

SPCC Experiments

1.Implementation of Lexical Analyzer in C / Java / Python

Lexical analysis is the first phase of a compiler. It takes modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into **a series of tokens**, by **removing any whitespace or comments** in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Token: Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

Example: In programming language, **keywords, constants, identifiers, strings, numbers, operators and punctuations symbols** can be considered as tokens.

```

import re

keyword =
['break','case','char','const','continue','default','do','i
nt','else','enum','extern','float','for','goto','if','long'
,'register','return','short','signed','sizeof','static','sw
itch','typedef','union','unsigned','void','volatile','while
']
built_in_functions =
['clrscr()','printf(',')','scanf(',')','getch(',')','main()']
operators =
['+','-','*','/','%','==','!=','>','<','>=','<=','&&','||',
'!','&','|','^','~','>>','<<','=','+=','-=','*=']
specialsymbol = ['@','#','$','_','!']
separator = [',',':',';','\n','\t','{','}','(',')','[',']']

file = open('lexical.txt','r+')
contents = file.read()
splitCode = contents.split() #split program in word based
on space
length = len(splitCode)      # count the number of word in
program
for i in range(0,length):
    if splitCode[i] in keyword:
        print("Keyword -->",splitCode[i])
        continue
    if splitCode[i] in operators:
        print("Operators --> ",splitCode[i])
        continue
    if splitCode[i] in specialsymbol:
        print("Special Operator -->",splitCode[i])
        continue

```

```
if splitCode[i] in built_in_functions:
    print("Built_in Function -->",splitCode[i])
    continue
if splitCode[i] in separator:
    print("Separator -->",splitCode[i])
    continue
if re.match(r'(#include*).*', splitCode[i]):
    print ("Header File -->", splitCode[i])
    continue
if re.match(r'^[-+]?[0-9]+$',splitCode[i]):
    print("Numerals --> ",splitCode[i])
    continue
if re.match(r"^[^d\W]\w*\Z", splitCode[i]):
    print("Identifier --> ",splitCode[i])
```

Steps:

Create list of tokens,Read file, split words, check if word belongs to keyword,symbol,special character,numerals,identifier and print.

☆☆☆Lexical Analyzer using FLEX

```
/** Definition Section has one variable
which can be accessed inside yylex()
and main() */
%{
int count = 0;
%}

/** Rule Section has three rules, first rule
matches with capital letters, second rule
matches with any character except newline and
third rule does not take input after the enter*/
%%
[A-Z] {printf("%s capital letter\n", yytext);
      count++;}
.      {printf("%s not a capital letter\n", yytext);}
\n {return 0;}
%%

/** Code Section prints the number of
capital letter present in the given input*/
int yywrap(){}
int main(){

// Explanation:
// yywrap() - wraps the above rule section
/* yyin - takes the file pointer
           which contains the input*/
/* yylex() - this is the main flex function
           which runs the Rule Section*/
// yytext is the text in the buffer

// Uncomment the lines below
```

```

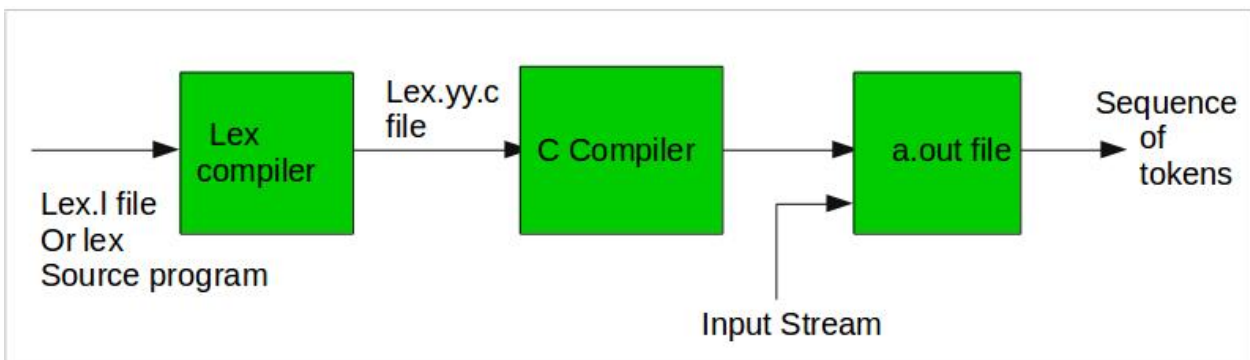
// to take input from file
// FILE *fp;
// char filename[50];
// printf("Enter the filename: \n");
// scanf("%s",filename);
// fp = fopen("example.txt","r");
// yyin = fp;

yylex();
printf("\nNumber of Capital letters "
      "in the given input - %d\n", count);

return 0;
}

```

★★★★ THE RULE SECTION WILL CHANGE BASED ON QUESTION



Program Structure:

In the input file, there are 3 sections:

1. **Definition Section:** The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in “%{ %}” brackets. Anything written in this brackets is copied directly to the file lex.yy.c

Syntax:

```
%{  
    // Definitions  
%}
```

2. **Rules Section:** The rules section contains a series of rules in the form: pattern action and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in “%% %%”.

Syntax:

```
%%  
pattern  action  
%%
```

Examples: Table below shows some of the pattern matches.

3. **User Code Section:** This section contains C statements and additional

functions. We can also compile these functions separately and load with the lexical analyzer.

Basic Program Structure:

```
%{  
// Definitions  
%}  
  
%%  
Rules  
%%  
  
User code section
```

2.Implement Lexical Analyzer using FLEX Count no of Vowels & Consonants

Code:

```
%{  
    int vow_count=0;  
    int const_count =0;  
%}  
  
%%  
[aeiouAEIOU] {vow_count++;}  
[a-zA-Z] {const_count++;}  
%%  
int yywrap(){  
int main()  
{  
    printf("Enter the string of vowels and consonants:");
```

```

yylex();
printf("Number of vowels are: %d\n", vow_count);
printf("Number of consonants are: %d\n", const_count);
return 0;
}

```

★ ★ ★ Steps

1. Define variables in definition section:
Here, vowel count and consonant count
2. Define Rules in rules section:
 - a. for matching vowels, Rule1 [aeiouAEIOU]
 - b. for consonants, rest of the alphabet [a-zA-Z]
3. Write print statements in user code section

EG)

```

%{
    DEFINITION SECTION
%}
%%
    RULES SECTION
%%
int yywrap(){}
int main(){
    yylex(); // SHOULD BE AFTER FILE READ!
    CODE (mostly print statements);
    return 0;
}

```


3.Implement Lexical Analyzer using FLEX Count no of Words, characters & lines

Definitions:

count_word,count_char,count_lines

Rules:

for word,

([a-zA-Z0-9])*

for lines,

\n

for char,

length of every word(yylen),new line(+1for each line),everything else (.)

CODE

```
%{
    int word_count,line_count,char_count=0;
}%

%%
([a-zA-Z0-9])* {word_count++;char_count+=yyLeng;}
\n {line_count++;char_count++;}
. {char_count++;}
%%

int yywrap(){
int main(){
    FILE *fp;
    fp=fopen("lol.txt","r");
```

```

yyin=fp;
yylex();
printf("Words %d\n",word_count);
printf("Lines %d\n",line_count);
printf("Chars %d\n",char_count);
return 0;
}

```

4.Implement Lexical Analyzer using FLEX

Count no of keywords,identifiers & operators

Rules:

keywords: make list

eg) "main"|"print"|"falana"|"dhimkana" {keyword++;}

identifiers: make rules (starts with underscore or alpha,and can have numbers)

`[a - z A - Z _][a - z A - Z 0 - 9 _]*`

STARTS WITH ONE ALPHABET OR UNDERSCORE
FOLLOWED BY ONE OR MORE ALPHABET OR NUMBER

operators: make list

CODE

```

%{
    int keyword,id,op;
}%

%%
"scanf"|"printf"|"main"|"void"|"int"|"return" {printf("%s
is a KEYWORD \n",yytext);keyword++;}
[a-zA-Z_][a-zA-Z0-9_]* {printf("%s is an

```

```

IDENTIFIER\n",yytext);id++;}
"+"|"="|"-"|"%"|"/" {printf("%s is an
OPERATOR\n",yytext);op++;}
. ;
%%

int yywrap(){
int main(){
    FILE *fp;
    fp=fopen("lol.c","r");
    yyin=fp;
    yylex();
    printf("Keywords %d\n",keyword);
    printf("Identifiers %d\n",id);
    printf("Operators %d\n",op);
    return 0;
}

```

eg [+|-|=|%|/] {operator++;}

5.Implement Lexical Analyzer using FLEX

Identify Even & odd integers

```

%{
    int i;
}%

%%
[0-9]+    {i=atoi(yytext);
            if(i%2==0)
                printf("%d is Even \n",i);
            else
                printf("%d is Odd \n",i);}

```

```

%%

int yywrap(){}
int main(){
    FILE *fp;
    fp=fopen("lol.txt","r");
    yyin=fp;
    yylex();
    return 0;
}

```

6.Implement Lexical Analyzer using FLEX Count of printf & scanf statements in C program\

CODE

```

%{
    int scan_count,print_count;
}%

%%

```

```

"scanf(" {scan_count++;}
"printf(" {print_count++;}
. ;
%%

int yywrap(){}
int main(){
    FILE *fp;
    fp=fopen("lol.c","r");
    yyin=fp;
    yylex();
    printf("Scan %d\n",scan_count);
    printf("Print %d\n",print_count);
    return 0;
}

```

INPUT FILE

```

#include <stdio.h>

int main()
{
    float num1;
    double num2;

    printf("Enter a number: ");
    scanf("%f", &num1);

    printf("Enter another number: ");
    scanf("%lf", &num2);

    printf("num1 = %f\n", num1);
    printf("num2 = %lf", num2);
}


```


```
return 0;  
}
```

SYNTAX ANALYSIS

Syntax Analysis is a second phase of the compiler design process in which the given input string is checked for the confirmation of rules and structure of the formal grammar. It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.

 **7. Write a program to find FIRST & FOLLOW Symbols for the given grammar.**

 **8. Implement Syntax Analyzer in C / Java / Python - Eg. LL(1) Generate the Predictive Parsing Table (take FIRST and FOLLOW as input for any grammar)**

 **9. Implement Syntax Analyzer in C / Java / Python - Eg. LL(1) Perform Parsing action for valid & invalid inputs based on the Parsing Table Generated**



10 Implementation of Operator Precedence Parser *

```
t = [
['error','>','>','>'],
['<','>','<','>'],
['<','>','>','>'],
['<','<','<','Accept']]
row = ['id','+','*','$']
col = ['id','+','*','$']

stack=[]
top = -1
stack.append('$')
string_length = int(input("Enter length of string : "))
l = []
print("Enter Input")
for i in range(0,string_length):
    a = input()
    l.append(a)
l.append('$')
print("Stack : ",stack)
print("Input Buffer : ",l)
res = 1
while(res == 1):
    b = stack[top]
    c = l[0]
    d = row.index(b)
    e = col.index(l[0])
    if t[d][e] == '<':
        print("\n",l[0]," Push on the stack")
        stack.append(l[0])
        l.remove(l[0])
        print("Stack : ",stack)
        print("Input Buffer : ",l)
    elif t[d][e] == '>':
        print("\n",l[0]," Pop from stack")
        stack.pop()
        print("Stack : ",stack)
        print("Input Buffer : ",l)
    elif b == c == 'id':
        print("Error")
        res = 0
```

```
elif b == c == '$':  
    print("String Accepted")  
    res = 0
```

YACC (3 parts)

 Introduction to yacc

TO RUN:

- 1) flex example.l
- 2) bison -d -y example.y
- 3) a.exe (WINDOWS) /a.o (LINUX)

11. Parser Generator Tool : YACC Simple Calculator

cal.l

```
%{  
    /* Definition section */  
    #include<stdio.h>  
    #include "y.tab.h"  
    extern int yylval;  
}%  
  
/* Rule Section */  
%%  
[0-9]+ {  
    yylval=atoi(yytext);  
    return NUMBER;  
}  
[\t] ;  
[\n] return 0;  
.  
return yytext[0];
```



```
%%

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

CAL.y

```
%{
    /* Definition section */
    #include<stdio.h>
    int flag=0;
}%

%token NUMBER

%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

/* Rule Section */
%%

ArithmeticExpression: E{

    printf("\nResult=%d\n", $$);

    return 0;

};

E: E '+' E {$$=$1+$3;}
```

```

| E '-' E {$$=$1-$3;}

| E '*' E {$$=$1*$3;}

| E '/' E {$$=$1/$3;}

| E '%' E {$$=$1%$3;}

| '(' E ')' {$$=$2;}

| NUMBER {$$=$1;}

;

%%

//driver code
void main()
{
    printf("\nEnter Any Arithmetic Expression :\n");

    yyparse();
    if(flag==0)

}

void yyerror()
{
    printf("\nEntered arithmetic expression is
Invalid\n\n");
    flag=1;
}

```

12. Parser Generator Tool : YACC Recognize nested IF stmt and display levels

.l

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

%%
"if" {return IF;}
[sS][0-9]* {return S;}
"<" | ">" | "==" | "<=" | ">=" | "!=" {return RELOP;}
[0-9]+ {return NUM;}
[a-zA-z_][a-zA-Z0-9_]* {return ID;}
\n {return NEWLINE;}
. {return yytext[0];}
%%

int yywrap(){
    return 0;
}
```

.y

```
%{
#include<stdio.h>
int count = 0;
%}

%token IF ID NUM RELOP NEWLINE S

%%
smt: if_smt NEWLINE {printf("Nested ifs=%d\n",count);}
    ;
if_smt : IF '(' cond ')' '{if_smt}' {count++;}
        | S
```

```

        ;
cond : x RELOP x
        ;
x : ID
  | NUM
  ;
%%

int yyerror(char *s){
    printf("Invalid Statement\n");
    return 0;
}

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

```



13. Parser Generator Tool : YACC Program to recognize a valid variable in C language

```

%{

    #include "y.tab.h"

%}
%%

[a-zA-Z][a-zA-Z_0-9]* return letter;

[0-9]                return digit;

.                    return yytext[0];

\n                   return 0;

%%

```

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

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

    int valid=1;
}%}

%token digit letter

%%

start : letter s
s : letter s
    | digit s
    |
    ;

%%

int yyerror()
{
```

```

        printf("\nIts not a identifier!\n");

        valid=0;

        return 0;
    }

    int main()
    {

        printf("\nEnter a name to tested for identifier ");

        yyparse();
        if(valid)
        {

            printf("\nIt is a identifier!\n");

        }

    }

```

14. Implement Intermediate Code Generation using LEX and YACC

```
%{
```

```

#include "y.tab.h"
#include<string.h>
%}

%%

[0-9]+ {strcpy(yylval.str, yytext); return NUM;}
[a-zA-Z_][a-zA-Z0-9_]* {strcpy(yylval.str, yytext); return ID;}
\n {return 0;}
[ \t] {}
. {return yytext[0];}
%%

int yywrap(){
    return 0;
}

```

```

%{
#include<stdio.h>
#include<string.h>
char *createT();
int tCount = 0;
%}

%union{char str[30];}
%left '+'
%left '-'
%left '*'
%left '/'
%token <str> ID
%token <str> NUM
%type <str> assign
%type <str> exp

%%

assign : ID '=' exp {printf("\n%s=%s", $1, $3);}

```

```

exp : '('exp')' {strcpy($$, $2);}
    | exp '+'exp {strcpy($$, createT()); printf("\n%s=%s+%s", $$, $1,
$3);}
    | exp '-'exp {strcpy($$, createT()); printf("\n%s=%s-%s", $$, $1,
$3);}
    | exp '*'exp {strcpy($$, createT()); printf("\n%s=%s*%s", $$, $1,
$3);}
    | exp '/'exp {strcpy($$, createT()); printf("\n%s=%s/%s", $$, $1,
$3);}
    | NUM {strcpy($$, $1);}
    | ID {strcpy($$, $1);}

```

```
%%
```

```

char *createT(){
    char snum[30], *ptr;
    sprintf(snum, "t%d", tCount);
    ptr = snum;
    tCount++;
    return ptr;
}

```

```

int main(){
    printf("Expression: ");
    yyparse();
    return 0;
}

```

```

int yyerror(char *s){
    printf("\nInvalid Expression");
    return 0;
}

```