SSGMCE/FRM/32-B

SSGMCE	SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGG.			**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET			
	EXPERIMENT TITLE: Design Predictive Parser for the given language.			
EXPERIMENT NO.: **SSGMCE/WI/IT/01/6IT01/07**			ISSUE NO.: 00	ISSUE DATE: 01.02.2022
REV. DATE:	REV. NO.:	DEPTT.: INFORMATION TECHNOLOGY		
LABORATORY: COMPILER DESIGN (CD)			SEMESTER: VI	PAGE: OF 6

1.0) AIM:

Design Predictive Parser for the given language.

2.0) OBJECTIVE:

After the completion of this experiment, predictive parser will be able to identify the tokens to verify syntax errors.

3.0) FACILITIES/ APPARATUS:

i) Hardware: Computer Machine

ii) Software: FLEX (fast lexical analyzer generator) for Windows- LEX and YACC (Bison) Installer for Windows 7/8.1/10 32-bit & 64-bit

4.0) THEORY:

A predictive parser is a recursive descent parser with no backtracking or backup. It is a top-down parser that does not require backtracking. At each step, the choice of the rule to be expanded is made upon the next terminal symbol. Predictive Parser is a method that implements the technique of Top-Down parsing without Backtracking. A predictive parser is an effective technique of executing recursive-descent parsing by managing the stack of activation records, particularly.

Consider

A -> A1 | A2 | … | An

If the non-terminal is to be further expanded to 'A', the rule is selected based on the current input symbol 'a' only.

Predictive Parsers has the following components –

- **Input Buffer** – The input buffer includes the string to be parsed followed by an end marker $ to denote the end of the string. Here a, +, b are terminal symbols.

a	+	b	$

Input String

- **Stack** – It contains a combination of grammar symbols with $ on the bottom of the stack. At the start of Parsing, the stack contains the start symbol of Grammar followed by $.
- **Parsing Table** – It is a two-dimensional array or Matrix M [A, a] where A is non-terminal and 'a' is a terminal symbol.

Algorithm to construct Predictive Parsing Table

Input – Context-Free Grammar G

Output– Predictive Parsing Table M

Method– For the production A → ɑ of Grammar G.

- For each terminal, a in FIRST (α) add A → ɑ to M [A, a].
- If ε is in FIRST (ɑ), and b is in FOLLOW (A), then add A → ɑ to M[A, b].
- If ε is in FIRST (ɑ), and $ is in FOLLOW (A), then add A → ɑ to M[A, $].
- All remaining entries in Table M are errors.

5.0) PROGRAM:

```
#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;
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()
{
    //void E();
    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("Enter the expression, place ; at the end and press Ctrl-Z to terminate \n");
    parser();<br>return 0;
}
```

6.0) OUTPUT OF PROGRAM

INPUT

```
Enter the expression, place ; at the end and press Ctrl-Z to terminate
a*b+c;
```

OUTPUT

```
Identifier: a
Identifier: b
Arithmetic Operator: *
Identifier: c
Arithmetic Operator: +
```

INPUT

```
5*7;
```

OUTPUT

```
Number: 5
Number: 7
Arithmetic Operator: *
```

INPUT

```
*2;
```

OUTPUT

```
line 5, Syntax error
```

7.0) CONCLUSION:

A lexical analyzer has been designed using LEX Program to scan reserved words and Identifiers of C Language.