Compiler and Translator Design (ITITC20) Practical File SEM-6

SUBMITTED BY:- Ansh Sarin
2019UIT3100

Write a program to implement a DFA to recognize the identifiers, keywords, constants, and comments of C language.

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <string>
#include <iostream>
using namespace std;

// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
{
   if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
      ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
      ch == '[' || ch == ']' || ch == '{' || ch == '}')
      return (true);
   return (false);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
   if (ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == '>' || ch == '<' ||
      ch == '=')
      return (true);
   return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str)
```

```
{
   if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
      str[0] == '3' || str[0] == '4' || str[0] == '5' ||
      str[0] == '6' || str[0] == '7' || str[0] == '8' ||
      str[0] == '9' || isDelimiter(str[0]) == true)
      return (false);
   return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str)
{
   if (!strcmp(str, "if") || !strcmp(str, "else") ||
      !strcmp(str, "while") || !strcmp(str, "do") ||
      !strcmp(str, "break") ||
       !strcmp(str, "continue") || !strcmp(str, "int")
      || !strcmp(str, "double") || !strcmp(str, "float")
      || !strcmp(str, "return") || !strcmp(str, "char")
      || !strcmp(str, "case") || !strcmp(str, "char")
      || !strcmp(str, "sizeof") || !strcmp(str, "long")
      || !strcmp(str, "short") || !strcmp(str, "typedef")
      || !strcmp(str, "switch") || !strcmp(str, "unsigned")
      || !strcmp(str, "void") || !strcmp(str, "static")
      || !strcmp(str, "struct") || !strcmp(str, "goto"))
      return (true);
   return (false);
}

// Returns 'true' if the string is an INTEGER.
bool isInteger(char* str)
{
   int i, len = strlen(str);

   if (len == 0)
      return (false);
   for (i = 0; i < len; i++) {
```

```
    if (str[i] != '0' && str[i] != '1' && str[i] != '2'
        && str[i] != '3' && str[i] != '4' && str[i] != '5'
        && str[i] != '6' && str[i] != '7' && str[i] != '8'
        && str[i] != '9' || (str[i] == '-' && i > 0))
        return (false);
  }
  return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char* str)
{
  int i, len = strlen(str);
  bool hasDecimal = false;

  if (len == 0)
    return (false);
  for (i = 0; i < len; i++) {
    if (str[i] != '0' && str[i] != '1' && str[i] != '2'
        && str[i] != '3' && str[i] != '4' && str[i] != '5'
        && str[i] != '6' && str[i] != '7' && str[i] != '8'
        && str[i] != '9' && str[i] != '.' ||
        (str[i] == '-' && i > 0))
        return (false);
    if (str[i] == '.')
        hasDecimal = true;
  }
  return (hasDecimal);
}

// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
  int i;
  char* subStr = (char*)malloc(
            sizeof(char) * (right - left + 2));
```

```
    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

// Parsing the input STRING.
void parse(char* str)
{
    int left = 0, right = 0;
    int len = strlen(str);

    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false)
            right++;

        if (isDelimiter(str[right]) == true && left == right) {
            if (isOperator(str[right]) == true)
                printf("'%c' IS AN OPERATOR\n", str[right]);

            right++;
            left = right;
        } else if (isDelimiter(str[right]) == true && left != right
                || (right == len && left != right)) {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)
                printf("'%s' IS A KEYWORD\n", subStr);

            else if (isInteger(subStr) == true)
                printf("'%s' IS AN INTEGER\n", subStr);

            else if (isRealNumber(subStr) == true)
                printf("'%s' IS A REAL NUMBER\n", subStr);
```

```
        else if (validIdentifier(subStr) == true
                && isDelimiter(str[right - 1]) == false)
            printf("'%s' IS A VALID IDENTIFIER\n", subStr);

        else if (validIdentifier(subStr) == false
                && isDelimiter(str[right - 1]) == false)
            printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
        left = right;
      }
   }
   return;
}

// DRIVER FUNCTION
int main()
{
    // maximum length of string is 100 here
  // char str[100] = "int a = b + 1c; ";

   char str[80];

   cin.getline(str,80);

   parse(str); // calling the parse function

   return (0);
}
```

OUTPUT->

```
anshs@Ansh MINGW64 ~/Desktop/compiler Prac/DetectTockens
$ ./a
int a=b+c
'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'b' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'c' IS A VALID IDENTIFIER
```

Write a program for predictive parse.

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>
#define SIZE 128
#define NONE -1
#define EOS '\0'
#define NUM 257
#define KEYWORD 258
#define ID 259
#define DONE 260
#define MAX 999

char lexemes[MAX];
char buffer[SIZE];
int lastchar=-1;
int lastentry=0;
int tokenval=DONE;
int lineno=1;
int lookahead;
void E();
struct entry
{
  char *lexptr;
  int token;
}
symtable[100];
struct entry
     keywords[]=
{"if",KEYWORD,"else",KEYWORD,"for",KEYWORD,"int",KEYWORD,"float",KEYWORD,
          "double",KEYWORD,"char",KEYWORD,"struct",KEYWORD,"return",KEYWORD,0,0
};
void Error_Message(char *m)
{
```

```
   fprintf(stderr,"line %d, %s \n",lineno,m);
   exit(1);
}
int look_up(char s[ ])
{
   int k;
   for(k=lastentry; k>0; k--)
      if(strcmp(symtable[k].lexptr,s)==0)
         return k;
   return 0;
}
int insert(char s[ ],int tok)
{
   int len;
   len=strlen(s);
   if(lastentry+1>=MAX)
      Error_Message("Symbpl table is full");
   if(lastchar+len+1>=MAX)
      Error_Message("Lexemes array is full");
   lastentry=lastentry+1;
   symtable[lastentry].token=tok;
   symtable[lastentry].lexptr=&lexemes[lastchar+1];
   lastchar=lastchar+len+1;
   strcpy(symtable[lastentry].lexptr,s);
   return lastentry;
}
/*void Initialize()
{
   struct entry *ptr;
   for(ptr=keywords;ptr->token;ptr+1)
      insert(ptr->lexptr,ptr->token);
}*/
int lexer()
{
   int t;
   int val,i=0;
```

```
while(1)
{
  t=getchar();
  if(t==' '||t=='\t');
  else if(t=='\n')
    lineno=lineno+1;
  else if(isdigit(t))
  {
    ungetc(t,stdin);
    scanf("%d",&tokenval);
    return NUM;
  }
  else if(isalpha(t))
  {
    while(isalnum(t))
    {
      buffer[i]=t;
      t=getchar();
      i=i+1;
      if(i>=SIZE)
        Error_Message("Compiler error");
    }
    buffer[i]=EOS;
    if(t!=EOF)
      ungetc(t,stdin);
    val=look_up(buffer);
    if(val==0)
      val=insert(buffer,ID);
    tokenval=val;
    return symtable[val].token;
  }
  else if(t==EOF)
    return DONE;
  else
  {
    tokenval=NONE;
```

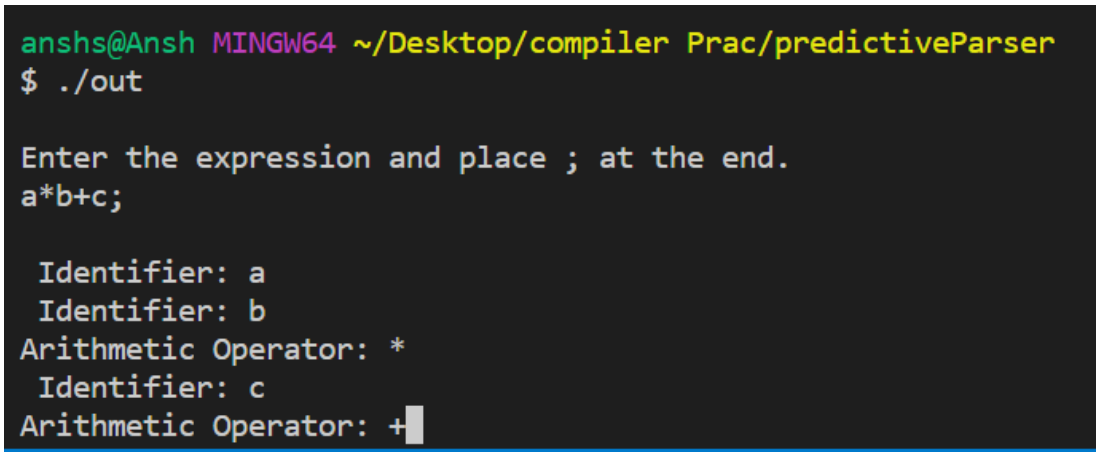
```
        return t;
      }
    }
}
void Match(int t)
{
   if(lookahead==t)
      lookahead=lexer();
   else
      Error_Message("Syntax error");
}
void display(int t,int tval)
{
   if(t=='+'||t=='-'||t=='*'||t=='/')
      printf("\nArithmetic Operator: %c",t);
   else if(t==NUM)
      printf("\n Number: %d",tval);
   else if(t==ID)
      printf("\n Identifier: %s",symtable[tval].lexptr);
   else
      printf("\n Token %d tokenval %d",t,tokenval);
}
void F()
{

   switch(lookahead)
   {
   case '(' :
      Match('(');
      E();
      Match(')');
      break;
   case NUM :
      display(NUM,tokenval);
      Match(NUM);
      break;
```

```
    case ID :
      display(ID,tokenval);
      Match(ID);
      break;
    default :
      Error_Message("Syntax error");
    }
}
void T()
{
    int t;
    F();
    while(1)
    {
      switch(lookahead)
      {
      case '*' :
        t=lookahead;
        Match(lookahead);
        F();
        display(t,NONE);
        continue;
      case '/' :
        t=lookahead;
        Match(lookahead);
        display(t,NONE);
        continue;
      default :
        return;
      }
    }
}
void E()
{
    int t;
    T();
```

```
  while(1)
  {
    switch(lookahead)
    {
    case '+' :
      t=lookahead;
      Match(lookahead);
      T();
      display(t,NONE);
      continue;
    case '-' :
      t=lookahead;
      Match(lookahead);
      T();
      display(t,NONE);
      continue;
    default :
      return;
    }
  }
}
void parser()
{
  lookahead=lexer();
  while(lookahead!=DONE)
  {
    E();
    Match(';');
  }
}
int main()
{
  char ans[10];
  printf("\nEnter the expression and place ; at the end.\n");
  parser();
        return 0;
```

}

OUTPUT->



Write a program to convert infix to postfix using Lex and Yacc.

Main.l->

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

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

Main.y->

```
%{
#include<stdio.h>
%}
%token NUM
%left '+' '-'
%left '*' '/'
%right NEGATIVE
%%
S:  E {printf("\n");}
   ;
E:  E '+' E {printf("+");}
   |  E '*' E {printf("*");}
   |  E '-' E {printf("-");}
   |  E '/' E {printf("/");}
   |  '(' E ')'
   |  '-' E %prec NEGATIVE {printf("-");}
   |  NUM     {printf("%d", yylval);}
   ;
%%

int main(){
   yyparse();
}

int yyerror (char *msg) {
   return printf ("error YACC: %s\n", msg);
}
```

OUTPUT->

```
anshs@Ansh MINGW64 ~/Desktop/compiler Prac/infixTopostfix
$ ./a
2+3/5*7-4
235/7*+4-
```

Write a program to implement symbols table.

```
//Implementation of symbol table
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
int main()
{
 int i=0,j=0,x=0,n;
 void *p,*add[5];
 char ch,srch,b[15],d[15],c;
 printf("Expression terminated by $:");
 while((c=getchar())!='$')
 {
  b[i]=c;
  i++;
 }
 n=i-1;
 printf("Given Expression:");
 i=0;
 while(i<=n)
 {
  printf("%c",b[i]);
  i++;
 }
 printf("\n Symbol Table\n");
 printf("Symbol \t addr \t type");
 while(j<=n)
 {
  c=b[j];
  if(isalpha(toascii(c)))
  {
   p=malloc(c);
   add[x]=p;
```

```
 d[x]=c;
 printf("\n%c \t %d \t identifier\n",c,p);
 x++;
 j++;
}
else
{
 ch=c;
 if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
 {
  p=malloc(ch);
  add[x]=p;
  d[x]=ch;
  printf("\n %c \t %d \t operator\n",ch,p);
  x++;
  j++;
 }}}
 return 0;}
```

OUTPUT->

```
anshs@Ansh MINGW64 ~/Desktop/compiler Prac/symbolTable
$ ./out
Expression terminated by $:n=v+t$
Given Expression:n=v+t
 Symbol Table
Symbol    addr     type
n         17190288         identifier

 =        17190784         operator

v         17190408         identifier

 +        17190536         operator

t         17177704         identifier
```

Write a program to implement simple calculator using Lex and Yacc.

Main.l->

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

%%

[0-9]+ {yylval=atoi(yytext); return NUMBER;}

">=" return GE;
"<=" return LE;
"!=" return NE;
"==" return EQ;
[\n] return 0;
[\t];

. return yytext[0];

%%
```

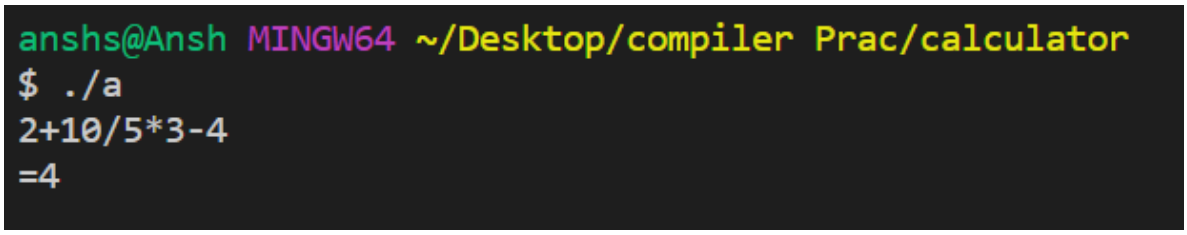
Main.y->

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

%token NAME NUMBER     // Declaration of Names token
%left GE LE NE EQ '<' '>' '%'              // Asscociativity
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS
```

```
%%

statement : NAME '=' exp

          |exp {printf("=%d\n",$1);}
          ;

exp : NUMBER {$$ == $1;}
          |exp '+' exp {$$ = $1 + $3 ;}
          |exp '-' exp {$$ = $1 - $3 ;}
          |exp '*' exp {$$ = $1 * $3 ;}
          |exp '/' exp {$$ = $1 / $3 ;}
          |exp '<' exp {$$ = $1 < $3 ;}
          |exp '>' exp {$$ = $1 > $3 ;}
          |exp '%' exp {$$ = $1 % $3 ;}
          |exp GE exp {$$ = $1 >= $3 ;}
          |exp LE exp {$$ = $1 <= $3 ;}
          |exp EQ exp {$$ = $1 == $3 ;}
          |exp NE exp {$$ = $1 != $3 ;}
          |exp '-' exp %prec UMINUS {$$ = -$2 ;}
          |'(' exp ')' {$$ = $2 ;}
;

%%

int main()
{
yyparse();
}

int yyerror()
{
}

int yywrap()
```

```
{
return 1;
}
```

OUTPUT->



Write a program to implement lexical analyzer for C language.

Main.l->

```
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.*\n {printf("%sThis is a PREPROCESSOR DIRECTIVE\n",yytext);}
auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|reg
ister|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while
{printf("\n%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
"*/" {COMMENT = 0;}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION: \n%s",yytext);}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n%s is an IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n%s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n%s is a NUMBER ",yytext);}
\{ {if(!COMMENT) printf("\nBLOCK BEGINS");}
\} {if(!COMMENT) printf("\nBLOCK ENDS");}
\) {if(!COMMENT);printf("\n)");}
```

```
= {if(!COMMENT) printf("\n%s is an ASSIGNMENT OPERATOR",yytext);}
\<= | \>= | \< | \== | \!= | \> {if(!COMMENT) printf("\n%s is a RELATIONAL
OPERATOR",yytext);}
\, | \; {if(!COMMENT) printf("\n%s is a SEPERATOR",yytext);}
%%
int main(int argc, char **argv)
{
FILE *file;
file=fopen("c_lex_analyser.txt","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{ return(1);
}
```

OUTPUT->

```
anshs@Ansh MINGW64 ~/Desktop/compiler Prac/lexicalAnalyserForC
$ ./a

void is a KEYWORD
FUNCTION:
main(
int is a KEYWORD
a is an IDENTIFIER
)

BLOCK BEGINS


int is a KEYWORD
a is an IDENTIFIER,
b is an IDENTIFIER,
c is an IDENTIFIER;

a is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
1 is a NUMBER ;

b is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
2 is a NUMBER ;

if is a KEYWORD (
a is an IDENTIFIER >
b is an IDENTIFIER
)

c is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
0 is a NUMBER ;

else is a KEYWORD

c is an IDENTIFIER
```

```
FUNCTION:
printf(
"The value of c: %d" is a STRING,
c is an IDENTIFIER
);

for is a KEYWORD (
int is a KEYWORD
i is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
0 is a NUMBER ;
i is an IDENTIFIER <
5 is a NUMBER ;
i is an IDENTIFIER++
)

i is an IDENTIFIER++;

return is a KEYWORD
0 is a NUMBER ;

BLOCK ENDS
```

Write a program to implement parser for C language.

Main.l->

```
%option yylineno

%{
	#include<stdio.h>
	#include"y.tab.h"
%}


%%
"#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];}
%%

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

Main.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("Parsing Successfull\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->

```
anshs@Ansh MINGW64 ~/Desktop/compiler Prac/parser
$ ./a
int a=b+c;
^Z
Parsing Successfull
```

Generate three address code for selected C statements.

Main.l->

```
%{
#include"y.tab.h"
extern char yyval;
%}

%%

[0-9]+ {yylval.symbol=(char)(yytext[0]);return NUMBER;}
[a-z] {yylval.symbol= (char)(yytext[0]);return LETTER;}
. {return yytext[0];}
\n {return 0;}

%%
```

Main.y->

```
%{
#include"y.tab.h"
#include<stdio.h>
char addtotable(char,char,char);


int index1=0;
char temp = 'A'-1;


struct expr{

char operand1;
char operand2;
char operator;
char result;
};
```

```
%}

%union{
char symbol;
}




%left '+' '-'
%left '/' '*'

%token <symbol> LETTER NUMBER
%type <symbol> exp
%%

statement: LETTER '=' exp ';' {addtotable((char)$1,(char)$3,'=');};
exp: exp '+' exp {$$ = addtotable((char)$1,(char)$3,'+');}
   |exp '-' exp {$$ = addtotable((char)$1,(char)$3,'-');}
   |exp '/' exp {$$ = addtotable((char)$1,(char)$3,'/');}
   |exp '*' exp {$$ = addtotable((char)$1,(char)$3,'*');}
   |'(' exp ')' {$$= (char)$2;}
   |NUMBER {$$ = (char)$1;}
   |LETTER {(char)$1;};

%%

struct expr arr[20];

void yyerror(char *s){
   printf("Errror %s",s);
}

char addtotable(char a, char b, char o){
```

```
    temp++;
    arr[index1].operand1 =a;
    arr[index1].operand2 = b;
    arr[index1].operator = o;
    arr[index1].result=temp;
    index1++;
    return temp;
}

void threeAdd(){

    int i=0;
    char temp='A';
    while(i<index1){
        printf("%c:=\t",arr[i].result);
        printf("%c\t",arr[i].operand1);
        printf("%c\t",arr[i].operator);
        printf("%c\t",arr[i].operand2);
        i++;
        temp++;
        printf("\n");
    }
}

int yywrap(){
    return 1;
}

int main(){
    printf("Enter the expression: ");
    yyparse();
    printf("\n");
    threeAdd();
    printf("\n");
    return 0;
}
```

OUTPUT->

```
anshs@Ansh MINGW64 ~/Desktop/compiler Prac/threeAddressCode
$ ./a
Enter the expression: a=b+c;

A:=     b       +       c
B:=     a       =       A
```