Compiler Design Lab Report

Name: YENDLURI SATHVIKA

Roll no: CH.EN.U4CSE22164

Course Code: 19CSE401

Basic Programs

1. **Aim:** Program to Identify Vowels and Consonants

Algorithm:

- Open the gedit text editor from Accessories under Applications menu.
- Specify the header file <stdio.h> between %{ and %}.
- Define the character patterns for vowels [aAeEiIoOuU], alphabets [a-zA-Z], whitespaces [\t\n], and other characters ..
- Use translation rules to print whether the character is a vowel, consonant, or not an alphabet character.
- Call yylex() inside the main() function to begin lexical analysis.
- Save the program as vowelconsonant.l using the LEX language.
- Run the program using the LEX compiler to generate lex.yy.c.
- The generated lex.yy.c contains tables and routines to match input characters.
- Compile lex.yy.c using a C compiler to create an executable file.
- Run the executable to check each character in the input and classify it.

Code:

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

%%

[aAeEiIoOuU]        { printf("%s is a VOWEL\n", yytext); }
[a-zA-Z]            { printf("%s is a CONSONANT\n", yytext); }
[ \t\n]             ;  // Ignore whitespace
.                   { printf("%s is not an alphabet character\n", yytext); }

%%

int main() {
    yylex();
    return 0;
}

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

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ cd Downloads
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ flex q1.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ gcc lex.yy.c -ll -o scanner
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ ./scanner
sathvika
s is a CONSONANT
a is a VOWEL
t is a CONSONANT
h is a CONSONANT
v is a CONSONANT
i is a VOWEL
k is a CONSONANT
a is a VOWEL
```

2. **Aim:** Program to Count Lines, Words, and Characters

Algorithm:

- Open the gedit text editor from Accessories under Applications menu.
- Include the header file <stdio.h> between %{ and %}.
- Declare and initialize line, word, and character counters.
- Define regular expressions for newline, whitespace, and words.
-
- Use translation rules to update the respective counters.
- Call yylex() inside the main() function.
- Print the final count of lines, words, and characters.
- Save the program as counter.l.
- Run the program using the LEX compiler to generate lex.yy.c.
- Compile lex.yy.c using a C compiler to produce the executable.
- Run the executable to perform the counting operation on input.

Code:

```
%{
#include <stdio.h>
int lines = 0, words = 0, chars = 0;
%}

%%

\n              { lines++; chars++; }
[ \t]+          { chars += yyleng; }
[^ \t\n]+       { words++; chars += yyleng; }

%%

int main() {
    yylex();
    printf("\nLines: %d\nWords: %d\nCharacters: %d\n", lines, words, chars);
    return 0;
}

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

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ flex q2.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ gcc lex.yy.c -ll -o scanner
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ ./scanner
sathvika
13102005
** ## %^&



Lines: 6
Words: 5
Characters: 32
```

- Use translation rules to identify and print whether input is float, integer, or not a number.
- Ignore whitespaces like tab, space, and newline.
- Call yylex() inside the main() function to start lexical analysis.
- Save the program as numcheck.l.
- Run the program using the LEX compiler to generate lex.yy.c.
- Compile lex.yy.c using a C compiler to get the executable.
- Run the executable to test inputs and identify the type of number.

Code:

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

%%

[0-9]+\.[0-9]+     { printf("%s is a FLOATING POINT number\n", yytext); }
[0-9]+             { printf("%s is an INTEGER\n", yytext); }
[ \t\n]            ; // Ignore whitespace
.                  { printf("%s is not a number\n", yytext); }

%%

int main() {
    yylex();
    return 0;
}

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

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ flex q3.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ gcc lex.yy.c -ll -o scanner
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ ./scanner
67.89
67.89 is a FLOATING POINT number
73
73 is an INTEGER
22
22 is an INTEGER
6431
6431 is an INTEGER
19
19 is an INTEGER
```

4. **Aim:** Program to Recognize C Keywords

Algorithm:

- Open the gedit text editor from Accessories under Applications menu.
- Include the header file <stdio.h> between %{ and %}.
- Define regular expressions for C keywords, identifiers, whitespaces, and other characters.
- Use translation rules to print whether input is a C keyword, identifier, or something else.
- Ignore spaces, tabs, and newline characters.
- Call yylex() in the main() function to begin lexical analysis.
- Save the program as keywordid.l.
- Run the program through the LEX compiler to generate lex.yy.c.
- Compile lex.yy.c using a C compiler to get the final executable.
- Run the executable to classify each token as keyword, identifier, or other.

Code:

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

%%

"int" |
"float" |
"return" |
"if" |
"else" |
"while" |
"for"                   { printf("%s is a C keyword\n", yytext); }

[a-zA-Z_][a-zA-Z0-9_]* { printf("%s is an identifier\n", yytext); }

[ \t\n]                 ; // Ignore spaces

.                       { printf("%s is something else\n", yytext); }

%%

int main() {
    yylex();
    return 0;
}

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

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ flex q4.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ gcc lex.yy.c -ll -o scanner
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ ./scanner
for
for is a C keyword
and
and is an identifier
can
can is an identifier
sathvika yendluri
sathvika is an identifier
yendluri is an identifier
15
1 is something else
5 is something else
```

5. **Aim:** Program to Recognize Operators

Algorithm:

- Open the gedit text editor from Accessories under Applications menu.
- Include the header file <stdio.h> between %{ and %}.
- Define regular expressions for relational operators, arithmetic/assignment operators, whitespaces, and other characters.
- Use translation rules to check and print whether input is a relational operator, arithmetic/assignment operator, or not an operator.
- Ignore whitespaces like tab and newline characters.
- Call yylex() inside the main() function to begin lexical analysis.
- Save the program as operatorcheck.l.

- Run the program through the LEX compiler to generate lex.yy.c.
- Compile lex.yy.c using a C compiler to get the executable.
- Run the executable to test and classify the input operators.

Code:

```
%{
#include <stdio.h>
%}
%%
"==" |
"!=" |
"<=" |
">=" |
"<" |
">"                    { printf("%s is a relational operator\n", yytext); }

"+" |
"-" |
"*" |
"/" |
"="                    { printf("%s is an arithmetic/assignment operator\n", yytext); }

[ \t\n]                ; // Ignore spaces

.                      { printf("%s is not an operator\n", yytext); }

%%

int main() {
    yylex();
    return 0;
}

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

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ flex q5.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ gcc lex.yy.c -ll -o scanner
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ ./scanner
%
% is not an operator
>
> is a relational operator
>=
>= is a relational operator
+
+ is an arithmetic/assignment operator
&
& is not an operator
```

EXPERIMENT NO – 1

Aim: To implement Lexical Analyzer Using Lex Tool

Algorithm:

- Open gedit text editor from Accessories in Applications.
- Specify the header files to be included inside the declaration part (i.e. between %{ and %}).
- Define the digits 0-9 and identifiers a-z and A-Z.
- Using translation rules, define the regular expressions for digit, keywords, identifiers, operators, header files etc. If matched with the input, store and display using yytext.
- Inside procedure main (), use yyin() to point to the current file being passed by the lexer.
- The specification of the lexical analyzer is prepared by creating a program lab1.l in the LEX language.
- The lab1.l program is run through the LEX compiler to produce equivalent C code named lex.yy.c.
- The program lex.yy.c consists of a table constructed from the regular expressions of lab1.l, along with standard routines that use the table to recognize lexemes.
- Finally, the lex.yy.c program is run through a C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

Code:

Lab1.l:

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

int COMMENT = 0;
%}

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

%%

#.*                     { printf("\n%s is a preprocessor directive", yytext); }

int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto                    { printf("\n\t%s is a keyword", yytext); }

"/*"                    { COMMENT = 1; printf("\n\t%s is a COMMENT", yytext); }

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

\{                      { if (!COMMENT) printf("\n BLOCK BEGINS"); }

\}                      { if (!COMMENT) printf("BLOCK ENDS "); }

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

\".*\"                  { if (!COMMENT) printf("\n\t%s is a STRING", yytext); }
```

```
[0-9]+                  { if (!COMMENT) printf("\n %s is a NUMBER", yytext); }

\)(\:)?                 { if (!COMMENT) { printf("\n\t"); ECHO; printf("\n"); } }

\(                      { ECHO; }

=                       { if (!COMMENT) printf("\n\t%s is an ASSIGNMENT OPERATOR", yytext); }

\<= |
\>= |
\< |
== |
\>                      { if (!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR", yytext); }

%%

int main(int argc, char **argv)
{
    FILE *file;
    file = fopen("var.c", "r");
    if (!file)
    {
        printf("Could not open the file\n");
        exit(0);
    }

    yyin = file;
    yylex();
    printf("\n");
    return 0;
}

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

Var.c:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
a=1;
b=2;
c=a+b;
printf("Sum:%d",c);
}
```

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ lex lab1.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ cc lex.yy.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ ./a.out

#include<stdio.h> is a preprocessor directive

#include<conio.h> is a preprocessor directive

        void is a keyword
FUNCTION
        main(
        )


 BLOCK BEGINS

        int is a keyword
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;

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

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

 c IDENTIFIER
        = is an ASSIGNMENT OPERATOR
 a IDENTIFIER+
 b IDENTIFIER;

FUNCTION
        printf(
        "Sum:%d" is a STRING,
 c IDENTIFIER
        )
;
BLOCK ENDS
```

- After the common part, append 'X' to modifiedGram to denote the new non-terminal.
- Create newGram to store the restructured productions from the remaining suffixes of part1 and part2.
- Display the final left-factored productions using printf().

Code:

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

int main() {
    char gram[100], part1[100], part2[100], modifiedGram[100], newGram[100];
    int i, j = 0, k = 0, pos = 0;

    printf("Enter Production : A->");
    fgets(gram, sizeof(gram), stdin);

    gram[strcspn(gram, "\n")] = 0;

    for (i = 0; gram[i] != '|' && gram[i] != '\0'; i++, j++) {
        part1[j] = gram[i];
    }
    part1[j] = '\0';

    if (gram[i] == '|') {
        i++;
    }
    for (j = 0; gram[i] != '\0'; i++, j++) {
        part2[j] = gram[i];
    }
    part2[j] = '\0';

    for (i = 0; i < strlen(part1) && i < strlen(part2); i++) {
        if (part1[i] == part2[i]) {
            modifiedGram[k] = part1[i];
            k++;
            pos = i + 1;
        } else {
            break;
        }
    }

    modifiedGram[k] = 'X';
    modifiedGram[k + 1] = '\0';


    j = 0;
    for (i = pos; i < strlen(part1); i++, j++) {
        newGram[j] = part1[i];
```

```c
    modifiedGram[k] = 'X';
    modifiedGram[k + 1] = '\0';


    j = 0;
    for (i = pos; i < strlen(part1); i++, j++) {
        newGram[j] = part1[i];
    }
    newGram[j++] = '|';
    for (i = pos; i < strlen(part2); i++, j++) {
        newGram[j] = part2[i];
    }
    newGram[j] = '\0';

    printf("\nA->%s", modifiedGram);
    printf("\nX->%s\n", newGram);

    return 0;
}
```

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ gcc qq.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ ./a.out
Enter Production : A->aE+bcD|aE+eIT

A->aE+X
X->bcD|eIT
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$
```