Compiler and Translator Design-INITC20 Practical File

NAME: Ravneet Singh Oberoi
ROLL NO.: 2021UIN3365
BRANCH: ITNS

INDEX

Practical 1

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

Code:-

```
#include <bits/stdc++.h>
using namespace std;
int isKeyword(char buffer[])
{
char keywords[32][10] =
{"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",
"unsigned", "void", "volatile", "while"};
int i, flag = 0;

for (i = 0; i < 32; ++i)
{
if (strcmp(keywords[i], buffer) == 0)
{
flag = 1;
break;
}
}
return flag;
}
int main()
{
   char ch, buffer[15], b[30], logical_op[] = "><", math_op[] = "+-*/=", numer[] =
".0123456789", other[] = ",;\(){}[]'':";
ifstream fin("lexicalinput.txt");
int mark[1000] = {0};
int i, j = 0, kc = 0, ic = 0, lc = 0, mc = 0, nc = 0, oc = 0, aaa = 0;
vector<string> k;
vector<char> id;
vector<char> lo;
vector<char> ma;
vector<string> nu;
vector<char> ot;
if (!fin.is_open())
{
cout << "error while opening the file\n";
```

```
 exit(0);
}

while (!fin.eof())
{
ch = fin.get();
for (i = 0; i < 12; ++i)
{
          if (ch == other[i])
{
int aa = ch;
if (mark[aa] != 1)
{
                   ot.push_back(ch);
mark[aa] = 1;
++oc;
}
}
}

for (i = 0; i < 5; ++i)
{
if (ch == math_op[i])
{
int aa = ch;
if (mark[aa] != 1)
{
                 ma.push_back(ch);
mark[aa] = 1;
++mc;
}
}
}
for (i = 0; i < 2; ++i)
{
if (ch == logical_op[i])
{
int aa = ch;
if (mark[aa] != 1)
{
lo.push_back(ch);
mark[aa] = 1;
++lc;
}
}
}
```

```
 if (ch == '0' || ch == '1' || ch == '2' || ch == '3' || ch == '4' || ch == '5' || ch == '6' || ch == '7' || ch == '8' || ch == '9' || ch == '.' || ch == ' ' || ch == '\n' || ch == ';')
{
            if (ch == '0' || ch == '1' || ch == '2' || ch == '3' || ch == '4' || ch == '5' || ch == '6' || ch == '7' || ch == '8' || ch == '9' || ch == '.')
b[aaa++] = ch;
if ((ch == ' ' || ch == '\n' || ch == ';') && (aaa != 0))
{
b[aaa] = '\0';
aaa = 0;
char arr[30];
strcpy(arr, b);
nu.push_back(arr);
++nc;
}
}

if (isalnum(ch))
{
buffer[j++] = ch;
}
else if ((ch == ' ' || ch == '\n') && (j != 0))
{
buffer[j] = '\0';
j = 0;

if (isKeyword(buffer) == 1)
{
k.push_back(buffer);
++kc;
}
else
{
if (buffer[0] >= 97 && buffer[0] <= 122)
{
if (mark[buffer[0] - 'a'] != 1)
{
id.push_back(buffer[0]);
++ic;
mark[buffer[0] - 'a'] = 1;
}
}
}
}
}

fin.close();
```

```
 printf("Keywords: ");
for (int f = 0; f < kc; ++f)
{
if (f == kc - 1)
{
cout << k[f] << "\n";
}
else
{
cout << k[f] << ", ";
}
}
printf("Identifiers: ");
for (int f = 0; f < ic; ++f)
{
if (f == ic - 1)
{
cout << id[f] << "\n";
}
else
{
cout << id[f] << ", ";
}
}
   printf("Math Operators: ");
for (int f = 0; f < mc; ++f)
{
if (f == mc - 1)
{
cout << ma[f] << "\n";
}
else
{
cout << ma[f] << ", ";
}
}
    printf("Logical Operators: ");
for (int f = 0; f < lc; ++f)
{
if (f == lc - 1)
{
cout << lo[f] << "\n";
}
else
{
cout << lo[f] << ", ";
}
```

```
 }
   printf("Numerical Values: ");
for (int f = 0; f < nc; ++f)
{
if (f == nc - 1)
{
cout << nu[f] << "\n";
}
else
{
cout << nu[f] << ", ";
}
}
printf("Others: ");
for (int f = 0; f < oc; ++f)
{
if (f == oc - 1)
{
cout << ot[f] << "\n";
}
else
{
cout << ot[f] << " ";
}
}

return 0;
}
```

Output:-

lexicalinput.txt:-

```
int a, b, c;
float d, e;
a = b = 5;
c = 6;
if ( a > b)
{
c = a - b;
e = d - 2.0;
}
else
{
d = e + 6.0;
b = a + c;
}
```

```
Keywords: int, float, if, else
Identifiers: a, b, c, d, e
Math Operators: =, -, +
Logical Operators: >
Numerical Values: 5, 6, 2.0, 6.0
Others: , ; ( ) { }
```

Practical 2

Write a program for predictive parser.

Code:-

```
#include<iostream>
#include<string>
#include<deque>
using namespace std;
int n,n1,n2;
int getPosition(string arr[], string q, int size)
{
for(int i=0;i<size;i++)
{
if(q == arr[i])
return i;
}
return -1;
}
int main()
{
   string prods[10],first[10],follow[10],nonterms[10],terms[10];
string pp_table[20][20] = {};
cout<<"Enter the number of productions : ";
cin>>n;
cin.ignore();
cout<<"Enter the productions"<<endl;
for(int i=0;i<n;i++)
{
getline(cin,prods[i]);
cout<<"Enter first for "<<prods[i].substr(3)<<" : ";
getline(cin,first[i]);
}
cout<<"Enter the number of Terminals : ";
cin>>n2;
cin.ignore();
cout<<"Enter the Terminals"<<endl;
for(int i=0;i<n2;i++)
{
cin>>terms[i];
}
terms[n2] = "$";
n2++;
cout<<"Enter the number of Non-Terminals : ";
cin>>n1;
cin.ignore();
for(int i=0;i<n1;i++)
{
```

```
 cout<<"Enter Non-Terminal : ";
getline(cin,nonterms[i]);
cout<<"Enter follow of "<<nonterms[i]<<" : ";
getline(cin,follow[i]);
}


cout<<endl;
cout<<"Grammar"<<endl;
for(int i=0;i<n;i++)
{
cout<<prods[i]<<endl;
}


for(int j=0;j<n;j++)
{
int row = getPosition(nonterms,prods[j].substr(0,1),n1);
if(prods[j].at(3)!='#')
{
for(int i=0;i<first[j].length();i++)
{
int col = getPosition(terms,first[j].substr(i,1),n2);
pp_table[row][col] = prods[j];
}
}
else
{
for(int i=0;i<follow[row].length();i++)
{
        int col = getPosition(terms,follow[row].substr(i,1),n2);
pp_table[row][col] = prods[j];
}
}
}
//Display Table
for(int j=0;j<n2;j++)
cout<<"\t"<<terms[j];
cout<<endl;
for(int i=0;i<n1;i++)
{
cout<<nonterms[i]<<"\t";
//Display Table
for(int j=0;j<n2;j++)
{
cout<<pp_table[i][j]<<"\t";
}
cout<<endl;
```

```
 }
//Parsing String
char c;
do{
string ip;
deque<string> pp_stack;
pp_stack.push_front("$");
pp_stack.push_front(prods[0].substr(0,1));
cout<<"Enter the string to be parsed : ";
getline(cin,ip);
ip.push_back('$');
cout<<"Stack\tInput\tAction"<<endl;
while(true)
{
for(int i=0;i<pp_stack.size();i++)
cout<<pp_stack[i];
cout<<"\t"<<ip<<"\t";
      int row1 = getPosition(nonterms,pp_stack.front(),n1);
      int row2 = getPosition(terms,pp_stack.front(),n2);
      int column = getPosition(terms,ip.substr(0,1),n2);
if(row1 != -1 && column != -1)
{
string p = pp_table[row1][column];
if(p.empty())
{
cout<<endl<<"String cannot be Parsed."<<endl;
break;
}
pp_stack.pop_front();
if(p[3] != '#')
{
for(int x=p.size()-1;x>2;x--)
{
                pp_stack.push_front(p.substr(x,1));
}
}
cout<<p;
}
else
{
if(ip.substr(0,1) == pp_stack.front())
{
if(pp_stack.front() == "$")
{
               cout<<endl<<"String Parsed."<<endl;
break;
}
```

```
 cout<<"Match "<<ip[0];
pp_stack.pop_front();
ip = ip.substr(1);
}
else
{
            cout<<endl<<"String cannot be Parsed."<<endl;
break;
}
}
cout<<endl;
}
cout<<"Continue?(Y/N) ";
cin>>c;
cin.ignore();
}while(c=='y' || c=='Y');
return 0;
}


//OUTPUT
//Enter the number of productions : 5
//Enter the productions
//S->aXYb
//Enter first for aXYb : a
//X->c
//Enter first for c : c
//X->#
//Enter first for # : #
//Y->d
//Enter first for d : d
//Y->#
//Enter first for # : #
//Enter the number of Terminals : 4
//Enter the Terminals
//a
//b
//c
//d
//Enter the number of Non-Terminals : 3
//Enter Non-Terminal : S
//Enter follow of S : $
//Enter Non-Terminal : X
//Enter follow of X : bd
//Enter Non-Terminal : Y
//Enter follow of Y : b
```

Output:-

```
Enter the number of productions : 5
Enter the productions
S->aXYb
Enter first for aXYb : a
X->c
Enter first for c : c
X->#
Enter first for # : #
Y->d
Enter first for d : d
Y->#
Enter first for # : #
Enter the number of Terminals : 4
Enter the Terminals
a
b
c
d
Enter the number of Non-Terminals : 3
Enter Non-Terminal : S
Enter follow of S : $
Enter Non-Terminal : X
Enter follow of X : bd
Enter Non-Terminal : Y
Enter follow of Y : b

Grammar
S->aXYb
X->c
X->#
Y->d
Y->#
        a       b       c       d       $
S       S->aXYb
X               X->#    X->c    X->#
Y               Y->#            Y->d
Enter the string to be parsed : acdb
Stack   Input   Action
S$      acdb$   S->aXYb
aXYb$   acdb$   Match a
XYb$    cdb$    X->c
cYb$    cdb$    Match c
Yb$     db$     Y->d
db$     db$     Match d
b$      b$      Match b
$       $
String Parsed.
```

Practical 3

Write a program to convert infix to postfix using lex and yacc.

Code:-

infix_to_postfix.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);
}
```

infix_to_postfix.l -

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

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

Output:-

```
5+3*7+8*9
537*+89*+
```

Practical 4
Write a program to implement symbols table.

Code:-

```
#include <bits/stdc++.h>
using namespace std;
unordered_map<string, string> m;
void display()
{
for (auto ite : m)
{
        cout << ite.first << " is present at " << ite.second << endl;
}
}
int main()
{
int n;
do
{
        printf("\nSYMBOL TABLE IMPLEMENTATION\n");
printf("1. INSERT\n");
printf("2. DISPLAY\n");
printf("3. DELETE\n");
printf("4. SEARCH\n");
printf("5. MODIFY\n");
printf("6. END\n");
printf("Enter your option : ");
cin >> n;
switch (n)
{
case 1:
{
string x, y;
cout << "Enter the label and address: ";
cin >> x >> y;
if (m.find(x) != m.end())
{
cout << "Label already present!" << endl;
}
else
{
m[x] = y;
cout << "Label added successfully!" << endl;
}
// display();
break;
}
```

```
 case 2:
{
display();
break;
}
case 3:
{
string x;
cout << "Enter the label: ";
cin >> x;
if (m.find(x) == m.end())
{
cout << "Label not present in symbol table!" << endl;
}
else
{
m.erase(x);
cout << "Label deleted successfully!" << endl;
}
// display();
break;
}
case 4:
{
printf("Enter the label to be searched : ");
string x;
cin >> x;
if (m.find(x) != m.end())
{
        cout << "The label is already in the symbol Table at address: " << m[x] << endl;
}
else
{
cout << "The label is not found in the symbol table" << endl;
}
break;
}
case 5:
{
string x;
cout << "Enter the label: ";
cin >> x;
cout << "Enter the new address of the label: ";
string y;
cin >> y;
if (m.find(x) == m.end())
{
```

```
 cout << "Label not present" << endl;
}
else
{
m[x] = y;
          cout << "Address updated successfully!" << endl;
}
// display();
break;
}
default:
{
break;
}
}

} while (n < 6);
return 0;
}
```

Output:-

```
SYMBOL TABLE IMPLEMENTATION
1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 6

Process returned 0 (0x0)   execution time : 96.384 s
Press any key to continue.
```

```
SYMBOL TABLE IMPLEMENTATION
1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 1
Enter the label and address: input_string 100
Label added successfully!

SYMBOL TABLE IMPLEMENTATION
1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 2
input_string is present at 100
```

```
SYMBOL TABLE IMPLEMENTATION
1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 3
Enter the label: input_string
Label deleted successfully!

SYMBOL TABLE IMPLEMENTATION
1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 4
Enter the label to be searched : input_string
The label is not found in the symbol table

SYMBOL TABLE IMPLEMENTATION
1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 1
Enter the label and address: out_str 100
Label added successfully!

SYMBOL TABLE IMPLEMENTATION
1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 5
Enter the label: out_str 200
Enter the new address of the label: Address updated successfully!
```

Practical 5
Write a program to implement simple calculator using Lex and Yacc.

Code:-

calc.y -

```
%{

#include <stdio.h>
#include <stdlib.h>
extern int yylex();
extern int yyparse();
extern FILE* yyin;

void yyerror(const char* s);
%}
%union {
int ival;
float fval;
}
%token<ival> T_INT
%token<fval> T_FLOAT
%token T_PLUS T_MINUS T_MULTIPLY T_DIVIDE T_LEFT T_RIGHT
%token T_NEWLINE T_QUIT
%left T_PLUS T_MINUS
%left T_MULTIPLY T_DIVIDE
%type<ival> expression
%type<fval> mixed_expression
%start calculation

%%

calculation:
| calculation line
;

line: T_NEWLINE
        | mixed_expression T_NEWLINE { printf("\tResult: %f\n", $1);}
        | expression T_NEWLINE { printf("\tResult: %i\n", $1); }
| T_QUIT T_NEWLINE { printf("bye!\n"); exit(0); }
;
```

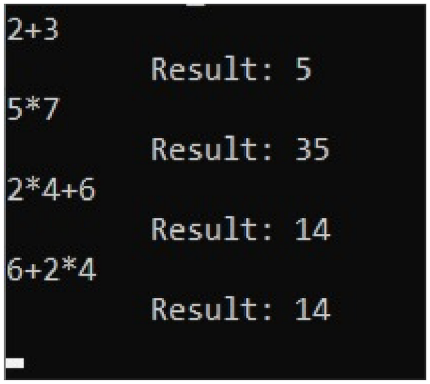
```
mixed_expression: T_FLOAT { $$ = $1; }
      | mixed_expression T_PLUS mixed_expression { $$ = $1 + $3; }
| mixed_expression T_MINUS mixed_expression { $$ = $1 - $3; }
    | mixed_expression T_MULTIPLY mixed_expression { $$ = $1 * $3; }
| mixed_expression T_DIVIDE mixed_expression { $$ = $1 / $3; }
| T_LEFT mixed_expression T_RIGHT { $$ = $2; }
| expression T_PLUS mixed_expression { $$ = $1 + $3; }
| expression T_MINUS mixed_expression { $$ = $1 - $3; }
       | expression T_MULTIPLY mixed_expression { $$ = $1 * $3; }
| expression T_DIVIDE mixed_expression { $$ = $1 / $3; }
| mixed_expression T_PLUS expression { $$ = $1 + $3; }
| mixed_expression T_MINUS expression { $$ = $1 - $3; }
       | mixed_expression T_MULTIPLY expression { $$ = $1 * $3; }
| mixed_expression T_DIVIDE expression { $$ = $1 / $3; }
          | expression T_DIVIDE expression { $$ = $1 / (float)$3; }
;

expression: T_INT { $$ = $1; }
| expression T_PLUS expression { $$ = $1 + $3; }
| expression T_MINUS expression { $$ = $1 - $3; }
| expression T_MULTIPLY expression { $$ = $1 * $3; }
| T_LEFT expression T_RIGHT { $$ = $2; }
;
%%

int main() {
yyin = stdin;
do {
yyparse();
} while(!feof(yyin));

return 0;
}
void yyerror(const char* s) {
fprintf(stderr, "Parse error: %s\n", s);
exit(1);
}
```

calc.l -

```
%option noyywrap
%{
#include <stdio.h>
```

```
#define YY_DECL int yylex()

#include "calc.tab.h"

%}

%%

[ \t] ; // ignore all whitespace
[0-9]+\.[0-9]+ {yylval.fval = atof(yytext); return T_FLOAT;}
[0-9]+ {yylval.ival = atoi(yytext); return T_INT;}
\n {return T_NEWLINE;}
"+" {return T_PLUS;}
"-" {return T_MINUS;}
"*" {return T_MULTIPLY;}
"/" {return T_DIVIDE;}
"(" {return T_LEFT;}
")" {return T_RIGHT;}
"exit" {return T_QUIT;}
"quit" {return T_QUIT;}
%%
```

Output:-



Practical 6
Write a program to implement lexical analyser for C language.

Code:-
%{lex_analyser.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|go
to|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|uns
igned|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("input.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{ return(1);
}
```

Output:-

input.c -

```
#include <stdio.h>
#define PI 3.14
struct inp
{
int a;
};
int check(int a, int b)
{
        return (a > b);
}
int main()
{
         struct inp ab;
int r = 5;
         printf("abc");
return 0;
}
```

```
#include <stdio.h>
This is a PREPROCESSOR DIRECTIVE
#define PI 3.14
This is a PREPROCESSOR DIRECTIVE

struct is a KEYWORD
inp is an IDENTIFIER

BLOCK BEGINS

int is a KEYWORD
a is an IDENTIFIER;

BLOCK ENDS;

int is a KEYWORD
FUNCTION:
check(
int is a KEYWORD
a is an IDENTIFIER,
int is a KEYWORD
b is an IDENTIFIER
)

BLOCK BEGINS

return is a KEYWORD (
a is an IDENTIFIER >
b is an IDENTIFIER
);

BLOCK ENDS

int is a KEYWORD
FUNCTION:
main(
)

BLOCK BEGINS

struct is a KEYWORD
inp is an IDENTIFIER
ab is an IDENTIFIER;

int is a KEYWORD
r is an IDENTIFIER
```

```
BLOCK BEGINS

struct is a KEYWORD
inp is an IDENTIFIER
ab is an IDENTIFIER;

int is a KEYWORD
r is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
5 is a NUMBER ;

FUNCTION:
printf(
"abc" is a STRING
);

return is a KEYWORD
0 is a NUMBER ;

BLOCK ENDS
```

Practical 7
Write a program to implement parser for C language.

Code:-
parser.l

```
%option yylineno

%{
#include<stdio.h>
#include"parser.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;
}
```

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       | struct_declarator_list ','

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 '[' ']'
                        | '[' ']'
param_list ')'          | direct_abstract_declarator '('
                        | '(' param_list ')'
                        | direct_abstract_declarator '(' ')'
                        | '(' ')'
stat                    ;
                        : labeled_stat

                        | exp_stat

                        | compound_stat
```

```
                    | selection_stat

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

                    | '{' stat_list '}'

                    | '{' decl_list '}'

                    | '{' '}'
stat_list

                    ;
                    : stat
selection_stat      | stat_list stat

                    ;
                    : IF '(' exp ')' stat
iteration_stat      %prec "then"
                    | IF '(' exp ')' stat ELSE stat
                    | SWITCH '(' exp ')' 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
jump_stat           | FOR '(' ';' exp ';' ')' stat
                    | FOR '(' ';' ';' exp ')' stat
                    | FOR '(' ';' ';' ')' 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
unary_operator                  | SIZEOF '(' type_name ')'
                        ;
                        : '&' | '*' | '+' | '-' | '~' | '!'
postfix_exp
                        ;
                        : primary_exp

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

                        | consts
argument_exp_list
                        | string

consts                  | '(' exp ')'
                        ;
                        : assignment_exp
                                        | argument_exp_list ',' assignment_exp
                        ;
                        : int_const
%%
                        | char_const
                        | float_const
                        | enum_const
                        ;
```

```
int main()
{
yyparse();
if(success)
printf("Parsing Successful\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:-

```
#include <stdio.h>
int main(){
int a=10;
printf("%d",&a);
return 0; }
Parsing Successful
```

Practical 8
Generate the three address code for selected C statements.

Code:-

three_address.l

```
%{
#include"three_address.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;}
%%
```

three_address.y

```
%{
#include"three_address.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:-

```
D:\CTD\CTD_Practical\8th_Three_Address>three_address
Enter the expression: a=b+c+d*e-f;

A:=     b       +       c
B:=     d       *       e
C:=     A       +       B
D:=     C       -       f
E:=     a       =       D
```

Create a GNU Makefile

Make reads a description of a project from a makefile (by default, called 'Makefile' in the current directory). A makefile specifies a set of compilation rules in terms of targets (such as executables) and their dependencies (such as object files and source files) in the following format:

target: dependencies
command

For each target, make checks the modification time of the corresponding dependency files to determine whether the target needs to be rebuilt using the corresponding command. Note that the command lines in a makefile must be indented with a single TAB character, not spaces.

GNU Make contains many default rules, referred to as implicit rules, to simplify the construction of makefiles. For example, these specify that '.o' files can be obtained from '.c' files by compilation, and that an executable can be made by linking together '.o' files. Implicit rules are defined in terms of make variables, such as CC (the C compiler) and CFLAGS (the compilation options for C programs), which can be set using VARIABLE=VALUE lines in the makefile. For C++ the equivalent variables are CXX and CXXFLAGS, while the make variable CPPFLAGS sets the preprocessor options. The implicit and user-defined rules are automatically chained together as necessary by GNU Make.

A simple 'Makefile' for the project above can be written as follows: