EXPERIMENT NO 1

Date of performance :

Date of submission :

Aim: To implement a two pass assembler.

Software Used: c,c++,java language

Theory:

If the source program is in assembly language of a machine, the object is in the machine language of the same machine and translator is executing on the same machine. The assemblere generates 2 files. The first file called the object file and is given .obj extension. The object file contains the binary code for instruction and information about object instruction. The second file generated by assembler is called list file with .lst extension. The list file contains assembly language statements, binary code for each instruction and offset for each instruction.

There are two phases of assembler design:

1. Analysis phase
2. Synthesis phase

The task performed by the two phases are:-

Analysis phase:

1. isolated label, mnemonic opcode,operands and comment of statement.
2. check validity of mnemonic opcode by consulting MOT.
3. check the number of operands required for an instruction by consulting MOT.
4. process the labels and symbols appropriately and fill ST.
5. update LC appropriately by consulting MOT for the length of insturction.
6. ignore comments.
7. takeb proper actions for pseudo opcodes by consulting POT.

Synthesis phase:

1. obtain machine opcode consulting to MOT corresponding to mnemonic.
2. Fill in address for symbols or labels at appropriate places by consulting ST.
3. write the above information in output object file.

1) ALGORITHM:

Pass1: Purpose- define symbols and literals

1. Determine length of machine instructions (MOTGET1)
2. Keep track of Location Counter (LC)
3. Remember values of symbols until Pass2 (STSTO)
4. Process some pseudo ops(POTGET1)
5. Remember literals (LITSTO)

Pass2: Purpose- generate object program

1. Look up value of symbols(STGET)
2. Generate instructions(MOTGET2)
3. Generate Data
4. Process pseudo ops(POTGET2)

2) PROBLEM SPECIFY

Source program			FIRST PASS Relative Mnemonic Address instruction	SECOND PASS Relative Mnemonic Address instruction

JOHN	START	0						
	USING	*,15						
	L	1,FIVE	0	L	1,-(0,15)	0	L	1,16(0,15)
	A	1,FOUR	4	A	1,-(0,15)	4	A	1,12(0,15)
	ST	1,TEMP	8	ST	1,-(0,15)	8	ST	1,20(0,15)
FOUR	DC	F'4'	12	4		12	4	
FIVE	DC	F'5'	16	5		16	5	
TEMP	DS	1F	20			20		
	END							

3) FORMAT OF DATABASE

1) PSEUDO OPCODE TABLE(POT) FOR PASS1 AND PASS2

Mnemonic Op-code (4 byte) (character)	Binary Op-code (1-byte) (Hexadecimal)	Instruction Length (2-bite) (binary)	Instruction Format (3 bit) (binary)	Not Use in this design (3 bit)
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
"ALbb"	5E	10	001	
"ALRb"	1E	01	000	
"ARbb"	1A	01	000	
	….	…..	…….	
"MVCb"	D2	11	100	

2) MACHINE-OP TABLE(MOT) FOR PASS1 AND PASS2

Pseudo-op (5-byte) (character)	Address of routine to process pseudo-op (3-byte=24bit addrss
"DROPb" "ENDbb" "EQUbb" "START" USING"	P1DROP P1END P1EQU P1START P1USING

3) SYMBOL TABLE(ST) FOR PASS1 AND PASS2

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

4) SAME FIELD USE FOR TABLE FOR LITERAL (LT)

5) BASE TABLE(BT) F0R PASS2

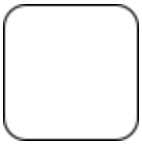
Availability indicator (1-byte) (character)	Designated relative address contents of base register(3-byte=24bit address) (hexadecimal)
"N" "N" . . "N"	- - -

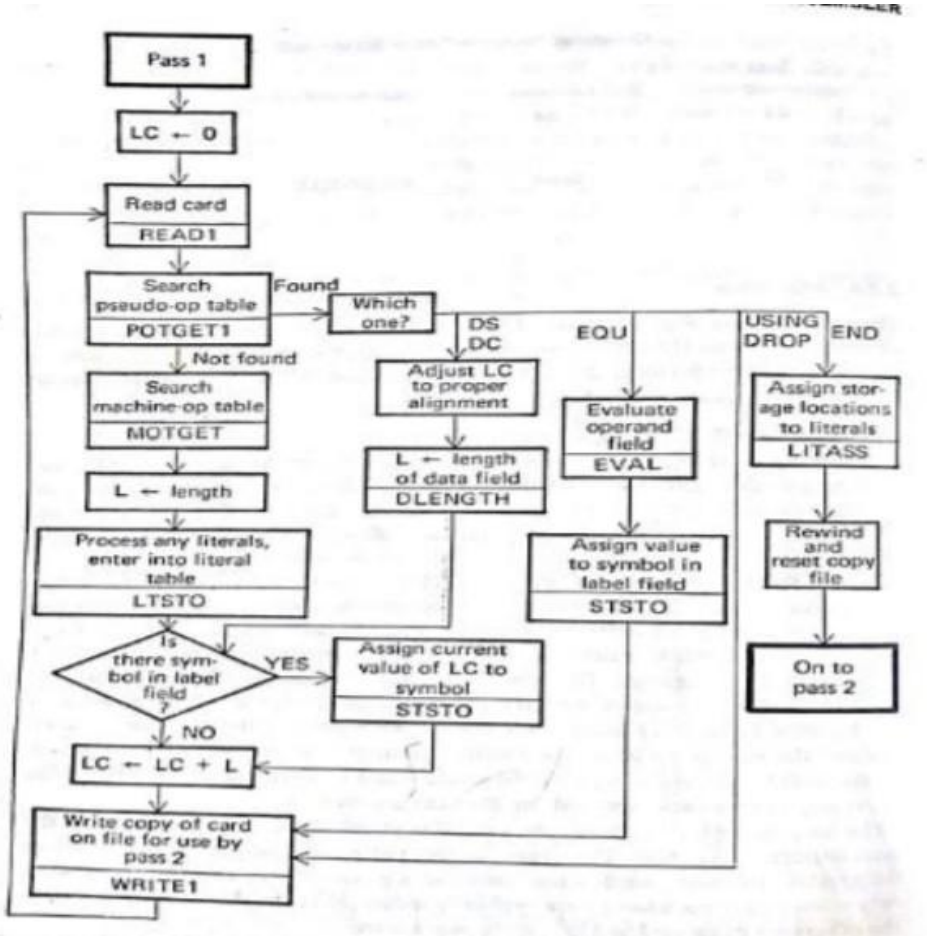
"Y"	00 00 00

Y=register specified in USING pseudo-op
N=register never specified in USING pseudo-op/subsequently made unavailable by the DROP pseudo-op

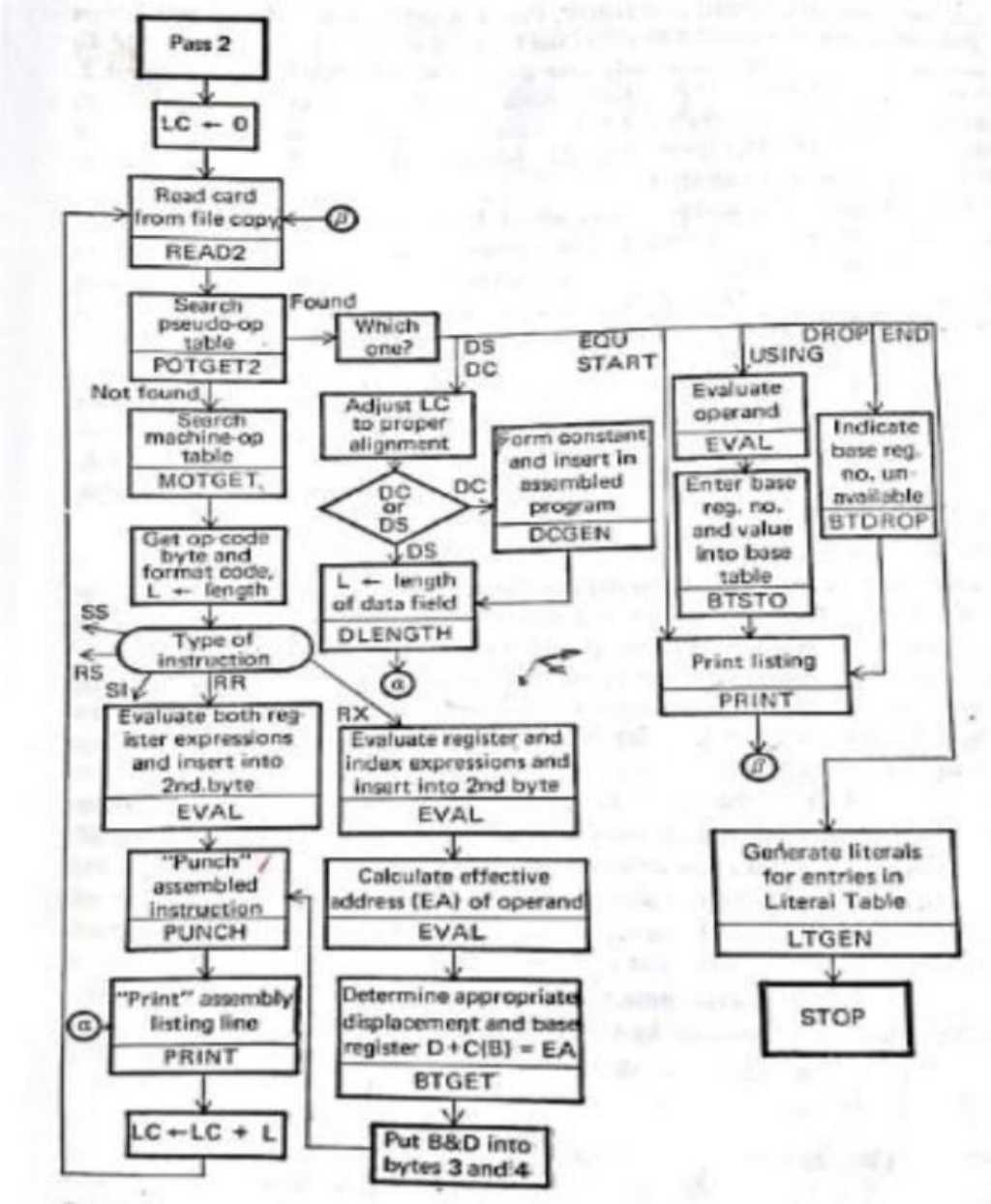
Flowchart:

Pass1:





Pass2:


Pass 2
LC ← 0
Read card from file copy
READ2
Search pseudo-op table
POTGET2
Found
Which one?
DS
DC
EQU
START
USING
DROP
END
Not found
Search machine-op table
MOTGET
Adjust LC to proper alignment
Form constant and insert in assembled program
DCGEN
DC or DS
DC
DS
Get op-code byte and format code, L ← length
L ← length of data field
DLENGTH
Evaluate operand
EVAL
Enter base reg. no. and value into base table
BTSTO
Indicate base reg. no. unavailable
BTDROP
SS
Type of instruction
RS
SI
RR
RX
Print listing
PRINT
Evaluate both register expressions and insert into 2nd byte
EVAL
Evaluate register and index expressions and insert into 2nd byte
EVAL
"Punch" assembled instruction
PUNCH
Calculate effective address (EA) of operand
EVAL
Generate literals for entries in Literal Table
LTGEN
"Print" assembly listing line
PRINT
Determine appropriate displacement and base register D+C(B) = EA
BTGET
STOP
LC ← LC + L
Put B&D into bytes 3 and 4

Program for TWO PASS ASSEMBLER

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{
char *code[9][4]={
{"PRG1","START","",""},
{"","USING","*","15"},
{"","L","",""},
{"","A","",""},
{"","ST","",""},
{"FOUR","DC","F",""},
{"FIVE","DC","F",""},
{"TEMP","DS","F",""},
{"","END","",""}
};
char av[2],avail[15]={'N','N','N','N','N','N','N','N','N','N','N','N','N','N','N'};
int i,j,k,count[3],lc[9]={0,0,0,0,0,0,0,0,0},loc=0;
clrscr();
printf("-----------------------------------\n");
printf("LABLE\t\t\OPCODE\n");
printf("-----------------------------------\n\n");
for(i=0;i<=8;i++)
{
for(j=0;j<=3;j++)
{
printf("%s\t\t",code[i][j]);
}
printf("\n");
}
getch();
printf("---------------------------");
printf("\n VALUES FOR LC:\n\n");
for(j=0;j<=8;j++)
{
if((strcmp(code[j][1],"START")!=0)&&(strcmp(code[j][1],"USING")!=0)&&(strcmp(code[j][1],"L")!=0)
)
lc[j]=lc[j-1]+4;
printf("%d\t",lc[j]);
}

printf("\n\nSYMBOL  TABLE:\n------------------------------ \n");
printf("SYMBOL\t\tVALUE\t\tLENGTH\t\tR/A");
printf("\n---------------------------\n");

for(i=0;i<9;i++)
{
if(strcmp(code[i][1],"START")==0)
{
printf("%s\t\t%d\t\t%d\t\t%c\n",code[i][0],loc,4,'R');
}
else if(strcmp(code[i][0],"")!=0)
```

```
{
printf("%s\t\t%d\t\t%d\t\t%c\n",code[i][0],loc,4,'R');
loc=loc+4;
}
else if(strcmp(code[i][1],"USING")==0){}
else
{loc=loc+4;}
}
printf("----------------------------------------");
printf("\n\nBASE TABLE:\n---------- \n");
printf("REG NO\t\tAVAILIBILITY\tCONTENTS OF BASE TABLE");
printf("\n----------\n");
for(j=0;j<=8;j++)
{
if(strcmp(code[j][1],"USING")!=0)
{}
else
{
strcpy(av,code[j][3]);
}
}
count[0]=(int)av[0]-48;
count[1]=(int)av[1]-48;
count[2]=count[0]*10+count[1];
avail[count[2]-1]='Y';

for(k=0;k<16;k++)
{
printf("%d\t\t%c\n",k,avail[k-1]);
}
printf("----------\n");
printf("Continue..??\n\n");
getch();
printf("PASS2 TABLE:\n\n");
printf("LABLE\t\tOP1\t\tLC\t\t");
printf("\n----------\n");
loc=0;
for(i=0;i<=8;i++)
{
for(j=0;j<=3;j++)
{
printf("%s\t\t",code[i][j]);
}
j=0;
printf("\n");
}
printf("----------");
getch();
}
```

OUTPUT ::-

LABLE TABLE::-

```
---------------------------------
LABLE          OPCODE
-----------------------------------

PRG1           START
               USING         *             15
               L
               A
               ST
FOUR           DC            F
FIVE           DC            F
TEMP           DS            F
               END
---------------------------
 VALUES FOR LC:

0      0      0      4      8      12      16      20      24

SYMBOL TABLE:
---------------------------------
SYMBOL         VALUE     LENGTH     R/A
-------------------
PRG1           0             4             R
FOUR           12            4             R
FIVE           16            4             R
TEMP           20            4             R
--------------------------

BASE TABLE:
------------------------------
REG NO         AVAILIBILITY    CONTENTS OF BASE TABLE
------------------------------
0
1              N
2              N
3              N
4              N
5              N
6              N
7              N
8              N
9              N
10             N
11             N
12             N
13             N
14             N
15             Y
---------------------------------------
```

Continue..??

PASS2 TABLE:

```
LABLE        OP1          LC
-----------------------
PRG1         START
             USING        *          15
             L
             A
             ST
FOUR          DC          F
FIVE          DC          F
TEMP          DS          F
             END
-----------------------
```

Conclusion: Hence implemented a two pass assembler.

SIGN AND REMARK

DATE

R1	R2	R4	R5	Total (15 Marks)	Signature

EXPERIMENT NO 2

Date of performance :

Date of submission :

Aim: to implement a two pass macro preprocessor.

Software Used: c, c++, java language

Theory:

Macro is a set of instructions, it contains: Macro definition, macro call. Macro definition contains 3 parts:

1. Macro body
2. header of the macro.
3. footer

Macro preprocessor, like an assembler, scans and processes lines of a text. In assembly language, lines are interrelated by addressing: a line can refer to another by its address or name, which must be available to the assembler. Moreover, the address assigned to each line depends upon proceeding lines, upon their address, and possibly upon their contents as well. If we consider macro definitions to constitute integral entities, we may say that the lines of our macro language are not so closley interrelated. Macro definitions refer to nothing outside themselves, and macro calls refer only to macro definitions.

Tasks performed by a macro preprocessor:

1. Recognize macro definitions
2. Save the definitions
3. Recognize macro calls
4. Expand calls and substitute arguments

Specification of database

Pass 1 database

1) the i/p macro source desk
2) the o/p macro source desk copy for use by pass 2
3) The macro definition table(MDT) use to store the body of the macro definition
4) The macro name table(MNT) use to store the names of defined macro
5) The macro definition table counter(MDTC) used to indicate the next available entry in the MDT
6) The macro name table counter (MNTC) used to indicate the next available entry in the MNT
7) The argument list array (ALA) used to substitute index markers for dummy arguments before storing a macro definition

Pass 2 database

1) the copy of the i/p macro source deck
2) the o/p expanded source deck to be used as i/p to the assembler
3) the macro definition table(MDT) create by pass 1

4) the macro name table (MNT) create by pass 1
5) the macro defination table pointer(MDTP) used to indicate macro call arguments for the index markers in the stored macro defination

problem defination

/* input code for macroprocess */

ADD A

MACRO ADD1 &ARG

LOAD ARG

MEND

MACRO PQR &A,&B,&C

ADD B

READ C

READ A

MEND

MACRO LMN

LOAD C

MEND

LOAD B

PQR 5,3,2

ADD1 1

LMN

SUB C

ENDP

MACRO NAME TABLE

Macro name	No. of parameter
ADD1	1
PQR	3

LMN	0

MAVRO DEFINATION TABLE(MDT)

INDEX	INSTRUCTION
1	LOAD #1
2	MEND
3	ADD #2
4	READ #3
5	READ #1
6	MEND
7	LOAD C
8	MEND

FORMAL VS POSITIONAL PARAMETER LIST

MACRO NAME	FOMRAL PARAMETER	POSTIONAL PARAMETER
ADD1	&ARG	#1
PQR	&A	#1
PQR	&B	#2
PQR	&C	#3

ACTURAL VS POSITIONAL PARAMETER LIST

MACRO NAME	ACTUAL PARAMETER	POSTIONAL PARAMETER
PQR	5	#1
PQR	3	#2
PQR	2	#3
ADD1	1	#1

EXPANDED CODE

INSTUCTION CODE
ADD A

LOAD B

ADD 2

READ 3

READ 5

LOAD 1

LOAD 2

SUB C

ENDP

Flowchart:

Pass 1

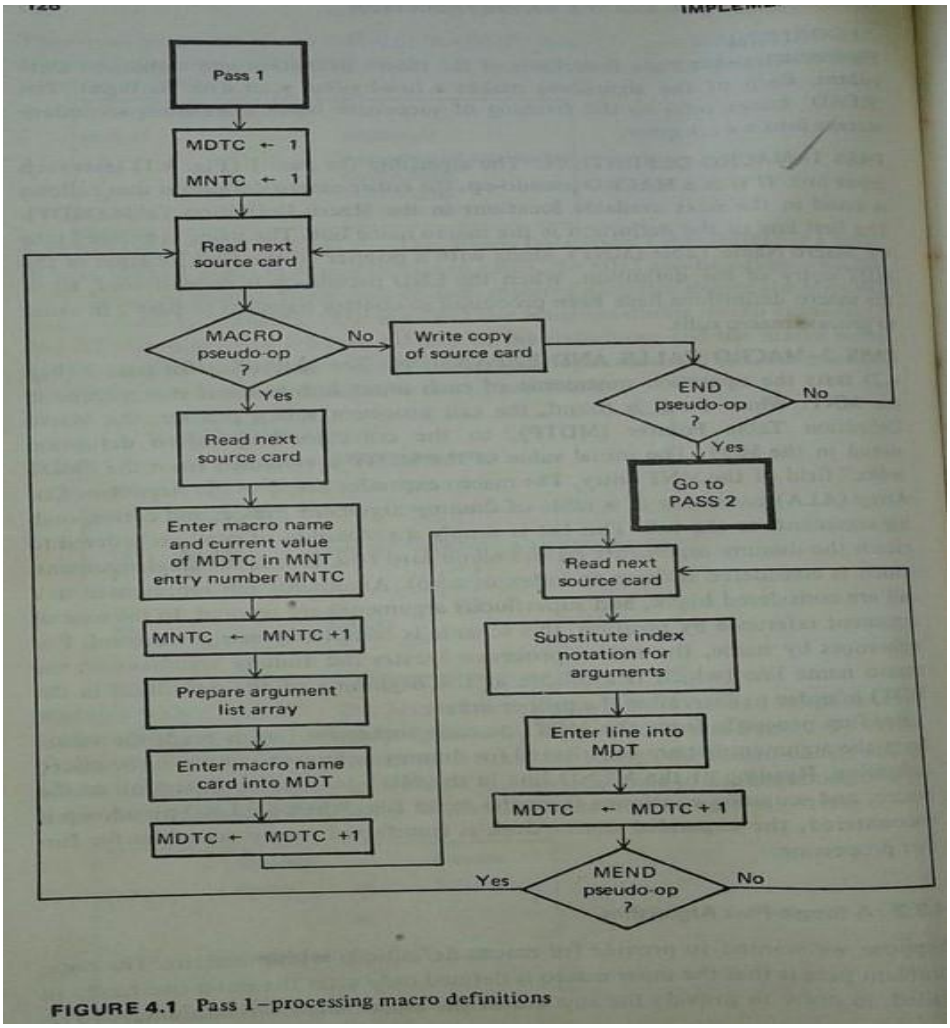


FIGURE 4.1 Pass 1–processing macro definitions

Pass 2:

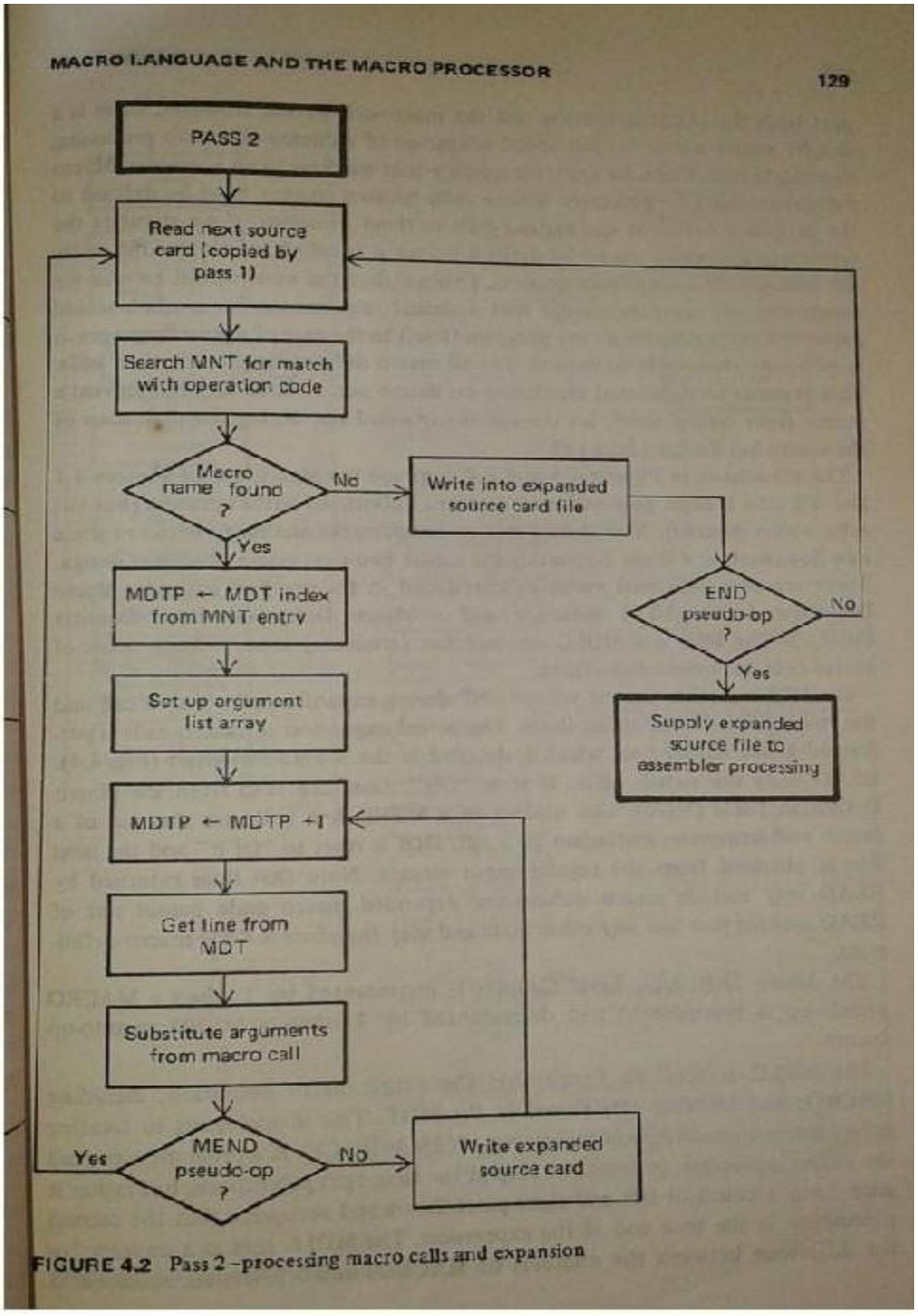


FIGURE 4.2 Pass 2 –processing macro calls and expansion

Program: TO IMPLEMENT **macro processor**

```
import java.io.*;
class Macroprocessor
{
	public static void main(String args[])
	{
		String code[][]={{"ADD","A","","",""},
		{"MACRO","ADD1","&ARG","",""},
		{"LOAD","ARG","","",""},
		{"MEND","","","","",""},
		{"MACRO","PQR","&A","&B","&C"},
		{"ADD","B","","",""},
		{"READ","C","","",""},
		{"READ","A","","",""},
		{"MEND","","","","",""},
		{"MACRO","LMN","","",""},
		{"LOAD","C","","",""},
		{"MEND","","","","",""},
		{"LOAD","B","","",""},
		{"PQR","5","3","2",""},
		{"ADD1","1","","",""},
		{"LMN","","",""},
		{"SUB","C","","",""},
		{"ENDP","","","","",}
		};
	String mn[]=new String[3],fpmn[]=new String[4],fp[]=new String[4],pp[]=new String[4];
	int parameter[]=new int[3],c=0,d=0,e=0,l=0;
	for(int i=0;i<18;i++)
	{
		if(code[i][0].equals("MACRO"))
		{
			mn[c]=code[i][1];
			for(int j=2;j<5;j++)
			{
				if(code[i][j]!="")
				{
					fpmn[e]=code[i][1];
					fp[e]=code[i][j];
					pp[e++]="#"+(++d);

				}
			} parameter[c++]=d;
			d=0;
	}
}
	String apmn[]=new String[4],ap[]=new String[4],app[]=new String[4];
	c=1;
	d=0;
	for(int i=0;i<18;i++)
	{
		for(int j=0;j<mn.length;j++)
		{
```

```
                if(code[i][0].equals(mn[j])&&code[i][1]!="")
                {
                        while(code[i][c]!="")
                        {
                                apmn[d]=code[i][0];
                                ap[d]=code[i][c];
                                app[d]="#"+c;
                                c++;
                                d++;
                        }
                        c=1;
                }
        }
}
System.out.println("macro name table");
System.out.println("_____");
System.out.println("macro name no. of parameter");
System.out.println("_____");
for(int i=0;i<mn.length;i++)
{System.out.println(mn[i]+"\t\t"  +parameter[i]);
}
System.out.println(" ----------\n \n");
System.out.println("macro definition table");
System.out.println(" ----------");
System.out.println("index \t instruction");
System.out.println(" ---------- ");
int index=1, i=0;
while(i<18)
{
        if(code[i][0].equals("MACRO"))
        {
                i++;
        while(code[i][0]!="MEND")
        {
                for(int j=0; j<fp.length; j++)
                {
                        if(("&"+code[i][1]).equals(fp[j]))
                        {
                                System.out.println((index++)+"\t"+code[i][0]+" "+pp[j]);
                                break;
                        }
                }
                i++;
        }
        System.out.println((index++)+"\t  MEND");
        }
        else
        {
                i++;
        }
 }
System.out.println(" ---------- \n \n");
System.out.println("Formal Vs Positional Parameter list");
System.out.println(" ----------");
```

```
            System.out.println("Macro Name \t Formal parameter \t Positional Parameter");
            System.out.println("-----------------------------------");
            for(i=0; i<fpmn.length;i++)
            {
                  System.out.println(fpmn[i]+"\t\t"+fp[i]+"\t\t\t"+pp[i]);
             }
            System.out.println("-----------------------------------------");
            System.out.println("actual Vs positional parameter");
            System.out.println("-----------------------------------------");
            System.out.println("macro name\t actual parameter\tpositional parameter");
            System.out.println("-----------------------------------------");
            for(i=0;i<apmn.length;i++)
{System.out.println(apmn[i]+"\t\t"+ap[i]+"\t\t\t"+app[i]);}
System.out.println("-----------------------------------\n\n");
  String pvalue[][]=new String[4][2];
for(i=0;i<4;i++)
{  for(int j=0;j<4;j++) {

if (fpmn[i].equals(apmn[j])&pp[i].equals( app[j]))
{   pvalue[i][0]=fp[i];pvalue[i][1]=ap[j];break;}}}
System.out.println("expanded code");
System.out.println("-------------------");System.out.println("instruction  code");
System.out.println("-----------------");
i=0;
while(i<18)
{
if(code[i][0].equals("ADD")||code[i][0].equals("SUB")||code[i][0].equals("ENDP")||code[i][0].equals("L
OAD"))
{System.out.println(code[i][0]+""+code[i][1]);
i++; }
else if(code[i][0].equals("MACRO"))
{i++;
while(code[i][0]!="MEND"){i++;}
i++; }
else{
int k=0;
while(k<18)
{  if (code[k][1].equals(code[i][0]))
{  k++;
while(code[k][0]!="MEND")
{
for(l=0;l<4;l++)
  if(("&"+code[k][l]).equals(pvalue[l][0]))
   System.out.println(code[k][0]+""+pvalue[l][l]);
}
k++;  }k++; }k++; i++; }
} } }
```

OUTPUT-

```
macro name table
--------------------------------------------------
macro name no. of parameter
--------------------------------------------------
ADD1         1
PQR          3
LMN          0
--------------------------------------------------

macro definition table
-----------------------------------
index    instruction
---------------------------------------
1   LOAD #1
2    MEND
3   ADD #2
4   READ #3
5   READ #1
6    MEND
7   LOAD #3
8    MEND
-----------------------------

Formal Vs Positional Parameter list
-------------------------------------------------------------------------------------------------
Macro Name   Formal parameter     Positional Parameter
-----------------------------------------------------------------
ADD1                &ARG                    #1
PQR                 &A                     #1
PQR                 &B                     #2
PQR                 &C                     #3
-----------------------------------------------------------------
actual Vs positional parameter
-------------------------------------------------------------------------------------------
macro name   actual parameter   positional parameter
-----------------------------------------------------------
PQR                5                      #1
PQR                3                      #2
PQR                2                      #3
ADD1                1                      #1
-----------------------------------------------------------

expanded code
-----------------------------
instruction code
-----------------------------
ADDA
LOADB
```

Conclusion: Thus we have implemented two pass macro preprocessor.

SIGN AND REMARK

DATE

R1	R2	R3	R4	R5	Total (15 Marks)	Signature

EXPERIMENT NO 3

Date of performance :

Date of submission :

Aim: To design a lexical analyzer for a language whose grammar is known.

Software Used: c,c++,java language

Theory:

Lexical analyzer is the first interface between the source program and the compiler. It reads source program character by character from a file or read in bulk from a file and store it in an array and then reading from an array character by character is really an issue. This is to be thought well as lexical anaysis and that too,get_next.char function is the most busy function. So if at all one has to gain in time, he/she has to think on this issue. It expresses the linear lexical structure. It also recognizes the logically cohesive units, so called as lexemes or tokens. The tokens could be keywords, identifiers, operators, labels, punctuations, symbols etc.

Activity of lexical analyzer:

Input: source program in HLL.

Output: Stream of tokens.

Many times you have to look ahead to confirm a token.

Example:- Do 51 =1.25 (Fortran)

This is not a Do statement, however it is valid assignment. In fortran blanks are allowed in identifier and therefore Do51 is valid identifier in fortran, provided 1.25 is there (and not 1, 25). Because 1, 25 would have been then, it would have been a valid Do statement, meaning to say that, keep on doing statement 5, till 1 reaches to 25 from 1. so in such case several characters are to be scanned to take a right decision. An example in c is '+' and '++'.

Algorithm:

1. Start
2. Initialise a character array storing all the key words from c language.
3. open source file in read mode.
4. Read one character at the current position of file pointer.
5. store the character, in a character array if it is not already present.
6. call the display symbol function to display the character if it is a symbol.
7. repeat these steps till end of file.
8. Rewind the file and initialize file pointer to start of file.
9. Read a character at file pointer position.
10. Check its ASCII value, if it falls with in the range of character; add it to the array of character string.
11. If the character is symbol, check the string whether it is a keyword check whether it is already present, add it to temporary array and display it with its priority.
12. If the word is include store the next string as constant in an array and the next value till end of line occurs as the value of constant.
13. If the word is data type store it in an array with flag set to indicate its data type.
14. Repeat from step 9, till end of file.
15. Display the array containing constants and their values as literals and variables and their data types as identifiers.
16. Close the source file.

17. Stop.

Problem definition:-

1)
```
int max(int a,int b)
{
        If(a<b)
        Return a;
        Else
        Return b;
}
```

2)
```
#include<stdio.h>
#include<conio.h>
#define pi=3.142
main()
{
int a,b,c;
float bc=0;
char p;
clrscr();
printf("enter a number");
scanf("%d%d",&a,&b);
c=a+b+bc
printf("%d",c);
prinf("enter a character");
printf("%c",p);
printf("character is",p);
getch();
return 0;
```

PROGRAM TO IMPLEMENT LEXICAL ANALYZER

LEX Program <x1,e>

```
%{
#include"y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {yylval=ato(yytext);
        return NUM;
        }
return yytext[0];
\n return 0;
%%
int yywrap();
{
return 1;
}
```

YACC Program <x1,y>

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

```
%token A NUM
%%
state: A'='E
        |E     {print("\n The result=%d\n",$1);}
        ;
E:E'+NUM       {$S=$1+$3;}
   |NUM       {$S=$1;}
   ;
%%
extern FILE *yyin;
main()
{
 do
 {
   yyparse();
 }while(!feof(yyin));
}
yyerror(char *s)
{
 fprintf(stderr,"%s\n",s);
}
```

OUTPUT:

```
[root@aap root]# lex x1.1
[root@aap root]# yacc -d x1.y
[root@aap root]# cc lex.yy.c.y.tab.c
[root@aap root]# ./a.out
4+6
The result=10
```

LEX Program <anbn.I>

```
%{
#include"y.tab.h"
%}
%%
a  {return A;}
b  {return B;}
.   {return(yytext[0]);}
\n return ('\n');
%%
int yywrap()
{
 return 1;
}
```

YACC Program <anbn.y>

```
%{
%}
%token A B
%%
statement:anbn'\n' {printf("\n Its a valid string!!!");
                    return 0;}
anbn:    A B
   |A anbn B
   ;
```

```
%%
main();
{
printf("\n Enter some valid string\n");
yyparse();
}
int yyerror(char *s)
{
printf("\nIt is not in anbn");
}
```

OUTPUT (run 1):

```
[root@aap root]# lex x2.1
[root@aap root]# yacc x2.y
[root@aap root]# cc lex.yy.c.y.tab.c
[root@aap root]# ./a.out

Enter some valid string
aabb
Its a valid string!!!
```

OUTPUT (run 2):

```
[root@aap root]# ./a.out

Enter some valid string
abbb
It is not in anbn
```

OUTPUT (run 3):

```
[root@aap root]# ./a.out

Enter some valid string

It is not in anbn
```

Conclusion: Lexical analyzer has been designed in c language.

SIGN AND REMARK

DATE

R1	R2	R3	R4	R5	Total (15 Marks)	Signature

EXPERIMENT NO 4

Date of performance :

Date of submission :

Aim: To implement a simple parser using Lex YACC

Software Used: c language

Theory:

A Parser for a Grammar is a program which takes in the Language string as it's input and produces either a corresponding Parse tree or an Error. When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword etc of the language, these substrings are called the tokens of the Language. The Lexical analysis can store all the recognized tokens in an intermediate file and give it to the Parser as an input. However it is more convenient to have the lexical Analyzer as a coroutine or a subroutine which the Parser calls whenever it requires a token. Parse trees are the Graphical representation of the grammar which filters out the choice for replacement order of the Production rules.

The Lex utility generates a 'C' code which is nothing but a yylex () function which can be used as an interface to YACC. A good amount of details on Lex can be obtained from the Man Pages itself. A Practical approach to certain fundamentals are given here. The General Format of a Lex File consists of three sections:

1. Definitions
2. Rules
3. User Subroutines

Definitions consists of any external 'C' definitions used in the lex actions or subroutines. e.g. all preprocessor directives like #include, #define macros etc. These are simply copied to the lex.yy.c file. The other type of definitions are Lex definitions which are essentially the lex substitution strings, lex start states and lex table size declarations.The Rules is the basic part which specifies the regular expressions and their corresponding actions. The User Subroutines are the function definitions of the functions that are used in the Lex actions.

Yacc is the Utility which generates the function 'yyparse' which is indeed the Parser. Yacc describes a context free, LALR (1) grammar and supports both bottom-up and top-down parsing. The general format for the YACC file is very similar to that of the Lex file.

1. Declarations
2. Grammar Rules
3. Subroutines

In Declarations apart from the legal 'C' declarations there are few Yacc specific declarations which begins with a %sign.

Algorithm:

Step 1. Identify the Terminal and Non-Terminal Symbols from the BNF and Lex.
Step 2. Try coding all the grammar rules in yacc with empty actions Compile,link it to Lex and check for conflicts. This is an easy way of validating the BNF for reduce/reduce and shift/reduce conflicts.
Step 3. Search for any reduce/reduce conflict. Resolve it in Lex.
Step 4. Resolve any shift/reduce conflicts. Details on resolving it given later.
Step 5. Write rules for all possible syntax errors.Details on error handling are given later.
Step 6. Code the yyerror function in subroutine section.
Step 6. Design the Data Structure which can be easily integrated with the grammar rules for syntax directed translation.
Step 7. From the Data Structures and Lex needs ,formulate the correct Stack.The stack must have pointers for all the data structures.
Step 8. Do the appropiate type binding to all tokens and yacc variables (non-terminals).
Step 9. Write all the data structures in a seperate file and inlude it in yacc.
Step 10.Code all the actions.
Step 11.Restrict the actions in case of error, i.e no data structure should be built but parsing should continue to get more errors.

Eliminating shift/reduce errors

1. use -d switch of yacc to create debug file(y.output). This file will contain the full transition diagram description and the points at which any conflict arises.
2. Try assigning precedence and associativity to tokens and literals by using %left %right %noassoc %prec. Note that precedence level in the same line is same and down the line increases.
3. In majority of the cases shift/reduce conflict is always in the vicinity of left/right recursions. These might not be due to associativity or precedence relations.e.g consider the rules

    s->XabY
    a->E|aXAY E=empty transition
    b->E|bXBY

The syntax of these rules says that there is block XY which can have zero or more blocks of type A & B. These rules have shift/reduce conflict on the symbol X since in s->AabY for making a transition from literal a to b with input X it has no way to tell if it should reduce or shift another token .

These rules can be rewritten as following

    s->XaY
    a->E|aXbY
    b->A|B

Syntax checking and error recovery.It is one of the toughest part of parsing. There are many functions like yyerrok etc. However not all yacc versions supports them. The simplest method is just to use the pseudo literal 'error'. In a rule whenever there is an error, yacc pushes a pseudo literal error and takes in next input. On identifying the rule it pops the stack and takes proper actions. Thus in this way the file pointer will always point to the right location and next rule can be looked for correctly. In our example code error class {yyerror("Missing Relation"); } says that if only class exists then error must be flaged. the literal error is pushed on the stack if relation is missing and then class is pushed. On reduction it calls yyerror with msg string.

Program :

```
//lex prgram
%{ #include “y.tab.h”
Extern int yylval
%}
%%
[0-9]+{yylval=atoi(yytext);
Return NUM;
}
Return yytext[0];
\n return 0;
%%
Int yywrap()
{
Return1;
}

// yacc program<x1.y>
%{
Include<stdio.h>
%}
% token A NUM
%%
State: A’=’E | E        {printf(“\n the result=%d\N”,$1);}
;
E:E’+’Nunm{$$=$1+$3;}
| NUM  { $$=$1;}
;
%%
Extern FILE yyin;
Main()
{
Do
{
Yyparse();
}
While(!feof(yyin))
}
Yyerror(char *s)
{
Fprintf(stderr,”%s\n”,s);
}
```

Output

```
[root@app root]$ lex x1.l
[root@app root]$ yacc –d x1.y
[root@app root]$ cc lex.yy.c y.tab.c
[root@app root]$ ./a.out
4+5
The result=10
```

Conclusion: Hence we have generated implemented parser using Lex YACC.

SIGN AND REMARK

DATE

R1	R2	R3	R4	R5	Total (15 Marks)	Signature

EXPERIMENT NO 5

Date of performance :

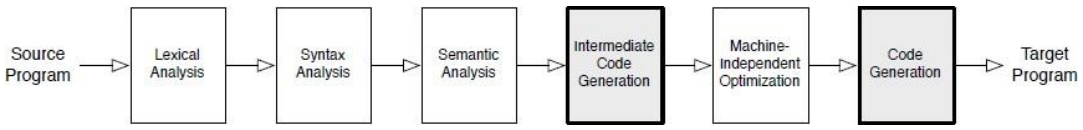
Date of submission :

Aim: To perform code optimization, considering the target machine to be X86.

Software Used: c,c++,java language

Theory:

Code Optimization: An optimizing compiler is perhaps one of the most complicated, most powerful, and most interesting programs in existence. This chapter will talk about optimizations, although this chapter will not include a table of common optimizations.



Optimization within a compiler is concerned with improving in some way the generated object code while ensuring the result is identical. Technically, a better name for this chapter might be "Improvement", since compilers only attempt to improve the operations the programmer has requested. Optimizations fall into three categories:

- Speed; improving the runtime performance of the generated object code. This is the most common optimisation
- Space; reducing the size of the generated object code
- Safety; reducing the possibility of data structures becoming corrupted (for example, ensuring that an illegal array element is not written to)

Unfortunately, many "speed" optimizations make the code larger, and many "space" optimizations make the code slower -- this is known as the space-time tradeoff.

Common Optimization Algorithms

Common optimization algorithms deal with specific cases:

1. Dead Code Elimination
2. Common Sub-expression Elimination
3. Copy Propagation
4. Code Motion
5. Induction Variable Elimination
6. Reduction In Strength
7. Function Chunking

1.Dead code Elimination:-

A variable said to be live in program if the value continue into its used sequentially variable is dead if it is be performed by eliminating such a dead code.

Eg. i-j;

……

x=i+10;

……

The optimization can be performed by eliminating the assignment statement i=j. this assignment statement is called dead assignment.

8. Common Sub-expression Elimination:-

This is an expression appearing repeatedly in program which is completed previously. If the operand of such sub expression don't get changed at all then result of sub expression is used instead of recomputing it each time.

Eg. t1:=4x; t1:=4*i

t2:=a[t1]; t2:=a[t1]

t3:=4*j; t3:=4*j

t4:=4*I; t5:=n

t5:=n t6:=b[t1]+t5

t6:=6[t4]+t5

9. Code Motion

It's a technique which moves the code outside the loop. Here is the name if there lies some expression in the loop whole result remain unchanged even after executing the loop for several time then such an expression should be placed just before the loop.

Eg. while(i<=max-1) n=max-1;

{ while(i<=n)

sum:=sum+a[i]; {

} sum:=sum+a[i];

}

10. Induction Variable Elimination

A variable x is called as induction variable of loop L if the value of variable get changed every time if the its either decremented / incremented by some constant

Eg. b1

i=t+1;

t1=4*j;

t2:=a[t1];

If t2<10 goto b1. in the above code the value of i& t1 are in block state.i.e. when value of I goto incremented by 1 the t1 get incremented by 4. hence i& t1 are induction variables.

11. Reduction In Strength

The strength of certain operands is higher than other eg. Strength of x is higher than t. in strength reduction technique the higher strength operands can be replaced by lower strength operand.

```
Eg for(i=1;i<=50;i++)
{
count :=1*7
}
```

```
temp:=7
for(i=10;i<=50;i++)
{   count:=temp;
        temp:=temp+7;
}
```

Here the value of count as 7,14,21& so on upto 50.

PROGRAM:-

```
Import java.ip.*;
Import java.util.*;
Import java.long.String;
Public class optimization
{
Public static void main(String args[]) throws IOException
{
DataInputStream in=new DataInputStream(System.in);
String s1,s2;
String code[]=new String[10];
System.out.println("Enter the string1:);
S1=in.readLine();
System.out.println("Enter the string2:);
S2=in.readLine();
If(s1.equals(s2))
{
System.out.println("enter string is duplicate");
S2=null;
}
Else
System.out.println("enter string is correct");
System.out.println("enter the line of code");
Int n=Integer.praseInt(in.readLine());
System.out.println("enter the code of program");
```

```
For(int i=0;i<=n;i++)
Code[i]=in.readLine();
For(int i=0;i<n;i++)
{
Char c[]=code[i].toCharArray();
Char d[]=code[i+1].toCharArray();
If ((c[0]=='I')&&(c[1]=='n')&&(c[2]=='t'))
If(d[3]==c[4])
System.out.println("the line"+code[i+1]+"will not be excuted since it's a dead code")'
Else
System.out.prinln("code is corrected");
}}}
```

Output:-

Enter the string1
Int i=0;
Enter the string2
Int j=3;
Enter string is corrected
Enter the line of code:
3
Enter the code of program
Int k=0;
If (k)
K=K+1;
The line if(k) will not be excuted since it's a dead code

Conclusion: Hence we have studied code optimization.

SIGN AND REMARK

DATE

R1	R2	R3	R4	R5	Total (15 Marks)	Signature

EXPERIMENT NO 6

Date of performance :

Date of submission :

Aim: To generate target code for the code optimized, considering the target machine to be X86.

Software Used: c language

Theory:

Simple code generation

In this sectin we will discuss the method of generating targe code from address statement. In this method computed result can be kept in register as long as possible eg.

X:=a+b;

Correspounding target code is " add b,r: here r hold value of a here cost=2

Or

Mov b,r, here r holds value

Add r,r0 here cost=2

The code generator algo. Used descriptor to keep track of register contents & address for names.

1. A register descriptor is used to keep track of what is currently in each register descriptor show initially all the register are empty as the code generation for block program the register will hold value of computations.
2. the add descriptor stores the location where current value of name can be found at runtime. The information about location can be stored in symbol table & is use to access the variables

the modes of operand address reusability

s->used to indicate value of operand in storage

r->used to indicate valued of operand in register.

Is->indicates that address of operand is stored in storage

Ir->indicate that address of operand is stored in register

Eg address descriptor

The address descriptor fields

The attributes mean type of operand:-it generally refer to the name of temporary variables.

The address mode indicate whether address is in storage location /in register.

The register descriptor can be shown as

By using register descriptor we can keep track of registers which are currently occupied. The status field is of Boolean type which is used to check whether the register is occupied with some data/not when status field holds the value 'true' then operand descriptor who is having the lasts value in register.

Algorithm:

```
 Gen_code(operator,Op1,op2)
{
If(op1.addressmode=’R’)
{
If(operator=’+’)
Generate(‘ADD op2, R0’);
Else if(operator=’-‘)
Generate(‘SUB op2, R0’);
Else if(operator=’*‘)
Generate(‘MUL op2, R0’);
Else if(operator=’/‘)
Generate(‘DIV op2, R0’);
}
Else if (op2.addressmode=’R’)
{

If(operator=’+’)
Generate(‘ADD op1, R0’);
Else if(operator=’-‘)
Generate(‘SUB op1, R0’);
Else if(operator=’*‘)
Generate(‘MUL op1, R0’);
Else if(operator=’/‘)
Generate(‘DIV op1, R0’);
}
Else{
Generate(‘MOV op2, R0’)
If(operator=’+’)
Generate(‘ADD op1, R0’);
Else if(operator=’-‘)
Generate(‘SUB op1, R0’);
Else if(operator=’*‘)
Generate(‘MUL op1, R0’);
Else if(operator=’/‘)
Generate(‘DIV op1, R0’);
}
}
```

Eg:=
X:=(a+b)*(c-d)+((e/f)*(a+b))

Program:

```
import java.io.*;
public class exp6
{
   public static void main(String args[])throws IOException
        {
        DataInputStream in=new DataInputStream(System.in);
        System.out.println("Enter the equation");
        String stmt=in.readLine();
        StringBuffer ans=new StringBuffer("");
        int reg=0;
        int count=0;
        char c2='a';
        int flag=0;
        for(int i=0;i<stmt.length();i++)
        {
                char c=stmt.charAt(i);
                if(i>0)
                {
                c2=stmt.charAt(i-1);
                }
                switch(c)
                {
                case'(':count++;
                break;
                case')':count--;
                break;
                case'+':if(count>0)
                {
                System.out.println("MOV "+stmt.charAt(i-1)+",R"+reg);
                System.out.println("ADD "+stmt.charAt(i+1)+",R"+reg);
                ans.append("R"+reg);
                reg++;
                }
                else
                {
                ans.append("+");
                }
                break;
                case'-':if(count>0)
                {
                System.out.println("MOV "+stmt.charAt(i-1)+",R"+reg);
                System.out.println("SUB  "+stmt.charAt(i+1)+",R"+reg);
                ans.append("R"+reg);
                reg++;
                }
                else
                {
                ans.append("-");
                }
                break;
                case'*':if(count>0)
```

```
{
System.out.println("MOV "+stmt.charAt(i-1)+",R"+reg);
System.out.println("MUL "+stmt.charAt(i+1)+",R"+reg);
ans.append("R"+reg);
reg++;
}
else
{
ans.append("*");
}
break;
case'/':if(count>0)
{
System.out.println("MOV "+stmt.charAt(i-1)+",R"+reg);
System.out.println("DIV "+stmt.charAt(i+1)+",R"+reg);
ans.append("R"+reg);
reg++;
}
else
{
ans.append("/");

}
break;
default:break;
}
flag++;
}
String ans1=new String(ans);
for(int i=0;i<ans1.length();i++)
{
char c=ans1.charAt(i);
switch(c)
{
case'+':System.out.println("ADD"+ans1.charAt(i-2)+ans1.charAt(i-1)+","+ans1.charAt(i+1)+ans1.charAt
                                                    (i+2));
                    break;
 case'-':System.out.println("SUB"+ans1.charAt(i-2)+ans1.charAt(i-1)+","+ans1.charAt(i+1)+ans1.charAt(i
                                                    +2));
                    break;
case'*':System.out.println("MUL"+ans1.charAt(i-2)+ans1.charAt(i-1)+","+ans1.charAt(i+1)+ans1.charAt
                                                    (i+2));
                    break;
  case'/':System.out.println("DIV"+ans1.charAt(i-2)+ans1.charAt(i-1)+","+ans1.charAt(i+1)+ans1.charAt(i
                                                    +2));
                    break;
                default :break;
}
}
}
}
```

OUTPUT-

```
Enter the equation
(a+b)*(c-d)+(e/f)*(a+b)
MOV a,R0
ADD b,R0
MOV c,R1
SUB d,R1
MOV e,R2
DIV f,R2
MOV a,R3
ADD b,R3
MUL R0,R1
ADD R1,R2
MUL R2,R3
```

Conclusion: Hence we have generated target code for code optimized considering target m/c to be x86.

SIGN AND REMARK

DATE

R1	R2	R3	R4	R5	Total (15 Marks)	Signature

EXPERIMENT NO 7

Date of performance :

Date of submission :

Aim: To implement a LR(0) parser.

Software Used: TURBOC3.

Theory:

1. Construc transition relation between states
   - Use algorithms Initial item setand Next item set
   - States are set of LR(0) items.
     - Shift items of the form P→α•Sβ
     - Reduce items of the form P→α•
2. Construct parsing table
   - If every state contains no conflicts use LR(0) parsing algorithm.
   - If states contain conflict.
     - Rewrite grammar or
     - Resolve conflict or
     - Use stronger parsing technique.

Constructing LR(0) parsing tables:
Items
The construction of these parsing tables is based on the notion of LR(0) items (simply called items here) which are grammar rules with a special dot added somewhere in the right-hand side. For example the rule E → E + B has the following four corresponding items:

E → • E + B
E → E • + B
E → E + • B
E → E + B •

Rules of the form A → ε have only a single item A → •. The item E → E • + B, for example, indicates that the parser has recognized a string corresponding with E on the input stream and now expects to read a '+' followed by another string corresponding with B.

Item sets
It is usually not possible to characterize the state of the parser with a single item because it may not know in advance which rule it is going to use for reduction. For example if there is also a rule E → E * B then the items E → E • + B and E → E • * B will both apply after a string corresponding with E has been read. Therefore it is convenient to characterize the state of the parser by a set of items, in this case the set { E → E • + B, E → E • * B }.

Extension of Item Set by expansion of non-terminals
An item with a dot before a nonterminal, such as E → E + • B, indicates that the parser expects to parse the nonterminal B next. To ensure the item set contains all possible rules the parser may be in the midst of parsing, it must include all items describing how B itself will be parsed. This means that if there are rules

such as B → 1 and B → 0 then the item set must also include the items B → • 1 and B → • 0. In general this can be formulated as follows:

If there is an item of the form A → v • Bw in an item set and in the grammar there is a rule of the form B → w' then the item B → • w' should also be in the item set.

Closure of item sets

Thus, any set of items can be extended by recursively adding all the appropriate items until all nonterminals preceded by dots are accounted for. The minimal extension is called the closure of an item set and written as clos(I) where I is an item set. It is these closed item sets that are taken as the states of the parser, although only the ones that are actually reachable from the begin state will be included in the tables.

Augmented grammar

Before the transitions between the different states are determined, the grammar is augmented with an extra rule

Table construction:

(0) S → E

where S is a new start symbol and E the old start symbol. The parser will use this rule for reduction exactly when it has accepted the whole input string.

For this example, the same grammar as above is augmented thus:

(0) S → E
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

It is for this augmented grammar that the item sets and the transitions between them will be determined.

Finding the reachable item sets and the transitions between them[edit]

The first step of constructing the tables consists of determining the transitions between the closed item sets. These transitions will be determined as if we are considering a finite automaton that can read terminals as well as nonterminals. The begin state of this automaton is always the closure of the first item of the added rule: S → • E:

Item set 0

S → • E
+ E → • E * B
+ E → • E + B
+ E → • B
+ B → • 0
+ B → • 1

The boldfaced "+" in front of an item indicates the items that were added for the closure (not to be confused with the mathematical '+' operator which is a terminal). The original items without a "+" are called the kernel of the item set.

Starting at the begin state (S0), all of the states that can be reached from this state are now determined. The possible transitions for an item set can be found by looking at the symbols (terminals and nonterminals) found following the dots; in the case of item set 0 those symbols are the terminals '0' and '1' and the nonterminals E and B. To find the item set that each symbol x leads to, the following procedure is followed for each of the symbols:

Take the subset, S, of all items in the current item set where there is a dot in front of the symbol of interest, x.
For each item in S, move the dot to the right of x.
Close the resulting set of items.
For the terminal '0' (i.e. where x = '0') this results in:

Item set 1
$B \rightarrow 0 \bullet$
and for the terminal '1' (i.e. where x = '1') this results in:

Item set 2
$B \rightarrow 1 \bullet$
and for the nonterminal E (i.e. where x = E) this results in:

Item set 3
$S \rightarrow E \bullet$
$E \rightarrow E \bullet * B$
$E \rightarrow E \bullet + B$
and for the nonterminal B (i.e. where x = B) this results in:

Item set 4
$E \rightarrow B \bullet$
Note that the closure does not add new items in all cases - in the new sets above, for example, there are no nonterminals following the dot. This process is continued until no more new item sets are found. For the item sets 1, 2, and 4 there will be no transitions since the dot is not in front of any symbol. For item set 3, note that that the dot is in front of the terminals '*' and '+'. For '*' the transition goes to:

Item set 5
$E \rightarrow E * \bullet B$
$+ B \rightarrow \bullet 0$
$+ B \rightarrow \bullet 1$
and for '+' the transition goes to:

Item set 6
$E \rightarrow E + \bullet B$
$+ B \rightarrow \bullet 0$
$+ B \rightarrow \bullet 1$
For item set 5, the terminals '0' and '1' and the nonterminal B must be considered. For the terminals, note that the resulting closed item sets are equal to the already found item sets 1 and 2, respectively. For the nonterminal B the transition goes to:

Item set 7
$E \rightarrow E * B \bullet$
For item set 6, the terminal '0' and '1' and the nonterminal B must be considered. As before, the resulting item sets for the terminals are equal to the already found item sets 1 and 2. For the nonterminal B the transition goes to:

Item set 8
$E \rightarrow E + B \bullet$
These final item sets have no symbols beyond their dots so no more new item sets are added, so the item generating procedure is complete. The finite automaton, with item sets as its states is shown below.

The transition table for the automaton now looks as follows:

Item Set	*	+	0	1	E	B
0			1	2	3	4
1						
2						
3	5	6				
4						
5			1	2		7
6			1	2		8
7						
8						

Program:-

To write a code for LR(0) Parser for following Production:

E->E+T

T->T*F/F

F->(E)/char

```
#include<string.h>
#include<conio.h>
#include<stdio.h>
int axn[][6][2]={
{{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
{{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},
{{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},
{{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},
{{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
{{100,5},{101,6},{101,6},{-1,-1},{101,6},{101,6}},
{{100,5},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1}},
{{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
{{-1,-1},{100,6},{-1,-1},{-1,-1},{100,11},{-1,-1}},
{{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},
{{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},
{{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}}
};
int  gotot[12][3]={1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,2,3,-1,-1,-1,-1,
```

```
9,3,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
int a[10];
char b[10];
int top=-1,btop=-1,i;
void push(int k)
{
if(top<9)
a[++top]=k;
}
void pushb(char k)
{
if(btop<9)
b[++btop]=k;
}
char TOS()
{
return a[top];
}
void pop()
{
if(top>=0)
top--;
}
void popb()
{
if(btop>=0)
b[btop--]='\0';
}
void display()
{
for(i=0;i<=top;i++)
printf("%d%c",a[i],b[i]);
}
void display1(char p[],int m)
{
int l;
printf("\t\t");
for(l=m;p[l]!='\0';l++)
printf("%c",p[l]);
printf("\n");
}
void error()
{
printf("\n\nSyntax Error");
}
void reduce(int p)
{
int len,k,ad; char src,*dest; switch(p)
                {
                case 1:dest="E+T"; src='E';
                break;
                case 2:dest="T"; src='E';
                break;
                case 3:dest="T*F"; src='T';
                break;
                case 4:dest="F"; src='T';
                break;
                case 5:dest="(E)"; src='F';
                break;
                case 6:dest="i"; src='F';
                break; default:dest="\0"; src='\0';
                break;
                }
                for(k=0;k<strlen(dest);k++)
                {
                pop();
                popb();
                } pushb(src); switch(src)
                {
                case 'E': ad=0; break;
                case 'T': ad=1; break;
                case 'F': ad=2; break;
                default: ad=-1; break;
                } push(gotot[TOS()][ad]);
                }
                int main()
                {
```

OUTPUT: -

Enter any String: - a+b*c

0 a+b*c

0a5 +b*c

0F3 +b*c

0T2 +b*c

0E1 +b*c

0E1+6 b*c

0E1+6b5 *c

0E1+6F3 *c

0E1+6T9 *c

0E1+6T9*7 c

0E1+6T9*7c5

Conclusion: - Thus we have successfully implemented LR(0) parser.

SIGN AND REMARK

R1	R2	R3	R4	R5	Total (15 Marks)	Signature