NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

PRINCIPLE OF COMPILER CONSTRUCTION

Lab Assignment

SUBMITTED BY: -

Name: *HARSH KUMAR*

Branch: CSAI

Roll no.:2021UCA1829

INDEX

PRACTICAL 1

AIM

Implement a two-pass assembler 8085/8086

CODE

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<conio.h>

struct opTable {
  char code[10], objcode[10];
} myOpT[3] = {
  "LDA", "00",
  "STA", "0C",
  "LDCH", "50"
};

struct symbolTable {
  char symbol[10];
  int addr;
} mySymTab[10];

int startAddress, locCounter, symCount = 0, length;
char line[20], label[8], opcode[8], operand[8], programName[10];

void checkLabel() {

  int k, dupSymbol = 0;

  for (k = 0; k < symCount; k++)
    if (!strcmp(label, mySymTab[k].symbol)) {
      mySymTab[k].addr = -1;
      dupSymbol = 1;
      break;
    }

  if (!dupSymbol) {
    strcpy(mySymTab[symCount].symbol, label);
    mySymTab[symCount++].addr = locCounter;
  }
}
```

```
void checkOpCode() {
  int k = 0, found = 0;
  for (k = 0; k < 3; k++)

    if (!strcmp(opcode, myOpT[k].code)) {
      locCounter += 3;
      found = 1;
      break;
    }

  if (!found) {
    if (!strcmp(opcode, "WORD")) locCounter += 3;
    else if (!strcmp(opcode, "RESW")) locCounter += (3 * atoi(operand));
    else if (!strcmp(opcode, "RESB")) locCounter += atoi(operand);
  }
}

void readLine() {
  char buff[8], word1[8], word2[8], word3[8];
  int i, j = 0, count = 0;
  label[0] = opcode[0] = operand[0] = word1[0] = word2[0] = word3[0] = '\0';
  for (i = 0; line[i] != '\0'; i++) {
    if (line[i] != ' ') buff[j++] = line[i];
    else {
      buff[j] = '\0';
      strcpy(word3, word2);
      strcpy(word2, word1);
      strcpy(word1, buff);
      j = 0;
      count++;
    }
  }
  buff[j - 1] = '\0';
  strcpy(word3, word2);
  strcpy(word2, word1);
  strcpy(word1, buff);
  switch (count) {
  case 0:
    strcpy(opcode, word1);
    break;
  case 1: {
    strcpy(opcode, word2);
    strcpy(operand, word1);
  }
  break;
  case 2: {
    strcpy(label, word3);
    strcpy(opcode, word2);
    strcpy(operand, word1);
```

```
  }
  break;
  }
}

void PASS1() {
  FILE * input, * inter;
  input = fopen("assemblycode.txt", "r");
  inter = fopen("intermediate.txt", "w");
  printf("LOCATION LABEL\tOPERAND\tOPCODE\n");
  printf("_____________________________________");
  fgets(line, 20, input);

  readLine();

  if (!strcmp(opcode, "START")) {

    startAddress = atoi(operand);
    locCounter = startAddress;
    strcpy(programName, label);

    fprintf(inter, "%s", line);
    fgets(line, 20, input);
  } else {
    programName[0] = '\0';
    startAddress = 0;
    locCounter = 0;
  }
  printf("\n %d\t %s\t%s\t %s", locCounter, label, opcode, operand);

  while (strcmp(line, "END") != 0) {

    readLine();
    printf("\n %d\t %s \t%s\t %s", locCounter, label, opcode, operand);
    if (label[0] != '\0') checkLabel();
    checkOpCode();
    fprintf(inter, "%s %s %s\n", label, opcode, operand);
    fgets(line, 20, input);
  }

  printf("\n %d\t\t%s", locCounter, line);
  fprintf(inter, "%s", line);

  fclose(inter);
  fclose(input);
}

void PASS2() {
  FILE * inter, * output;
  char record[30], part[6], value[5];
```

```
  int currtxtlen = 0, foundopcode, foundoperand, chk, operandaddr, recaddr = 0;
  inter = fopen("intermediate.txt", "r");
  output = fopen("output.txt", "w");
  fgets(line, 20, inter);

  readLine();
  if (!strcmp(opcode, "START")) fgets(line, 20, inter);
  printf("\n\nObject Code\n");
  printf("\nH^ %s ^ %d ^ %d ", programName, startAddress, length);
  fprintf(output, "\nH^ %s ^ %d ^ %d ", programName, startAddress, length);
  recaddr = startAddress;
  record[0] = '\0';
  while (strcmp(line, "END") != 0) {
    operandaddr = foundoperand = foundopcode = 0;
    value[0] = part[0] = '\0';
    readLine();
    for (chk = 0; chk < 3; chk++) {
      if (!strcmp(opcode, myOpT[chk].code)) {
        foundopcode = 1;
        strcpy(part, myOpT[chk].objcode);

        if (operand[0] != '\0') {
          for (chk = 0; chk < symCount; chk++)

            if (!strcmp(mySymTab[chk].symbol, operand)) {
              itoa(mySymTab[chk].addr, value, 10);
              strcat(part, value);
              foundoperand = 1;
            }
          if (!foundoperand) strcat(part, "err");
        }
      }
    }
    if (!foundopcode) {
      if (strcmp(opcode, "BYTE") == 0 || strcmp(opcode, "WORD") || strcmp(opcode,
"RESB")) {
        strcat(part, operand);
      }
    }
    if ((currtxtlen + strlen(part)) <= 8) {
      strcat(record, "^");
      strcat(record, part);

      currtxtlen += strlen(part);
    } else {
      printf("\nT^ %d ^%d %s", recaddr, currtxtlen, record);
      fprintf(output, "\nT^ %d ^%d %s", recaddr, currtxtlen, record);
      recaddr += currtxtlen;
      currtxtlen = strlen(part);
      strcpy(record, part);
```

```
    }
    fgets(line, 20, inter);
  }
  printf("\nT^ %d ^%d %s", recaddr, currtxtlen, record);
  fprintf(output, "\nT^ %d ^%d %s", recaddr, currtxtlen, record);
  printf("\nE^ %d\n", startAddress);
  fprintf(output, "\nE^ %d\n", startAddress);
  fclose(inter);
  fclose(output);
}

int main() {
  PASS1();
  length = locCounter - startAddress;
  PASS2();
  getch();
}
```

OUTPUT

assemblycode.txt file

assemblycode.txt - Notepad

File Edit Format View Help

```
MYCODE START 1000
STA
LOOP1 JMP LOOP2
LDA
LOOP2 JMP LOOP1
RESB 09
LDA
STA
JMP LOOP1
END
```

Output Generated

```
LOCATION LABEL  OPERAND OPCODE
_____________________________________
 1000    MYCODE START    1000
 1000           STA
 1003    LOOP1  JMP      LOOP2
 1003           LDA
 1006    LOOP2  JMP      LOOP1
 1006           RESB     09
 1015           LDA
 1018           STA
 1021           JMP      LOOP1
 1021           END

Object Code

H^ MYCODE ^ 1000 ^ 21
T^ 1000 ^7 ^0C^LOOP2
T^ 1007 ^7 00^LOOP1
T^ 1014 ^6 09^00^0C
T^ 1020 ^5 LOOP1
E^ 1000
```

PRACTICAL 2

AIM

Develop a lexical analyzer for “C” using LEX tool.

CODE

LexicalAnalyzer.l

```
/* Definition Section */
%{
    int FLAG=0; // Flag for Comment
%}

identifier [a-zA-Z][a-zA-Z0-9]*

/* Rule Section */
%%
#.*\n {printf("\nPREPROCESSOR DIRECTIVE\n%s",yytext);}

auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if
|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigne
d|void|volatile|while {printf("\nKEYWORD  %s",yytext);}

"/*" {FLAG = 1;}
"*/" {FLAG = 0;}

{identifier}\( {if(!FLAG)printf("\n\nFUNCTION\n%s",yytext);}

{identifier}(\[[0-9]*\])? {if(!FLAG) printf("\nIDENTIFIER %s",yytext);}

\".*\" {if(!FLAG)printf("\nSTRING %s ",yytext);}

[0-9]+ {if(!FLAG) printf("\nNUMERIC LITERAL %s",yytext);}

\{ {if(!FLAG) printf("\nBEGINNING OF BLOCK");}
\} {if(!FLAG) printf("\nENDING OF BLOCK");}

\) {if(!FLAG);printf("\n)");}
= {if(!FLAG) printf("\nASSIGNMENT OPERATOR %s",yytext);}

\<= |
\>= |
\< |
```

```
\== |
\!= |
\> {if(!FLAG) printf("\nRELATIONAL OPERATOR %s",yytext);}

\, |
\;  {if(!FLAG) printf("\nSEPARATOR %s",yytext);}

%%

/* Driver Section */
int main(int argc, char **argv)
{
    FILE *file;
    file=fopen("Code.c","r");
    if(!file)
    {
        printf("Could not open the file! :(");
        exit(1);
    }
    yyin=file;
    yylex();
    printf("\n");
    return 0;
}

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

OUTPUT

Code.c

Start here | Code.c

```c
#include <stdio.h>

int main(int x)
{
    /* Declaring Variables */
    int x, y, z;
    x = 7;
    y = 5;

    /* Comparing x and y */
    if (x > y)
        z = 0;
    else
        z = -1;

    /* Printing the result */
    printf("The value of z: %d", z);

    /* for loop */
    for (int i = 0; i < 5; i++)
        i++;

    return 0;
}
```

Terminal

```
PREPROCESSOR DIRECTIVE
#include <stdio.h>


KEYWORD  int

FUNCTION
main(
KEYWORD  int
IDENTIFIER x
)

BEGINNING OF BLOCK


KEYWORD  int
IDENTIFIER x
SEPARATOR ,
IDENTIFIER y
SEPARATOR ,
IDENTIFIER z
SEPARATOR ;

IDENTIFIER x
ASSIGNMENT OPERATOR =
NUMERIC LITERAL 7
SEPARATOR ;

IDENTIFIER y
ASSIGNMENT OPERATOR =
NUMERIC LITERAL 5
SEPARATOR ;



KEYWORD  if (
IDENTIFIER x
RELATIONAL OPERATOR >
IDENTIFIER y
)

IDENTIFIER z
ASSIGNMENT OPERATOR =
NUMERIC LITERAL 0
SEPARATOR ;
KEYWORD  else

IDENTIFIER z
ASSIGNMENT OPERATOR = -
NUMERIC LITERAL 1
SEPARATOR ;



FUNCTION
printf(
STRING "The value of z: %d"
SEPARATOR ,
IDENTIFIER z
)
SEPARATOR ;


KEYWORD  for

KEYWORD  for (
KEYWORD  int
IDENTIFIER i
ASSIGNMENT OPERATOR =
NUMERIC LITERAL 0
SEPARATOR ;
IDENTIFIER i
RELATIONAL OPERATOR <
NUMERIC LITERAL 5
SEPARATOR ;
IDENTIFIER i++
)

IDENTIFIER i++
SEPARATOR ;


KEYWORD  return
NUMERIC LITERAL 0
SEPARATOR ;

ENDING OF BLOCK
```

PRACTICAL 3

AIM

Represent ‘C’ language using Context Free Grammar

Context Free Grammar

```
The Context Free Grammar for C language can be given by G = (V, T, S, P) where:

V = set of non-terminals
  = {program_unit, translation_unit, external_decl, function_definition, decl,
decl_list, decl_specs, storage_class_spec, type_spec, type_qualifier,
struct_or_union_spec, struct_or_union, struct_decl_list, init_declarator_list,
init_declarator, struct_decl, spec_qualifier_list, struct_declarator_list,
struct_declarator_list, struct_declarator, enum_spec, enumerator_list, enumerator,
declarator, direct_declarator, pointer, type_qualifier_list, param_list, param_decl,
id_list, initializer, initializer_list, type_name, abstract_declarator,
direct_abstract_declarator, stat, labeled_stat, exp_stat,compound_stat, stat_list,
selection_stat, iteration_stat, jump_stat, exp assignment_exp, assignment_operator,
conditional_exp, logical_or_exp, logical_and_exp, inclusive_or_exp, exclusive_or_exp,
and_exp, equality_exp, relational_exp, shift_expression, additive_exp, mult_exp,
cast_exp, unary_exp, unary_operator, postfix_exp, primary_exp, argument_exp_list,
consts, int_const, char_const, float_const, id, string, enumeration_const,
storage_const, type_const, qual_const, struct_const, enum_const, DEFINE, IF, ELSE, FOR,
DO, WHILE, BREAK, SWITCH, CONTINUE, RETURN, CASE, DEFAULT, GOTO, SIZEOF, PUNC, or_const,
and_const, eq_const, shift_const, rel_const, inc_const, point_const, HEADER}

T = set of terminals
  = {All ASCII characters}

S = start symbol = program_unit

P = set of productions
program_unit                    -> HEADER program_unit
                                          | DEFINE primary_exp program_unit

                                          | translation_unit


translation_unit                -> external_decl

                                          | translation_unit external_decl
```

```
external_decl                        -> function_definition
                                           | decl

function_definition                  -> decl_specs declarator decl_list compound_stat
                                           | declarator decl_list compound_stat
                                           | decl_specs declarator  compound_stat

                                           | declarator compound_stat

decl                                 -> decl_specs init_declarator_list ';'

                                           | decl_specs ';'

decl_list                            -> decl
                                           | decl_list decl

decl_specs                           -> storage_class_spec decl_specs
                                           | storage_class_spec
                                           | type_spec decl_specs

                                           | type_spec

                                           | type_qualifier decl_specs
                                           | type_qualifier

storage_class_spec              -> storage_const

type_spec                            -> type_const

                                           | struct_or_union_spec
                                           | enum_spec

type_qualifier                       -> qual_const

struct_or_union_spec            -> struct_or_union id '{' struct_decl_list '}' ';'
                                           | struct_or_union id

struct_or_union                      -> struct_const

struct_decl_list                -> struct_decl
                                           | struct_decl_list struct_decl

init_declarator_list            -> init_declarator
                                           | init_declarator_list ',' init_declarator

init_declarator                      -> declarator
                                           | declarator '=' initializer

struct_decl                          -> spec_qualifier_list struct_declarator_list ';'
```

```
spec_qualifier_list                    -> type_spec spec_qualifier_list
                                             | type_spec
                                             | type_qualifier spec_qualifier_list
                                             | type_qualifier

struct_declarator_list          -> struct_declarator
                                             | struct_declarator_list ','
struct_declarator

struct_declarator               -> declarator
                                             | declarator ':' conditional_exp
                                             | ':' conditional_exp

enum_spec                              -> enum_const id '{' enumerator_list '}'
                                             | enum_const '{' enumerator_list '}'
                                             | enum_const id

enumerator_list                        -> enumerator
                                             | enumerator_list ',' enumerator

enumerator                             -> id
                                             | id '=' conditional_exp

declarator                             -> pointer direct_declarator
                                             | direct_declarator

direct_declarator               -> id

                                             | '(' declarator ')'

                                             | direct_declarator '[' conditional_exp ']'

                                             | direct_declarator '['  ']'
                                             | direct_declarator '(' param_list ')'

                                             | direct_declarator '(' id_list ')'

                                             | direct_declarator '('  ')'


pointer                                      -> '*' type_qualifier_list
                                             | '*'
                                             | '*' type_qualifier_list pointer
                                             | '*' pointer

type_qualifier_list                    -> type_qualifier
                                             | type_qualifier_list type_qualifier

param_list                             -> param_decl
                                             | param_list ',' param_decl
```

```
param_decl                          -> decl_specs declarator
                                          | decl_specs abstract_declarator
                                          | decl_specs

id_list                                   -> id
                                          | id_list ',' id

initializer                         -> assignment_exp
                                          | '{' initializer_list '}'
                                          | '{' initializer_list ',' '}'

initializer_list                -> initializer
                                          | initializer_list ',' initializer

type_name                           -> spec_qualifier_list abstract_declarator
                                          | spec_qualifier_list

abstract_declarator                 -> pointer
                                          | pointer direct_abstract_declarator
                                          |     direct_abstract_declarator

direct_abstract_declarator      -> '(' abstract_declarator ')'
                                          | direct_abstract_declarator '['
conditional_exp ']'
                                          | '[' conditional_exp ']'
                                          | direct_abstract_declarator '[' ']'
                                          | '[' ']'
                                          | direct_abstract_declarator '(' param_list
')'
                                          | '(' param_list ')'
                                          | direct_abstract_declarator '(' ')'
                                          | '(' ')'

stat                                -> labeled_stat

                                          | exp_stat

                                          | compound_stat

                                          | selection_stat

                                          | iteration_stat
                                          | jump_stat

labeled_stat                    -> id ':' stat
                                          | CASE int_const ':' stat
                                          | DEFAULT ':' stat

exp_stat                            -> exp ';'
```

```
                                        | ';'

compound_stat                       -> '{' decl_list stat_list '}'

                                        | '{' stat_list '}'

                                        | '{' decl_list    '}'

                                        | '{' '}'


stat_list                           -> stat

                                        | stat_list stat


selection_stat                      -> IF '(' exp ')' stat
                          %prec "then"
                                        | IF '(' exp ')' stat ELSE stat
                                        | SWITCH '(' exp ')' stat

iteration_stat                      -> WHILE '(' exp ')' stat
                                        | DO stat WHILE '(' exp ')' ';'
                                        | FOR '(' exp ';' exp ';' exp ')' stat
                                        | FOR '(' exp ';' exp ';'       ')' stat
                                        | FOR '(' exp ';' ';' exp ')' stat
                                        | FOR '(' exp ';' ';' ')' stat
                                        | FOR '(' ';' exp ';' exp ')' stat
                                        | FOR '(' ';' exp ';' ')' stat
                                        | FOR '(' ';' ';' exp ')' stat
                                        | FOR '(' ';' ';' ')' stat

jump_stat                           -> GOTO id ';'
                                        | CONTINUE ';'
                                        | BREAK ';'
                                        | RETURN exp ';'
                                        | RETURN ';'

exp                                     -> assignment_exp
                                        | exp ',' assignment_exp

assignment_exp                      -> conditional_exp
                                        | unary_exp assignment_operator
assignment_exp

assignment_operator                 -> PUNC
                                        | '='

conditional_exp                     -> logical_or_exp
                                        | logical_or_exp '?' exp ':' conditional_exp
```

```
logical_or_exp                      -> logical_and_exp
                                          | logical_or_exp or_const logical_and_exp

logical_and_exp                     -> inclusive_or_exp
                                          | logical_and_exp and_const inclusive_or_exp

inclusive_or_exp                -> exclusive_or_exp
                                          | inclusive_or_exp '|' exclusive_or_exp

exclusive_or_exp                -> and_exp
                                          | exclusive_or_exp '^' and_exp

and_exp                                   -> equality_exp
                                          | and_exp '&' equality_exp

equality_exp                    -> relational_exp
                                          | equality_exp eq_const relational_exp

relational_exp                      -> shift_expression
                                          | relational_exp '<' shift_expression
                                          | relational_exp '>' shift_expression
                                          | relational_exp rel_const shift_expression

shift_expression                -> additive_exp
                                          | shift_expression shift_const additive_exp

additive_exp                    -> mult_exp
                                          | additive_exp '+' mult_exp
                                          | additive_exp '-' mult_exp

mult_exp                            -> cast_exp
                                          | mult_exp '*' cast_exp
                                          | mult_exp '/' cast_exp
                                          | mult_exp '%' cast_exp

cast_exp                            -> unary_exp
                                          | '(' type_name ')' cast_exp

unary_exp                           -> postfix_exp
                                          | inc_const unary_exp
                                          | unary_operator cast_exp
                                          | SIZEOF unary_exp
                                          | SIZEOF '(' type_name ')'

unary_operator                      -> '&' | '*' | '+' | '-' | '~' | '!'


postfix_exp                         -> primary_exp
```

```
                                        | postfix_exp '[' exp ']'
                                        | postfix_exp '(' argument_exp_list ')'
                                        | postfix_exp '(' ')'
                                        | postfix_exp '.' id
                                        | postfix_exp point_const id
                                        | postfix_exp inc_const

primary_exp                        -> id

                                        | consts

                                        | string

                                        | '(' exp ')'

argument_exp_list               -> assignment_exp
                                        | argument_exp_list ',' assignment_exp

consts                             -> int_const

                                        | char_const
                                        | float_const
                                        | enumeration_const

int_const                    -> [0-9]+

char_const                   -> "'"."'"



float_const                  -> [0-9]+"."[0-9]+

id                           -> [a-zA-z_][a-zA-z_0-9]*

string                       -> \".*\"

enum_const                   -> "enum"

storage_const                -> "auto"
                             | "register"
                             | "static"
                             | "extern"
                             | "typedef"
```

```
type_const                    -> "void"
                               |  "char"
                               |  "short"
                               |  "int"
                               |  "long"
                               |  "float"
                               |  "double"
                               |  "signed"
```

```
                              |  "unsigned"

qual_const                    -> "const"
                              |  "volatile"

struct_const                  -> "struct"
                              |  "union"

DEFINE                        -> "#define"[ ]+[a-zA-z_][a-zA-z_0-9]*

IF                            -> "if"

ELSE                          -> "else"

FOR                           -> "for"

DO                            -> "do"

WHILE                         -> "while"

BREAK                         -> "break"

SWITCH                        -> "switch"

CONTINUE                      -> "continue"

RETURN                        -> "return"

CASE                          -> "case"

DEFAULT                       -> "default"

GOTO                          -> "goto"

SIZEOF                        -> "sizeof"

PUNC                          -> "*="
                              |  "/="
                              |  "+="
                              |  "%="
                              |  ">>=""-
                              |  ="
                              |  "<<="
                              |  "&="
                              |  "^="
                              |  "|="

or_const                      -> "||"

and_const                     -> "&&"
```

```
eq_const                    -> "=="
                            |  "!="

shift_const                 -> ">>"
                            |  "<<"

rel_const                   -> "<="
                            |  ">="

inc_const                   -> "++"
                            |  "--"

point_const                 -> "->"

HEADER                      -> "#include"[ ]+<[a-zA-z_][a-zA-z_0-9.]*>
```

PRACTICAL 4

AIM

Develop a simple calculator using LEX and YACC tools.

CODE

cal.l

```
/* Definition section */
%{
#include<stdio.h>
#include "cal.tab.h"
extern int yylval;
%}

/* Rule Section */
%%
[0-9]+ {
            yylval=atoi(yytext);
            return NUMBER;

      }
[\t] ;
[\n] return 0;
. return yytext[0];
%%

/* Driver Section */
int yywrap()
{
      return 1;
}
```

cal.y

```
/* Definition section */
%{
#include<stdio.h>
int flag=0;
%}

%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'


/* Rule Section */
%%
ArithmeticExpression: E{
            printf("Answer : %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;}
;

%%

// Driver Code
void main()
{
      printf("\nEnter Expression : ");
      yyparse();
}

void yyerror(char *a)
{
      printf("Invalid Arithmetic Expression\n");
      flag=1;
}
```

OUTPUT

```
Enter the Expression : a+b*c
Three Address Code
T = b*c
T1 = a+T
```

PRACTICAL 5

AIM

Develop a Parser for “C” language using LEX and YACC tool

CODE

parser.l

```
%option yylineno

/* Definition Section */
%{
      #include<stdio.h>
      #include"parser.tab.h"
%}

/* Rules Section */
%%
"#include"[ ]+<[a-zA-z_][a-zA-z_0-9.]*>     {return HEADER;}
"#define"[ ]+[a-zA-z_][a-zA-z_0-9]* {return DEFINE;}
"auto"|"register"|"static"|"extern"|"typedef" {return storage_const;}
"void"|"char"|"short"|"int"|"long"|"float"|"double"|"signed"|"unsigned"     {return
type_const;}
"const"|"volatile" {return qual_const;}
"enum" {return enum_const;}
"struct"|"union" {return struct_const;}
"case" {return CASE;}
"default" {return DEFAULT;}
"if" {return IF;}
"switch" {return SWITCH;}
"else" {return ELSE;}
"for" {return FOR;}
"do" {return DO;}
"while" {return WHILE;}
"goto" {return GOTO;}
"continue" {return CONTINUE;}
"break"      {return BREAK;}
"return" {return RETURN;}
"sizeof" {return SIZEOF;}
"||" {return or_const;}
"&&" {return and_const;}
"=="|"!=" {return eq_const;}
"<="|">=" {return rel_const;}
">>"|"<<" {return shift_const;}
"++"|"--" {return inc_const;}
"->" {return point_const;}
"*="|"/="|"+="|"%="|">>="|"-="|"<<="|"&="|"^="|"|="      {return PUNC;}
```

```
[0-9]+ {return int_const;}
[0-9]+"."[0-9]+    {return float_const;}
"'"."'"      {return char_const;}
[a-zA-z_][a-zA-z_0-9]* {return id;}
\".*\" {return string;}
"//"(\\.|[^\n])*[\n]
                                                                                ;
[/][*]([^*]|[*]*[^*/])*[*]+[/]
;
[ \t\n]
                                                                                ;
";"|"="|","|"{"|"}"|"("|")"|"["|"]"|"*"|"+"|"-
"|"/"|"?"|":"|"&"|"|"|"^"|"!"|"~"|"%"|"<"|">"                                {return
yytext[0];}
%%

/* User Code Section */
int yywrap(void)
{
    return 1;
}
```

parser.y

```
%{
      #include<stdio.h>
      int yylex(void);
      int yyerror(const char *s);
      int success = 1;
%}

%token int_const char_const float_const id string storage_const type_const qual_const
struct_const enum_const DEFINE
%token IF FOR DO WHILE BREAK SWITCH CONTINUE RETURN CASE DEFAULT GOTO SIZEOF PUNC
or_const and_const eq_const shift_const rel_const inc_const
%token point_const ELSE HEADER
%left '+' '-'
%left '*' '/'
%right UMINUS
%nonassoc "then"
%nonassoc ELSE

%start program_unit
%%
program_unit                    : HEADER program_unit
                                            | DEFINE primary_exp program_unit

                                            | translation_unit


                                            ;
translation_unit                : external_decl

                                            | translation_unit external_decl
```

```
                                        ;
external_decl                     : function_definition
                                        | decl
                                        ;
function_definition               : decl_specs declarator decl_list compound_stat
                                        | declarator decl_list compound_stat
                                        | decl_specs declarator  compound_stat

                                        | declarator compound_stat
                                        ;
decl                              : decl_specs init_declarator_list ';'

                                        | decl_specs ';'
                                        ;
decl_list                         : decl
                                        | decl_list decl
                                        ;
decl_specs                        : storage_class_spec decl_specs
                                        | storage_class_spec
                                        | type_spec decl_specs

                                        | type_spec

                                        | type_qualifier decl_specs
                                        | type_qualifier
                                        ;
storage_class_spec              : storage_const
                                        ;
type_spec                         : type_const

                                        | struct_or_union_spec
                                        | enum_spec
                                        ;
type_qualifier                    : qual_const
                                        ;
struct_or_union_spec            : struct_or_union id '{' struct_decl_list '}' ';'
                                        | struct_or_union id
                                        ;
struct_or_union                   : struct_const
                                        ;
 struct_decl_list               : struct_decl
                                        | struct_decl_list struct_decl
                                        ;
init_declarator_list            : init_declarator
                                        | init_declarator_list ',' init_declarator
                                        ;
init_declarator                   : declarator
                                        | declarator '=' initializer
                                        ;
struct_decl                       : spec_qualifier_list struct_declarator_list ';'
                                        ;
spec_qualifier_list               : type_spec spec_qualifier_list
                                        | type_spec
                                        | type_qualifier spec_qualifier_list
                                        | type_qualifier
                                        ;
struct_declarator_list          : struct_declarator
```

```
                                        | struct_declarator_list ','
struct_declarator
                                        ;
struct_declarator               : declarator
                                        | declarator ':' conditional_exp
                                        | ':' conditional_exp
                                        ;
enum_spec                           : enum_const id '{' enumerator_list '}'
                                        | enum_const '{' enumerator_list '}'
                                        | enum_const id
                                        ;
enumerator_list                     : enumerator
                                        | enumerator_list ',' enumerator
                                        ;
enumerator                          : id
                                        | id '=' conditional_exp
                                        ;
declarator                          : pointer direct_declarator
                                        | direct_declarator
                                        ;
direct_declarator               : id

                                        | '(' declarator ')'

                                        | direct_declarator '[' conditional_exp ']'

                                        | direct_declarator '['  ']'
                                        | direct_declarator '(' param_list ')'

                                        | direct_declarator '(' id_list ')'

                                        | direct_declarator '('  ')'


                                        ;
pointer                                 : '*' type_qualifier_list
                                        | '*'
                                        | '*' type_qualifier_list pointer
                                        | '*' pointer
                                        ;
type_qualifier_list                 : type_qualifier
                                        | type_qualifier_list type_qualifier
                                        ;
param_list                          : param_decl
                                        | param_list ',' param_decl
                                        ;
param_decl                          : decl_specs declarator
                                        | decl_specs abstract_declarator
                                        | decl_specs
                                        ;
id_list                                 : id
                                        | id_list ',' id
                                        ;
initializer                         : assignment_exp
                                        | '{' initializer_list '}'
                                        | '{' initializer_list ',' '}'
                                        ;
initializer_list                : initializer
                                        | initializer_list ',' initializer
```

```
                                        ;
type_name                           : spec_qualifier_list abstract_declarator
                                        | spec_qualifier_list
                                        ;
abstract_declarator                 : pointer
                                        | pointer direct_abstract_declarator
                                        |     direct_abstract_declarator
                                        ;
direct_abstract_declarator      : '(' abstract_declarator ')'
                                        | direct_abstract_declarator '['
conditional_exp ']'
                                        | '[' conditional_exp ']'
                                        | direct_abstract_declarator '[' ']'
                                        | '[' ']'
                                        | direct_abstract_declarator '(' param_list
')'
                                        | '(' param_list ')'
                                        | direct_abstract_declarator '(' ')'
                                        | '(' ')'
                                        ;
stat                                : labeled_stat

                                        | exp_stat

                                        | compound_stat

                                        | selection_stat

                                        | iteration_stat
                                        | jump_stat
                                        ;
labeled_stat                    : id ':' stat
                                        | CASE int_const ':' stat
                                        | DEFAULT ':' stat
                                        ;
exp_stat                            : exp ';'
                                        | ';'
                                        ;
compound_stat                       : '{' decl_list stat_list '}'

                                        | '{' stat_list '}'

                                        | '{' decl_list    '}'

                                        | '{' '}'

                                        ;
stat_list                           : stat

                                        | stat_list stat

                                        ;
selection_stat                      : IF '(' exp ')' stat
                           %prec "then"
                                        | IF '(' exp ')' stat ELSE stat
                                        | SWITCH '(' exp ')' stat
                                        ;
iteration_stat                      : WHILE '(' exp ')' stat
```

```
                                        | DO stat WHILE '(' exp ')' ';'
                                        | FOR '(' exp ';' exp ';' exp ')' stat
                                        | FOR '(' exp ';' exp ';'       ')' stat
                                        | FOR '(' exp ';' ';' exp ')' stat
                                        | FOR '(' exp ';' ';' ')' stat
                                        | FOR '(' ';' exp ';' exp ')' stat
                                        | FOR '(' ';' exp ';' ')' stat
                                        | FOR '(' ';' ';' exp ')' stat
                                        | FOR '(' ';' ';' ')' stat
                                        ;
jump_stat                         : GOTO id ';'
                                        | CONTINUE ';'
                                        | BREAK ';'
                                        | RETURN exp ';'
                                        | RETURN ';'
                                        ;
exp                                     : assignment_exp
                                        | exp ',' assignment_exp
                                        ;
assignment_exp                    : conditional_exp
                                        | unary_exp assignment_operator
assignment_exp
                                        ;
assignment_operator               : PUNC
                                        | '='
                                        ;
conditional_exp                   : logical_or_exp
                                        | logical_or_exp '?' exp ':' conditional_exp
                                        ;
logical_or_exp                    : logical_and_exp
                                        | logical_or_exp or_const logical_and_exp
                                        ;
logical_and_exp                   : inclusive_or_exp
                                        | logical_and_exp and_const inclusive_or_exp
                                        ;
inclusive_or_exp              : exclusive_or_exp
                                        | inclusive_or_exp '|' exclusive_or_exp
                                        ;
exclusive_or_exp              : and_exp
                                        | exclusive_or_exp '^' and_exp
                                        ;
and_exp                                 : equality_exp
                                        | and_exp '&' equality_exp
                                        ;
equality_exp                  : relational_exp
                                        | equality_exp eq_const relational_exp
                                        ;
relational_exp                    : shift_expression
                                        | relational_exp '<' shift_expression
                                        | relational_exp '>' shift_expression
                                        | relational_exp rel_const shift_expression
                                        ;
shift_expression              : additive_exp
                                        | shift_expression shift_const additive_exp
                                        ;
additive_exp                  : mult_exp
                                        | additive_exp '+' mult_exp
                                        | additive_exp '-' mult_exp
```

```
                                        ;
mult_exp                            : cast_exp
                                        | mult_exp '*' cast_exp
                                        | mult_exp '/' cast_exp
                                        | mult_exp '%' cast_exp
                                        ;
cast_exp                            : unary_exp
                                        | '(' type_name ')' cast_exp
                                        ;
unary_exp                           : postfix_exp
                                        | inc_const unary_exp
                                        | unary_operator cast_exp
                                        | SIZEOF unary_exp
                                        | SIZEOF '(' type_name ')'
                                        ;
unary_operator                      : '&' | '*' | '+' | '-' | '~' | '!'

                                        ;
postfix_exp                         : primary_exp

                                        | postfix_exp '[' exp ']'
                                        | postfix_exp '(' argument_exp_list ')'
                                        | postfix_exp '(' ')'
                                        | postfix_exp '.' id
                                        | postfix_exp point_const id
                                        | postfix_exp inc_const
                                        ;
primary_exp                         : id


                                        | consts

                                        | string

                                        | '(' exp ')'
                                        ;
argument_exp_list               : assignment_exp
                                        | argument_exp_list ',' assignment_exp
                                        ;
consts                              : int_const

                                        | char_const
                                        | float_const
                                        | enum_const
                                        ;
%%

int main()
{

    yyparse();
    if(success)
      printf("Successfully Parsed\n");
    return 0;
}
```

```
int yyerror(const char *msg)
{
	extern int yylineno;
	printf("Parsing Failed\nLine Number: %d %s\n",yylineno,msg);
	success = 0;
	return 0;
}
```

OUTPUT

Code.c

Start here | Code.c

```c
#include <stdio.h>
#define E 2.718
#define PI 3.14159

// Function to Calculate the Area of a Circle
float AreaOfCircle(float radius)
{
    return PI * radius * radius;
}

// Main Function
int main()
{
    int radius = 10;
    double area = AreaOfCircle(radius);
    printf("Area of Circle with Radius %d is %f", radius, area);
    return 0;
}
```

Output Generated

```
Successfully Parsed
```

PRACTICAL 6

AIM

Add assignment statement, if then else statement and while loop to the calculator and generate the three-address code for the same.

CODE

```
#include <stdio.h>
#include <string.h>

int i, choice, j, l, address = 100;
char userInput[10], expr[10], expr1[10], expr2[10], id1[5], op[5], id2[5];

int main()
{
	printf("Enter the Expression : ");
	scanf("%s", userInput);
	strcpy(expr, userInput);
	l = strlen(expr);
	expr1[0] = '\0';

	for (i = 0; i < 2; i++)
	{
		if (expr[i] == '+' || expr[i] == '-')
		{
			if (expr[i + 2] == '/' || expr[i + 2] == '*')
			{
				strrev(expr);
				j = l - i - 1;
				strncat(expr1, expr, j);
				strrev(expr1);
				printf("Three Address Code\nT = %s\nT1 = %c%cT\n", expr1,
expr[j + 1], expr[j]);
			}

			else
			{
				strncat(expr1, expr, i + 2);
			printf("Three Address Code\nT = %s\nT1 = T%c%c\n", expr1, expr[i +
2], expr[i + 3]);
			}
		}
```

```
            else if (expr[i] == '/' || expr[i] == '*')
            {
                  strncat(expr1, expr, i + 2);
            printf("Three Address Code\nT = %s\nT1 = T%c%c\n", expr1, expr[i + 2],
expr[i + 3]);
            }
      }

      return 0;
}
```

OUTPUT

```
Enter the Expression : a+b*c
Three Address Code
T = b*c
T1 = a+T
```