    15,SETUP
         SR    TOTAL,TOTAL
AC       EQU   2
INDEX  EQU   3
TOTAL  EQU   4
DATABASE EQU 13
SETUP  EQU   *
         USING SETUP,15
         L     DATABASE,=A(DATA1)
         USING DATAAREA,DATABASE
         SR    INDEX,INDEX
LOOP   L     AC,DATA1(INDEX)
         AR    TOTAL,AC
         A     AC,=F'6'
         ST    AC,SAVE(INDEX)
         A     INDEX,=F'4'
         C     INDEX,=F'8000'
         BNE   LOOP
         LR    1,TOTAL
         BR    14
         LTORG
SAVE   DS    2000F
DATAAREA EQU *
DATA1  DC    F'01'
       END
```

Machine Code:

machinecode - Note...

File Edit View

```
LN LC Instruction/datum
3     0      LA 15,6(0,15)
4     4      SR 4,4
11    6      L 13,42(0,15)
13    10     SR 3,3
14    12     L 2,0(3,13)
15    16     AR 4,2
16    18     A 2,46(0,15)
17    22     ST 2,58(3,15)
18    26     A 3,50(0,15)
19    30     C 3,54(0,15)
20    34     BC 7,6(0,15)
21    38     LR 1,4
22    40     BCR 15,14
23    48     8064
23    52     6
23    56     4
23    60     8000
24    64      .
26    8064   01
```

Ln 1, Col 1 100% Windows (CRLF) UTF-8

Conclusion- Thus we have Implemented program for pass 2 of two pass Assembler.

Experiment No. 6

Aim: Design an implementation of pass 1 of two pass macro processor.

Requirement: Java(jdk-11) IDE and printout pages

Theory:

In Pass-I the macro definitions are searched and stored in the macro definition table and the entry is made in macro name table.

SPECIFICATION OF DATABASES

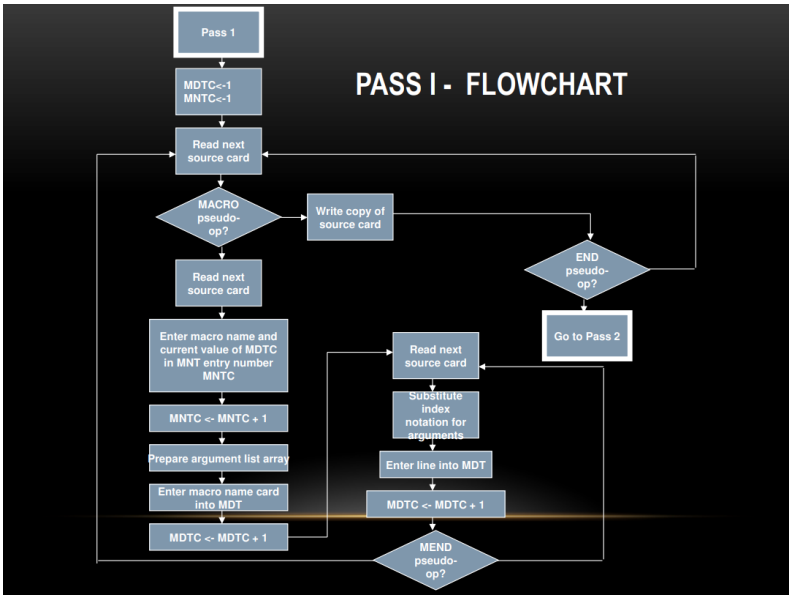
1. The input macro source program.
2. The output macro source program to be used by Pass2.
3. Macro-Definition Table (MDT), to store the body of macro defns .
4. Macro-Definition Table Counter (MDTC), to mark next available entry MDT.
5. Macro- Name Table (MNT) - store names of macros.
6. Macro Name Table counter (MNTC) - indicate the next available entry in MNT.
7. Argument List Array (ALA) - substitute index markers for dummy arguments before storing a macro-deff.

ALGORITHM

1. Pass1 of macro processor makes a line-by-line scan over its input.
2. Set MDTC = 1 as well as MNTC = 1.
3. Read next line from input program.
4. If it is a MACRO pseudo-op, the entire macro definition except this (MACRO) line is stored in MDT. The name is entered into Macro Name Table along with a pointer to the first location of MDT entry of the definition.

5. When the END pseudo-op is encountered all the macro-defns have been processed, so control is transferred to pass2

FLOWCHART



Code:

```
import java.util.*;
import java.lang.*;
import java.io.*;

class pass1
{
   public static void main(String []args)
   {
      BufferedReader reader;
      int lc=0,mnti=0,mdti=0,i,j,li=0,alai=0,alac=0,alasi=0,prgi=0;
      String[] mdt = new String[200];
      String[] mnt = new String[100];
      String[] ala = new String[100];
      int[] mntin = new int[100];
      int[] alain = new int[100];
      int[][] alas = new int[100][3];
      String[] prgstat = new String[200];
      try
      {
         reader = new BufferedReader(new FileReader("prg.txt"));
         String line = reader.readLine();
         String[] words = line.split("\\s+");
         //System.out.println(sym[0]+" "+symtab[0][0]);
```

```
        while (!line.trim().equals("END"))
        {
           if(words[0].equals("MACRO"))
           {
              li=0;alac=0;
              //System.out.println("yes");
              while(!words[0].equals("MEND"))
              {
                 line = reader.readLine();
                 words = line.split("\\s+");
                 if(li==0)
                 {
                    mnt[mnti] = words[0];
                    String[] op = words[1].split(",");
                    alas[alasi][0] = alai; alas[alasi][1] = mnti;alas[alasi][2]=op.length;
                    for(i=0;i<op.length;i++)
                    {
                       ala[alai] = op[i];
                       alain[alai] = alac;
                       alac++;
                       alai++;
                    }
                    mntin[mnti] = mdti;
                    mdt[mdti] = line;
                    mnti++;
                    mdti++;
                    alasi++;
                    li++;
                 }
                 else
                 {
                    for(i=alas[alasi-1][0];i<alai;i++)
                    {
                       if(line.contains(ala[i])==true) line =
line.replace(ala[i],"#"+Integer.toString(alain[i]));
                    }
                    mdt[mdti] = line;
                    mdti++;
                 }
              }
              //System.out.println(line);
           }
           else
           {
              prgstat[prgi] = line;
              prgi++;
           }
           line = reader.readLine();
           words = line.split("\\s+");
        }
```

```
        reader.close();
        prgstat[prgi] = line;
        prgi++;
        //for(i=0;i<mnti;i++) System.out.println(mnt[i]+" "+mntin[i]);
        //for(i=0;i<mdti;i++) System.out.println(i+" "+mdt[i]);
        //for(i=0;i<alai;i++) System.out.println(alain[i]+" "+ala[i]);

        try(OutputStream fw = new FileOutputStream("macro_name_table.txt"))
        {
           for(i=0;i<mnti;i++)
           {
              // SR NO       macro name      MDTindex
              String content = i+" "+mnt[i]+"
"+mntin[i]+System.getProperty("line.separator");
              fw.write(content.getBytes(),0,content.length());
           }
        }
        catch (IOException e) { e.printStackTrace(); }
        System.out.println("Check file macro_name_table.txt");
        try(OutputStream fw = new FileOutputStream("macro_definition_table.txt"))
        {
           for(i=0;i<mdti;i++)
           {
              // SR NO       macro definition
              String content = i+" "+mdt[i]+System.getProperty("line.separator");
              fw.write(content.getBytes(),0,content.length());
           }
        }
        catch (IOException e) { e.printStackTrace(); }
        System.out.println("Check file macro_definition_table.txt");
        try(OutputStream fw = new FileOutputStream("argument_list_array_pass_1.txt"))
        {
           for(i=0;i<alai;i++)
           {
              // SR NO       argument index in mdt       argument name
              String content = i+" "+alain[i]+" "+ala[i]+System.getProperty("line.separator");
              fw.write(content.getBytes(),0,content.length());
           }
        }
        catch (IOException e) { e.printStackTrace(); }
        System.out.println("Check file argument_list_array_pass_1.txt");
        try(OutputStream fw = new FileOutputStream("prg_intermidiate.txt"))
        {
           for(i=0;i<prgi;i++)
           {
              // program line
              String content = prgstat[i]+System.getProperty("line.separator");
              fw.write(content.getBytes(),0,content.length());
           }
        }
```

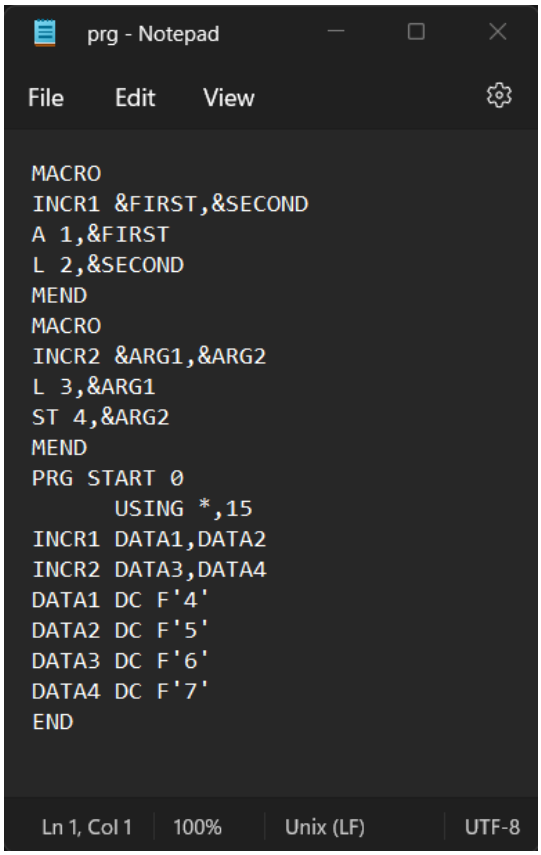
```
        catch (IOException e) { e.printStackTrace(); }
        System.out.println("Check file prg_intermidiate.txt");
        // This file is for background processing. This stores the start index of arguments of a
macro in the ala as the ala is the same for all macros.
        try(OutputStream fw = new FileOutputStream("alas.txt"))
        {
           for(i=0;i<alasi;i++)
           {
              // alastartindex      mntindex      number of arguments in that macro
              String content = alas[i][0]+" "+alas[i][1]+"
"+alas[i][2]+System.getProperty("line.separator");
              fw.write(content.getBytes(),0,content.length());
           }
        }
        catch (IOException e) { e.printStackTrace(); }
     }
     catch (IOException e) { e.printStackTrace(); }
  }
}
```

Output:

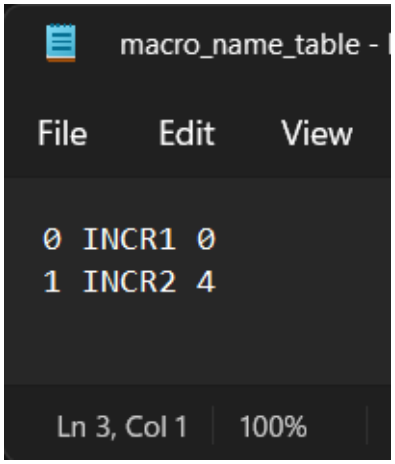
Execution:

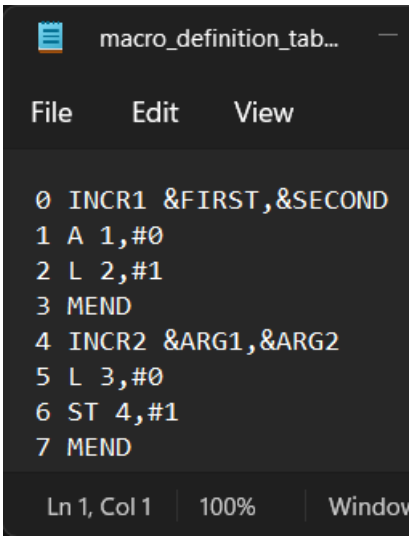
```
MINGW64:/c/Users/adnan/onedrive/desktop/college/sem6/spcc/exp6

adnan@LAPTOP-M72BKN5C MINGW64 ~/onedrive/desktop/college/sem6/spcc/exp6 (main)
$ javac pass1.java

adnan@LAPTOP-M72BKN5C MINGW64 ~/onedrive/desktop/college/sem6/spcc/exp6 (main)
$ java pass1
Check file macro_name_table.txt
Check file macro_definition_table.txt
Check file argument_list_array_pass_1.txt
Check file prg_intermidiate.txt

adnan@LAPTOP-M72BKN5C MINGW64 ~/onedrive/desktop/college/sem6/spcc/exp6 (main)
$
```

Input File:

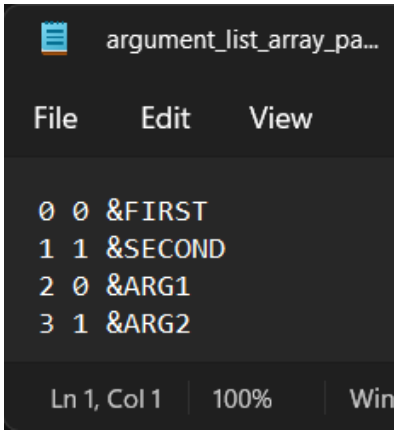


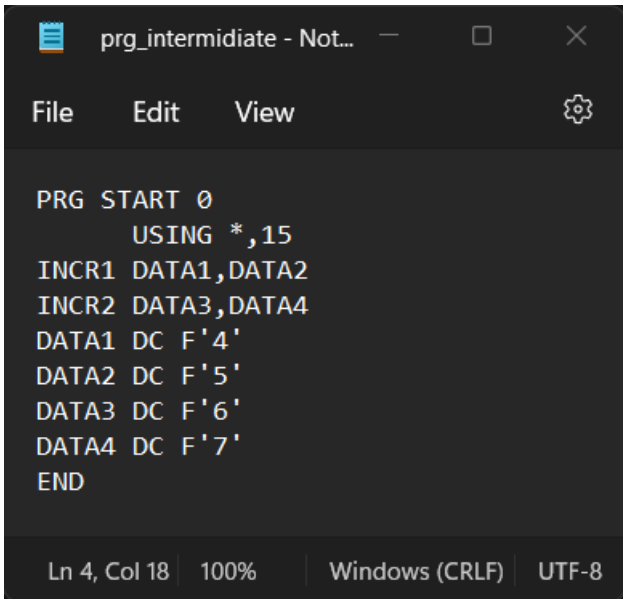
Macro name table and definition table:





Argument list array pass1 and program intermediate:





Conclusion: Thus we have Implemented program for pass 1 of two pass Macro Processor.

Experiment No. 7

Aim: Design an implementation of pass II of two pass macro processor.

Requirement: Java(jdk-11) IDE and printout pages

Theory:

In Pass-II the macro calls are identified and the arguments are placed in the appropriate place and the macro calls are replaced by macro definitions.

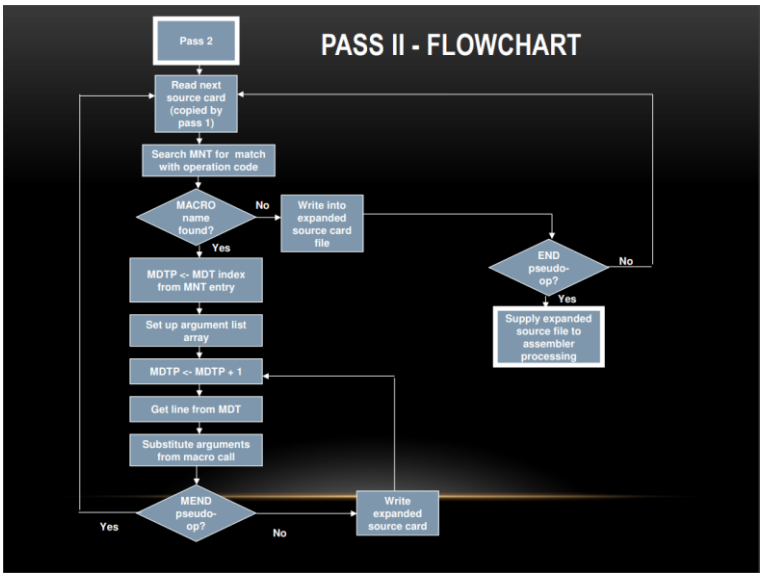
SPECIFICATION OF DATABASES

1. The input is from Pass1.
2. The output is expanded source to be given to assembler.
3. MDT and MNT are created by Pass1.
4. Macro-Definition Table Pointer (MDTP), used to indicate the next line of text to be used during macro expansion.
5. Argument List Array (ALA), used to substitute macro call arguments for the index markers in the stored macro-defns

ALGORITHM

1. This algorithm reads one line of i/p prog. at a time.
2. For each Line it checks if op-code of that line matches any of the MNT entry.
3. When match is found (i.e. when call is pointer called MDTF to corresponding macro defns stored in MDT.
4. The initial value of MDTP is obtained from MDT index field of MNT entry.
5. The macro expander prepares the ALA consisting of a table of dummy argument indices & corresponding arguments to the call.

6. Reading proceeds from the MDT, as each successive line is read, The values form the argument list one substituted for dummy arguments indices in the macro defn .
7. Reading MEND line in MDT terminates expansion of macro & scanning continues from the input file.
8. When END pseudo-op encountered, the expanded source program is given to the assembler.



Code:

```
import java.io.*;
import java.util.*;
import java.lang.*;

class pass2
{
   static int lc=0,mnti=0,mdti=0,i,j,li=0,alai=0,alac=0,alasi=0,prgi=0;
   static String[] mdt = new String[200];
   static String[] mnt = new String[100];
   static String[] ala = new String[100];
   static int[] mntin = new int[100];
   static int[] alain = new int[100];
   static int[][] alas = new int[100][3];
   static String[] prgstat = new String[200];

   public static int ifmacro(String name)
   {
      for(i=0;i<mnti;i++)
      {
         if(name.equals(mnt[i])) return i;
      }
      return -1;
```

```
    }

    public static void macroexp(int mi)
    {
        try
        {
            BufferedReader r;
            int ai=0,al=0;
            r = new BufferedReader(new FileReader("macro_definition_table.txt"));
            String line = r.readLine();
            String[] words = line.split("\\s+");
            while(true)
            {
                if(Integer.parseInt(words[0]) == mntin[mi]) break;
                line = r.readLine();
                words = line.split("\\s+");
            }
            //System.out.println(words[1]);
            for(i=0;i<alasi;i++)
            {
                if(alas[i][1]==mi)
                {
                    ai = alas[i][0];
                    al = alas[i][2];
                }
            }
            while(!words[1].equals("MEND"))
            {
                if(ifmacro(words[1])!=-1)
                {
                    line = r.readLine();
                    words = line.split("\\s+");
                    continue;
                }
                else
                {
                    for(i=ai;i<ai+al;i++)
                    {
                        //System.out.println("#"+Integer.toString(alain[i]));
                        if(words[2].contains("#"+Integer.toString(alain[i]))==true) words[2] =
words[2].replace("#"+Integer.toString(alain[i]),ala[i]);
                    }
                    String content = words[1]+" "+words[2];
                    prgstat[prgi] = content;
                    prgi++;
                }
                line = r.readLine();
                words = line.split("\\s+");
            }
        }
```

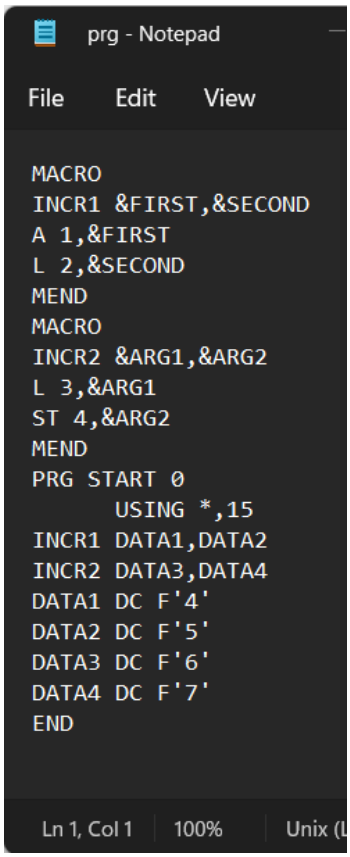
```
    catch (IOException e) { e.printStackTrace(); }
  }
  public static void main(String []args)
  {
    BufferedReader reader;
    try
    {
      reader = new BufferedReader(new FileReader("macro_name_table.txt"));
      String line = reader.readLine();
      while(line!=null)
      {
        String[] words = line.split("\\s+");
        mnt[mnti] = words[1];
        mntin[mnti]= Integer.parseInt(words[2]);
        mnti++;
        line = reader.readLine();
      }
      //for(i=0;i<mnti;i++) System.out.println(mnt[i]+" "+mntin[i]);
      reader = new BufferedReader(new FileReader("macro_definition_table.txt"));
      line = reader.readLine();
      while(line!=null)
      {
        String[] words = line.split("\\s+");
        mdt[mdti] = words[1];
        mdti++;
        line = reader.readLine();
      }
      //for(i=0;i<mdti;i++) System.out.println(i+" "+mdt[i]);
      reader = new BufferedReader(new FileReader("argument_list_array_pass_1.txt"));
      line = reader.readLine();
      while(line!=null)
      {
        String[] words = line.split("\\s+");
        alain[alai] = Integer.parseInt(words[1]);
        ala[alai] = words[2];
        alai++;
        line = reader.readLine();
      }

      reader = new BufferedReader(new FileReader("alas.txt"));
      line = reader.readLine();
      while(line!=null)
      {
        String[] words = line.split("\\s+");
        alas[alasi][0] = Integer.parseInt(words[0]);
        alas[alasi][1] = Integer.parseInt(words[1]);
        alas[alasi][2] = Integer.parseInt(words[2]);
        alasi++;
        line = reader.readLine();
      }
```

```
    //for(i=0;i<alai;i++) System.out.println(alain[i]+" "+ala[i]);
  }
  catch (IOException e) { e.printStackTrace(); }
  try
  {
    reader = new BufferedReader(new FileReader("prg_intermidiate.txt"));
    String line = reader.readLine();
    String[] words = line.split("\\s+");
    while (!line.trim().equals("END"))
    {
      int ai=0;
      int macval = ifmacro(words[0]);
      //System.out.println(words[0]+" "+macval);
      if(macval!=-1)
      {
        //System.out.println(macval);
        String[] op = words[1].split(",");
        for(i=0;i<alasi;i++)
        {
          if(alas[i][1]==macval)
          {
            ai = alas[i][0];
          }
        }
        for(i=ai;i<ai+op.length;i++)
        {
          ala[i]=op[i-ai];
          //System.out.println(ala[i]);
        }
        macroexp(macval);
      }
      else
      {
        prgstat[prgi] = line;
        prgi++;
      }
      line = reader.readLine();
      words = line.split("\\s+");
    }
    //for(i=0;i<prgi;i++) System.out.println(i+" "+prgstat[i]);
    reader.close();
  }
  catch (IOException e) { e.printStackTrace(); }
  try(OutputStream fw = new FileOutputStream("prg_expanded.txt"))
  {
    for(i=0;i<prgi;i++)
    {
      // program statement
      String content =prgstat[i]+System.getProperty("line.separator");
      fw.write(content.getBytes(),0,content.length());
```

```
        }
      }
      catch (IOException e) { e.printStackTrace(); }
      try(OutputStream fw = new FileOutputStream("argument_list_array_pass_2.txt"))
      {
         for(i=0;i<alai;i++)
         {
            // SR NO       argument index in mdt       argument name(Replaced with actual
arguments)
            String content = i+" "+alain[i]+" "+ala[i]+System.getProperty("line.separator");
            fw.write(content.getBytes(),0,content.length());
         }
      }
      catch (IOException e) { e.printStackTrace(); }
      System.out.println("Check file argument_list_array_pass_2.txt");
      System.out.println("Check file prg_expanded.txt");
   }
}
```

Output:

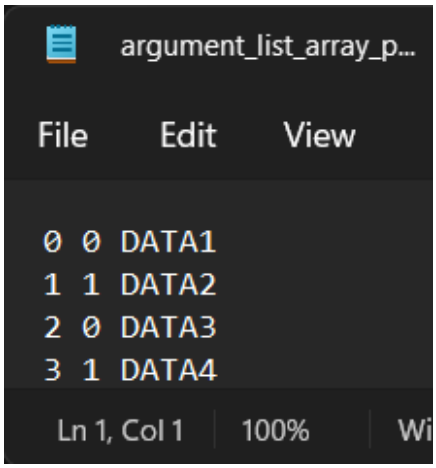
Input File:

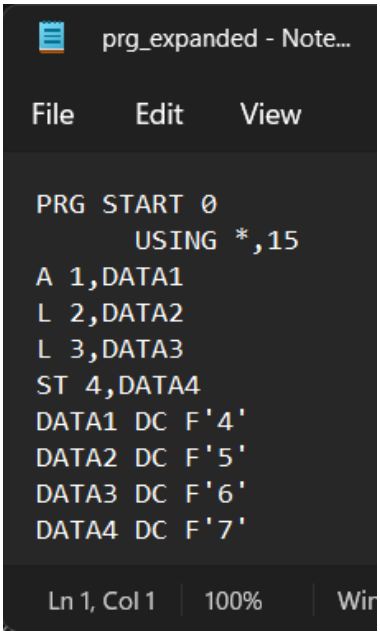


Execution:

```
MINGW64:/c/Users/adnan/onedrive/Desktop/college/sem6/spcc/exp7

adnan@LAPTOP-M72BKN5C MINGW64 ~/onedrive/Desktop/college/sem6/spcc/exp7 (main)
$ javac pass2.java

adnan@LAPTOP-M72BKN5C MINGW64 ~/onedrive/Desktop/college/sem6/spcc/exp7 (main)
$ java pass2
Check file argument_list_array_pass_2.txt
Check file prg_expanded.txt

adnan@LAPTOP-M72BKN5C MINGW64 ~/onedrive/Desktop/college/sem6/spcc/exp7 (main)
$
```

Argument list array pass2 and expanded program:





Conclusion: Thus we have Implemented program for pass 2 of two pass macro processor.

Experiment No. 8

Aim: Implementation of Lexical Analyzer

Requirement: Compatible version of java

Theory:

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. the lexical analyzer produces as output a token of the form: <token-name; attribute-value> that it passes on to the subsequent phase, syntax analysis.

Token-name is an abstract symbol that is used during syntax analysis.

Attribute-value points to an entry in the symbol table for this token.

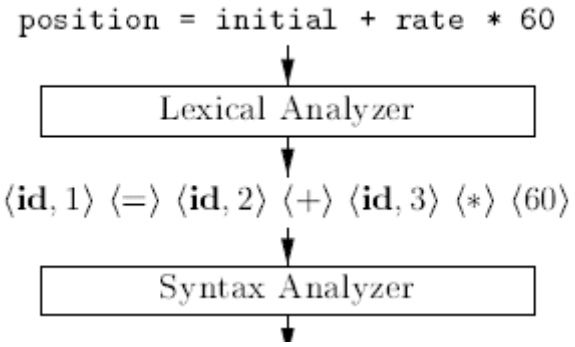
Information from the symbol-table entry is needed for semantic analysis and code generation. For example, suppose a source program contains the assignment statement

position = initial + rate * 60

The representation of the above assignment statement after lexical analysis as the sequence of tokens

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

In this representation, the token names =, +, and * are abstract symbols for the assignment, addition, and multiplication operators, respectively.



Code:

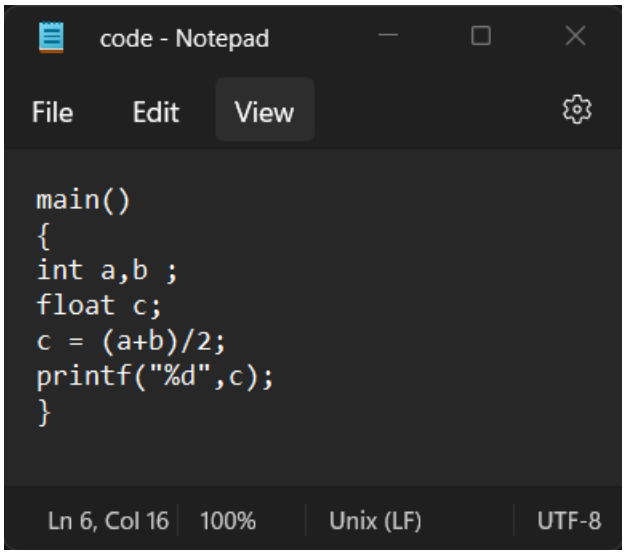
```
import java.util.*;
import java.io.*;

class Lex {

   public static void main(String[] args) {

      List<String> keywords =
Arrays.asList("break","else","auto","case","char","const","continue","default","do","double",
"else","enum","extern","float","for","goto","if","int","long","register","return","short","signe
d","sizeof","static","struct","switch","typedef","union","unsigned","void","volatile","while");

      List<String> operators = Arrays.asList("+","-
","*","/","%","==","!=","||","=","|","&&","&","{","}","(",")");
      List<String> delimiter = Arrays.asList(";");
      List<String> functions = Arrays.asList("printf","main","scanf");
      Vector<String> identifiers = new Vector<>();
      try{
         BufferedReader reader = new BufferedReader(new FileReader("code.txt"));
         String line = reader.readLine();
         int i;

         while(line!=null) {
            if(line.length()==1) {
               if(operators.contains(line))
System.out.print("Operator"+operators.indexOf(line)+" ");
            }
            else {
               String temp="";int igb=0;
               for(i=0;i<line.length();i++) {
                  if(line.charAt(i) == ' ') {
                     if(keywords.contains(temp))
System.out.print("Keyword"+keywords.indexOf(temp)+" ");
                     else if(operators.contains(temp))
System.out.print("Operator"+operators.indexOf(temp)+" ");
                     //System.out.print(temp+"//");
```

```
                temp="";
            }
            else if(line.charAt(i) == '(') {
                if(functions.contains(temp)) {
                    System.out.print("Function"+functions.indexOf(temp)+" ");
                    igb=1;
                    temp="";
                }
                else {
                    System.out.print("Operator"+operators.indexOf("(")+" ");
                }
                //System.out.print(temp+"//");
            }
            else if(line.charAt(i) == ')') {
                if(igb==1) igb=0;
                else {
                    if(temp!="") {
                        if(keywords.contains(temp))
System.out.print("Keyword"+keywords.indexOf(temp)+" ");
                        else if(functions.contains(temp))
System.out.print("Function"+functions.indexOf(temp)+" ");
                        else {
                            System.out.print("Identifier");
                            if(!identifiers.contains(temp)) identifiers.add(temp);
                            System.out.print(identifiers.indexOf(temp)+" ");
                        }
                    }
                    System.out.print("Operator"+operators.indexOf(")")+" ");
                }
                //System.out.print(temp+"//");
                temp="";
            }
            else if(igb==1) continue;
            else if(line.charAt(i) == ',') {
                if(keywords.contains(temp))
System.out.print("Keyword"+keywords.indexOf(temp)+" SpecialCharacter");
                else if(functions.contains(temp))
System.out.print("Function"+functions.indexOf(temp)+" SpecialCharacter");
                else {
                    System.out.print("Identifier");
                    if(!identifiers.contains(temp)) identifiers.add(temp);
                    System.out.print(identifiers.indexOf(temp)+" SpecialCharacter ");
                }
                //System.out.print(temp+"//");
                temp="";
            }
            else if(operators.contains(Character.toString(line.charAt(i)))) {
                if(temp!="") {
                    if(keywords.contains(temp))
System.out.print("Keyword"+keywords.indexOf(temp)+" ");
```

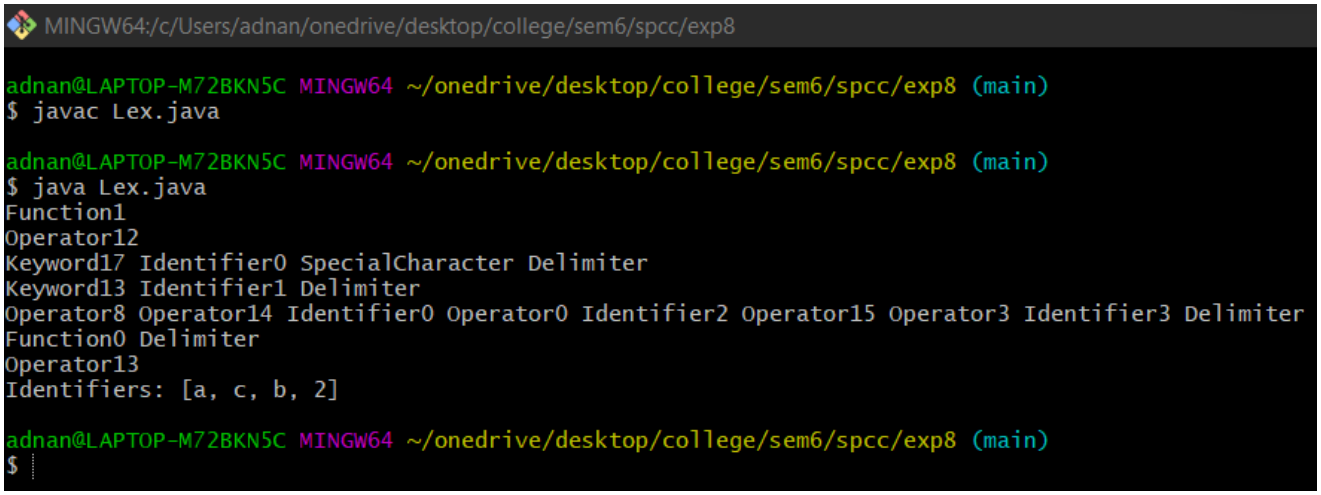
```
                        else if(functions.contains(temp))
System.out.print("Function"+functions.indexOf(temp)+" ");
                        else {
                            System.out.print("Identifier");
                            if(!identifiers.contains(temp)) identifiers.add(temp);
                            System.out.print(identifiers.indexOf(temp)+" ");
                        }
                    }
                    System.out.print("Operator"+operators.indexOf(Character.toString(line.cha
rAt(i)))+" ");
                    //System.out.print(temp+"//");
                    temp="";
                }
                else if(delimiter.contains(Character.toString(line.charAt(i)))) {
                    if(temp!="") {
                        if(keywords.contains(temp))
System.out.print("Keyword"+keywords.indexOf(temp)+" ");
                        else if(functions.contains(temp))
System.out.print("Function"+functions.indexOf(temp)+" ");
                        else {
                            System.out.print("Identifier");
                            if(!identifiers.contains(temp)) identifiers.add(temp);
                            System.out.print(identifiers.indexOf(temp)+" ");
                        }
                    }
                    System.out.print("Delimiter ");
                    //System.out.print(temp+"//?");
                    temp="";
                }
                else temp+=Character.toString(line.charAt(i));
            }

        }
        System.out.println();
        line = reader.readLine();

      }
      System.out.println("Identifiers: "+identifiers);
    }catch(IOException e){}

  }

}
```

Output:

Input File:



Execution:



Conclusion: Thus we have successfully implemented Lexical Analyzer in Java.

Experiment No. 09

Aim- Implementation of parser

Requirement- Java and printout pages

Theory- A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens, interactive commands, or program instructions and breaks them up into parts that can be used by other components in programming.

The overall process of parsing involves three stages:

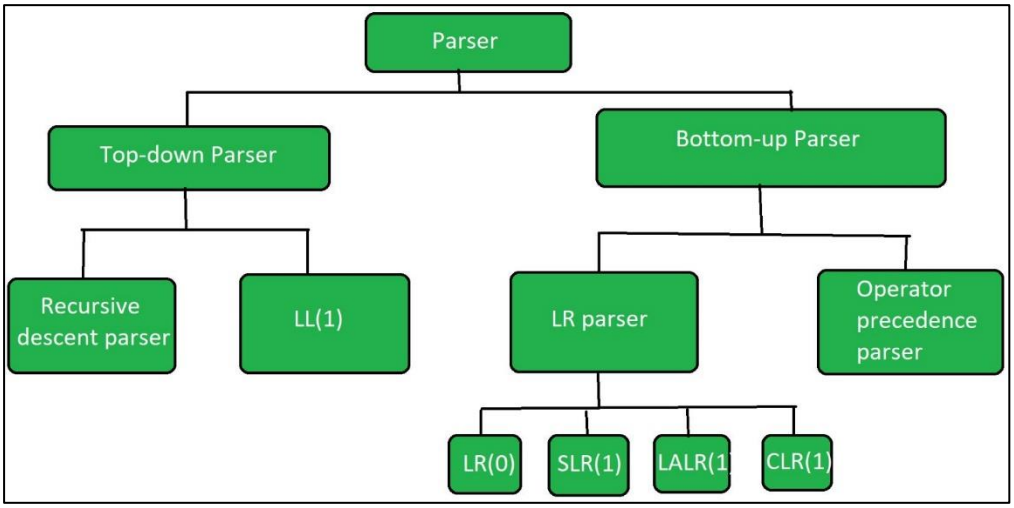
1. **Lexical Analysis:** A lexical analyzer is used to produce tokens from a stream of input string characters, which are broken into small components to form meaningful expressions. A token is the smallest unit in a programming language that possesses some meaning (such as +, -, *, “function”, or “new” in JavaScript).
2. **Syntactic Analysis:** Checks whether the generated tokens form a meaningful expression. This makes use of a context-free grammar that defines algorithmic procedures for components. These work to form an expression and define the particular order in which tokens must be placed.
3. **Semantic Parsing:** The final parsing stage in which the meaning and implications of the validated expression are determined and necessary actions are taken.

A parser's main purpose is to determine if input data may be derived from the start symbol of the grammar. This is achieved as follows:

- **Top-Down Parsing**: Involves searching a parse tree to find the left-most derivations of an input stream by using a top-down expansion. Parsing begins with the start symbol which is transformed into the input symbol until all symbols are translated and a parse

tree for an input string is constructed. Examples include LL parsers and recursive-descent parsers. Top-down parsing is also called predictive parsing or recursive parsing.

- **Bottom-Up Parsing:** Involves rewriting the input back to the start symbol. It acts in reverse by tracing out the rightmost derivation of a string until the parse tree is constructed up to the start symbol This type of parsing is also known as shift-reduce parsing. One example is an LR parser.



Code-

Lex Code:

```
%{
#include "y.tab.h"
%}
%%
[ /t/n]                 { ; }
"select"                 { return Select; }
"from"                   { return from; }
"distinct"               { return distinct; }
"where"                   { return where; }
"like"                  { return like; }
"and"                    { return and; }
```

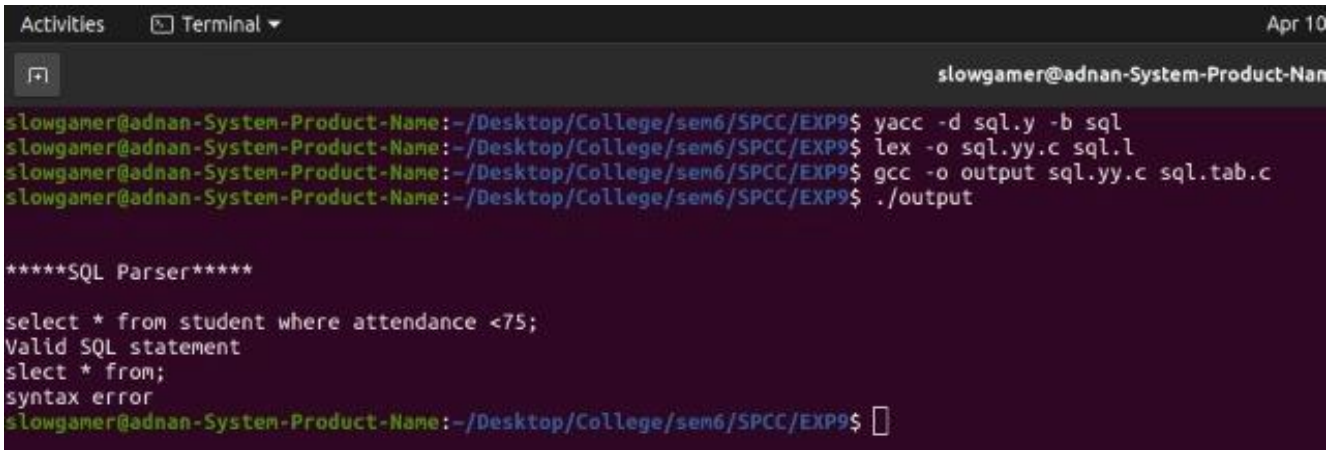
```
"or"                    { return or; }
[0-9]+                  { return number; }
[A-Za-z]([A-Za-z]|[0-9])*       { return id; }
"<="                    { return le; }
">="                    { return ge; }
"=="                    { return eq; }
"!="                    { return ne; }
.                       { return yytext[0]; }

%%

int yywrap(void) { return 1; }
```

Yacc Code:

```
%{

void yyerror (char *s);
int yylex(void);
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

%}

%start start

%token Select from distinct where like and or number id le ge eq ne

%left and or
%left '<' '>' le ge eq ne

%right '='

%%

start   :  sql_a ';'            { printf("Valid SQL statement"); }
        |
        ;

sql_a   :  Select attributes from tables sql_b      { ; }
        | Select distinct attributes from tables sql_b      { ; }
        | Select distinct attributes from tables      { ; }
        | Select attributes from tables      { ; }
        ;

sql_b   :  where condition    { ; }
        ;

attributes :   id ',' attributes
```

```
            | '*'
            | id
            ;

tables    :    id ',' tables
            | id
            ;

condition  :   condition or condition
            | condition and condition
            | E
            ;

E     :       F '=' F
            | F '<' F
            | F '>' F
            | F le F
            | F ge F
            | F eq F
            | F ne F
            | F or F
            | F and F
            | F like F
            ;

F   :         id
            | number
            ;

%%

int main(void)
{
   printf("\n\n*****SQL Parser*****\n\n");
   return yyparse();
}

void yyerror (char *s) { fprintf (stderr, "%s\n", s); }
```

Output-



Conclusion- We have successfully implemented SQL parser using LEX and YACC.

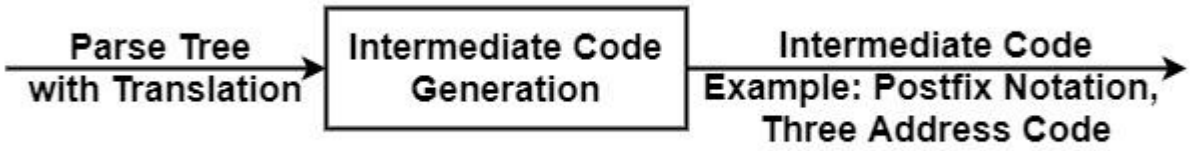
Experiment No. 10

Aim: To implement intermediate code generator.

Requirements: Compatible version of Java.

Theory:

Intermediate code can translate the source program into the machine program. Intermediate code is generated because the compiler can't generate machine code directly in one pass. Therefore, first, it converts the source program into intermediate code, which performs efficient generation of machine code further. The intermediate code can be represented in the form of postfix notation, syntax tree, directed acyclic graph, three address codes, Quadruples, and triples.

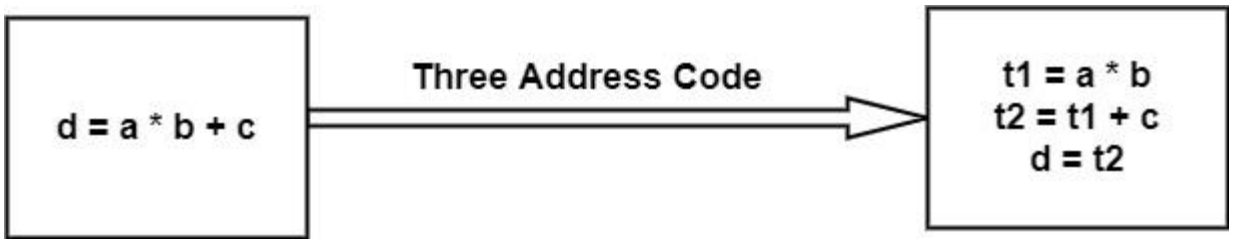


Intermediate Code Generation

Example of Intermediate Code Generation −

- **Three Address Code**− These are statements of form c = a op b, i.e., in which there will be at most three addresses, i.e., two for operands & one for Result. Each instruction has at most one operator on the right-hand side.

Example of three address code for the statement



Advantages of Intermediate Code Generation

- It is Machine Independent. It can be executed on different platforms.
- It creates the function of code optimization easy. A machine-independent code optimizer can be used to intermediate code to optimize code generation.
- It can perform efficient code generation.
- From the existing front end, a new compiler for a given back end can be generated.
- Syntax-directed translation implements the intermediate code generation, thus by augmenting the parser, it can be folded into the parsing

Code:

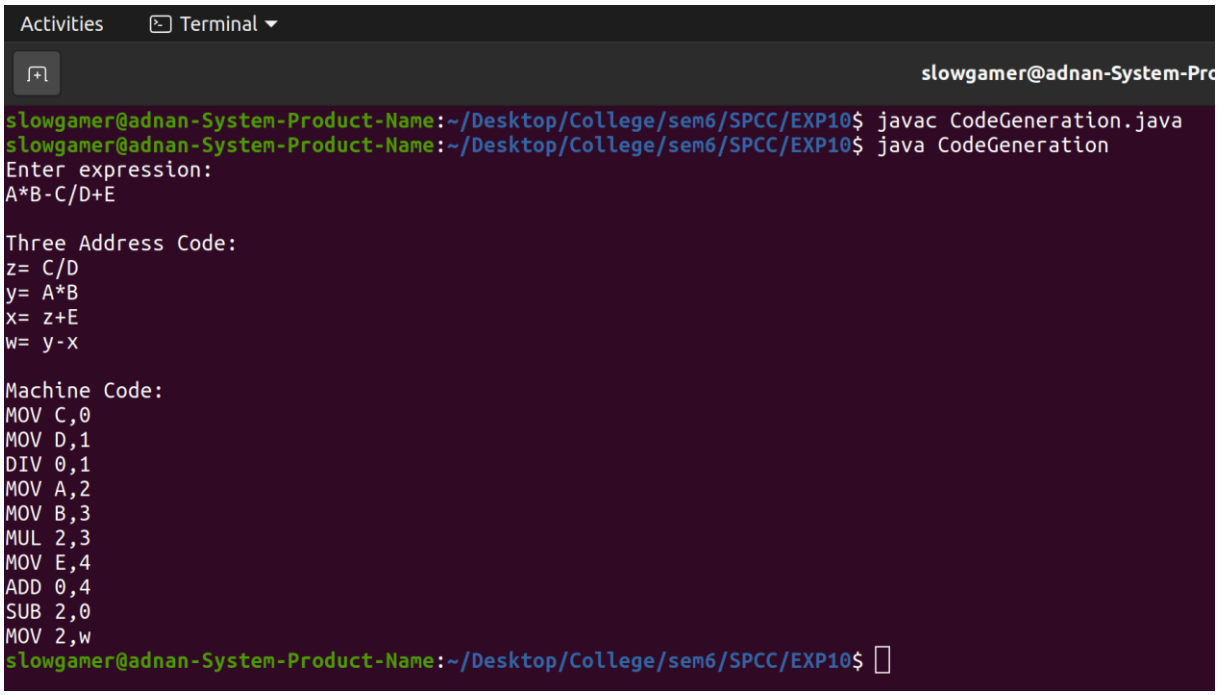
```
import java.util.*;
import java.io.*;

class CodeGeneration {

   static Vector<String> threeadd = new Vector<>();
   static char tempVar='z';
   static int regcounter=0;
   static Map<String,Integer> reg = new HashMap<>();
   static Vector<String> machinecode = new Vector<>();

   public static void main(String []args) {

      String input;int i;
      Scanner sc = new Scanner(System.in);

      System.out.println("Enter expression: ");
      input = sc.nextLine();

      generateThreeAddressCode(input);

      System.out.println();
      System.out.println("Three Address Code: ");
      for(i=0;i<threeadd.size();i++) System.out.println(threeadd.get(i));

      generateMachineCode();

      System.out.println();
      System.out.println("Machine Code: ");
```

```
    for(i=0;i<machinecode.size();i++) System.out.println(machinecode.get(i));
  }

  public static void generateThreeAddressCode(String s) {

    String after="";
    if(s.contains("=") && s.length()==3) {
      threeadd.add(s);
      return;
    }
    else if(s=="") return;

    if(s.contains("(")) {
      int start = s.indexOf("(");
      int end = s.indexOf(")");
      String temp = tempVar+"= "+s.substring(start+1,end);
      after = s.substring(0,start)+tempVar+s.substring(end+1);
      tempVar--;
      threeadd.add(temp);
    }
    else if(s.contains("/")) {
      int c = s.indexOf("/");
      String temp = tempVar+"= "+s.substring(c-1,c)+"/"+s.substring(c+1,c+2);
      after = s.substring(0,c-1)+tempVar+s.substring(c+2);
      tempVar--;
      threeadd.add(temp);
    }

    else if(s.contains("*")) {
      int c = s.indexOf("*");
      String temp = tempVar+"= "+s.substring(c-1,c)+"*"+s.substring(c+1,c+2);
      after = s.substring(0,c-1)+tempVar+s.substring(c+2);
      tempVar--;
      threeadd.add(temp);
    }

    else if(s.contains("+")) {
      int c = s.indexOf("+");
      String temp = tempVar+"= "+s.substring(c-1,c)+"+"+s.substring(c+1,c+2);
      after = s.substring(0,c-1)+tempVar+s.substring(c+2);
      tempVar--;
      threeadd.add(temp);
    }

    else if(s.contains("-")) {
      int c = s.indexOf("-");
      String temp = tempVar+"= "+s.substring(c-1,c)+"-"+s.substring(c+1,c+2);
      after = s.substring(0,c-1)+tempVar+s.substring(c+2);
      tempVar--;
      threeadd.add(temp);
```

```
        }
        //System.out.println(after);
        generateThreeAddressCode(after);
    }

    public static void generateMachineCode() {

        int i,p=0,f=0;
        for(i=0;i<threeadd.size();i++) {
            String s = threeadd.get(i);
            String temp="";

            if(s.contains("+")) p = s.indexOf("+");
            else if(s.contains("-")) p = s.indexOf("-");
            else if(s.contains("*")) p = s.indexOf("*");
            else if(s.contains("/")) p = s.indexOf("/");
            else if(s.contains("=")) {
                p = s.indexOf("=");
                temp = "MOV "+reg.get(s.substring(p+1))+","+s.substring(0,1);
                machinecode.add(temp);
                temp="";
                f=1;
                continue;
            }

            if(reg.containsKey(s.substring(p-1,p)) == false) {
                reg.put(s.substring(p-1,p),regcounter);
                temp = "MOV "+s.substring(p-1,p)+","+regcounter;
                machinecode.add(temp);
                regcounter++;
            }
            //System.out.println(temp);
            temp="";
            if(reg.containsKey(s.substring(p+1,p+2)) == false) {
                reg.put(s.substring(p+1,p+2),regcounter);
                temp = "MOV "+s.substring(p+1,p+2)+","+regcounter;
                machinecode.add(temp);
                regcounter++;
            }
            //System.out.println(temp);
            temp="";

            if(s.contains("+")) temp = "ADD ";
            else if(s.contains("-")) temp = "SUB ";
            else if(s.contains("*")) temp = "MUL ";
            else if(s.contains("/")) temp = "DIV ";

            temp += reg.get(s.substring(p-1,p))+","+reg.get(s.substring(p+1,p+2));
            int r= reg.get(s.substring(p-1,p));
            reg.remove(s.substring(p-1,p));
```

```
            reg.put(s.substring(0,1),r);
            machinecode.add(temp);
            //System.out.println(temp);
        }
        if(f==0) {
            String temp="",s=threeadd.get(i-1);
            temp = "MOV "+reg.get(s.substring(0,1))+","+s.substring(0,1);
            machinecode.add(temp);
        }
    }
}
```

Output:



Conclusion: We have successfully implemented intermediate code generation program for arithmetic expression in JAVA.

Experiment No. 11

AIM: Case study on LLVM tool

THEORY: What is LLVM?

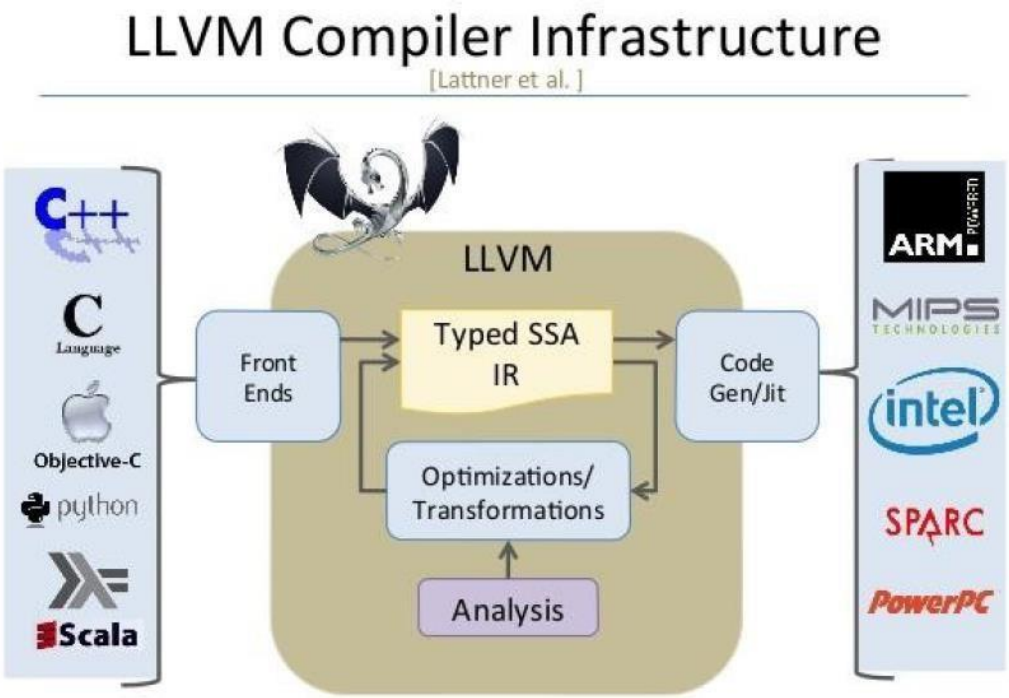
- LLVM is an acronym that stands for low level virtual machine. It also refers to a compiling technology called the LLVM project, which is a collection of modular and reusable compiler and toolchain technologies.
- The LLVM project has grown beyond its initial scope as the project is no longer focused on traditional virtual machines.
- LLVM is a compiler and a toolkit for building compilers, which are programs that convert instructions into a form that can be read and executed by a computer.
- The LLVM project is a collection of modular and reusable compiler and tool chain technologies. LLVM helps build new computer languages and improve existing languages.
- It automates many of the difficult and unpleasant tasks involved in language creation, such as porting the output code to multiple platforms and architectures.

HOW DOES A LLVM COMPILER WORK ?

- On the front end, the LLVM compiler infrastructure uses clang —a compiler for programming languages C, C++ and CUDA - to turn source code into an interim format. Then the LLVM clang code generator on the back end turns the interim format into final machine code.
- The compiler has five basic phases:
- Lexical Analysis - Converts program text into words and tokens (everything apart from words, such as spaces and semicolons).
- Parsing - Groups the words and tokens from the lexical analysis into a form that makes sense.
- Semantic Analyser — Identifies the types and logics of the programs.

- Optimization - Cleans the code for better run-time performance and addresses memory-related issues.
- Code Generation - Turns code into a binary file that is executable LLVM COMPILER

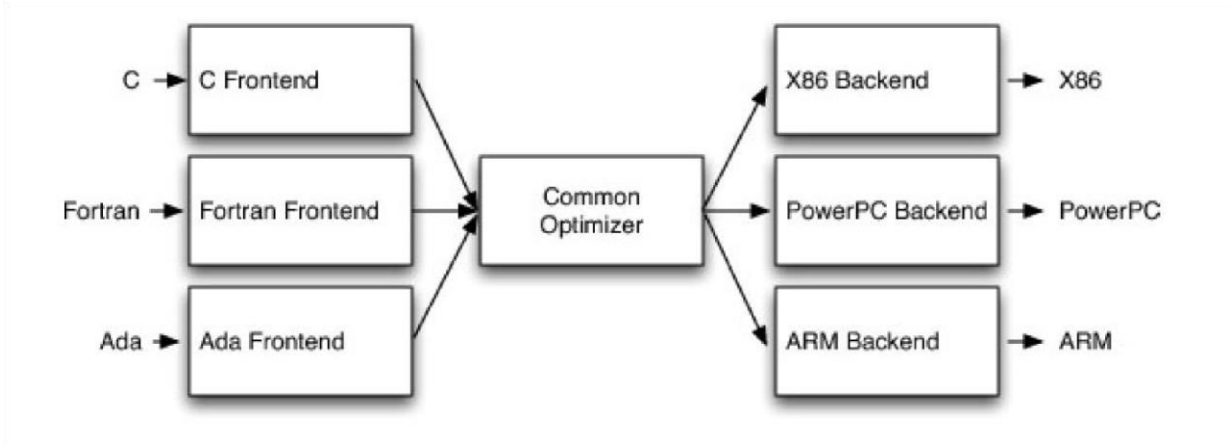
ARCHITECTURE:



At the frontend you have Perl, and many other high level languages. At the backend, you have the natives code that runs directly on the machine.

At the centre is your intermediate code representation. If every high level language can be represented in this LLVM IR format, then analysis tools based on this IR can be easily reused - that is the basic rationale.

LLVM IR:



The LLVM project/infrastructure: This is an umbrella for several projects that, together, form a complete compiler: frontends, backends, optimizers, assemblers, linkers, libc++, compiler-rt, and a JIT engine. The word "LLVM" has this meaning, for example, in the following sentence: "LLVM consists of several projects".

An LLVM-based compiler: This is a compiler built partially or completely with the LLVM infrastructure. For example, a compiler might use LLVM for the frontend and backend but use GCC and GNU system libraries to perform the final link. LLVM has this meaning in the following sentence, for example: "I used LLVM to compile C programs to a MIPS platform".

LLVM libraries: This is the reusable code portion of the LLVM infrastructure. For example, LLVM has this meaning in the sentence: "My project uses LLVM to generate code through its Just-in-Time compilation framework".

LLVM core: The optimizations that happen at the intermediate language level and the backend algorithms form the LLVM core where the project started. LLVM has this meaning in the following sentence: "LLVM and Clang are two different projects".

The LLVM IR: This is the LLVM compiler intermediate representation. LLVM has this meaning when used in sentences such as "I built a frontend that translates my own language to LLVM".

Three Primary LLVM Components:

The LLVM Virtual Instruction Set –

1. The common language- and target-independent IR
2. Internal (IR) and external (persistent) representation

A collection of well-integrated libraries - Analyses, optimizations, code generators, JIT compiler, garbage collection support, profiling, ...

A collection of tools built from the libraries - Assemblers, automatic debugger, linker, code generator, compiler driver, modular optimizer,..

<u>Why LLVM ?</u>

- A need for a compiler that allows better diagnostics
- Better integration with IDEs, a license that is compatible with commercial products.

- Fast compiler that is easy to develop and maintain.
- Useful for performing optimizations and transformations on code

Code Generation Using LLVM:

The LLVM target-independent code generator is a framework that provides a suite of reusable components for translating the LLVM internal representation to the machine code for a specified target—either in assembly form (suitable for a static compiler) or in binary machine code format (usable for a JIT compiler).

The LLVM target-independent code generator consists of six main components:

1. Abstract target description interfaces which capture important properties about various aspects of the machine, independently of how they will be used. These interfaces are defined in include/llvm/Target/.
2. Classes used to represent the code being generated for a target. These classes are intended to be abstract enough to represent the machine code for any target machine. These classes are defined in include/llvm/CodeGen/. At this level, concepts like “constant pool entries” and “jump tables” are explicitly exposed.
3. Classes and algorithms used to represent code at the object file level, the MC Layer. These classes represent assembly level constructs like labels, sections, and instructions. At this level, concepts like “constant pool entries” and “jump tables” don’t exist.

4. Target-independent algorithms used to implement various phases of native code generation (register allocation, scheduling, stack frame representation, etc). This code lives in lib/CodeGen/.

5. §. Implementations of the abstract target description interfaces for particular targets. These machine descriptions make use of the components provided by LLVM, and can optionally provide custom target-specific passes, to build complete code generators for a specific target. Target descriptions live in lib/Target/.
6. 6. The target-independent JIT components. The LLVM JIT is completely target independent (it uses the TargetJlTlnfo structure to interface for target-specific issues. The code for the target- independent JIT lives in lib/ExecutionEngine/JIT.

Depending on which part of the code generator you are interested in working on, different pieces of this will be useful to you. In any case, you should be familiar with the target description and machine code representation classes. If you want to add a backend for a new target, you will need to implement the target description classes for your new target and understand the LLVM code representation. If you are interested in implementing a new code generation algorithm, it should only depend on the target-description and machine code representation classes, ensuring that it is Portable.

Uses of LLVM:

LLVM is a modular compiler infrastructure: — Primary focus is on providing good interfaces & robust components — LLVM can be used for many things other than simple static compilers.

LLVM provides language- and target-independent components: — Does not force use of JIT, GC, or a particular object model — Code from different languages can be linked together and optimized.

LLVM is well designed and provides aggressive functionality: — Inter procedural optimization, link-time/install-time optimization today.

CONCLUSION: Thus we have studied about LLVM and code generation using LLVM.

76_ Adnan Shaikh

Page No.: 1
Date. / /

Subject :- SPCC

Sem :- VI

Assignment No :- 1

Q.1) What is system programming? List some system programs and write their functions.

→ System Software or System program :-

System software is the software which is required to run the hardware parts of computer and other application software.

System program is a set of program which are developed to operate, control and extend processing capabilities of computer itself.

- Types of system software :-

1. Standard System program
   - It contains assembler, processor, linker, compiler, debugger, loaders, editors etc

2. Operating System Software
   - It include all operating system software which can be used to create interface between hardware and user application.
   - Eg :- Windows, DOS, UNIX, Linux etc.

(1) Assembler :-

- Assembler is a system software which convert programmes written in Assembly language into Machine language.

(2) Macro processor :-
- A Macro processor is a program which copes a bunch of text from one place to another. It replaces the macro cell into macro definition.

(3) Loader :-
- Loader performs the function of placing object code into main memory for execution purpose.
- To do this loader translate object code into executable form.

(4) Linker :-
- Linker is a program in a system which helps to link a object module of program into a single object file.

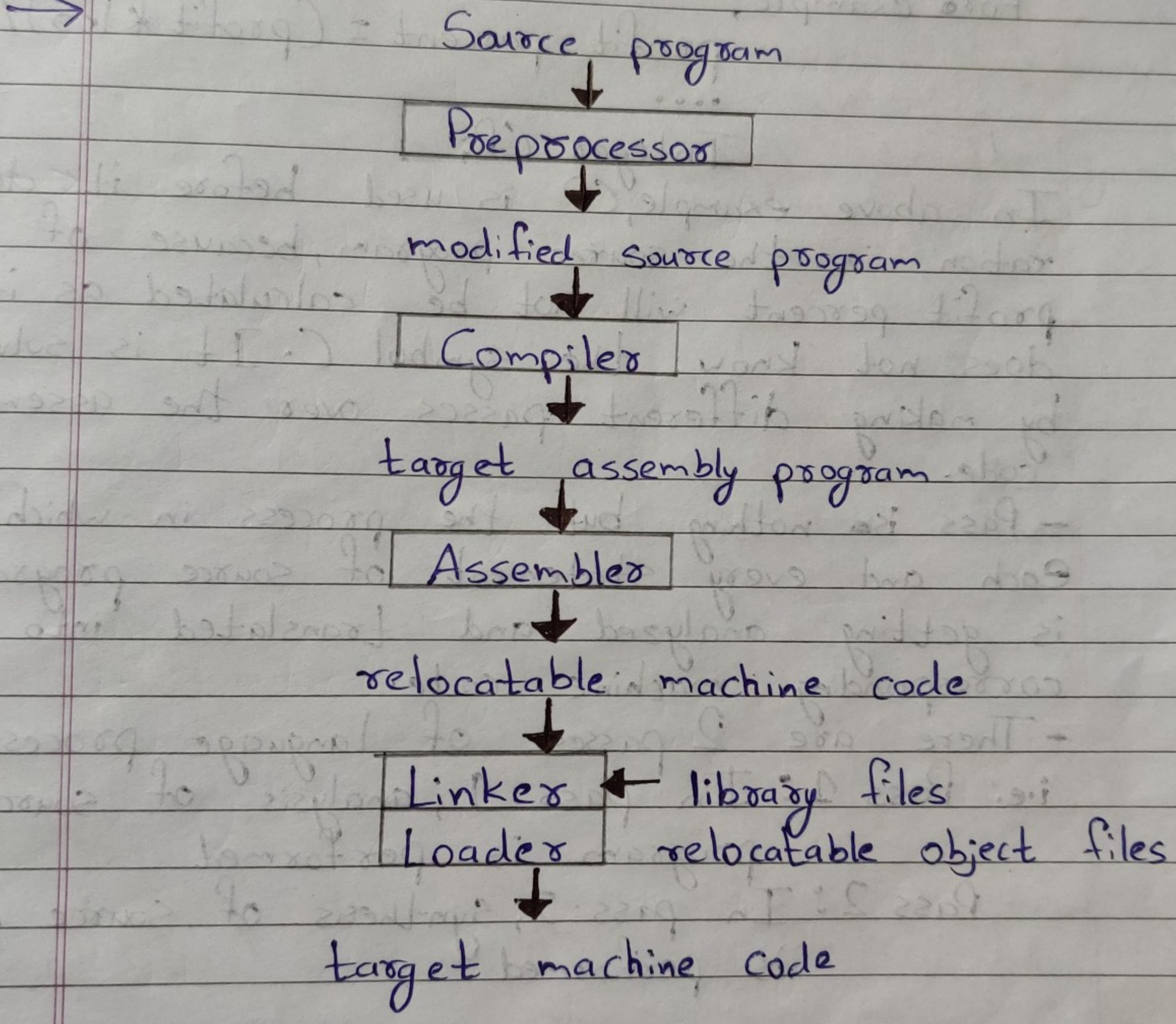
(5) Compiler :-
- Compiler is a system program that translates high level language into low level language.

(6) ~~Inter~~ Interpreter :-
- It is also a translator like a compiler but interpreter takes one line at a time and translate it into object code, then go for next line and interprets. This process continue until entire source code is interpreted.

(7) Operating System :-
- It acts as a interface between computer user and computer hardware.
- Eg :- Windows, IOS, Linux etc.

Q.2) Indicate the order in which following System program are used from developing program upto its execution:- Assembler, Loader, Linker, Macro processor, Compiler, Editor

→



Q.3) Explain forward reference problem and how it is handled in assemble design.

→ - When statement is processing, some of the symbol are used but they are not declared anywhere in the source program.
- Due to this the processor does not find its corresponding memory address and gets failed to generate target program.

Page No.: 4
Date. / /

- If the symbol is used before its declaration then the Synthesis cannot continue its task. Such problems is called as forward reference problem.

For example,

```
profit percent = (profit * 100)/CP
....
Integer CP;
```

In above example, CP is used before its declaration in the source program, because of this profit percent will not be calculated as it does not know the symbol C. It is solved by making different passes over the assembly code.

- Pass is nothing but the process in which each and every statement of source program is getting analyzed and translated into its corresponding machine code.
- There are 2 passes of language processor i.e. Pass 1: In pass 1, analysis of source program is performed

Pass 2: In pass 2, synthesis of source program is performed

Q.4) Explain with neat flowchat and database working of two ~~part~~ pass assembler.

→ In two pass assembler the forward reference problem can be resolved easily because in first pass all the LC processing is done and all the symbols that are used in the source program has the ~~valued~~ valid memory address associated with them.

- So the second pass only use the symbol and generate the target code by using the address found in symbol table.

- Pass 1 :-

1) It seperates the labels mnemonic opcode and operand fields of a statement.
2) Validate the mnemonic opcode through opcode table.
3) Build symbol and literal table.
4) Perform the LC processing.
5) Generate the intermediate code.

- Pass 2 :-

1) Take the address of symbols from symbol table addresses of literals from literal table and address of opcode from opcode table.
2) Synthesize the target program.

Page No.: 6
Date. / /

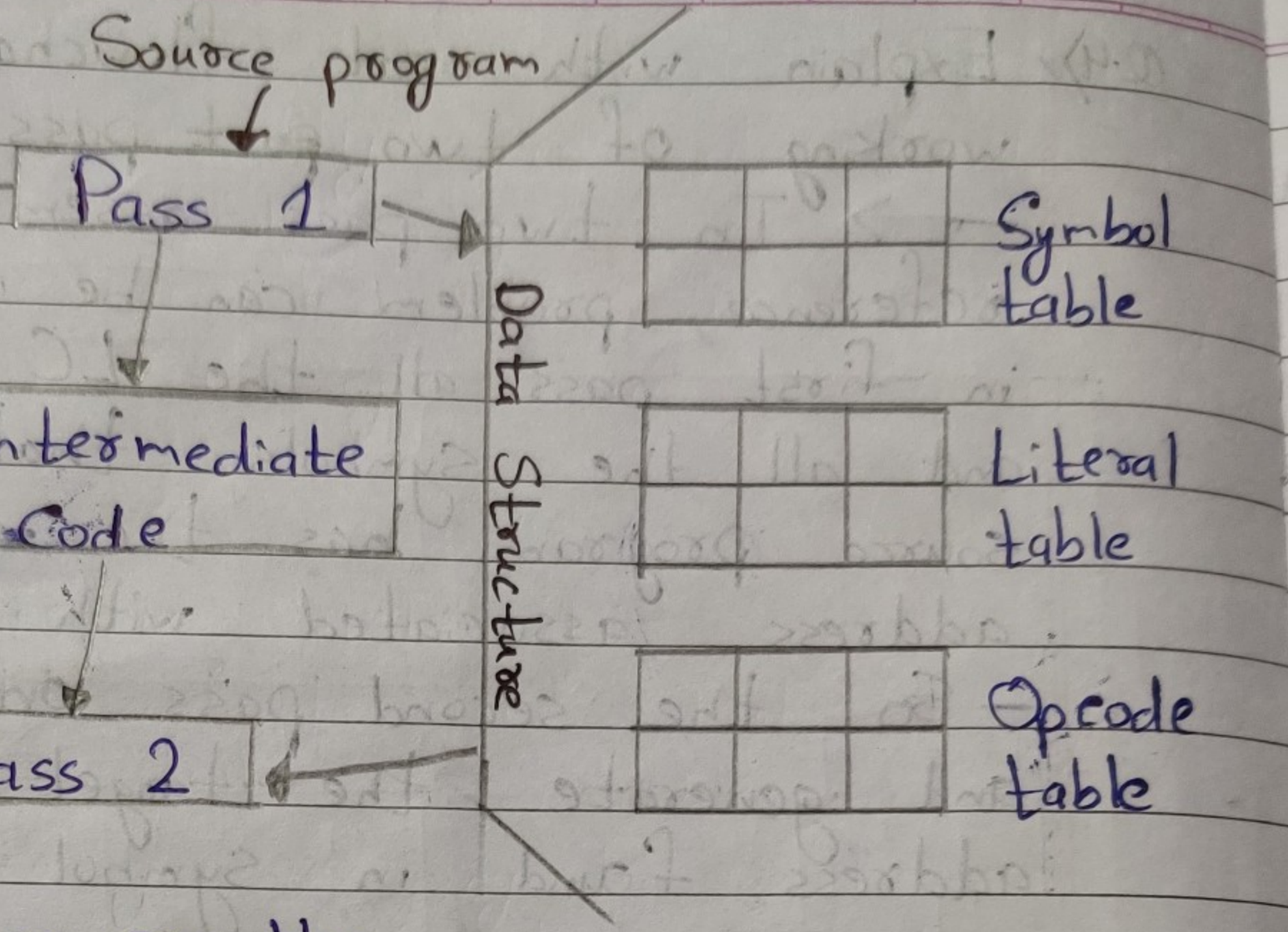


Fig - Two Pass assembler

Page No.: 7
Date. / /

Q.5) Explain Single pass assemblers with neat flowchart and databases.

→

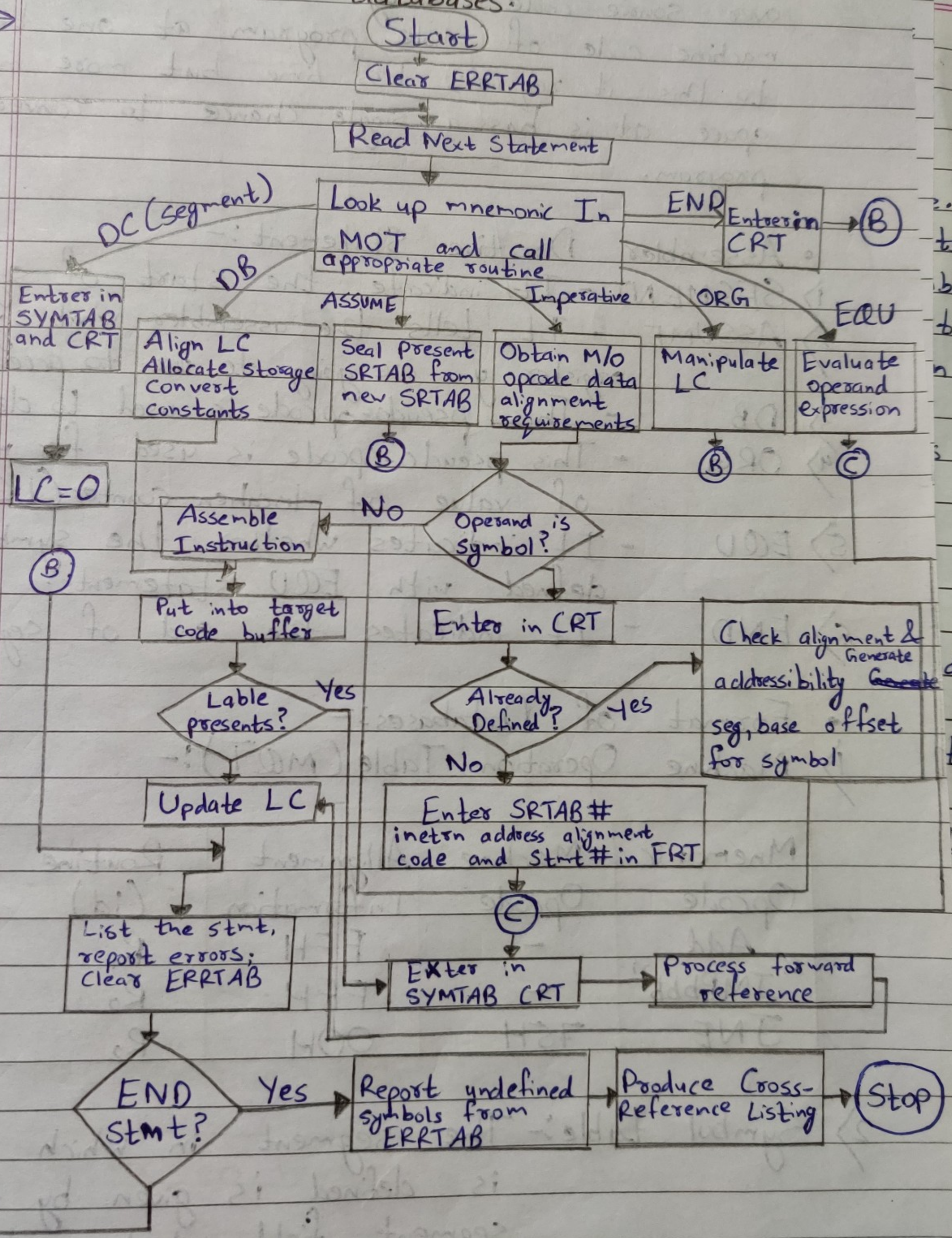


Fig :- Flowchart of the Single Pass Assembler

- In single pass assembler the assembler passes over source code exactly once and converts equivalent machine code of whole program at one time. Due to this it requires less time but more memory space at is has a single chance to convert the program.

- Assembler Directive Statement :-

1) SEGMENT - It indicate the start of segments.
2) ASSUME - It tells the assembler what segment register you are going to use to access a segment.
3) DB - This pseudo-opcode is used to define bytes.
4) ORG - This pseudo-opcode is used for manipulation of value of locahen counter (LC).
5) EQU - It indicates whether the symbol is defined with EQU statements.
6) END - It indicates the end of segment.

- Format of Databases :-

1) Machine Operation Table (MOT) :-

Mnemonic Opcode	Machine Opcode	Allignment Information	Routine (id)
Add	-	FFH	R_1
"JMPbbb"	-	FFH	R_2
JNE	75H	OOH	R_2

2) Symbol table :- The segment in which the symbol is defined is given by the owner segment field which contain the SYMTAB entry.

3) SEGMENT REGISTER TABLE (SRTAB):-
Whenever an assembler encounter, ASSUME_Stmt it stores the previous SRTAB on the stack and new SRTAB is created.

Segment Reg	Segment Name

4) STORED SEGMENT REGISTER TABLE (STSRT):-

	Segment Register	Segment Name	
00	ES		#1 STSRT
01	CS (code segment)		
10	SS (Stack -11-)		
11	DS (Data -11-)		
00	ES		
01	CS		
10	SS		STSRT #2
11	DS		

5) Forward reference table (FRT):-

Pointer to Next Entry (2)	Entry # in STSRT (1)	Instrn Address	Usage Code (1)	Source Stmt (2)

6) Cross Reference Table (CRT):-

Pointer to next entry	Source stmt #

76_Adnan Shaikh

Q.6) Explain two pass macro processor with neat flowchart and database.

→ - It is used for, identifying the macro name and performing expansion.

- Features of Macroprocessor :-

i) Recognized the macro definition
ii) Save macro definition
iii) Recognized the macro call
iv) Perform macro expansion

- Forward reference problem :-

- The assembler specifies that the macro definition should occur anywhere in the program
- So there can be changes of macro call before its definition which gives rise to the forward reference problem and macro

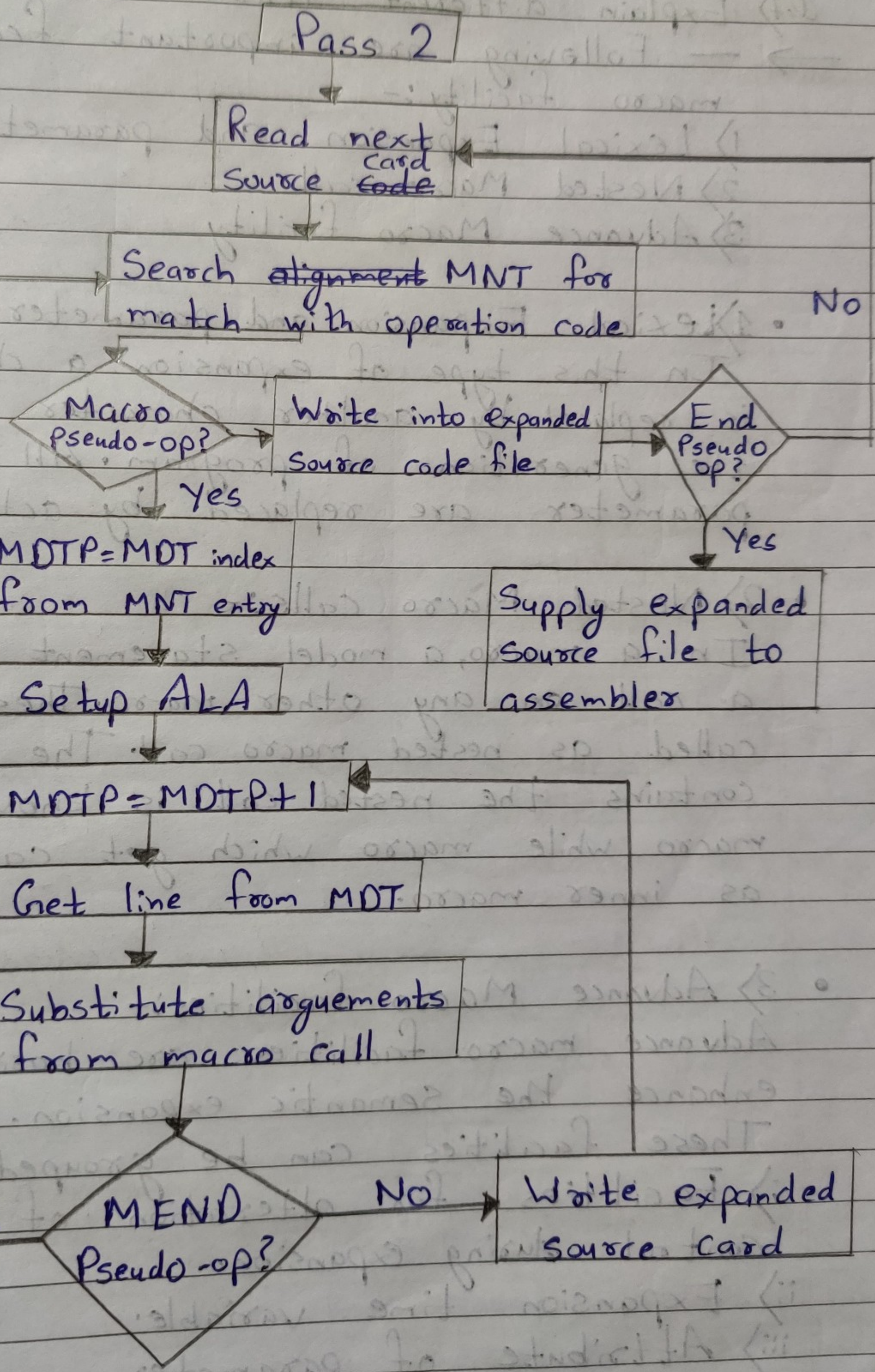
1. Pass 1 :- Recognize macro definition, save macro definition.
2. Pass 2 :- Recognize macro call, perform macro expansion.

- Database required for pass 2 :-

In Pass 2, we perform recognize remicro call and macro expansion.

i) Copy File :- It is a file which contain the output given from pass 1.
ii) MNT :- It is used for recognizing macro call and expansion
iii) MDTP :- It is used to point to index of MDT. The starting index is given by MNT.
iv) ALA :- It is used to replace the index notation by its actual value.
v) ESC :- It is used to contain the expanded macro call which is given to the assembler for further processing.

Pass 2


Read next source card
Search MNT for match with operation code
No
Macro Pseudo-op?
Write into expanded Source code file
End Pseudo op?
Yes
MDTP=MDT index from MNT entry
Yes
Supply expanded Source file to assembler
Setup ALA
MDTP=MDTP+1
Get line from MDT
Substitute arguements from macro call
Yes
MEND Pseudo-op?
No
Write expanded Source card

Q.7) Explain different features of macro facility

→ — Following are important features of macro facility :-

1) Lexical Expansion and parameter substitutioner?
2) Nested Macro call
3) Advance Macro facility

- 1) Lexical Expansion and parameter substitution :-
In this type of expansion a character string is replaced by another character string in the generation of program. All the formal parameter are replaced by actual parameter

- 2) Nested Macro calls :-
In a macro, a model statement can constitute a call on any other macro. These calls are called as nested macro call. The macro which contains the nested call is known as outer macro while macro which get called is known as inner macro.

- 3) Advance Macro facilities :-
Advance macro facilities are basically used to enhance the semantic expansion.
These facilities can be grouped into :-
i) Facilities for alteration of flow of control during expansion.
ii) Expansion time variable.
iii) Attribute of parameter.

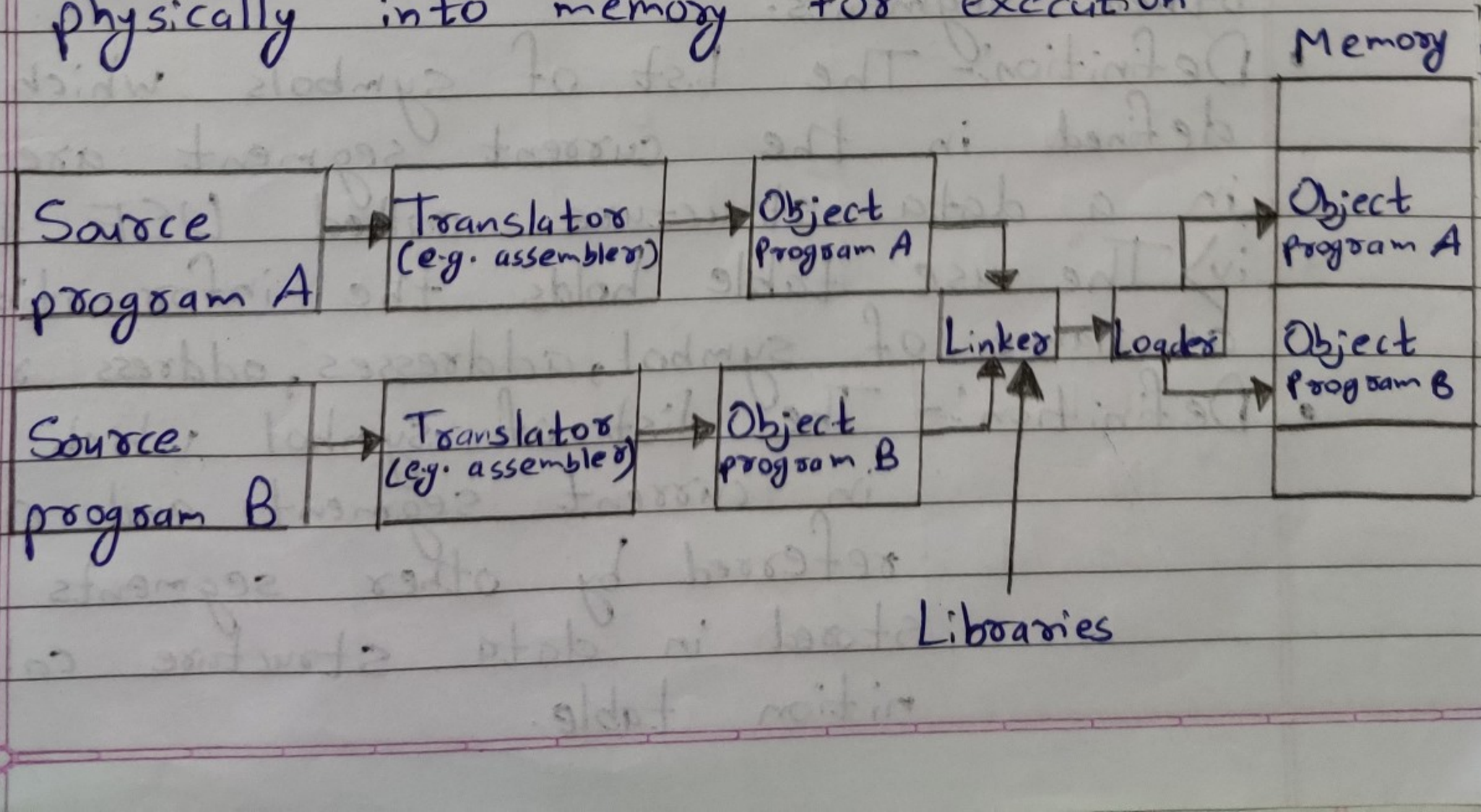
Page No.: 13
Date. / /

Q.8) What are different function of loader? Explain in brief.

→ Function of loader :-

i) Allocation
ii) Linking
iii) Relocation
iv) Loading

- i) Allocation :- Allocates memory space for program
- ii) Linking :- Binding of objects program code with necessary library routine or other object program codes to generate executable program.
- iii) Relocation :- It changes the memory address of address sensitive instruction of program so that it can be loaded at an address different from the location originally specified.
- iv) Loading :- Finally, executable file content (i.e. Machine instruction and data) will be placed physically into memory for execution



76. Ainan Shaikh

Q.9) Explain the working of a direct linking loader with proper example. Clearly show the entries in different database built by direct linking loader.

→ • It is a type of relocatable loader.
• The direct linking loader is most common type of loader.
• The loader cannot have direct access to the source code.
• And to place object code in the memory there are 2 situations:- Either distance of the object code could be absolute or can be relative addresses
• The assembler should give the following information to the loader:-
i) The length of the object code segment.
ii) The list of all symbol which are not defined in the current segment but can be used in current segment.
iii) The list of all symbol which are defined in current segment but can be referred by other segments.
• Definition:- The list of symbols which are not defined in the current segment are stored in a data structure called USE table.
iv) The use table holds the information such as name of symbol, addresses, address relativity.
• Definition:- The list of symbol which are defined in current segment and can be referred by other segments are stored in data structure called definition table.

v) The definition table holds information such as address, symbol.

* Object decks for direct linking loader:-
1) External symbol Dictionary (ESD) record:-
- Entries and Eternals
2) Text (TXT) Records:-
- Control the actual object code tognslated version of source program.
3) Relocation and Linkage Directory (RLD) Record:-
- Contains relocation information like location in the program whose contents depend on the address at which program is placed.
- These information are:-
- Location of each constant that needs to be changed due to relocation.
- By what it has to be changed.
4) END Record:-
- Specifies the starting address for execution

Page No.: 16
Date. / /

* Example:-

```
John    START
        ENTRY RESULT
        EXTERNAL                        SUM
LOAD    1,POINTER               0  LOAD  1,48
LD-ADWR 15,ASUM                 4  LD-ADWR 15,56
BALR    14 15                   8  BALR  14,15
STORE   1 ~~RESULT~~ RESULT     IC 4LT   0,0

TABLE   DC      1,7,9,10,13             28,0001
                                        2A,0007
                                        2C,0009
                                        30,000A
                                        34,000D
POINTER DATA    ADDR(TABLE)             48,0028
RESULT  NUM     0                       52,0000
ASUM    DATA    ADDR(SUM)               56, ???? External
        END
```

- SD - Segment Definition
- Local Definition
- External Reference

* RLD record:-

Symbol	Flag	Length	Relative Location
JOHN	+	4	48
SUM	+	4	56

* ESD record:-

Symbol	Type	Relative Location	Length
JOHN	SD	0	64
Result LD	52		
SUM	ER	-	-

Q.10) Explain different loader scheme

→ Types of loader :-

1) Absolute loader :-
It is a simplaced type of loader scheme. In this scheme the loader simply accepts the machine language code produced by assembler and place it into main memory at the location specified by the assembler. The task of an absolute loader is virtually trivial. Absolute loader is simply to implement but it has several disadvantages.

2) Compile and Go Loader :-
A compile and go loader is one of the loader in which assembler lies in memory and loader itself does the process of assembling and loading machine instructions and data at specified memory location. Instruction are read line by line. Machine code is generated and directly put in the memory at some known address. After completion of assembly process, assembler assign starting address of the program to locate counter.

3) General Loader :-
The source program is converted to object program by some translator. The loader accept these object modules and put machine instruction and data in an executable form at their assigned memory. The loader occupies some portion of main memory. The size of loader is smaller than assembler.

4) Subroutine Linkages / Program linking loader :-
In a given program, it is often needed to perform a particular subtask; Many times on different data values. Such a subtask is called as subroutine.

5) Relocating Loaders (BSS) :-
Loaders that allow for program relocation are called relocating / relative loader i.e. it loads a program in specific area of memory, rebcate it so that it can execute correctly.

6) Direct Linking Loaders :-
It is a type of relocatable loader. The direct linking loader is the most common type of loader. The loader cannot have direct access to the source code and to place the object code in the memory there are two situations :- Either address of subject could be 'absolute' or can be relative. If at all the address is relative then it is the assemblers who informs the loader about the relative adresses.

7) Dynamic Linking Loader :-
Dynamic loader is the loader that actually intersect the "calls" and loads the necessary procedure is called over super-visor or simply flipper. Thus over all scheme is called Dynamic loading or load on call.

76_ Adnan Shaikh

Date
Page 1

Subject: SPCC Sem: VI

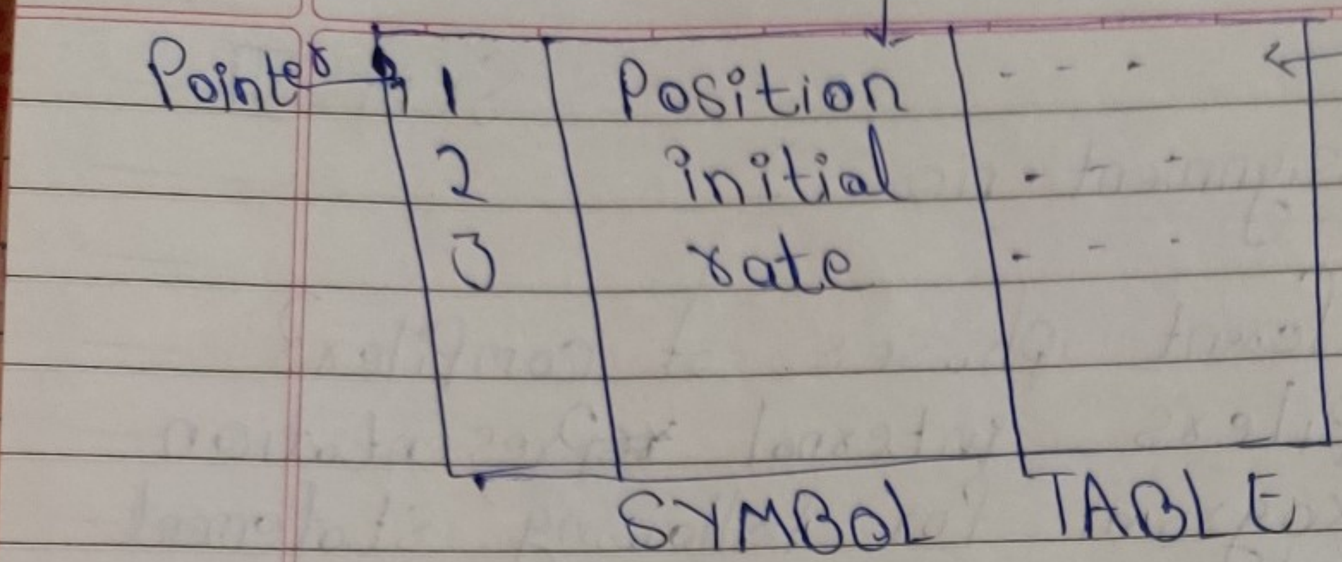
Assignment no. 2

1. What are different phases of compiler? Illustrate compilers internal representation of source program for following statement after each phase position = initial + rate * 60.

① Lexical Analysis: (a) The first phase of a compiler is called lexical analysis or scanning.
(b) The lexical analyzer reads the stream of characters making up source program and groups the character into meaningful sequences called lexemes.
(c) For each lexeme, the lexical analyzer produces as output a token of the form
<token-name, attribute-value>
that it passes on to the subsequent phase, Syntax analysis. In the token,
(d) In the token, the first component token-name is an abstract symbol that is used during Syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.
(e) Information from the symbol-table entry is needed for semantic analysis and code generation.

Lexical analysis of position = initial + rate * 60 is given by:

① - <id, 1> <=> <id, 2> <+> <id, 3> <*> <60>
For some tokens attributes values are implicit.



② Syntax Analysis: (a) The second phase of compiler is syntax analysis or parsing.
(b) The parser uses the first components of the tokens produced by the lexical analyzer to create a tree like intermediate representation that depicts the grammatical structure of the token stream.
(c) The intermediate node represents operation & the children represent the arguments of the operation.
(d) A syntax tree for the token stream (1) is:

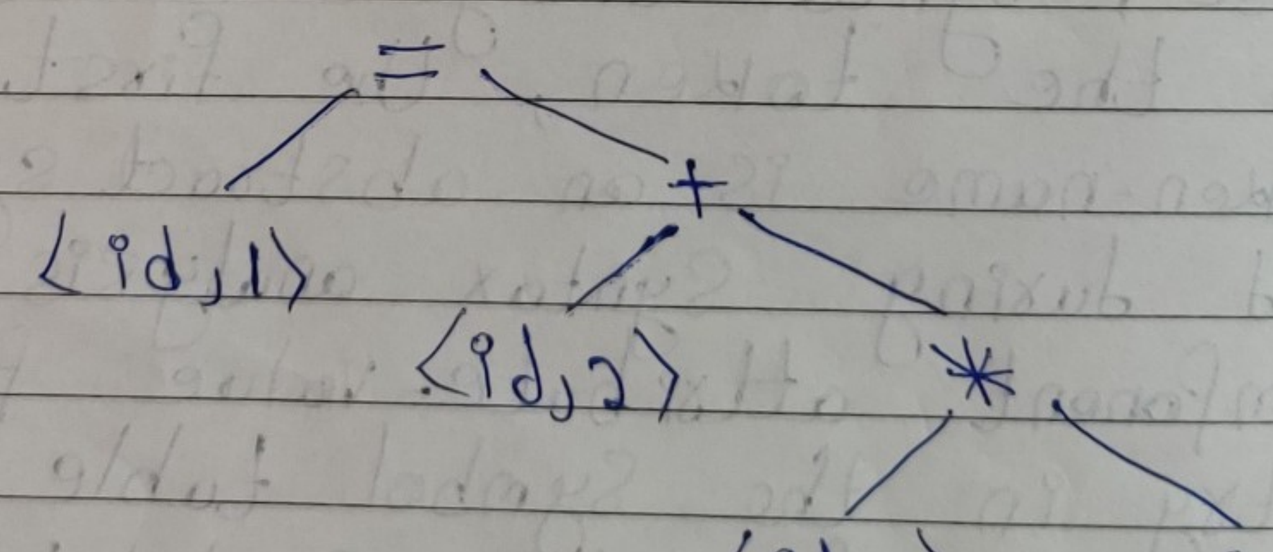


Fig: 1
SYNTAX TREE

(3) Semantic Analysis: (a) The Semantic analyzer uses the syntax tree and the information in the Symbol table to check the source program for Semantic consistency with the language definition
(b) It also gathers & check type information and saves it in either Syntax tree or the Symbol table, for subsequent use during intermediate-code generation.
(c) The language specification may permit some type conversions called coercions.
(d) Semantic Analysis of fig 1.:

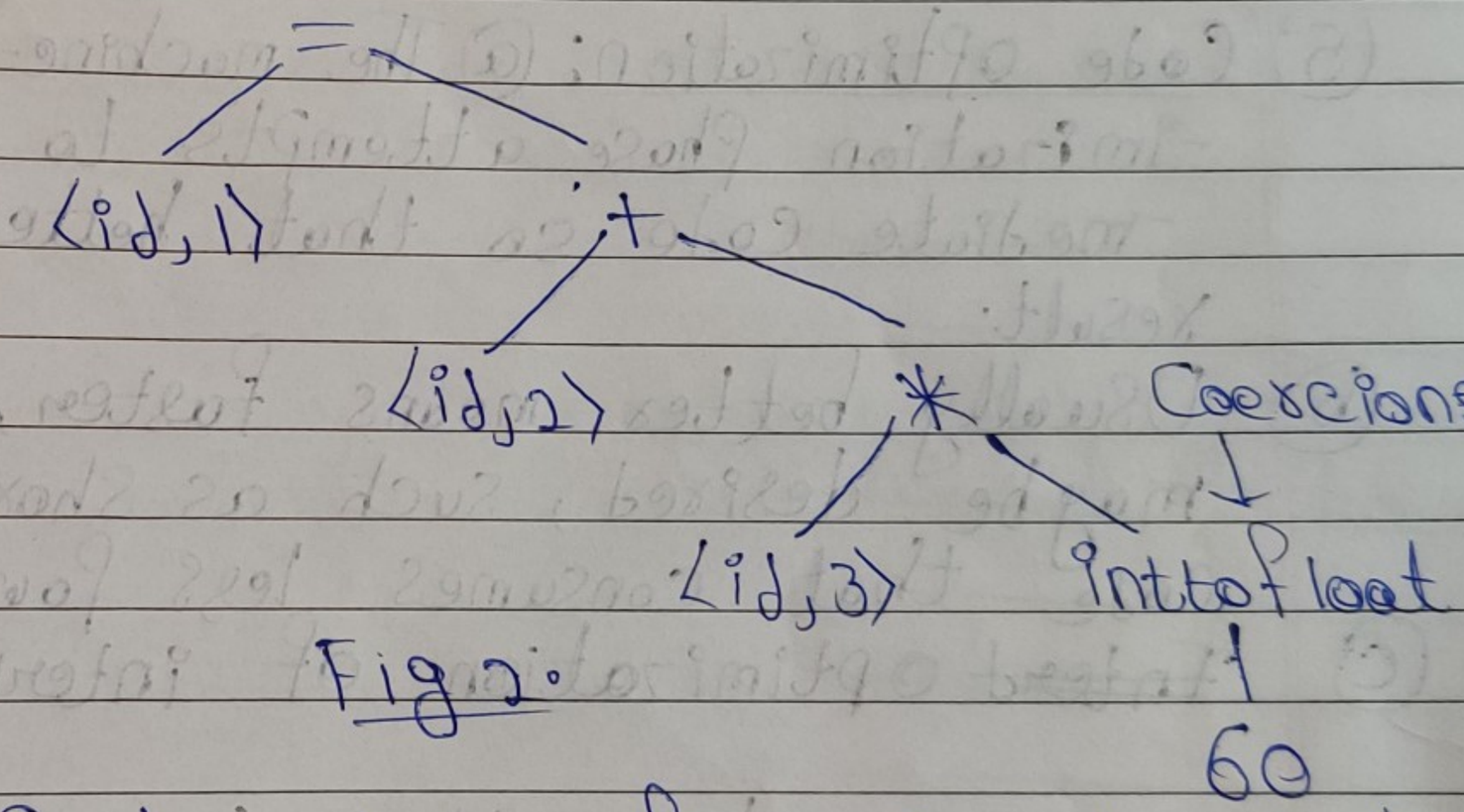


Fig 2.

Syntax tree after Semantic Analysis

(4) Intermediate Code generation: (a) Many compilers generate an explicit low level or machine-like Intermediate representation, which we can think of as a Program for an abstract machine.
(b) This intermediate representation should have two important properties: It should be easy to Produce and it should be easy to translate into the target machine.

(c) We consider a intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

(d) Intermediate code of sytax tree shown in fig. 2:

$$t_1 = \text{inttofloat}(60)$$
$$t_2 = id3 * t_1 \quad \text{--- (2)}$$
$$t_3 = id2 + t_2$$
$$id1 = t_3$$

(5) Code optimization: (a) The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

(b) Usually better means faster, but other objective may be desired, such as shorter code, or target code that consumes less power.

(c) ~~Intert~~ optimization of intermediate code in (1):

$$t_1 = id3 * 60.0 \quad \text{--- (3)}$$
$$id_1 = id_2 + t_1$$

(6) Code Generation: (a) The code generator takes as input an intermediate representation of the source program and maps ~~it~~ it into the target language.

(b) If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.

(c) Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

(c) Intermediate code
(d) Machine code for intermediate code in (3)

```
LDF   R2, id3        (Immediate constant)
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

2.) Eliminate left recursion in the following grammar (Remove Direct and Indirect recursion)

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$

Ans) Left recursion in A : $A \rightarrow Ac$ & $A \xrightarrow{*} Aa$

$\therefore S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$

$\therefore A \rightarrow Ad_1 \mid Ad_2 \mid \cdots \mid B_1 \mid B_2 \mid \cdots$
is equivalent to
$A \rightarrow B_1A' \mid B_2A' \mid \cdots$
$A' \rightarrow d_1A' \mid d_2A' \mid d_3A^3 \mid \cdots \mid \epsilon$

In our case, $d_1 = c$, $d_2 = ad$
$B_1 = bd$, $B_2 = \epsilon$

$\therefore A \rightarrow Ad_1 \mid Ad_2 \mid B_1 \mid B_2$
Can be rewritten as:

Date
Page 6

$A \rightarrow B_1 A' \mid B_2 A'$
$A A' \rightarrow \alpha_1 A' \mid \alpha_2 A'$

Re substituting α_i & B_i
$S \rightarrow Aa \mid b$
$\therefore A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$

3. Consider the following grammar
$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$ to construct the Predictive Parser table.

Ans) ① Eliminating Left Recursion

$E \rightarrow E+T \mid T$
$\therefore E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow T*F \mid F$
$\therefore T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$

Non-recursive Left recursive Grammar:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id.$

(2) First and Follow:

First(E) = { (, id }

First(E') = { +, ε }

First(T) = { (, id }

First(T') = { *, ε }

First(F) = { (, id }

~~Follow(E) = { (, id~~

Follow(E) = {), $ }

Follow(E') = {), $ }

Follow(T) = { +,), $ }

Follow(T') = { +,), $ }

Follow(F) = { *, +,), $ }

Predictive Parser table:

	id	(	)	+	*	$
E	E→TE'	E→TE'				
E'			E'→ε	E'→+TE'		E'→ε
T	T→FT'	T→FT'				
T'			T'→ε	T'→ε	T'→*FT'	T'→ε
F	F→id	F→(E)				

$E \rightarrow TE' \rightarrow First(TE') = First(T) = \{c, id\}$

$E' \rightarrow +TE' \rightarrow First(+TE') = \{+\}$

$E' \rightarrow \epsilon \rightarrow First(\epsilon) = \{\} \Rightarrow Follow(E')$
$Follow(E') = \{), \$\}$

$T' \rightarrow FT' \rightarrow First(F) = \{c, id\}$

$T' \rightarrow *FT' \rightarrow First(*) = \{*\}$

$T' \rightarrow \epsilon \rightarrow First(\epsilon) = \{\} \Rightarrow Follow(T')$

& $Follow(T') = \{+,), \$\}$

$F \rightarrow (E) \rightarrow First(c) = \{c\}$
$F \rightarrow id \rightarrow First(id) = \{id\}$

Date
Page 9

4. Differentiate Top down and Bottom-up parsing techniques. Explain Shift reduce Parser with example.

Ans)

Top-down	Bottom-up
a) Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.	A Bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root.
2) left most Derivation is used	Right most derivation is used.
3) It attempts to find the left most derivat-ions for an input string.	It attempts to reduce the input string to Start symbol of a grammar.
4) It's main decision is to select what production rule to use in order to construct the string.	Its main decision is to select when to use a production rule to reduce the string to get the Starting Symbol.

Shift reduce parser:

Shift-reduce parsing is a form of bottom parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

We use $ to mark the bottom of the stack and also the right end of the input. Conventionally, In 'bottom-up parsing, we show the top of the stack on the right. Initially, the stack is empty, and the string w is on the input, as follows:

STACK	INPUT
$	w$

During a left-to-right scan of the input string the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of the appropriate production.
The parser repeats this cycle until it has detected an error or until the stack contains the start symbol & the input is empty:

STACK	Input
$S	$

E.g. Consider, Grammar:

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

and input: a*b id

Initial Configuration

STACK	Input
$	a*b$

Scan the postfix string from the left,

76_Afnan Shaikh

Date
Page 11

STACK	Input	Action
$	a * b $	Shift
$ a	* b $	Reduce
$ F	* b $	Reduce
$ T	* b $	Shift
$ T *	b $	Shift
$ T * b	$	Reduce
$ T * F	$	Reduce
$ E	$	Halt, Successful Parsing completion

Q6) Construct SLR Parsing table for following grammar.

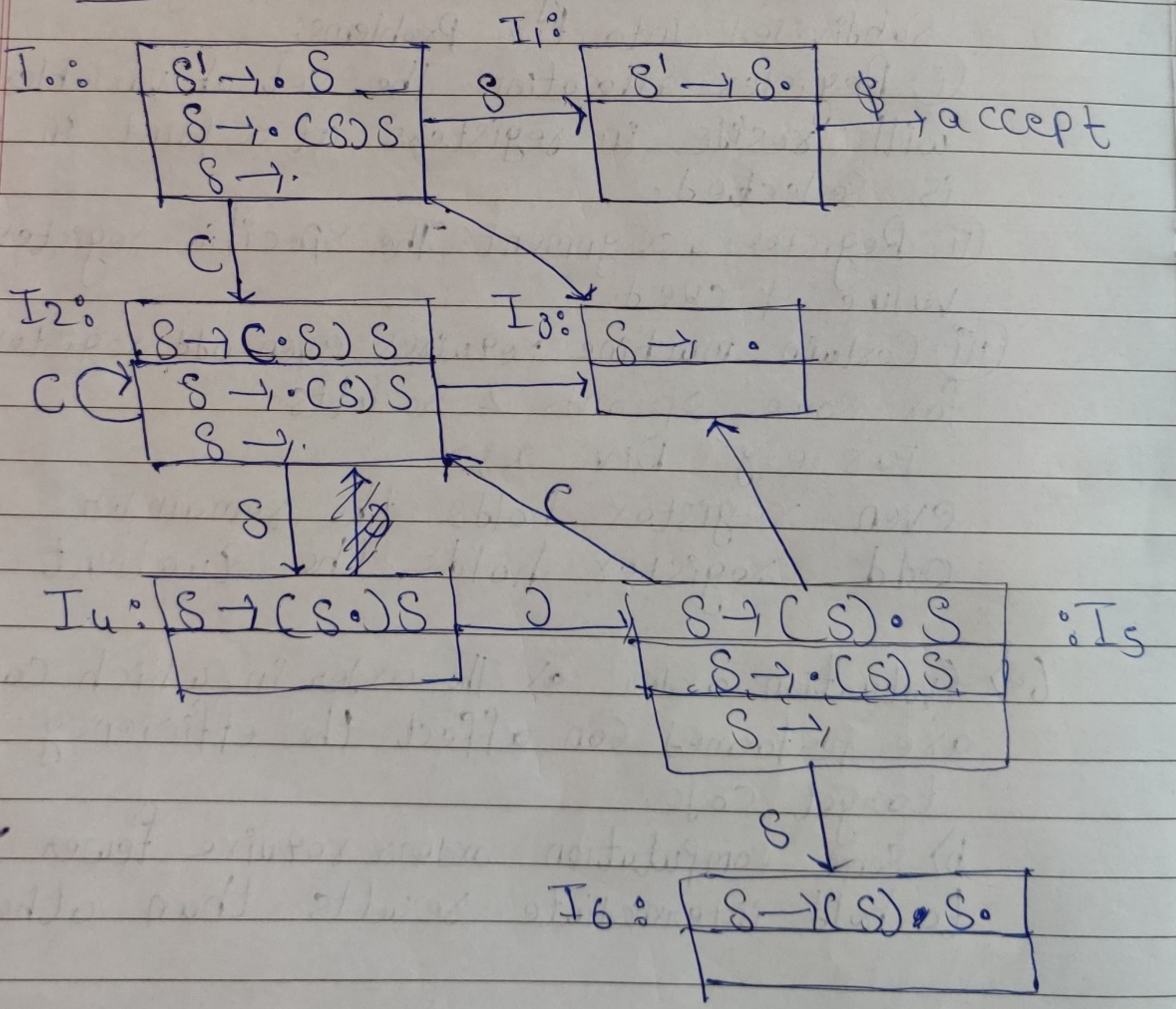
$S \rightarrow (S)S$
$S \rightarrow \varepsilon$

Soln: Augmented:

$S' \rightarrow S$ (1)
$S \rightarrow (S)S$ (2)
$S \rightarrow \varepsilon$ (3)

Follow(S') = {$} ; Follow(S) = {$,)}

Items Sets:



State	Actions				GoTo
	(	)		$	S
0	S2		S3		1
1				acc	
2	S2		S3		4
3		r3		r3	
4		S5			
5	S2	~~S2~~	S3		6
6		r2		r2	

Q5) For the given grammar below construct operator precedence relations matrix, assuming *, + are binary operators and id as terminal symbol & E as non-terminal Symbol.

E → E+E ; E → E*E ; E → id.

Soln: ① Construct operator precedence relation matrix

$\Sigma = \{ id, +, *, \$ \}$

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Apply operator precedence parsing algorithm to obtain skeletal syntax tree for the statement: id + id * id

Input String $ id + id * id $

$\Rightarrow$ $ $\lessdot$ id $\gtrdot$ + $\lessdot$ id $\gtrdot$ * $\lessdot$ id $\gtrdot$ $

$\Rightarrow$ $ E + E * E $

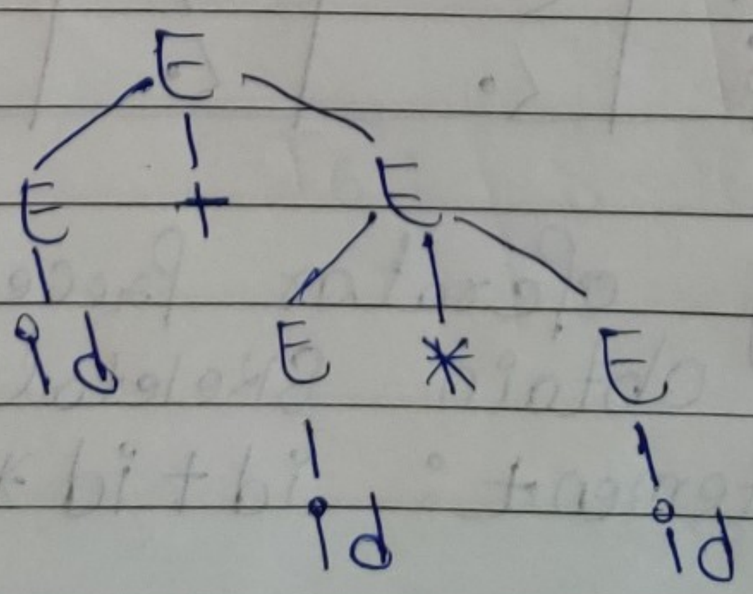
$\Rightarrow$ $ + * $ $\Rightarrow$ $ $\lessdot$ + $\lessdot$ * $\gtrdot$ $

$\Rightarrow$ ~~$ +~~ $ $\lessdot$ + $ $\Rightarrow$ $ $\lessdot$ + $\gtrdot$ $ $\Rightarrow$ $ $ $\Rightarrow$ Accepted

Using Stack: {a $\lessdot$ b, ~~a~~ a $\doteq$ b} $\Rightarrow$ Shift; a $\gtrdot$ b reduce

Stack	Input String	a op b	Action
$	id+id*id$	$ $\lessdot$ id	Shift
$ id	+id*id$	id $\gtrdot$ +	reduce
$ E	+id*id$	$ $\lessdot$ +	Shift
$ E+	id*id$	+ $\lessdot$ id	Shift
$ E+id	*id$	id $\gtrdot$ *	Reduce
$ E+E	*id$	+ $\lessdot$ *	Shift
$ E+E*	id$	* $\lessdot$ id	Shift
$ E+E*id	$	id $\gtrdot$ $	Reduce
$ E+E*E	$	* $\gtrdot$ $	Reduce
$ E+E	$	+ $\gtrdot$ $	Reduce
$ E	$	$ $\doteq$ $	Accept

Tree:



Q.6) Construct

Q.7) Write a Short note on Syntax Directed definitions.

Ans) (1) A Syntax-directed definition (SDD) is a Context-free grammar together with attributes and rules.

(2) Attributes are associated with grammar Symbols & rules are associated with the productions. If X is a Symbol and a is one of its attributes, then we write X.a to denote the value of 'a' at a particular parse-tree node labeled X.

(3) Attributes may be of any Kind: numbers, types, table, references, or Strings, for instance.

(1) There are two Kinds of ~~non-term~~ attributes for nonterminals:

(a) A Synthesized attribute for a nonterminal A at a parse-tree Node N is defined by Semantic rule associated with the production at N.

(6) An inherited attribute for a non-terminal B at a parse-tree Node N is defined by a semantic rule associated with the production at the parent of N.

(5) An SDD that involves only synthesized attributes is called S-attributed

(6) In an S-attributed SDD, value of head is computed with the production body. i.e parent value is computed using children value.

(7) The "L-attributed translations" (Left-to-Right) which encompass virtually all translations that can be performed during parsing is mostly applied

(8) e.g of infix-to-postfix translator:

Production	Semantic Rule
$E \rightarrow E_1 + T$	E.code = E_1.code ‖ T.code ‖ '+'

code is an attribute & ‖ used for concatenation. E & E_1 are same non-terminal but subscript is used for distinguish between instances.

(9) Rules can be written as follow:

$E \rightarrow E_1 + T$ { print '+' }

The code fragment appears between '{' '}' which is called as Semantic action

(10) The position of action in a production body determines the order in which the action is executed.

Date
Page 13

8.) Explain various form of intermediate code used by compiler

Ans) The various intermediate code forms are
a) Polish notation
b) Syntax tree
c) Three address codes: Triples and Quadruples
d) ~~Quadruples~~

a) Polish notation: The postfix notation for an Expression E can be defined inductively as follows:
1. If E is a variable or constant, then the postfix notation for E is E itself.
2. If E is an expression of the form E_1 op E_2, where op is any binary operator, then the postfix notation for E is $E_1' E_2'$ op, where E_1' and E_2' are the postfix notation for E_1 and E_2.
3. If E is a parenthesized expression of the form (E_1) then the postfix notation for E is the same as the postfix notation for E_1.

e.g.: $(9-5)+2$ $\xrightarrow{\text{Postfix}}$ $E=(E_1)+E_2$

$(E_1)=E_1=9-5=95-$
$E_2=2$
$\therefore E=(E_1)+E_2=(E_1)E_2+=95-2+$

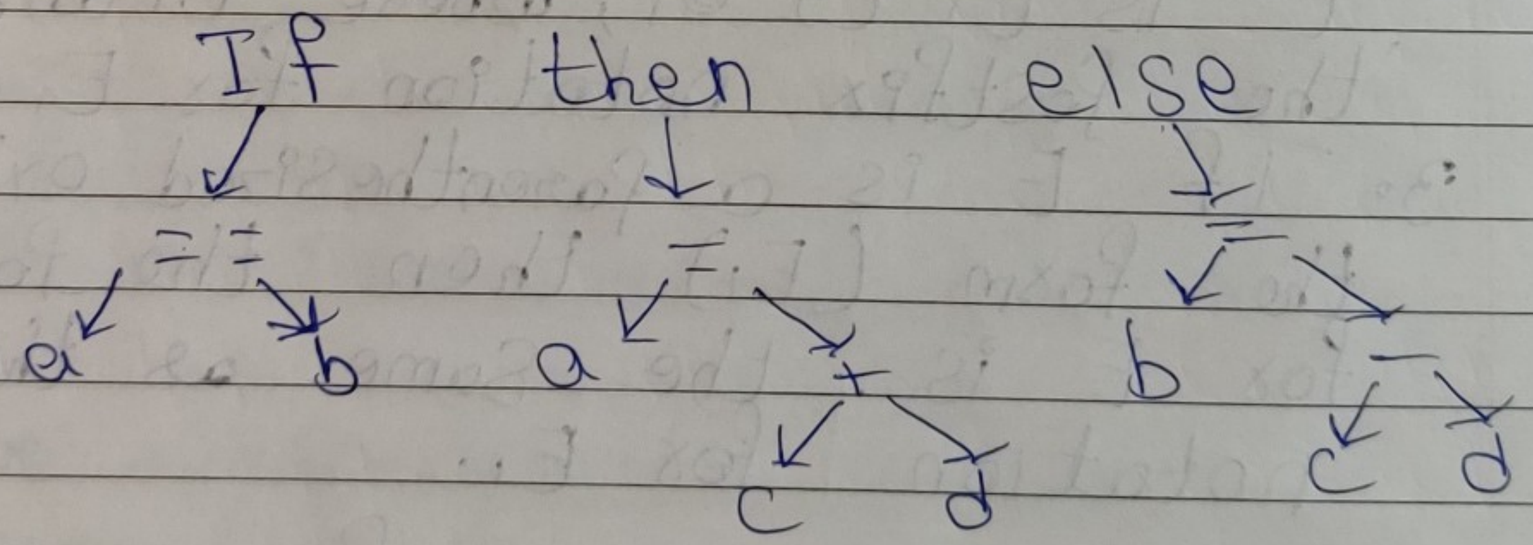
To evaluate postfix expression we repeatedly scan the postfix string from the left,

until we find an operator. Then, look to its left we find operands. Then, look to the left for the proper number of operands, & replace them by the result. Then repeat the process, continuing to the right & searching for another operator.

e.g: 95-2+ => 9-5 => 42+ => 4+2 => 6

b) Syntax tree: The parse tree itself is a useful intermediate language representation for a source program, especially in optimizing compilers where the intermediate code needs to extensively restructure.
We use an abstract Syntax tree, a tree in which each leaf represents an operand & each interior node an operator.

e.g: If a==b then a=c+d else b=c-d



c) Three-Address Code: ① In three address code, there is atmost one operator on the right side of an instruction. e.g: a = b+c.
② No more than 2 variables appear on right side & 1 on left of assignment operator.
③ It eliminates the ambiguity by evaluating only one operation at a time.

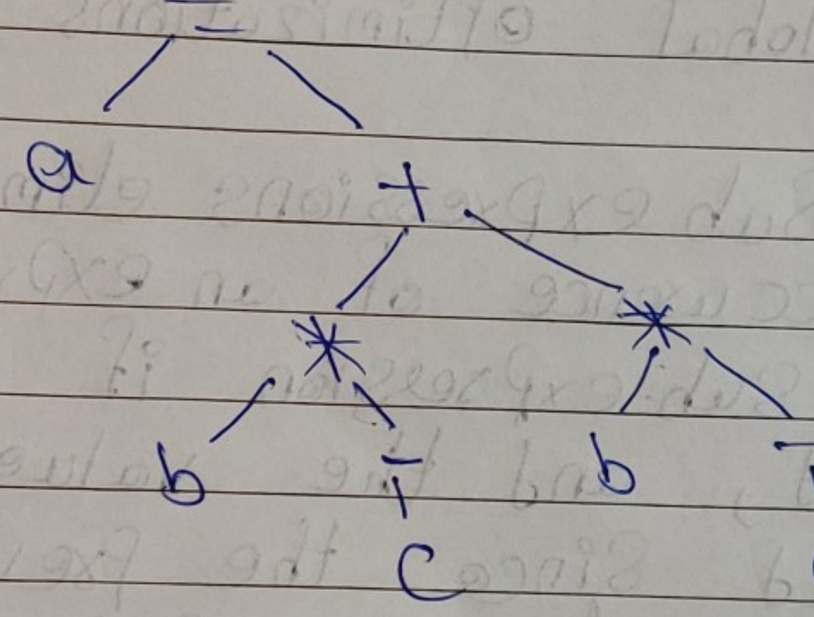
76_Adnan Shaikh

Date
Page 15

(i) Quadruples: A quadruple has four fields: op, arg1, arg2 & result
The evaluation is result of arg1 & arg2 of op is stored in result.

(ii) Triples: A triple has three fields: op, arg1 & arg2.
Triples store result of arg1 op arg2 in its position rather than explicit temporary name.

e.g:



=> a = b * -c + b * -c

Triple:

address	Instruction		op	arg1	arg2
35	(0)	0	minus	c	
36	(1)	1	*	b	(0)
37	(2)	2	minus	c	
38	(3)	3	*	b	(2)
39	(4)	4	+	(1)	(3)
40	(5)	5	=	a	(4)

Quadruple:

	op	arg1	arg2	result		op	arg1	arg2	result
0	minus	c		t1	4	+	t2	t4	t5
1	*	b	t1	t2	5	=	t5		a
2	min	c		t3					
3	4 *	b	t3	t4					

9.) Explain the principal sources of optimization

Ans) ① There are number of ways in which a compiler can improve a program without changing the function it computes.
Function preserving transformations example:
ⓐ Common Sub expressions elimination.
ⓑ Copy propagation;
ⓒ Dead-code elimination.
ⓓ Constant folding
The other transformations come up primarily when global optimizations are performed.

② Common Sub expressions elimination:
ⓐ An occurence of an expression E is called a common sub-expression if E was previously computed, and the values in E haven't changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.
ⓑ For example:

$t_1 := 4*i$
$t_2 := a[t_1]$
$t_3 := 4*i$ ← common
$t_4 := n$
$t_5 := b[t_3] + t_4$

optimized

$t_1 := 4*i$
$t_2 := a[t_1]$
$t_3 := n$
$t_4 := b[t_1] + t_3$

③ Copy propagation: ⓐ Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but it gives an opportunity to eliminate x.
for e.g: $x = pi;$ $A = x * r * r;$

optimized : A:= Pi*r*r
↑ x eliminated.

(4) Dead-code eliminations: (a) A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
(b) example:

```
int compute_dead(int c){
int a:= c * 2;
int b:= a*c; → dead code
return a;
}
```

in above code b is assigned but never used it can eliminated from there.

(5) Constant folding: (a) Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
(b) e.g: a:= 3.14157/2 optimized→ a:= 1.570

(6) Loop optimizations: (a) In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of expressions in an inner loop is decreased, even if we increase the amount of code outside the loop.
(b) Three techniques are important for loop optimization.
(i) Code motion
(ii) Induction-variable elimination.
(iii) Reduction in strength.

(7) Code Motion: (a) An important modification that decreases the amount of code in a loop is code motion.
(b) This transformation takes an expression that yields the same result independent of the number of times a loop is executed (loop invariant) and places the expression in the outter loop if exist or "before the loop" if entry point exist.

e.g:

```
while(i<=limit-2){          optimized →   t=limit-2
#not Changing →  Stmt;                    while(i<=t){
#limit          }                           Stmt;
                                          }
```

limit-2 will remain same through out the loop instead of calculating it each time we move it out & calculated once.

(8) Induction variables: (a) Loops are usually processed inside out. ~~for example:~~ j = k;

(b) for e.g: j = k ; v = ~~so do~~ some value

```
while
do{
    j = j-1
    t1 = 4*j
    t2 = a[t1]
}while t2 > v
```

In above example values of j & t1 remain in lock-step; every time j = j-1 ⟹ t1 decreases by 4. i.e $t_1 = 4j_1 = 4(j-1) = 4j-4$
j_1 only implies previous value replaced with new one. Such identifiers are called induction.

(c) When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction variable elimination.

Foxe.g $j = k$; $t_1 = 4 * j$; $v =$ Some value

```
do {
    j = j - 1
    t1 = t1 - 4
    t2 = a[t1]
} while t2 > v
```

Here, we get ride of multiplication by substituting subtraction which is much cheaper & initialize t_1 at entry point of loop because it had no origin before assigning it value.

(9) Reduction in Strength (a) Reduction in Strength replaces expensive operations by equivalent cheaper ones on the target machine.

(b) for e.g: $x * x$ is cheaper to implement than x^2 which required a call to exponentiation routine.

$\therefore$ x^2 Optimized → $x * x$

4.) (a) Explain the various issues in the design of code generation.

Ans) (1) Input to code generator: (a) The input to the code generation consists of the intermediate representation of the source produced by front end, together with the information in the symbol table to determine run-time address of the data objects denoted by the names in the intermediate representation.

(b) Intermediate representation can be:
(i) Postfix Notation (ii) Three address code
(iii) Virtual Machine (Stack machine code)
(iv) Syntax tree.

(2) Target Program: (a) The output of the code generator is the target program. The output may be:
(i) Absolute machine language: It can be placed in a fixed memory location and can be executed immediately.
(ii) Relocatable machine language: It allows subprograms to be compiled separately.
(iii) Assembly language: Code generation is made easier.

(3) Memory management: (a) Names in the source program are mapped to addresses of data objects in run-time memory by front-end & code generator.
(b) It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
(c) Labels in three-address statements have to be converted to addresses for instructions.

Date
Page 21

4.) Instruction Selection: a) The instructions of target machine should be complete & uniform.
b) Instruction speeds & machine idioms are important factors when efficiency of target program is considered.
c) The quality of the generated code is determined by its speed and size.

5.) Register allocation: a) Instructions involving register operands are shorter & faster than those involving operands in memory. The use of registers ~~~~ is subdivided into two problems:
(i) Register allocation: The set of variables that will reside in registers at a point in the program is selected.
(ii) Register assignment: The specific register that a value picked.
(iii) Certain machine requires even-odd register pairs for some operands & results.
For e.g: Div x, y
even register holds the remainder.
odd register holds the quotient.

6.) Evaluation order: a) The order in which computations are performed can affect the efficiency of the target code.
b) Some computation orders require fewer registers to hold intermediate results than others.