**WEEK-1**

**AIM:** To write a program to count number of spaces, characters, tabs and newlines.

**DESCRIPTION:**

Lex is a computer program that generates lexical analyzers and was written by Mike Lesk and Eric Schmidt. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

We need to read each character from the input file and analyse them and then categorize them into characters, spaces, tabs, etc. Scan the entire file and check for the relevant conditions and printthe count accordingly.

**ALGORITHM:**

* In the definition section include the libraries string.h and math.h and predefine check function.
* In the rules section match the given input to characters,spaces,lines,tabs and run the main function with yytext as parameter where yytext is the input buffer.
* In the user code section create main function, read input and run the yylex() function.
* create few new variable such as cc, ct, cs, cl each used to store the count of characters, spaces, lines and tab in the given input.
* using the regular expressions, find the number of occurrences of characters, spaces, lines and tab in the given input.
* After the while loop is terminated print the count of each occurrence.

**CODE:**

file = open("example.txt","r")

data = file.read()

text=file.read()

count\_tab=0

count\_space=0

count\_newline=0

count\_chars=0;

for char in data:

if char=='\t':

count\_tab+=1

elif char==' ':

count\_space+=1

elif char=='\n':

count\_newline+=1

else:

count\_chars+=1

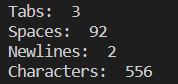
print("Tabs: ", count\_tab)

print("Spaces: ", count\_space)

print("Newlines: ", count\_newline)

print("Characters: ", count\_chars)

**OUTPUT:**



**CONCLUSION:**

Hence, the above program is used to count the number of lines, spaces and tabs in given input.

**AIM:** To write a program to implement Scanner Generator using lex.

**DESCRIPTION:**

We can use flex a to automatically generate the lexical analyzer/scanner for the lexical atoms of a language. But here we are trying to make a Scanner generator without using lex.

**ALGORITHM:**

1. In the definition section include the libraries stdioh, string.h and ctype.h.
2. Create a file in read mode.
3. Check each and every character and categorize whether the token is keyword, identifier, operator, comment, special symbols and generate the token.
4. Print the lineno, tokenno, token, lexeme in another file in write mode.

**CODE:**

/\*STANDALONE SCANNER PROGRAM\*/

#include<stdio.h>

#include<ctype.h>

#include<string.h>

int main()

{

FILE \*input, \*output;

int l=1;

int t=0;

int j=0;

int i,flag;

char ch,str[20];

input = fopen("input.txt","r");

output = fopen("output.txt","w");

char keyword[30][30] = {"int","main","if","else","do","while"};

fprintf(output,"Line no. \t Token no. \t Token \t Lexeme\n\n");

while(!feof(input))

{

i=0;

flag=0;

ch=fgetc(input);

if( ch=='+' || ch== '-' || ch=='\*' || ch=='/' )

{

fprintf(output,"%7d\t\t %7d\t\t Operator\t %7c\n",l,t,ch);

t++;

}

else if( ch==';' || ch=='{' || ch=='}' || ch=='(' || ch==')' || ch=='?' || ch=='@' || ch=='!' || ch=='%')

{

fprintf(output,"%7d\t\t %7d\t\t Special symbol\t %7c\n",l,t,ch);

t++;

}

else if(isdigit(ch))

{

fprintf(output,"%7d\t\t %7d\t\t Digit\t\t %7c\n",l,t,ch);

t++;

}

else if(isalpha(ch))

{

str[i]=ch;

i++;

ch=fgetc(input);

while(isalnum(ch) && ch!=' ')

{

str[i]=ch;

i++;

ch=fgetc(input);

}

str[i]='\0';

for(j=0;j<=30;j++)

{

if(strcmp(str,keyword[j])==0)

{

flag=1;

break;

}

}

if(flag==1)

{

fprintf(output,"%7d\t\t %7d\t\t Keyword\t %7s\n",l,t,str);

t++;

}

else

{

fprintf(output,"%7d\t\t %7d\t\t Identifier\t %7s\n",l,t,str);

t++;

}

}

else if(ch=='\n')

{

l++;

}

}

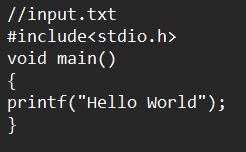
fclose(input);

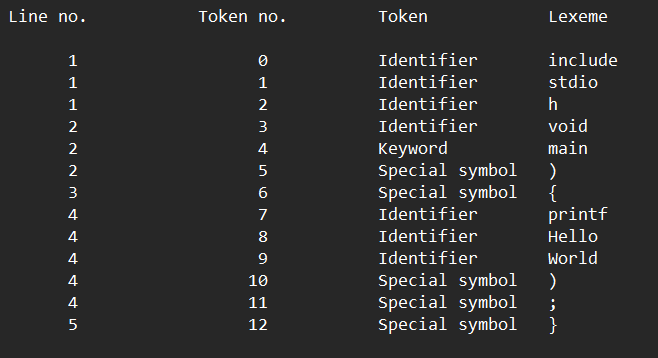
fclose(output);

return 0;

}

**OUTPUT:**





**CONCLUSION:**

Hence, the above program is used to implement scanner generator using lex.

**WEEK-2**

**AIM:** Write a program to count number of vowels and consonants for a given input.

**DESCRIPTION:**

Lex is a computer program that generates lexical analyzers and was written by Mike Lesk and Eric Schmidt. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language

In the below program we will be implementing a lex program on how to count the number of vowels, consonants for a given input using lex.

**ALGORITHM:**

* In the definition section include the libraries string.h and math.h and predefine check function.
* In the rules section match the given input to vowels or consonants and run the main function with yytext as parameter where yytext is the input buffer.
* In the user code section create main function, read input and run the yylex() function.
* Create few new variable such vow\_count and constance\_count each used to store the count of vowels and consonents in the given input.
* Using the regular expressions, find the number of occurrences of vowels and consonants in the given input.
* Print the count of each occurrence.

**CODE:**

%{

int vow\_count=0;

int const\_count =0;

%}

%%

[aeiouAEIOU] {vow\_count++;}

[a-zA-Z] {const\_count++;}

%%

int yywrap(){}

int main()

{

printf("Enter a string: ");

yylex();

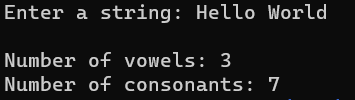
printf("Number of vowels: %d\n", vow\_count);

printf("Number of consonants: %d\n", const\_count);

return 0;

}

**OUTPUT:**



**CONCLUSION:**

Hence, the above program is used to count the number of vowels and consonants for a given input.

**AIM:** To write a program to count number of spaces, characters, words and newlines using flex.

**DESCRIPTION:**

Lex is a computer program that generates lexical analyzers and was written by Mike Lesk and Eric Schmidt. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language

In the below program we will be implementing a lex program on how to count the number of characters, word, spaces, and end of line in the given input.

**ALGORITHM:**

* In the definition section include the libraries string.h and math.h and predefine check function.
* In the rules section match the given input to characters, word, spaces, and end of line and run the main function with yytext as parameter where yytext is the input buffer.
* In the user code section create main function, read input and run the yylex() function.
* Create few new variable nw, nl, ns and cc each used to store the count of characters, word, lines and spaces in the given input.
* Using the regular expressions, find the number of occurrences of characters, word, lines and spaces in the given input.
* Print the count of each occurrence.

%{

#include<stdio.h>

int lc=0,sc=0,tc=0,ch=0,wc=0;

%}

%%

[\n] { lc++; ch+=yyleng;}

[ ] { sc++; ch+=yyleng;}

[ \t] { tc++; ch+=yyleng;}

[^\t\n ]+ { wc++; ch+=yyleng;}

%%

int yywrap(){

return 1;

}

int main(){

printf("Input:\n");

yylex();

printf("No. of lines: %d\n",lc);

printf("No. of spaces: %d\n",sc);

printf("No. of tabs: %d\n", tc);

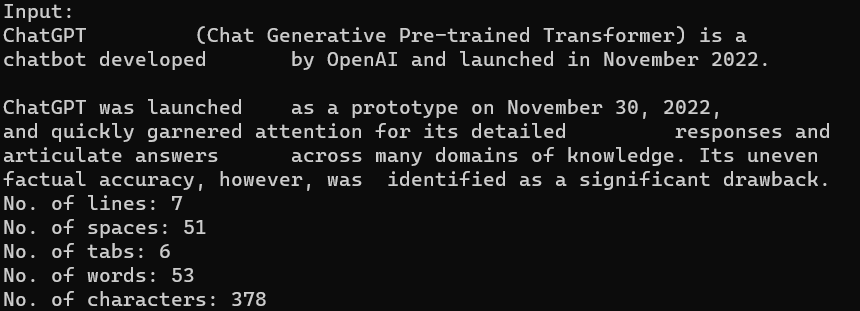
printf("No. of words: %d\n", wc);

printf("No. of characters: %d\n", ch);

return 0;

}

**OUTPUT:**



**CONCLUSION:**

Therefore, the above code gives us the count of number of characters, word, spaces and end of line present in the given input.

**AIM:** To write a program to identify and count number if positive and negative integers using flex.

**DESCRIPTION:**

Lex is a computer program that generates lexical analyzers and was written by Mike Lesk and Eric Schmidt. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

In the below program we will be implementing a lex program on how to count the number of positive and negative integers present in the given input.

**ALGORITHM:**

* In the definition section include the libraries string.h and math.h and predefine check function.
* In the rules section match the given input to positive or negative numbers and run the main function with yytext as parameter where yytext is the input buffer.
* In the user code section create main function, read input and run the yylex() function.
* Create few new variables such neg and pos each used to store the count of positive or negative numbers in the given input.
* Using the regular expressions, find the number of occurrences of positive or negative numbers in the given input.
* Print the count of each occurrence.

**CODE:**

%{

int posno=0;

int negno=0;

int posfrac=0;

int negfrac=0;

%}

DIGIT [0-9]

%%

\+?{DIGIT}+ posno++; printf("%s is positive integer\n",yytext);

-{DIGIT}+ negno++; printf("%s is negative integer\n",yytext);

\+?{DIGIT}\*\.{DIGIT}+ posfrac++; printf("%s is positive fraction\n",yytext);

-{DIGIT}\*\.{DIGIT}+ negfrac++; printf("%s is negative fraction\n",yytext);

. ;

%%

int main()

{

yylex();

printf("\nNo. of Positive integer: %d", posno);

printf("\nNo. of Negative integer: %d", negno);

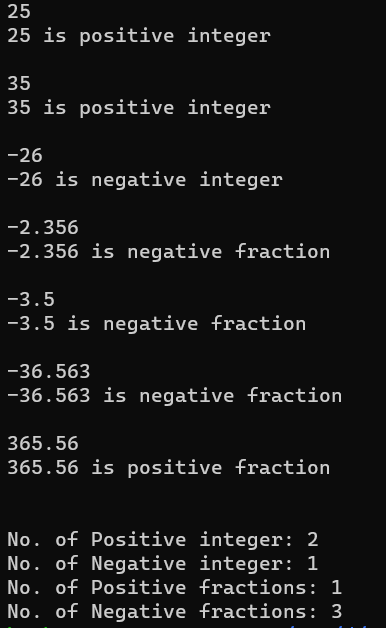
printf("\nNo. of Positive fractions: %d", posfrac);

printf("\nNo. of Negative fractions: %d\n", negfrac);

return 0;

}

**OUTPUT:**



**CONCLUSION:**

Therefore, the above code gives us the count of number of positive and negative integers present in the given input.

**AIM:** To write a program to check whether the given number is Armstrong number or not using flex.

**DESCRIPTION:**

Every lex program consists of 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

%%

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.

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

Basic Program Structure:

%{

// Definitions

%}

%%

Rules

%%

User code section

In this program we check whether the given number is armstrong or not where an armstrong number is equal to sum of power of length for each of its digits.

**ALGORITHM:**

* In the definition section include the libraries string.h and math.h and predefine check function.
* In the rules section match the given input to digits and run the check function with yytext as parameter where yytext is the input buffer.
* In the user code section create main function ,read input and run the yylex() function
* Then create the check function with parameter as an array of characters .
* Convert the given string into number using a for loop and iterating through string and store it in the variable num.
* Create a new variable y and while num is greater than 0 increment y by pow(num%10,len) where len is length of the input string and divide num by 10.
* After the while loop is terminated the given number is armstrong or else its not Armstrong.

**CODE:**

%{

#include <stdio.h>

#include <string.h>

#include <math.h>

void check(char \*);

%}

%%

[0-9]+ check(yytext);

%%

int yywrap(){}

int main(){

printf("Enter number: ");

yylex();

return 0;

}

void check(char\* a)

{

int len = strlen(a), i, num = 0;

for (i = 0; i < len; i++)

num = num \* 10 + (a[i] - '0');

int x = 0, y = 0, temp = num;

while (num > 0) {

y = num % 10;

x = x + y\*y\*y;

num = num / 10;

}

if (x == temp)

printf("%d is an Armstrong number\n", temp);

else

printf("%d is Not an Armstrong number\n", temp);

}

**OUTPUT:**



****

**CONCLUSION:**

Hence we have found whether a number is Armstrong or not using the lex program in which we wrote a separate check function which runs when matched with the above regular expression and gives us the correct output.

**AIM:** Write a program to identify octal or hexadecimal number using lex.

**DESCRIPTION:**

Every lex program consists of 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

%%

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.

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

Basic Program Structure:

%{

// Definitions

%}

%%

Rules

%%

User code section

In this program we check whether the given input is octal or hexadecimal.

**ALGORITHM:**

* In the definitions section we need not include anything since we dont perform any string operations or arithmetic operations for the given program.
* In the rules section write the regular expressions which match hexadecimal or octal number.
* The regular expression for hexadecimal is **[o][x|X][0-9A-F]+**
* The regular expression for octal is **[o][0-7]+**
* If the given input is hexadecimal print (“this is a hexadecimal number”).
* If the given input is octal print (“this is a octal number”).
* And in the usercode section we just write the main function and call yylex() function inside it we also call the yywrap() function to wrap up the rules section.

**CODE:**

%{

/\*program to identify octal and hexadecimal numbers\*/

%}

Oct [o][0-7]+

Hex [o][x|X][0-9A-F]+

%%

{Hex} printf("this is a hexadecimal number");

{Oct} printf("this is an octal number");

%%

main()

{

yylex();

}

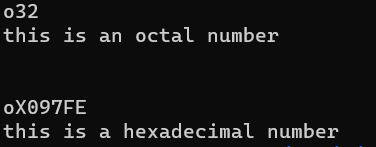
int yywrap()

{

return 1;

}

**OUTPUT:**



**CONCLUSION:**

Hence we have found whether the given number is hexadecimal or octal using the corresponding regular expressions in the rules section and by running the yylex() function in the main program of user code section.

**AIM:** Towrite a program to capitalize the given comments using lex.

**DESCRIPTION:**

Every lex program consists of 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

%%

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.

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

Basic Program Structure:

%{

// Definitions

%}

%%

Rules

%%

User code section

In this program we capitalize the given input using toupper function

**ALGORITHM:**

* In the definitions section include <stdio.h> and <ctype.h> libraries.
* In the rules section if the given input matches the regular expression [a-zA-Z]+ i.e, if the given input only contains alphabets we perform the following steps.
* For each and every character in yytext we perform yytext[i]=toupper(yytext[i]) and print it.
* Else if the character is not an alphabet we print it directly without any operation.
* In the main function we only call the yylex() function which helps us to run rules section.

**CODE:**

%{

#include<stdio.h>

#include<ctype.h>

int k;

void display(char \*);

%}

letter [a-z]

com [//]

%%

{com} {k=1;}

{letter} {if(k==1) display(yytext);}

%%

main()

{

yylex();

}

void display(char \*s)

{

int i;

for(i=0;s[i]!='\0';i++)

printf("%c", toupper(s[i]));

}

int yywrap()

{

return 1;

}

**OUTPUT:**

****

**CONCLUSION:**

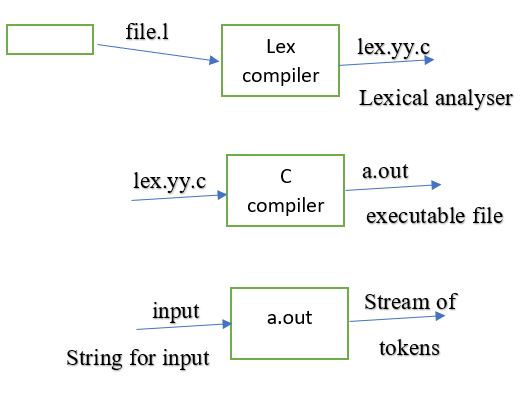
Hence we have capitalized the given input by writing a special code for it in the rules section where we use regular expression to match the input to alphabets and convert them to uppercase using the toupper() function.

**AIM:** To write a program scanner generator using lex.

**DESCRIPTION:**

Lex is a lexical analysis that can be used to identify specific text strings.

Scanner generator is also called as lexical analyzer.



**ALGORITHM:**

* In the definition section include the libraries stdio.h.
* In the rules section if the given input matches the regular expression then we print the whether the token is a identifier, keyword, operator etc.
* In the main function we only call the yylex() function which helps us to run rules section.

**CODE:**

%{

#include <stdio.h>

%}

DIGIT [0-9]

ALPHA [a-zA-Z]

IDENT {ALPHA}({ALPHA}|{DIGIT})\*

%%

#.\* {printf("PREPROCESSOR DIRECTIVE: %s \n",yytext);}

int|double|char {printf("Keyword: %s\n",yytext);}

"+"|"-"|"\*"|"/" {printf("Arithmetic Operator: %s\n",yytext);}

"="|"+="|"-="|"\*="|"/=" {printf("Assignment Operator: %s\n",yytext);}

"=="|">="|"<="|"!="|">"|"<" {printf("Assignment Operator: %s\n",yytext);}

"(" {printf("Open Parenthesis: %s\n",yytext);}

")" {printf("Closed Parenthesis: %s\n",yytext);}

"{" {printf("Block Begins: %s\n",yytext);}

"}" {printf("Block Ends: %s\n",yytext);}

";" {printf("Semicolon: %s\n",yytext);}

{IDENT} { printf("IDENTIFIER: %s\n", yytext); }

{DIGIT}+ { printf("INTEGER: %s\n", yytext); }

. { printf("INVALID: %s\n", yytext); }

%%

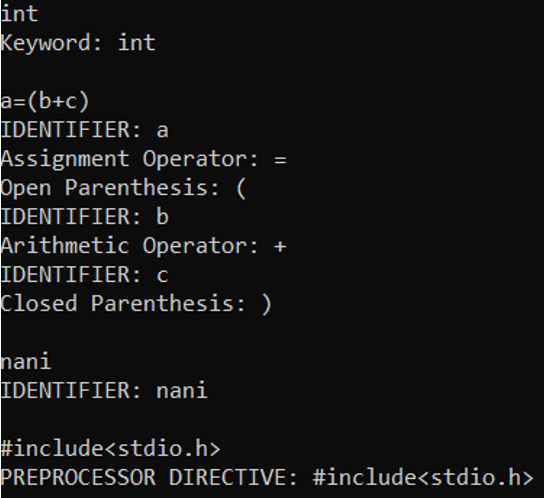
int main(void) {

yylex();

return 0;

}

**OUTPUT:**

****

**CONCLUSION:**

From the above program, we can generate the stream of tokens using lexical analyzer.

**AIM:** To write a program to find complete real precision using lex.

**DESCRIPTION:**

Every lex program consists of 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

%%

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.

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

Basic Program Structure:

%{

// Definitions

%}

%%

Rules

%%

User code section

In this program we find the real precision value of the given decimal i.e, the number of real digits after the decimal point

**ALGORITHM:**

* In the definition section we include <stdio.h> library.
* In the rules section we match the given input to the regular expression [0-9]+\.[0-9]+ and perform below steps.
* Declare a pointer ptr pointing to starting index of yytext (input).
* while ptr!=’.’ increment ptr.
* then while ptr==0 increment ptr.
* declare a new variable num\_digits=0.
* while (\*ptr) increment ptr and also num\_digits.
* print(precision , num\_digits).
* In the main function we only call yylex() function which helps us to run rules section.

**CODE:**

%{

#include <stdio.h>

%}

%%

[0-9]+\.[0-9]+ {

char \*ptr = yytext;

int num\_digits = 0;

while (\*ptr != '.') {

ptr++;

}

ptr++;

while (\*ptr == '0') {

ptr++;

}

while (\*ptr) {

num\_digits++;

ptr++;

}

//printf("Number: %s\n", yytext);

printf("Precision: %d decimal places\n", num\_digits );

}

. {

}

%%

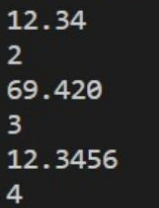
int main() {

yylex();

return 0;

}

**OUTPUT:**

****

**CONCLUSION:**

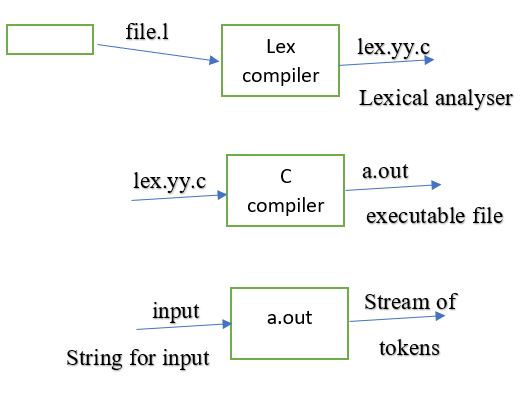
Hence, we have found the real precision value of the given decimal number by matching the given input to above regular expression and writing a special code block for it where we use pointer for finding the first nonzero element after decimal point and count number of digits after it which is the real precision value.

**WEEK-3**

**AIM:** To write a program to find regular expression of digit, identifier and keywords starting with letter “a”.

**DESCRIPTION:**

Lex is a lexical analysis that can be used to identify specific text strings.



**ALGORITHM:**

* In the definition section we include <stdio.h> library.
* In the rules section if the given input matches the regular expression then prints the string starts with the letter ‘a’ if flag==1 or else we print the string is not accepted.
* In the main function we only call the yylex() function which helps us to run rules section.

**CODE:**

%{

int flag = 0;

%}

%%

[aeiouAEIOU].[a-zA-Z0-9.]+ flag=1;

[a-zA-Z0-9]+

%%

int main()

{

yylex();

if(flag==1){

printf("Accepted");

}

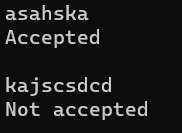
else{

printf("Not Accepted");

}

}

**OUTPUT:**

****

**CONCLUSION:**

From the above program, we can find a identifier, keyword, digit starting with letter ‘a’.

**WEEK-4**

**AIM:** To write a program to compute FIRST, FOLLOW functions and Predictive Parsing Table for a given grammar.

**DESCRIPTION:**

FIRST(X) for a grammar symbol X is the set of terminals that begin the strings derivable from X. Rules to compute FIRST set: If x is a terminal, then FIRST(x) = {'x' } If x-> Є, is a production rule, then add Є to FIRST(x).

Follow(X) to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form. In RHS of A -> aBb, b follows Non-Terminal B, i.e., FOLLOW(B) = {b}, and the current input character read is also b. Hence the parser applies this rule. And it is able to get the string “ab” from the given grammar.

Here the 1st L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

**ALGORITHM:**

1) FIRST Function

FIRST (α) is defined as the collection of terminal symbols which are the first letters of strings derived from α.

FIRST (α) = {α |α →∗ αβ for some string β }

If X is Grammar Symbol, then First (X) will be −

If X is a terminal symbol, then FIRST(X) = {X}

If X → ε, then FIRST(X) = {ε}

If X is non-terminal & X → a α, then FIRST (X) = {a}

2) FOLLOW function

If S is the start symbol, FOLLOW (S) ={$}

If production is of form A → α B β, β ≠ ε.

1. If FIRST (β) does not contain ε then, FOLLOW (B) = {FIRST (β)}

OR

(b) If FIRST (β) contains ε (i. e. , β ⇒\* ε), then

FOLLOW (B) = FIRST (β) − {ε} ∪ FOLLOW (A)

∵ when β derives ε, then terminal after A will follow B.

If production is of form A → αB, then Follow (B) ={FOLLOW (A)}.

3) Parsing Table

Step 1: First check all the essential conditions mentioned above and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

FIRST(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.

FOLLOW(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha). Find First(α) and for each terminal in First(α), make entry A –> α in the table.

If First(α) contains ε (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.

If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $.

**CODE:**

from collections import defaultdict

d = defaultdict(list)

f = defaultdict(list)

fo = defaultdict(list)

p = defaultdict(dict)

d = {'E':['TG'],'G':['+TG','e'],'T':['FW'],'W':['\*FW','e'],'F':['(E)','c']}

# First Functions

for i,j in d.items():

z = i

k = 0

while k<len(d[z]):

if 'a'<=d[z][k][0]<='z' or d[z][k][0] in '(+)\*' or d[z][k][0]=='e':

f[i].append(d[z][k][0])

k+=1

else:

z=d[z][k][0]

k = 0

# Follow Functions

for z in d.keys():

for i,j in d.items():

for k in d[i]:

if z in k:

if z=='E':

fo['E'].extend('$')

n = k.index(z)+1

if n<len(k):

if k[n] in '()+\*' or ('a'<=k[n]<='z' and k[n]!='e'):

fo[z].extend(k[n])

if k[n]=='e':

fo[z].extend(fo[i])

elif 'A'<=k[n]<='Z':

fo[z].extend([i for i in f[k[n]] if i!='e'])

fo[z].extend(fo[k[n]])

else:

if z!=i:

fo[z].extend(fo[i])

print("First Functions of all Non-Terminals")

for i,j in f.items():

print(i,'==>',set(j))

print("\nFollow Functions of all Non-Terminals")

for i,j in fo.items():

print(i,'==>',set(j))

# Parsing Table

# Parsing Table

print("\nParsing Table")

for i in d.keys():

k = f[i]

if 'c' in k:

p[i].update({'c':d[i]})

else:

p[i].update({'c':"Error"})

if '+' in k:

p[i].update({'+':d[i]})

else:

p[i].update({'+':"Error"})

if '\*' in k:

p[i].update({'\*':d[i]})

else:

p[i].update({'\*':"Error"})

if '(' in k:

p[i].update({'(':d[i]})

else:

p[i].update({'(':"Error"})

if ')' in k:

p[i].update({')':d[i]})

else:

p[i].update({')':"Error"})

if '$' in k:

for m in fo[k]:

p[i].update({'m':d[i]})

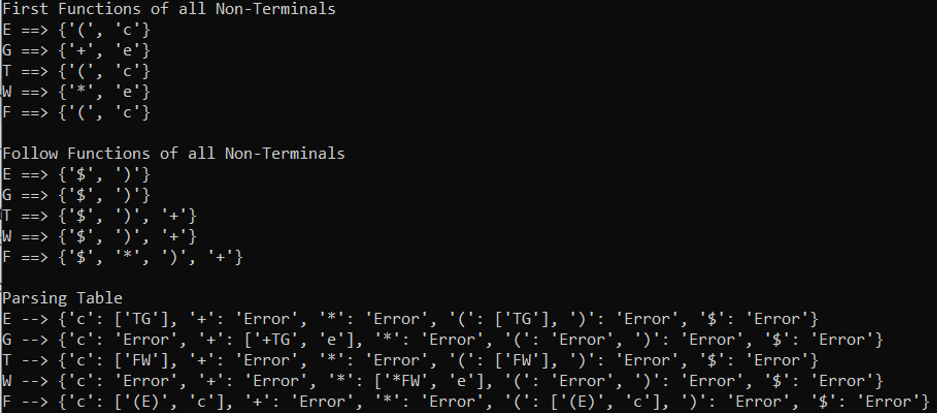
else:

p[i].update({'$':"Error"})

for i,j in p.items():

print(i,'-->',j)

**OUTPUT:**

****

**CONCLUSION:**

Hence, we have computed first, follow and parsing table using the given algorithms.

**WEEK-5**

**AIM:** To write a program for Recursive Decent Parser

**DESCRIPTION:**

A parser that uses collection of recursive procedures for parsing the given input string is called Recursive descent (RD) parser. It is a kind of Top-Down Parser. A top-down parser builds the parse tree from the top to down, starting with the start non-terminal. A Predictive Parser is a special case of Recursive Descent Parser, where no Back Tracking is required. By carefully writing a grammar means eliminating left recursion and left factoring from it, the resulting grammar will be a grammar that can be parsed by a recursive descent parser.

**CODE:**

print("E->TE'\nE'->+TE'/@\nT->FT'\nT'->\*FT'/@\nF->(E)/i\n")

global s

s=list(input("Enter the string: "))

global i

i=0

def match(a):

global s

global i

if(i>=len(s)):

return False

elif(s[i]==a):

i+=1

return True

else:

return False

def F():

if(match("(")):

if(E()):

if(match(")")):

return True

else:

return False

else:

return False

elif(match("i")):

return True

else:

return False

def Tx():

if(match("\*")):

if(F()):

if(Tx()):

return True

else:

return False

else:

return False

else:

return True

def T():

if(F()):

if(Tx()):

return True

else:

return False

else:

return False

def Ex():

if(match("+")):

if(T()):

if(Ex()):

return True

else:

return False

else:

return False

else:

return True

def E():

if(T()):

if(Ex()):

return True

else:

return False

else:

return False

if(E()):

if(i==len(s)):

print("String is accepted")

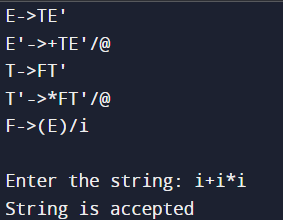
else:

print("String is not accepted")

else:

print("string is not accepted")

**OUTPUT:**

****

**CONCLUSION:**

Therefore, Recursive Descent Parser is thus implemented.

**AIM:** To write a program to check whether an input string for the given LL(1) parser is accepted or not.

**DESCRIPTION:**

LL(1) parsing is a top-down parsing method in the syntax analysis phase of compiler design. Required components for LL(1) parsing are input string, a stack, parsing table for given grammar, and parser. Here, we discuss a parser that determines that given string can be generated from a given grammar(or parsing table) or not.

Let the given grammar is G = (V, T, S, P) where V-variable symbol set, T-terminal symbol set, S- start symbol, P- production set.

**ALGORITHM:**

LL(1) Parser algorithm:

Input- 1. stack = S //stack initially contains only S.

2. input string = w$ where S is the start symbol of grammar, w is given string, and $ is used for the end of string.

3. PT is a parsing table of given grammar in the form of a matrix or 2D array.

Output- determines that given string can be produced by given grammar(parsing table) or not, if not then it produces an error.

**CODE:**

# Define the LL(1) grammar rules and parse table

grammar\_rules = {

'S': [('a', 'A'), ('b', 'B')],

'A': [('c',), ('d',)],

'B': [('c',), ('e',)],

}

parse\_table = {

('S', 'a'): ('a', 'A'),

('S', 'b'): ('b', 'B'),

('A', 'c'): ('c',),

('A', 'd'): ('d',),

('B', 'c'): ('c',),

('B', 'e'): ('e',),

}

def ll1\_parse(input\_string):

# Initialize the stack with the start symbol

stack = ['S']

input\_idx = 0

while stack:

top = stack.pop()

if top in grammar\_rules:

# Lookup the production in the parse table

try:

prod = parse\_table[top, input\_string[input\_idx]]

except KeyError:

return False

# Push the RHS of the production onto the stack in reverse order

stack.extend(reversed(prod))

elif top == input\_string[input\_idx]:

# Match the terminal symbol

input\_idx += 1

else:

# Mismatched terminal symbol, report an error

return False

# If we've consumed the entire input string, the parse was successful

return input\_idx == len(input\_string)

# Example usage

input\_str = 'acb'

if ll1\_parse(input\_str):

print(f'Input string "{input\_str}" was accepted.')

else:

print(f'Input string "{input\_str}" was not accepted.')

**OUTPUT:**

****

**CONCLUSION:**

Hence, LL(1) parser is thus implemented and input strings are verified.

**WEEK-5**

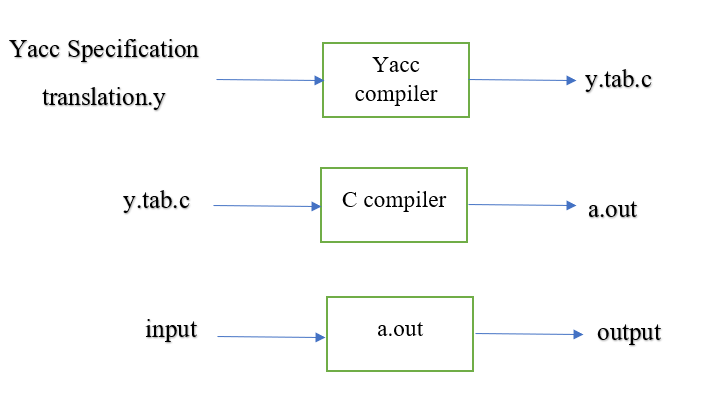
**AIM:** Towrite a program to validate an arithmetic expression using yacc tool.

**DESCRIPTION:**

Yacc is known as Yet another Compiler-compiler.It is used to produce the source code of syntactic analyser of the language produce by LALR(1) grammar.The input of YACC is the rule or grammar and the output is a c program.

YACC translate.y

This command converts the file translate.y into a C file y.tab.c. It represents an LALR parser in C with some other user’s prepared C routines.By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.

****

Lex is a lexical analysis that can be used to identify specific text strings.Yacc is a grammar parser,it reads text and can be used to turn a sequence of words into a structured format for processing.

**ALGORITHM:**

* In the definition section we include <stdio.h>, y.tab.h library.
* In the rules section if the given input matches the regular expression gives the required output for the given input.
* Define the grammar for the arithmetic expressions using the Yacc syntax.
* This grammar defines an arithmetic expression as a sequence of numbers and arithmetic operators (+, -, \*, /, and ^) with the appropriate precedence and associativity.
* Implement the actions associated with each grammar rule. In this case, we only need to check that the input string conforms to the grammar. Therefore, we can simply return the entered arithmetic expression is valid, and entered arithmetic expression is invalid otherwise.
* We define a main function that calls the yyparse function to parse the input string. The yyparse function returns 0 if the input string is valid, and a nonzero value otherwise. We then print an appropriate message based on the return value of yyparse.

**CODE:**

**arith.l**

%{

#include<stdio.h>

#include "y.tab.h"

%}

%%

[a-zA-Z]+ return VARIABLE;

[0-9]+ return NUMBER;

[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()

{

return 1;

}

**arith.y**

%{

#include<stdio.h>

%}

%token NUMBER

%token VARIABLE

%left '+' '-'

%left '\*' '/' '%'

%left '(' ')'

%%

S: VARIABLE'='E {

printf("\nEntered arithmetic expression is Valid\n\n");

return 0;

}

E:E'+'E

|E'-'E

|E'\*'E

|E'/'E

|E'%'E

|'('E')'

| NUMBER

| VARIABLE

;

%%

void main()

{

printf("\nEnter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");

yyparse();

}

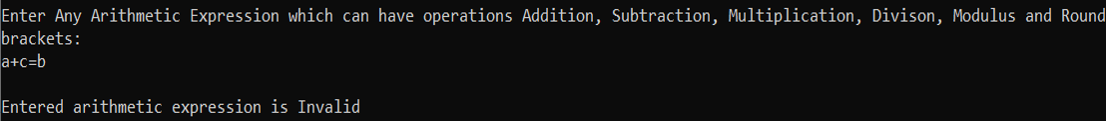
void yyerror()

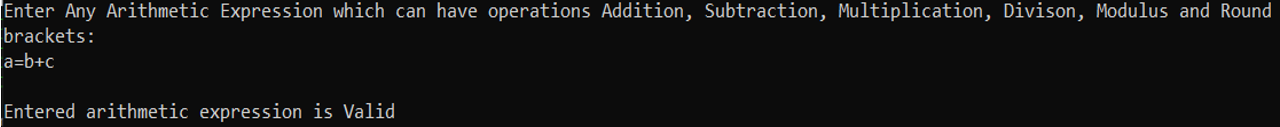
{

printf("\nEntered arithmetic expression is Invalid\n\n");

}

**OUTPUT:**

****

****

**CONCLUSION:**

From the above program, I understood how to validate whether a given arithmetic expression is valid or not using yacc tool.

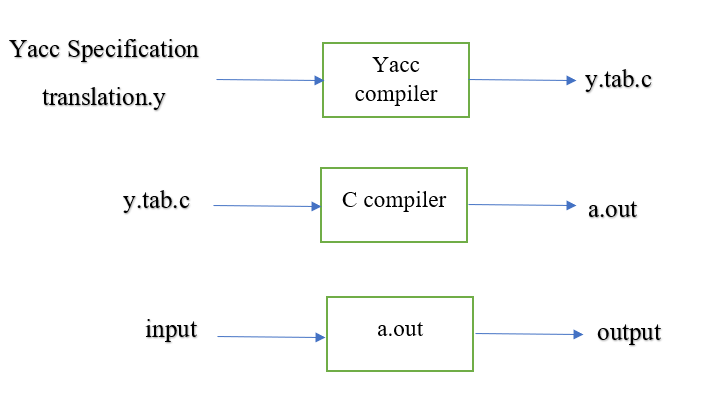
**AIM:** To write a program to check whether a given token is a number or an alphabet.

**DESCRIPTION:**

Yacc is known as Yet Another Compiler-Compiler. It is used to produce the source code of syntactic analyser of the language produce by LALR(1) grammar. The input of YACC is the rule or grammar and the output is a C program.

**YACC translate.y**

This command converts the file translate.y into a C file y.tab.c. It represents an LALR parser in C with some other user’s prepared C routines. By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.

****

Lex is a lexical analysis that can be used to identify specific text strings. Yacc is a grammar parser, it reads text and can be used to turn a sequence of words into a structured format for processing.

**ALGORITHM:**

* In the definition section we include <stdio.h>, y.tab.h library.
* In the rules section if the given input matches the regular expression gives the required output for the given input.
* In the rules section match the given input to digits with yytext as parameter where yytext is the input buffer.
* We define a main function that calls the yyparse function to parse the input string.
* In the rules section if the given input matches with the number, then given an input is a number, if given input matches with alphabet print input is an alphabet.

**CODE:**

**numalpha.l**

%{

#include "y.tab.h"

%}

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

[a-zA-Z]+ { yylval = \*yytext; return ALPHABET; }

[\n] return 0;

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

**numalpha.y**

%{

#include <stdio.h>

%}

%token NUMBER ALPHABET

%%

input: NUMBER { printf("Input is a number.\n"); }

| ALPHABET { printf("Input is an alphabet.\n"); }

;

%%

int main(void) {

yyparse();

return 0;

}

int yyerror(const char \*msg) {

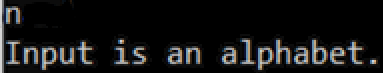
fprintf(stderr, "%s\n", msg);

return 0;

}

**OUTPUT:**

****

****

**CONCLUSION:**

From the above program, I have how to check whether a given token is a number or an alphabet using yacc tool.