

EXPERMENT-I

Aim: Design a Lexical analyzer for the given language. The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value.

DESCRIPTION:

A **lexical analyzer** (or lexer) is the first phase of a compiler, responsible for scanning the input source code and breaking it into meaningful tokens. The tokens are then passed to the syntax analyzer (parser) for further processing.

Functions of a Lexical Analyzer:

1. **Tokenization:** The lexer divides the input source code into valid lexemes and categorizes them into tokens (keywords, identifiers, operators, etc.).
2. **Ignoring Whitespaces and Comments:** Spaces, tabs, new lines, and comments are ignored since they are not relevant to syntax checking.
3. **Error Handling:** It identifies invalid lexemes and reports lexical errors.

Components of a Lexical Analyzer:

- **Input Buffer:** Holds the source program text.
- **Scanner:** Reads characters from the buffer and forms lexemes.
- **Symbol Table:** Stores identifiers and keywords.
- **Finite Automata (DFA/NFA):** Used for pattern matching to recognize tokens.

Token Types in the Given Language

The language defined in the problem consists of the following token types:

1. **Keywords:** Reserved words like if, else, while, for, etc.
2. **Identifiers:** Variable names, function names (restricted to a reasonable length).
3. **Operators:** +, -, *, /, =, <, >, ==, etc.
4. **Literals:** Numeric constants (123, 4.56) and string literals ("hello").
5. **Punctuation:** ,, :, (), {}, [], etc.
6. **Comments:** Ignored (e.g., // single-line, /* multi-line */).
7. **Whitespaces:** Spaces, tabs, and new lines (ignored).

Manual Calculation Example

Consider the following code snippet:

```
int main() {  
    // This is a comment  
    int a = 10;  
    float b = 20.5;  
    a = a + b;  
}
```

Step 1: Remove Comments and Extra Whitespaces After ignoring the comment and extra spaces, the relevant input remains:

```
int main() {  
    int a = 10;  
    float b = 20.5;  
    a = a + b;  
}
```

Step 2: Tokenization

Lexeme	Token Type
int	Keyword
main	Identifier
()	Punctuation
{	Punctuation
int	Keyword
a	Identifier
=	Operator
10	Literal
;	Punctuation
float	Keyword
b	Identifier
=	Operator
20.5	Literal

;	Punctuation
a	Identifier
=	Operator
a	Identifier
+	Operator
b	Identifier
;	Punctuation
}	Punctuation

Step 3: Symbol Table

Identifier	Type	Value
main	Function	-
a	int	10
b	float	20.5

PROGRAM:

```
#include<ctype.h>
#include<stdio.h>
#include<string.h>
#define MAX_INPUT_SIZE 1000
#define MAX_LINE_SIZE 100
#define MAX_TOKENS 100
int main()
{
int sy,o,nt,v,c,i=0,j,k,kw,keyword_index=0,variable_index=0,
constant_index=0,operator_index=0,symbol_index=0;
    char s[MAX_INPUT_SIZE]=" ";
    char g[MAX_LINE_SIZE];
    char keywords[MAX_TOKENS][20];
    char variables[MAX_TOKENS][20];
    char constants[MAX_TOKENS][20];
    char operators[MAX_TOKENS];
    char symbols[MAX_TOKENS];
    char constant[20];
```

```

char word[20];
int already_exists,is_keyword;
const char
*keyword_list[]={ "int","float","if","else","while","return","for","break","continue","char",
"double","void","do","switch","case","default","goto","main" };
const int keyword_count=sizeof(keyword_list)/sizeof(keyword_list[0]);
const char *operator_list="+=-*/><==/";
const char *symbol_list=";(){ }[]%&";
printf("Enter program (type $ on a new line to terminate \n");
while(1){
    if(!fgets(g,sizeof(g),stdin))
        break;
    if(g[strlen(g)-1]=='\n')
        g[strlen(g)-1]='\0';
    if(strcmp(g,"$")==0)
        break;
    if(strlen(s)+strlen(g)>=MAX_INPUT_SIZE-1){
        printf("Error:Input too long!\n");
        return 1;
    }
    strcat(s,g);
    strcat(s," ");
}
i=0;
while(s[i]!='\0'){
    already_exists=0;
    is_keyword=0;
    if(isdigit(s[i])){
        j=i;
        while(isdigit(s[i+1]))
            i++;
        strncpy(constant,&s[j],i-j+1);
        constant[i-j+1]='\0';
        for(c=0;c<constant_index;c++){
            if(strcmp(constants[c],constant)==0){

```

```

        already_exists=1;
        break;
    }
}
if(!already_exists)
    strcpy(constants[constant_index++],constant);
}
else if(isalpha(s[i])){
    j=i;
    while(isalnum(s[i+1])||s[i+1]=='_')i++;
    strncpy(word,&s[j],i-j+1);
word[i-j+1]='\0';
for(k=0;k<keyword_count;k++){
    if(strcmp(word,keyword_list[k])==0){
        is_keyword=1;
        for(kw=0;kw<keyword_index;kw++){
            if(strcmp(keywords[kw],word)==0){
                already_exists=1;
                break;
            }
        }
        if(!already_exists)
            strcpy(keywords[keyword_index++],word);
        break;
    }
}
if(!is_keyword){
    already_exists=0;
    for(v=0;v<variable_index;v++){
        if(strcmp(variables[v],word)==0){
            already_exists=1;
            break;
        }
    }
    if(!already_exists)

```

```

        strcpy(variables[variable_index++],word);
    }
}
else if(strchr(operator_list,s[i])){
    already_exists=0;
    for(o=0;o<operator_index;o++){
        if(operators[o]==s[i]){
            already_exists=1;
            break;
        }
    }
    if(!already_exists)
        operators[operator_index++]=s[i];
}
else if(strchr(symbol_list,s[i])){
    already_exists=0;
    for(sy=0;sy<symbol_index;sy++){
        if(symbols[sy]==s[i]){
            already_exists=1;
            break;
        }
    }
    if(!already_exists)
        symbols[symbol_index++]=s[i];
}
i++;
}
printf("Different types of tokens are listed in the given program:");
printf("\nkeywords:\n");
for(kw=0;kw<keyword_index;kw++){
    printf("%s ",keywords[kw]);
}
printf("\nvariables:\n");
for(v=0;v<variable_index;v++){
    printf("%s ",variables[v]);
}

```

```

    }
    printf("\nconstants:\n");
    for(c=0;c<constant_index;c++){
        printf("%s ",constants[c]);
    }
    printf("\noperators:\n");
    for(o=0;o<operator_index;o++){
        printf("%c ",operators[o]);
    }
    printf("\nsymbols:\n");
    for(sy=0;sy<symbol_index;sy++){
        printf("%c ",symbols[sy]);
    }
    printf("\n");
    return 0;
}

```

OUTPUT:

```

Enter Program (type $ on a new line to terminate input):
void main()
{
    int a=3,b=2,c;
    c=a+b;
}
$

Keywords:
void main int
Variables:
a b c
Constants:
3 2
Operators:
= +
Symbols:
( ) { , ; }

```

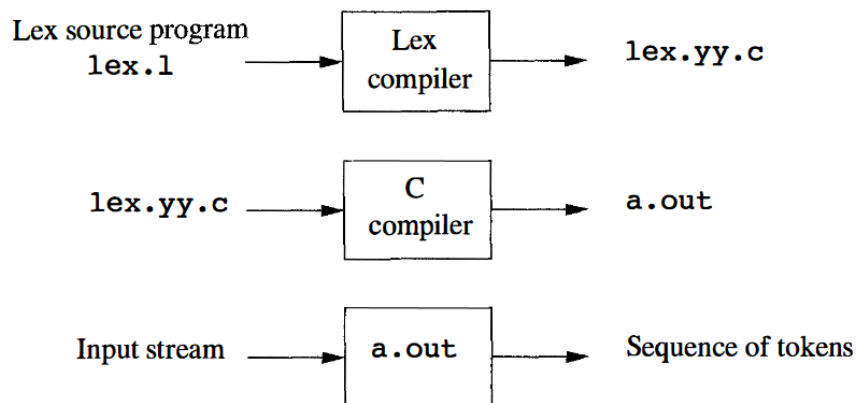
EXPERIMENT-II

Aim: Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating stools.

DESCRIPTION:

A Lex is a program that is used to generate lexical analyzer or scanner or tokenizers. First, the lexical analyzer create a program **lex.l** in the Lex language, then **lex.l** is run through the **Lex** compiler to produce a 'C' program **lex.yy.c**. The program **lex.yy.c** consists of a tabular representation of a transition diagram constructed from the regular expressions of **lex.l**, then uses this table to recognize lexeme. **Lex.yy.c** is run through the 'C' compiler to produce the object program **a.out**. **a.out** is a lexical analyzer that transforms an input stream into a sequence of tokens.

Execution Procedure of Lex program:



Lex specification:

A LEX program consists of three parts

declarations
%%
translation rules
%%
auxiliary functions

Note: Each section is separated by %%

Declaration section: The declaration section includes declarations of variables, constants and regular definitions. A regular definition is a sequence of regular expressions where each regular expression is given a name for notational convenience.

d₁→r₁
d₂→r₂
:
:

$$d_n \rightarrow r_n$$

Where d_1, d_2, \dots, d_n is a distinct name, and each r_1, r_2, \dots, r_n is a regular expressions.

Translation rules:

The translation rules are a set of statements, which defines the action to be performed for each regular expression in the translation rules of a LEX program.

P1 {action1}

P2 {action2}

:

:

Pn {action n}

Where each P_i is a regular expression and each action 'i' is program.

Auxiliary functions: The third section contain whatever additional functions are used in the actions. These functions can be compiled separately and loaded with the lexical analyzer.

PROGRAM:

1st we need to write a lex program with the extension of .l (lex123.l)

```
% {
int com=0;
% }
identifier [a-zA-Z]*
digit [0-9]
letter [a-zA-Z]
%%
#.* { printf("\n%s is PRE PROCESSOR DIRECTIVE",yytext); }
void|int|float|if { if(!com) printf("\n Keyword: %s ",yytext); }
"/*" { com=1;printf("\n Comments"); }
"*/" { com=0; }
[a-z]+ { if(!com) printf("\n Variables: %s ",yytext); }
{identifier}\( { if(!com) printf("\n Function Name: %s",yytext); }
\{ { if(!com) printf("\n Block Begins "); }
\} { if(!com) printf("\n Block Ends "); }
".*\\" { if(!com) printf("\n String is %s ",yytext); }
[0-9]+ { if(!com) printf("\n Number is %s ",yytext); }
\\ { if(!com) printf("\n\t"); }
= { if(!com) printf("\n Assigment Operator %s ",yytext); }
```

```

\<=|\>=|\>|\<|\== { if(!com) printf("\n Relational Operators: %s  ",yytext); }
\+|\-|\*|\%|\ { if(!com) printf("\n Arithmetic Operators: %s",yytext); }
%%

void main(argc,argv)
int argc;
char **argv;
{
if(argc>1)
{
FILE *file;
file=fopen(argv[1],"r");
if(!file)
{
printf("could not open file %s",argv[1]);
exit(0);
}
yyin=file;
}
yylex();
}
int yywrap()
{
return(0);
}

```

2nd write a simple c program (temp.c)

```

/* Addition of two numbers */
#include<stdio.h>
void main()
{
int a,b,c;
a=5;
b=2;
c=a+b;
printf(" \nThe sum is %d",c);
}

```

OUTPUT:

```
cse@lab-lunixserver:~/kusuma$ lex lexer.l
cse@lab-lunixserver:~/kusuma$ gcc lex.yy.c
cse@lab-lunixserver:~/kusuma$ ./a.out temp.c
```

```
#include<stdio.h> is PRE PROCESSOR DIRECTIVE
```

```
Keyword: void
```

```
Function Name: main()
```

```
Block Begins
```

```
Keyword: int
```

```
Variables: a ,
```

```
Variables: b ,
```

```
Variables: c ;
```

```
Variables: a
```

```
Assignment Operator =
```

```
Number is 5 ;
```

```
Variables: b
```

```
Assignment Operator =
```

```
Number is 2 ;
```

```
Variables: c
```

```
Assignment Operator =
```

```
Variables: a +
```

```
Variables: b ;
```

```
Function Name: printf(
```

```
String is " \nThe sum is %d" ,
```

```
Variables: c );
```

```
Block Ends
```

EXPERIMENT - III

Aim: Write a C program to construct predictive/ LL(1) parsing table for the given CFG

Grammar rules

A->**aBa**

B->**bB/ε**

DESCRIPTION:

LL(1) Parser: An LL(1) parser is a type of top-down, non-backtracking parser that uses a single-token lookahead to determine the correct production rule to apply. It is commonly implemented using recursive descent or a table-driven parsing approach.

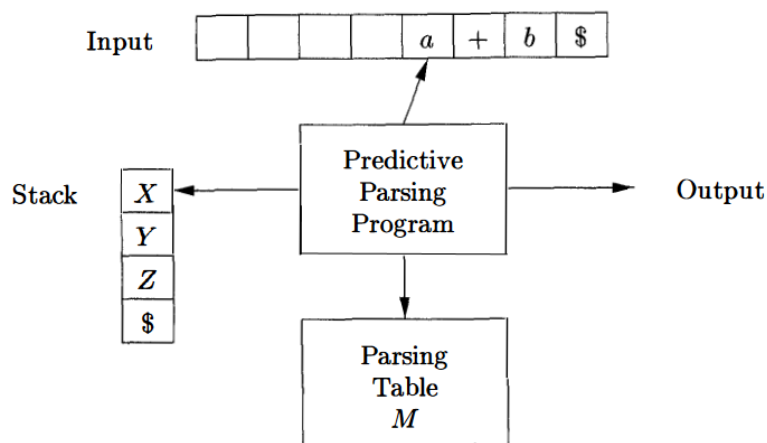
- "L" → Left-to-right scanning of input.
- "L" → Leftmost derivation.
- "1" → One-symbol lookahead.

It is used for parsing **LL (1) grammars**, which are context-free grammars that can be parsed using one-token lookahead without ambiguity.

Block Diagram: A **Predictive Parser** is a top-down, non-backtracking parser that uses a **parsing table** to determine which production rule to apply.

Components

1. **Input Buffer** → Holds the input string.
2. **Stack** → Stores grammar symbols, initially \$ (end marker) and the start symbol.
3. **Parsing Table (M)** → Guides parsing decisions using **FIRST** and **FOLLOW** sets.
4. **Predictive Parsing Program** → Reads input, consults **M**, expands non-terminals, and matches terminals.



Manual Calculation: Solve the given problem and write it on the left side of the paper.

PROGRAM:

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

```
#include<stdlib.h>
```

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

```
char prod[3][10] = {"A->aBa", "B->bB", "B->&"};
```

```
char first[3][10] = {"a", "b", "&"};
```

```
char follow[3][10] = {"$", "a", "a"};
```

```
char table[3][4][10];
```

```
char input[10];
```

```
int top = -1;
```

```
char stack[25];
```

```
char curp[20];
```

```
void push(char item)
```

```
{
```

```
    stack[++top] = item;
```

```
}
```

```
void pop()
```

```
{
```

```
    top = top - 1;
```

```
}
```

```
void display()
```

```
{
```

```
    int i;
```

```
for (i = top; i >= 0; i--)
```

```
    printf("%c", stack[i]);
```

```
}
```

```
int numr(char c)
```

```
{
```

```
    switch (c)
```

```
{
```

```

        case 'A': return 1;
        case 'B': return 2;
        case 'a': return 1;
        case 'b': return 2;
        case '&': return 3;
    }
    return 1;
}

int main()
{
    char c;
    int i,j,k,n;
    for(i=0;i<3;i++){
        for(j=0;j<4;j++){
            strcpy(table[i][j],"EMPTY");
        }
    }

    printf("\nGrammar\n");
    for(i=0;i<3;i++)
        printf("%s\n",prod[i]);
    printf("\nfirst={ %s, %s,%s }", first[0], first[1], first[2]);
    printf("\nfollow={ %s,%s }\n", follow[0], follow[1]);

    printf("\nPredictive parsing table for the given grammar :\n");
    strcpy(table[0][0],"");
    strcpy(table[0][1],"a");
    strcpy(table[0][2],"b");
    strcpy(table[0][3],"$");
    strcpy(table[1][0],"A");
    strcpy(table[2][0],"B");
    for(i=0;i<3;i++)
    {
        if(first[i][0]!='&')
            strcpy(table[numr(prod[i][0]))[numr(first[i][0])],prod[i]);
        else

```

```

strcpy(table[numr(prod[i][0])][numr(follow[i][0])],prod[i]);
}
printf("\n-----\n");
for(i=0;i<3;i++) {
for(j=0;j<4;j++)
{
printf("%-30s",table[i][j]);
if(j==3)
printf("\n-----\n");
}
}
printf("Enter the input string terminated with $ to parse:-");
scanf("%s",input);
for(i=0;input[i]!='\0';i++){
if((input[i]!='a')&&(input[i]!='b')&&(input[i]!='$'))
{
printf("Invalid String");
exit(0);
}
}
if(input[i-1]!='$')
{
printf("\n\nInput String Entered Without End Marker $");
exit(0);
}
push('$');
push('A');
i=0;
printf("\n\n");
printf("Stack\tInput\tAction");
printf("\n-----\n");
while(input[i]!='$'&& stack[top]!='S')
{
display();
printf("\t\t%s\t",input+i);

```

```

if(stack[top]==input[i])
{
printf("\tMatched %c\n", input[i]);
pop();
i++;
}
else
{
if(stack[top]>=65&&stack[top]<92)
{
strcpy(curp,table[numr(stack[top]))[numr(input[i])]);
if(!(strcmp(curp,"e")))
{
printf("\nInvalid String - Rejected\n");
exit(0);
}
else
{
printf("\tApply production %s\n",curp);
if(curp[3]=='&')
pop();
else
{
pop();
n=strlen(curp);
for(j=n-1;j>=3;j--)
push(curp[j]);
}
}
}
}
display();
printf("\t\t%s\t", (input+i));
printf("\n-----\n");

```



```

if(stack[top]=='$'&&input[i]=='$')
{
printf("\nValid String - Accepted\n");
}
else
{
printf("Invalid String - Rejected\n");
}
getch();
}

```

OUTPUT:

```

Grammar
A->aBa
B->bB
B->&

first={a, b,&}
follow={$,a}

Predictive parsing table for the given grammar :
-----
                a                b
$
-----
A                A->aBa                EMPTY
    EMPTY
-----
B                B->&                B->bB
    EMPTY
-----
Enter the input string terminated with $ to parse:-abba$_

```

Stack	Input	Action
A\$	abba\$	Apply production A->aBa
aBa\$	abba\$	Matched a
Ba\$	bba\$	Apply production B->bB
bBa\$	bba\$	Matched b
Ba\$	ba\$	Apply production B->bB
bBa\$	ba\$	Matched b
Ba\$	a\$	Apply production B->&
a\$	a\$	Matched a
\$	\$	

Valid String - Accepted

EXPERIMENT – IV

Aim: Write a C program for implementation of LR parsing algorithm to accept a given input string.

DESCRIPTION:

Construct LR Parsing:

LR Parsing: LR Parsing is a bottom-up parsing technique used to analyze deterministic context-free grammars. It reads input from left to right (L) and produces a rightmost derivation in reverse (R). It is efficient, deterministic, and widely used in compilers.

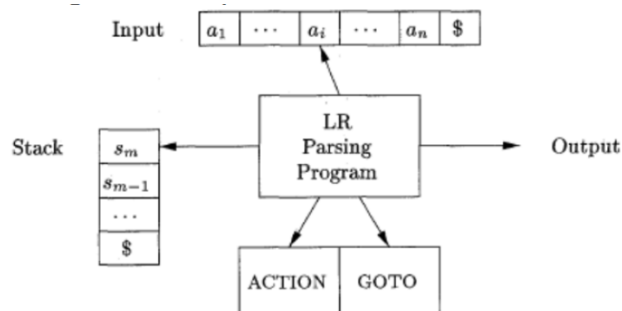
Bottom-up syntax analysis technique that can be used to parse a large class of context-free grammars. This technique is called LR(k) parsing. A general notation for the representation of LR grammars is given by LR(K).

Where, L- left to right scanning of the input

R- R for constructing a rightmost derivation in reverse

K- The no. of input symbols of lookahead that are used in making parsing decisions.

Block Diagram:



Types of LR Parsers

1. LR(0) → No lookahead, handles simple grammars.
2. SLR(1) → Uses Follow sets, better than LR(0).
3. CLR(1) (Canonical LR) → Uses lookaheads, powerful but large tables.
4. LALR(1) → Merges CLR(1) states, used in YACC/Bison.

Parsing Steps

1. Initialize stack with \$ and start state.
2. Read input symbol, check the ACTION table.
3. Perform Actions:
 - SHIFT → Move to a new state.

- REDUCE → Apply production rule.
 - ACCEPT → Successful parse.
 - ERROR → Reject input.
4. Repeat until stack is empty & input is consumed.

Manual Calculation: Solve the given problem and write it on the left side of the paper.

Check string acceptance for valid and invalid input strings based on parsing table.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int point=0,top;
char inp[20],stack[100];
void push(char);
void pop();
void s0(char);
void s1(char);
void s2(char);
void s3(char);
void s4(char);
void s5(char);
void s8(char);
void push(char k)
{
if(top==99)
{
printf("string not accepted");
}
else
stack[++top]=k;
}
void pop()
{
if(top== -1)
{
```

```
printf("string not accepted");
}
else
top--;
}
void s0(char l)
{
if(l=='e')
{
printf("shift3 \n");
push(l);
push('3');
point++;
}
else if(l=='d')
{
printf("shift4 \n");
push(l);
push('4');
point++;
}
else
{
printf("string not accepted");
exit(0);
}
}
void s1(char l)
{
if(l=='$')
{
push('$');
printf("string accepted");
exit(0);
}
```

```
}  
void s2(char l)  
{  
if(l=='e')  
{  
printf("shift3 \n");  
push(l);  
push('3');  
point++;  
}  
else if(l=='d')  
{  
printf("shift4 \n");  
push(l);  
push('4');  
point++;  
}  
else  
{  
printf("string not accepted");  
exit(0);  
}  
}  
void s3(char l)  
{  
if(l=='e')  
{  
printf("shift3 \n");  
push(l);  
push('3');  
point++;  
}  
else if(l=='d')  
{  
printf("shift4 \n");
```

```
push(l);
push('4');
point++;
}
else
{
printf("string not accepted");
exit(0);
}
}

void s4(char l)
{
if(l=='e' || l=='d' || l=='$')
{
printf("reduce C->d \n");
pop();
pop();
push('C');
}
else
{
printf("string not accepted");
exit(0);
}
}

void s5(char l)
{
if(l=='$')
{
printf("reduce S->CC \n");
pop();
pop();
pop();
pop();
push('S');
```

```

    }
else
{
printf("string not accepted");
exit(0);
}
}
void s8(char l)
{
if(l=='e' || l=='d' || l=='$')
{
printf("reduce C->eC \n");
pop();
pop();
pop();
pop();
push('C');
}
else
{
printf("string not accepted");
exit(0);
}
}

main()
{
int i,j=0,s=0,a,x;
top=-1;

printf("Grammer:\n");
printf("S->CC \n C->eC \n C->d \n");
printf("enter the ip $:");
scanf("%s",&inp);
printf("\n");

```

```

printf("STACK \t INPUT \t ACTON \n");
Printf("“ _____ \n”");
push('$');
push('0');
while(1)
{
j=top;
a=0;
printf("\n");
while(a<=j)
{
printf("%c",stack[a]);
a++;
}
printf("\t");
s=point;
while(inp[s]!='\0')
{
printf("%c",inp[s]);
s++;
}
printf("\t");
if(stack[top]=='0') s0(inp[point]);
else if(stack[top]=='1') s1(inp[point]);
else if(stack[top]=='2') s2(inp[point]);
else if(stack[top]=='3') s3(inp[point]);
else if(stack[top]=='4') s4(inp[point]);
else if(stack[top]=='5') s5(inp[point]);
else if(stack[top]=='8') s8(inp[point]);
else if(stack[top]=='S')
{
x=top-1;
if(stack[x]=='0')
{ printf("shift1");
push('1'); }

```



```

else
printf("string not accepted");
}
else if(stack[top]=='C')
{
x=top-1;
if(stack[x]=='0') {
printf("shift2");
push('2'); }
else if(stack[x]=='2')
{ printf("shift5");
push('5');
}
else if(stack[x]=='3')
{ printf("shift8");
push('8');
}
else
printf("string not accepted");
}
else
{
printf("string not accepted");
exit(0);
}
}
}

```

OUTPUT :

Grammer :

S->CC

C->eC

C->d

enter the ip \$:eedd\$

STACK	INPUT	ACTION
--------------	--------------	---------------

\$0	eedd\$	shift3	
\$0e3	edd\$	shift3	
\$0e3e3	dd\$	shift4	
\$0e3e3d4	d\$	reduce C->d	
\$0e3e3C	d\$	shift8	
\$0e3e3C8	d\$	reduce C->eC	
\$0e3C	d\$	shift8	
\$0e3C8	d\$	reduce C->eC	
\$0C	d\$	shift2	
\$0C2	d\$	shift4	
\$0C2d4	\$	reduce C->d	
\$0C2C	\$	shift5	
\$0C2C5	\$	reduce S->CC	
\$0S	\$	shift1	
\$0S1	\$	string accepted	

Grammer:

S->CC

C->eC

C->d

enter the ip \$:eed\$

STACK	INPUT	ACTON
\$0	eed\$	shift3
\$0e3	ed\$	shift3
\$0e3e3	d\$	shift4
\$0e3e3d4	\$	reduce C->d
\$0e3e3C	\$	shift8
\$0e3e3C8	\$	reduce C->eC
\$0e3C	\$	shift8
\$0e3C8	\$	reduce C->eC
\$0C	\$	shift2
\$0C2	\$	string not accepted_

EXPERIMENT - V

Aim: Write program to Convert the BNF rules into YACC form and write code to generate Intermediate Representation form (Quadruple).

Description:

Backus-Naur Form (BNF) Rules

Backus-Naur Form (BNF) is a formal notation used to describe the syntax of programming languages. It consists of production rules that define how sentences in a language are formed.

YACC (Yet another Compiler Compiler) Tool

YACC (Yet Another Compiler Compiler) is a tool used to generate parsers for programming languages. It takes grammar rules written in BNF (Backus-Naur Form) and generates a syntax analyzer (parser) for a given language.

Key Features of YACC:

- Automates the process of creating a parser.
- Works with LALR(1) (Look-Ahead Left-to-Right) grammars.
- Uses tokens from a lexical analyzer (Lex tool).
- Generates a C program for syntax analysis.

Structure of a YACC Program

A YACC program has three sections, separated by %%:

% {

C declarations (header files, global variables)

% }

YACC declarations (tokens, grammar rules)

%%

Grammar rules (productions)

%%

C code (main function, helper functions)

YACC vs LEX

Feature	YACC	LEX
Purpose	Syntax Analysis (Parsing)	Lexical Analysis (Tokenization)
Input	Grammar rules (BNF)	Regular expressions
Output	C code for a parser y.tab.c	C code for a lexical analyzer lex.yy.c

Program:

int.1

```
% {  
  
#include "y.tab.h"  
  
#include <stdio.h>  
  
#include <string.h>  
  
int LineNo=1;  
  
% }  
  
identifier [a-zA-Z][_a-zA-Z0-9]*  
number [0-9]+|([0-9]*\.[0-9]+)  
  
%%  
  
main\(\) return MAIN;  
  
if return IF;  
  
else return ELSE;  
  
while return WHILE;  
  
int |  
  
char |  
  
float return TYPE;  
  
{ identifier } { strcpy(yylval.var,yytext);  
return VAR;}  
  
{ number } { strcpy(yylval.var,yytext);  
return NUM;}  
  
\< |  
  
\> |  
  
\>= |  
  
\<= |  
  
== { strcpy(yylval.var,yytext); return RELOP;}  
  
[ \t] ;  
  
\n LineNo++;  
  
. return yytext[0];  
  
%%
```

int.y

```
% {
#include<string.h>
#include<stdio.h>
struct quad
{
char op[5];
char arg1[10];
char arg2[10];
char result[10];
}QUAD[30];
struct stack
{
int items[100];
int top;
}stk;
int Index=0,tIndex=0,StNo,Ind,tInd;
extern int LineNo;
% }

%union
{
char var[10];
}

%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'
%%

PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
;
```

```

CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONNST
| WHILEST
;
DESCT: TYPE VARLIST
;
VARLIST: VAR ',' VARLIST
| VAR
;
ASSIGNMENT: VAR '=' EXPR{
strcpy(QUAD[Index].op,"=");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,$1);
strcpy($$,QUAD[Index++].result);
}
;
EXPR: EXPR '+' EXPR {AddQuadruple("+",$1,$3,$$);}
| EXPR '-' EXPR {AddQuadruple("-", $1,$3,$$);}
| EXPR '*' EXPR { AddQuadruple("*",$1,$3,$$);}
| EXPR '/' EXPR { AddQuadruple("/", $1,$3,$$);}
| '-' EXPR { AddQuadruple("UMIN", $2, "", $$);}
| '(' EXPR ')' {strcpy($$, $2);}
| VAR
| NUM
;

```

```

CONDST: IFST{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST ELSEST
;
IFST: IF '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
} BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
};
ELSEST: ELSE{
tInd=pop();
Ind=pop();
push(tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}

```

```

BLOCK{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
};
CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
| VAR
| NUM
;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
WHILELOOP: WHILE '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);

```



```

Index++;
}
;
%%
extern FILE *yyin;
int main(int argc,char *argv[])
{
FILE *fp;
int i;
if(argc>1)
{
fp=fopen(argv[1],"r");
if(!fp)
{
printf("\n File not found");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t -----""\n\t\t Pos Operator Arg1 Arg2 Result" "\n\t\t
-----");
for(i=0;i<Index;i++)
{
printf("\n\t\t %d\t %s\t %s\t %s\t
%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);
}
printf("\n\t\t -----");
printf("\n\n");
return 0;
}

```

```

void push(int data)
{
stk.top++;
if(stk.top==100)
{
printf("\n Stack overflow\n");
exit(0);
}
stk.items[stk.top]=data;
}

int pop()
{
int data;
if(stk.top== -1)
{
printf("\n Stack underflow\n");
exit(0);
}
data=stk.items[stk.top--];
return data;
}

void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{
strcpy(QUAD[Index].op,op);
strcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex++);
strcpy(result,QUAD[Index++].result);
}

yyerror()
{
printf("\n Error on line no:%d",LineNo);
}

```

Input:

```
$vi test.c
```

```
main()
```

```
{
```

```
int a,b,c;
```

```
if(a<b)
```

```
{
```

```
a=a+b;
```

```
}
```

```
while(a<b)
```

```
{
```

```
a=a+b;
```

```
}
```

```
if(a<=b)
```

```
{
```

```
c=a-b;
```

```
}
```

```
else
```

```
{
```

```
c=a+b;
```

```
}
```

```
}
```

Output:

```
$ lex int.l
```

```
$ yacc -d int.y
```

```
$ gcc lex.yy.c y.tab.c -ll -lm
```

```
$ ./a.out test.c
```

Pos	Operator	Arg1	Arg2	Result
0	<	a	b	t0
1	==	t0	FALSE	5
2	+	a	b	t1
3	=	t1		a
4	GOTO			5
5	<	a	b	t2
6	==	t2	FALSE	10
7	+	a	b	t3
8	=	t3		a
9	GOTO			5
10	<=	a	b	t4
11	==	t4	FALSE	15
12	-	a	b	t5
13	=	t5		c
14	GOTO			17
15	+	a	b	t6
16	=	t6		c

EXPERIMENT - VI

Aim: Write program to generate machine code from the intermediate code of Quadruple representation.

Description:

Intermediate Code Representation (Three-Address Code - TAC):

Intermediate Code Representation (ICR) is a crucial phase in compiler design that helps in code optimization and machine-independent analysis. One of the most common forms of ICR is Three-Address Code (TAC).

What is Three-Address Code (TAC)?

- TAC breaks complex expressions into simple instructions.
- Each instruction has at most three operands (hence the name "three-address").
- It is machine-independent and serves as a bridge between high-level code and machine code.

The general forms of TAC:

General Form: $x = y \text{ op } z$ // Binary operation (e.g., +, -, *, /, %)

$x = \text{op } y$ // Unary operation (e.g., -, !)

$x = y$ // Simple assignment

Three-Address Code (TAC) can be represented in three different ways:

1. Quadruples (4-Tuples)
2. Triples (3-Tuples)
3. Indirect Triples

Target Program Representation

The main types of target program representations include:

1. Absolute Machine Code
2. Relocatable Machine Code
3. Assembly Language Representation

1. Absolute Machine Code

Absolute machine code is a directly executable binary representation of the program. It is fully prepared for execution and requires no further linking or relocation.

Example (Pure Binary Representation)

Instruction	Binary Representation	Hex Representation
MOV AX, 5	10111000 00000101	B8 05
ADD AX, BX	00000011 11011000	03 D8
MOV [1000], AX	10110000 00001000 00110100	B0 08 34

2. Relocatable Machine Code

Relocatable machine code is not directly executable. Instead of absolute memory addresses, it contains placeholders for memory locations, which are resolved during linking.

Example (x86 Relocatable Machine Code)

Instruction	Binary (Placeholder Used)	Hex Representation
MOV AX, VAR1	10111000 XXXX XXXX (XX = VAR1 Address)	B8 XX XX
ADD AX, BX	00000011 11011000	03 D8
MOV [VAR2], AX	10110000 YYYY YYYY (YY = VAR2 Address)	B0 YY YY

3. Assembly Language Representation

Assembly language is a human-readable version of machine instructions. It is a step above pure machine code and needs an assembler to convert it into executable machine code.

Example (x86 Assembly Code)

```
MOV AX, 10    ; Load 10 into AX
MOV BX, 20    ; Load 20 into BX
ADD AX, BX    ; AX = AX + BX
MOV [RESULT], AX ; Store result
```

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int label[20];
int no=0;
int main()
{
FILE *fp1,*fp2;
```

```

char fname[10],op[10],ch;
char operand1[8],operand2[8],result[8];
int i=0,j=0;
printf("\n Enter filename of the intermediate code");
scanf("%s",fname);
fp1=fopen(fname,"r");
fp2=fopen("target.txt","w");
if(fp1==NULL || fp2==NULL)
{
printf("\n Error opening the file");
exit(0);
}
while(!feof(fp1))
{
fprintf(fp2,"\n");
fscanf(fp1,"%s",op);
i++;
if(check_label(i))
fprintf(fp2,"\nlabel#%d",i);
if(strcmp(op,"print")==0)
{
fscanf(fp1,"%s",result);
fprintf(fp2,"\n\t OUT %s",result);
}
if(strcmp(op,"goto")==0)
{
fscanf(fp1,"%s %s",operand1,operand2);
fprintf(fp2,"\n\t JMP %s,label#%s",operand1,operand2);
label[no++]=atoi(operand2);
}
if(strcmp(op,"[]")==0)
{
fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t STORE %s[%s],%s",operand1,operand2,result);
}
}

```

```

if(strcmp(op,"uminus")==0)
{
fscanf(fp1,"%s %s",operand1,result);
fprintf(fp2,"\n\t LOAD -%s,R1",operand1);
fprintf(fp2,"\n\t STORE R1,%s",result);
}
switch(op[0])
{
case '*': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t MUL R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '+': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t ADD R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '-': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t SUB R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '/': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t DIV R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '%': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);

```



```

fprintf(fp2, "\n \t DIV R1,R0");
fprintf(fp2, "\n \t STORE R0,%s",result);
break;
case '=': fscanf(fp1,"%s %s",operand1,result);
fprintf(fp2, "\n \t STORE %s %s",operand1,result);
break;
case '>': j++;
fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2, "\n \t LOAD %s,R0",operand1);
fprintf(fp2, "\n \t JGT %s,label#%s",operand2,result);
label[no++]=atoi(result);
break;
case '<': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2, "\n \t LOAD %s,R0",operand1);
fprintf(fp2, "\n \t JLT %s, label#%d",operand2,result);
label[no++]=atoi(result);
break;
} }
fclose(fp2); fclose(fp1);
fp2=fopen("target.txt","r");
if(fp2==NULL)
{
printf("Error opening the file\n");
exit(0);
}
do
{
ch=fgetc(fp2);
printf("%c",ch);
}while(ch!=EOF);
fclose(fp1);
return 0;
}
int check_label(int k)
{

```

```

int i;
for(i=0;i<no;i++)
{
if(k==label[i])
return 1;
}
return 0;
}

```

OUTPUT:

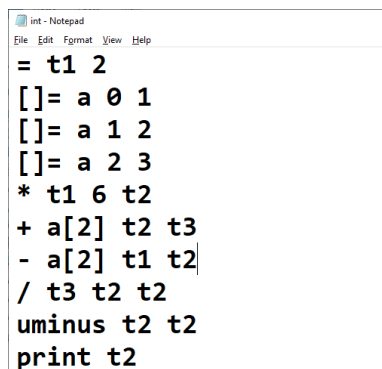
```

C:\TURBOC3\BIN>TC

Enter filename of the intermediate codeint.txt

```

Input (Quadruple representation) is provided in a separate text file named "int.txt" as shown below



```

int - Notepad
File Edit Format View Help
= t1 2
[] = a 0 1
[] = a 1 2
[] = a 2 3
* t1 6 t2
+ a[2] t2 t3
- a[2] t1 t2
/ t3 t2 t2
uminus t2 t2
print t2

```

Result (Assembly Code) is stored in "target.txt", as shown below:

```

STORE t1 2
STORE a[0],1
STORE a[1],2
STORE a[2],3
LOAD t1,R0
LOAD 6,R1
MUL R1,R0
STORE R0,t2

LOAD a[2],R0

```

LOAD t2,R1
ADD R1,R0
STORE R0,t3

LOAD a[2],R0
LOAD t1,R1
SUB R1,R0
STORE R0,t2

LOAD t3,R0
LOAD t2,R1
DIV R1,R0
STORE R0,t2

LOAD -t2,R1
STORE R1,t2
OUT t2

EXPERIMENT - VII

Aim: Write a C program to Implement of Symbol Table.

Description:

Definition:

A symbol table is a data structure used by a compiler to store and manage information about identifiers (variables, functions, objects, classes, etc.) used in a program. It serves as a lookup table for the compiler during various phases of compilation.

Role of Symbol Table in Compilation:

The symbol table plays a crucial role in different phases of the compiler:

1. Lexical Analysis
 - Recognizes identifiers (variables, functions, etc.) and adds them to the symbol table.
2. Syntax Analysis (Parsing)
 - Ensures correct usage of identifiers according to grammar rules.
3. Semantic Analysis
 - Checks type consistency and scope of identifiers.
4. Intermediate Code Generation
 - Helps in generating optimized intermediate representations.
5. Code Optimization
 - Assists in eliminating redundant computations and improving performance.
6. Code Generation
 - Provides memory addresses and offsets for efficient machine code generation.

Contents of a Symbol Table

A typical symbol table stores the following information:

Identifier	Type	Scope	Memory Address	Value
X	int	global	0x600003e3a0	10
Y	float	local	0x600003e3b0	3.14
sum()	func	global	0x600003e3c0	-

- Identifier Name: Name of the variable, function, or object.
- Type: Data type (e.g., int, float, char, function).
- Scope: Local, global, or block-level.
- Memory Address: Address allocated during runtime.
- Value: Initial or assigned value (if applicable).

Operations on Symbol Table

A symbol table supports the following operations:

1. Insertion: Adds new identifiers during lexical analysis.
2. Lookup/Search: Checks if an identifier is already declared.
3. Modification: Updates information (e.g., assigned values).
4. Deletion: Removes identifiers after scope completion.

Implementation of Symbol Table

A symbol table can be implemented using different data structures:

Data Structure	Advantages	Disadvantages
Array	Simple and fast lookup	Fixed size, inefficient for large programs
Linked List	Dynamic size	Slower search time ($O(n)$)
Hash Table	Fast search ($O(1)$ avg)	More memory usage
Binary Search Tree (BST)	Efficient for sorted data	Slower than hash table
Trie (Prefix Tree)	Good for language processing	High memory usage

A symbol table is an essential part of a compiler, helping in identifier management, type checking, and memory allocation. Efficient implementation can significantly improve compiler performance and code optimization.

PROGRAM:

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

#define MAX_IDENTIFIERS 50 // Maximum number of identifiers
#define MAX_LENGTH 100    // Maximum length of input expression
// Structure to store symbol table entry
struct Symbol {
    char name[MAX_LENGTH]; // Identifier name
    char type[10];         // Data type (int, float, etc.)
    char scope[10];        // Scope (global, local)
    void *address;         // Memory address
    char value[20];        // Value (if assigned)
```

```

};

// Symbol table array
struct Symbol symbolTable[MAX_IDENTIFIERS];
int i, symbolCount = 0; // Number of identifiers stored

// Function to add an identifier to the symbol table
void insertSymbol(char name[], char type[], char scope[], char value[]) {
    struct Symbol *s = &symbolTable[symbolCount]; // Get next free entry
    strcpy(s->name, name);
    strcpy(s->type, type);
    strcpy(s->scope, scope);
    s->address = malloc(strlen(name) + 1); // Allocate memory
    strcpy(s->value, value);
    symbolCount++; // Increase count
}

// Function to display the symbol table
void displaySymbolTable() {
    printf("\nSymbol Table:\n");
    printf("-----\n");
    printf("Name    Type    Scope    Address    Value\n");
    printf("-----\n");
    for (i = 0; i < symbolCount; i++) {
        printf("%-10s %-8s %-8s %p  %s\n",
            symbolTable[i].name,
            symbolTable[i].type,
            symbolTable[i].scope,
            symbolTable[i].address,
            symbolTable[i].value);
    }
    printf("-----\n");
}

// Function to check if a word is a data type
int isDataType(char *word) {
    return (strcmp(word, "int") == 0 || strcmp(word, "float") == 0 || strcmp(word, "char") == 0);
}

void main() {

```

```

char input[MAX_LENGTH], c;
int i = 0, j = 0;
char lastType[10] = ""; // Store last detected type
clrscr();
printf("Enter expression terminated by $: ");
while ((c = getchar()) != '$' && i < MAX_LENGTH - 1) { // Read input
    input[i] = c;
    i++;
}
input[i] = '\0'; // Null-terminate string
printf("\nGiven Expression: %s\n", input);
// Parse input and store identifiers
while (j < i) {
    if (isalpha(input[j])) { // If character is alphabetic (identifier start)
        char identifier[MAX_LENGTH];
        int k = 0;
        // Collect the whole word
        while (isalnum(input[j])) {
            identifier[k++] = input[j++];
        }
        identifier[k] = '\0'; // Null-terminate
        // Check if it's a datatype
        if (isDataType(identifier)) {
            strcpy(lastType, identifier); // Store last found datatype
        } else {
            char value[20] = "0"; // Default value
            // Check if next char is '=' to capture assigned value
            if (input[j] == '=') {
                j++; // Move past '='
                k = 0;
                while (isdigit(input[j]) || input[j] == '.') {
                    value[k++] = input[j++];
                }
                value[k] = '\0'; // Null-terminate value
            }
        }
    }
}

```

```

        // Insert into symbol table
        insertSymbol(identifier, lastType, "global", value);
    }
} else {
    j++; // Move to next character
}
}

// Display the final symbol table
displaySymbolTable();

// Free allocated memory
for (i = 0; i < symbolCount; i++) {
    free(symbolTable[i].address);
}

getch();
}

```

Output:

```

C:\TURBOC3\BIN>TC
Enter expression terminated by $: float num=7.5;int k=5;$

Given Expression: float num=7.5;int k=5;

Symbol Table:
-----
Name      Type      Scope      Address      Value
-----
num       float     global     23BC         7.5
k         int       global     23C4         5
-----

```