NMAM INSTITUTE OF TECHNOLOGY
Department of Computer Science & Engineering

Compiler Design Lab Manual

By,
Sunil Kumar B.L.
Senior Lecturer,
Department of CSE,
NMAMIT, Nitte

COMPILER DESIGN LAB MANUAL

LEX PROGRAMS:

1. Program to count the number of vowels and consonants in a given string.

```
%{
        #include<stdio.h>
        int vowels=0;
        int cons=0;
%}
%%
[aeiouAEIOU] {vowels++;}
[a-zA-Z] {cons++;}
%%
int yywrap()
{
        return 1;
}
main()
{
        printf("Enter the string.. at end press ^d\n");
        yylex();
        printf("No of vowels=%d\nNo of
consonants=%d\n",vowels,cons);
}
```

2. Program to count the number of characters, words, spaces and lines in a given input file.

```
%{
        #include<stdio.h>
        Int c=0, w=0, s=0, l=0;
%}
WORD  [^   \t\n,\.:]+
EOL  [\n]
BLANK  [ ]
%%
{WORD} {w++; c=c+yyleng;}
{BLANK} {s++;}
{EOL} {l++;}
.  {c++;}
%%
int yywrap()
{
        return 1;
}
main(int argc, char *argv[])
{
        If(argc!=2)
        {
                printf("Usage: <./a.out> <sourcefile>\n");
                exit(0);
        }
        yyin=fopen(argv[1],"r");
        yylex();
        printf("No of characters=%d\nNo of words=%d\nNo of
        spaces=%d\n No of lines=%d",c,w,s,l);
}
```

3. Program to count no of:
    a) +ve and –ve integers
    b) +ve and –ve fractions

```
%{
    #include<stdio.h>
    int posint=0, negint=0,posfraction=0, negfraction=0;
%}
%%
[-][0-9]+ {negint++;}
[+]?[0-9]+ {posint++;}
[+]?[0-9]*\.[0-9]+ {posfraction++;}
[-][0-9]* \.[0-9]+ {negfraction++;}
%%
int yywrap()
{
    return 1;
}

main(int argc, char *argv[])
{
    If(argc!=2)
    {
        printf("Usage: <./a.out> <sourcefile>\n");
        exit(0);
    }
    yyin=fopen(argv[1],"r");
    yylex();
    printf("No of +ve integers=%d\n No of –ve integers=%d\n No of
+ve
    fractions=%d\n No of –ve fractions=%d\n", posint, negint,
    posfraction, negfraction);
}
```

4. Program to count the no of comment line in a given C program. Also
eliminate them and copy that program into separate file

```
%{
        #include<stdio.h>
        int com=0;
%}
%s COMMENT
%%
“/*”[.]*”*/” {com++;}
“/*” {BEGIN COMMENT ;}
<COMMENT>”*/” {BEGIN 0; com++ ;}
<COMMENT>\n {com++ ;}
<COMMENT>. {;}
.|\n {fprintf(yyout,”%s”,yytext);
%%
int yywrap()
{
        return 1;
}

main(int argc, char *argv[])
{
        If(argc!=2)
        {
                printf(“Usage: <./a.out> <sourcefile> <destn file>\n”);
                exit(0);
        }
        yyin=fopen(argv[1],”r”);
        yyout=fopen(argv[2],”w”);
        yylex();
        printf(“No. of comment lines=%d\n”,com);
}
```

5. Program to count the no of ‘scanf’ and ‘printf’ statements in a C program. Replace them with ‘readf’ and ‘writef’ statements respectively.

```
%{
        #include<stdio.h>
        int pc=0, sc=0;
%}
%%
“printf” { fprintf(yyout,”writef”); pc++;}
“scanf” { fprintf(yyout,”readf”); sc++;}
%%
int yywrap()
{
        return 1;
}

main(int argc, char *argv[])
{
        if(argc!=2)
        {
                printf(“Usage: <./a.out> <sourcefile> <destn file>\n”);
                exit(0);
        }
        yyin=fopen(argv[1],”r”);
        yyout=fopen(argv[2],”w”);
        yylex();
        printf(“No of printf statements = %d\n No of scanf
        statements=%d\n”, pc, sc);
}
```

6. Program to recognize a valid arithmetic expression and identify the identifiers and operators present. Print them separately.

```
%{
        #include<stdio.h>
        #include<string.h>
        int noprt=0, nopnd=0, valid=1, top=-1, m, l=0, j=0;
        char opnd[10][10], oprt[10][10], a[100];
%}
%%
“(“ { top++; a[top]=’(‘ ; }
“{“ { top++; a[top]=’{‘ ; }
“[“ { top++; a[top]=’[‘ ; }
“)” { if(a[top]!=’(‘)
        {
          valid=0;  return;
        }
        else
          top--;
    }
“}” { if(a[top]!=’{‘)
        {
          valid=0;  return;
        }
        else
          top--;
    }
“]” { if(a[top]!=’[‘)
        {
          valid=0;  return;
        }
        else
          top--;
    }
“+”|”-“|”*”|”/” {       noprt++;
                         strcpy(oprt[l], yytext);
```

```
                        l++;
                    }
[0-9]+|[a-zA-Z][a-zA-Z0-9_]* {nopnd++;
                        strcpy(opnd[j],yytext);
                         j++;
                        }
%%
int yywrap()
{
        return 1;
}

main()
{
        int k;
        printf("Enter the expression.. at end press ^d\n");
        yylex();
        if(valid==1 && i==-1 && (nopnd-noprt)==1)
        {
                printf("The expression is valid\n");
                printf("The operators are\n");
                for(k=0;k<l;k++)
                        Printf("%s\n",oprt[k]);
                for(k=0;k<l;k++)
                        Printf("%s\n",opnd[k]);
        }
        else
                Printf("The expression is invalid");
}
```

7. Program to recognize whether a given sentence is simple or compound.

```
%{
        #include<stdio.h>
        Int is_simple=1;
%}
%%
[   \t\n]+[aA][nN][dD][   \t\n]+ {is_simple=0;}
[   \t\n]+[oO][rR][   \t\n]+ {is_simple=0;}
[   \t\n]+[bB][uU][tT][   \t\n]+ {is_simple=0;}
. {;}
%%
int yywrap()
{
        return 1;
}

main()
{
        int k;
        printf("Enter the sentence.. at end press ^d");
        yylex();
        if(is_simple==1)
                {
                        Printf("The given sentence is simple");
                }
        else
                {
                        Printf("The given sentence is compound");

                }
```

8. Program to recognize and count the number of identifiers in a given
input file.

```
%{
        #include<stdio.h>
        int id=0;
%}
%%
[a-zA-Z][a-zA-Z0-9_]* { id++ ; ECHO; printf("\n");}
.+ { ;}
\n { ;}
%%
int yywrap()
{
        return 1;
}

main (int argc, char *argv[])
{
        if(argc!=2)
        {
                printf("Usage: <./a.out> <sourcefile>\n");
                exit(0);
        }
        yyin=fopen(argv[1],"r");
        printf("Valid identifires are\n");
        yylex();
        printf("No of identifiers = %d\n",id);
}
```

YACC PROGRAMS:

1. Program to test the validity of a simple expression involving operators
   +, -, * and /

Yacc Part

```
%token NUMBER ID NL
%left '+' '-'
%left '*' '/'
%%
stmt : exp NL { printf("Valid Expression"); exit(0);}
        ;
exp :   exp '+' exp
        | exp '-' exp
        | exp '*' exp
        | exp '/' exp
        | '(' exp ')'
        | ID
        | NUMBER
        ;
%%
int yyerror(char *msg)
{
        printf("Invalid Expression\n");
        exit(0);
}
main ()
{
        printf("Enter the expression\n");
        yyparse();
}
```

Lex Part

```
%{
        #include “y.tab.h”
%}
%%
[0-9]+ { return DIGIT; }
[a-zA-Z][a-zA-Z0-9_]* { return ID; }
\n { return NL ;}
. { return yytext[0]; }
%%
```

2. Program to recognize nested IF control statements and display the levels of nesting.

Yacc Part

```
%token IF  RELOP  S  NUMBER  ID
%{
        int count=0;
%}
%%
stmt : if_stmt { printf("No of nested if statements=%d\n",count);
exit(0);}
        ;
if_stmt : IF '(' cond ')' if_stmt {count++;}
           | S;
           ;
cond :  x RELOP x
        ;
x : ID
  | NUMBER
  ;
%%
int yyerror(char *msg)
{
        printf("Invalid Expression\n");
        exit(0);
}
main ()
{
        printf("Enter the statement");
        yyparse();
}
```

Lex Part

```
%{
        #include “y.tab.h”
%}
%%
“if” { return IF; }
[sS][0-9]* {return S;}
“<”|”>”|”==”|”!=”|”<=”|”>=” { return RELOP; }
[0-9]+ { return NUMBER; }
[a-zA-Z][a-zA-Z0-9_]* { return ID; }
\n { ; }
. { return yytext[0]; }
%%
```

3. Program to check the syntax of a simple expression involving operators
   +, -, * and /

Yacc Part

```
%token NUMBER ID NL
%left '+' '-'
%left '*' '/'
%%
stmt : exp NL { printf("Valid Expression"); exit(0);}
        ;
exp :   exp '+' exp
        | exp '-' exp
        | exp '*' exp
        | exp '/' exp
        | '(' exp ')'
        | ID
        | NUMBER
        ;
%%
int yyerror(char *msg)
{
        printf("Invalid Expression\n");
        exit(0);
}
main ()
{
        printf("Enter the expression\n");
        yyparse();
}
```

Lex Part

```
%{
        #include “y.tab.h”
%}
%%
[0-9]+ { return NUMBER; }
[a-zA-Z][a-zA-Z0-9_]* { return ID; }
\n { return NL ;}
. { return yytext[0]; }
%%
```

4. Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.

Yacc Part

```
%token DIGIT LETTER NL UND
%%
stmt : variable NL { printf("Valid Identifiers\n"); exit(0);}
        ;

variable : LETTER alphanumeric
          ;

alphanumeric: LETTER alphanumeric
          | DIGIT alphanumeric
          | UND alphanumeric
          | LETTER
          | DIGIT
          | UND
          ;
%%
int yyerror(char *msg)
{
        printf("Invalid Expression\n");
        exit(0);
}
main ()
{
        printf("Enter the variable name\n");
        yyparse();
}
```

Lex Part

```
%{
        #include “y.tab.h”
%}
%%
[a-zA-Z] { return LETTER ;}
[0-9] { return DIGIT ; }
[\n] { return NL ;}
[_] { return UND; }
. { return yytext[0]; }
%%
```

5. Program to evaluate an arithmetic expression involving operating +, -,
* and /.

Yacc Part

```
%token NUMBER ID NL
%left '+' '-'
%left '*' '/'
%%
stmt : exp NL { printf("Value = %d\n",$1); exit(0);}
        ;
exp :   exp '+' exp { $$=$1+$3; }
        | exp '-' exp { $$=$1-$3; }
        | exp '*' exp { $$=$1*$3; }
        | exp '/' exp { if($3==0)
                        {
                                printf("Cannot divide by 0");
                                exit(0);
                        }
                        else
                                $$=$1/$3;
                        }
        | '(' exp ')' { $$=$2; }
        | ID { $$=$1; }
        | NUMBER { $$=$1; }
        ;
%%
int yyerror(char *msg)
{
        printf("Invalid Expression\n");
        exit(0);
```

```
}
main ()
{
        printf("Enter the expression\n");
        yyparse();
}
```

Lex Part

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

6. Program to recognize strings ‘aaab’, ‘abbb’, ‘ab’ and ‘a’ using grammar
   ($a^n b^n$, n>=0)

Yacc Part

```
%token A  B  NL
%%
stmt : s NL { printf("Valid String\n"); exit(0) ;}
        ;
s : A s B
  |
 ;
%%
int yyerror(char *msg)
{
        printf("Invalid String\n");
        exit(0);
}
main ()
{
        printf("Enter the String\n");
        yyparse();
}
```

Lex Part

```
%{
        #include "y.tab.h"
```

```
%}
%%
[aA] { return A; }
[bB] { return B; }
\n { return NL ;}
. { return yytext[0]; }
%%
```

7. Program to recognize the grammar ($a^n b$, n>=10)

```
%token A  B  NL
%%
stmt : A A A A A A A A A A s B NL
        {
                Printf("Valid"); exit(0);
        }
        ;
s : s A
  |
 ;
int yyerror(char *msg)
{
        printf("Invalid String\n");
        exit(0);
}
main ()
{
        printf("Enter the String\n");
        yyparse();
}
```

Lex Part

```
%{
        #include "y.tab.h"
%}
%%
```

```
[aA] { return A; }
[bB] { return B; }
\n { return NL ;}
. { return yytext[0]; }
%%
```

Steps to Execute Lex Program:

```
lex <pgm name>
cc lex.yy.c –ll
./a.out
```

Steps to execute YACC program:

```
yacc –d <yacc_pgm name>
lex <lex_pgm_name>
cc y.tab.c lex.yy.c –ly –ll
./a.out
```

Compiler lab Mini Project

Develop **one pass** mini-C compiler .The compiler should give final output in 8086 assembly language. MASM assembler (or debug utility) may be used to assemble (and execute) this code. **Mini-C compiler has to be written in 4 modules**. Final complete compiler is generated by linking stepwise refinement of these modules. Each module has to be tested by a sample test input. For the final compiler, Mini-C test program will be the input and the equivalent 8086 program will be the output.

Mini-C language is a subset of C language, with little variations in its syntax.
In Mini-C, **basic data type available is int**. Mini-C is not case sensitive. In Mini-C language, set of operators is subdivided into groups as follows:

a. **arithmetic operators of addition, subtraction, multiplication and division**
b. **relational operators like ==, !=, <=, >=, < and >**

Transfer of information from or to the input or output devices is caused by scan and print statements within the program.
Mini-C compiler should **support if statement and for statement**.

Features **not included** are:

1. char type, float type, array type and any user defined types
2. functions
3. while statement, compound statement etc..

4. file handling, pointers
5. gotos and hence labels

Module-1:

This is a main module which will later call other modules appropriately. EXE version of this module can be used along with command line arguments to run the complete Mini-C compiler

Eg: MCOMPILE.EXE TEST.MC

Here TEST.MC is the name of the test input program written in Mini-C. The module should create TEST.ASM file which should contain the equivalent 8086 code for the input mini C program. .ASM file has to be created only if there are no errors in the input program. This module must have following functions

1) Function MYLEX() will separate the tokens. For the time being restrict, this function to separate the tokens of only first line of the program. The first line of program may be

   main() ; /* this is comment */

   It also must delete comments.

2) Function ERROR() which should display errors like

   a)TEST.MC file not found

   b) syntax error in line 'n'

   c)semicolon expected in line 'n' etc..

   Since line number also has to be given with the error message, a counter for the line number should be maintained, while reading from the input file

3) Function MAIN() calls above two functions. It should open the input file TEST.MC, scan the first line (ie main()) of the input program and if no errors are found in that line, should create output file TEST.ASM

Module-2:

1) Function MAIN() written in module 1 must be extended to handle variable declarations of the input sample program.
2) Function MYLEX() written in module 1 must also be extended to separate tokens in the variable declaration part of the input sample program
3) A new function VAR_CHECK() must be written to identify different allowed types of variables (In this case it is only integer type). A binary tree has to be created for storing the information of each and every variable. Each node of the tree should contain variable name and its type(in this case it is only integer). Tree must be sorted lexicographically based on its variable names. This requires two more functions: one to create the tree and another to search in the tree.
4) Data definitions must be written onto the output file TEST.ASM
5) Function ERROR() written in the module 1 must be extended to include error messages like:
    a) redeclaration of variable
    b) syntax error in variable declaration in line n

Sample input: TEST.MC

```
main()
{ /* this is comment*/
  int a,b;
 }
```

Output : TEST.ASM

```
data segment
 a db ?
 b db ?
data segment ends
 code segment
 assume cs:code, ds:data
 start: mov ax, data
        mov ds,ax
         mov ah,4ch
         int 21h
        code ends
        end start
```

Module-3:

Arithmetic Expression Parser

This module is for parsing the arithmetic expression. **Recursive Descent Parsing** (a type of top down parsing) is to be used. Input arithmetic expression (infix notation) has to be scanned for the symbols and these symbols have to be pushed on to the stack. 8086 code has to be generated and stored in the output file simultaneously. Expression may also have array elements as its operands.

1. Function MYLEX() written in module 1 has to be extended to separate tokens in an arithmetic expression (to simplify whole thing, you may assume that identifiers are made of only single alphabets)
2. A new function EXPRESSION() has to be written which should call another function TERM(), which in turn should call the function FACTOR(). FACTOR() should call recursively the function EXPRESSION().

   Function FACTOR() should scan the input symbols of the arithmetic expression and push onto the stack the identifier name and its type. Function TERM() should generate 8086 code for multiplication and division. Function EXPRESSION() should generate 8086 code for addition and subtraction.

[Note: Details about recursive descent parser are available in the text book.

Structure of code will be as follows:

```
EXPRESSION()
   {TERM();
     EXPRESSION'()
    }

EXPRESSION'()
  { if   (token== + or  -)
      MYLEX();
       TERM();
        Generate code for add or sub as
             POP AX, POP BX; ADD or SUB AX,BX; PUSH AX;
      EXPRESSION'()
  }

TERM()
     { FACTOR()
       TERM'()
      }

TERM'()
     { if (token == * or /)
        MYLEX()
       FACTOR()
      Generate code for mul and div;
      TERM'()
      }

FACTOR()
     { if (token =='(')
         {MYLEX();
            EXPRESSION();
          If(token== ')')
               { MYLEX(); }
         }
      Else if (token type == id or num)
        {generate code for id or num  as
               MOV AX, token; PUSH AX;
               MYLEX();}
      }
```

3. Function ERROR() has to extended to include some more error

messages like :

a) parenthesis expected in line n
b) variable not defined
c) assign expected in line n etc…

Sample input : TEST.MC

```
main()
{ int a,b,d,e,f;
  int c
 c= d+e*((a-b)/(e+f)):
}
```

Output:TEST.ASM

Equivalent 8086 assembly code

Module 4:

1. Function STATEMENT() has to be written which should parse and generate 8086 code for the following constructs

   a) If (a relop b) then assignstmt1 else assignstmt2;
   (assignstmts can have arithmetic expressions on the RHS)
   b) for (a=1;a<10;a++)

   Note that these control statements can be realized by conditional and unconditional jump instructions and compare instructions of 8086. Hence function for lable generation has to be written

2. Functions for generating 8086 code for scan and read statements have to be written.

Eg: For reading an integer from keyboard,MINI-C statement may be

```
scan(a);
```

Equivalent 8086 code may be:

```
MOV AH,01
INT 21h
MOV a, AL
```

For displaying an integer, MINI-C statement may be

```
print (a)
```

Equivalent 8086 code may be

```
MOV DL,a
MOV AH,02
INT 21h
```

(in fact, the integer (2 digit) has to be converted to ASCII before using the above code. So integer to ascii conversion is initially required in the compiler and then the ascii values are to be used as the second parameter in MOV DL,a)

3. ERROR() routine has to be extended to display errors like
   a) then expected
   b) paranthesis missing in for statement
   c) no argument in scan or print statement etc…

4. All the modules have to be linked to get a complete MINI-C compiler.

Some Other small C problems :

1. *a.* Program in C to remove comments from a C file
   (both nested and non nested)
   b. Program to check whether given identifier is valid or not
   (as per the lexical rules of id)

2. To read a file containing syntactically correct C program, and display the tokens and their categories like id, constant, reserved word, relational operator, arithmetic operator.

3. Given an arithmetic expression, using recursive descent method, to print the order in which the operations are performed for given precedence and associativity rules.