

Sr. No	EXPERIMENT NAME
1	Program to implement Lexical Analyzer.
2	Program to eliminate Left Recursion from given grammar.
3	Program to implement Parser.
4	Program to study and implement LEX and YACC tools.(To check whether it is identifier or not)
5	Program to study and implement LEX and YACC tools.(To check whether the entered no is integer or floating point number)
6	W.A. P. to find no. of word count in sentence.
7	W.A.P. to implement Single pass Assembler
8	W. A. P. to find character count in a word.

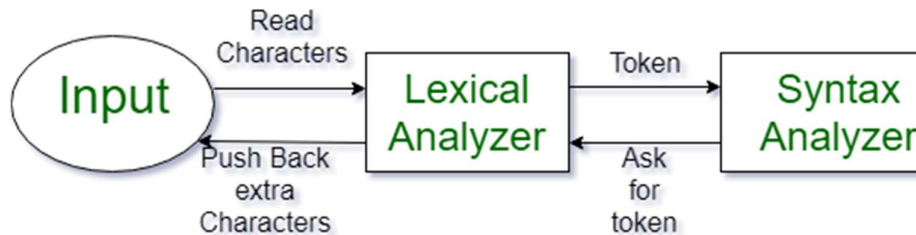
Lexical Analysis

Lexical analysis, also known as scanning or tokenization, is the first phase of a compiler that processes the source code written in a programming language. Its primary goal is to break down the input code into a sequence of tokens. A token is a meaningful and atomic unit of the source code that represents a specific element of the programming language, such as keywords, identifiers, operators, and literals.

Lexical Analyzer

A lexical analyzer, also known as a lexer or tokenizer, is a component of a compiler that performs lexical analysis on the source code written in a programming language. Its primary task is to read the input source code and break it down into a sequence of tokens, which are the smallest units of meaning in the language.

The lexical analyzer plays a crucial role in the compilation process by transforming the raw source code into a stream of tokens that can be easily processed by subsequent phases of the compiler.



What is a token?

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

These tokens are identified by the lexical analyzer during the lexical analysis phase of the compilation process. Types of lexical tokens found in many programming languages:

1. Keywords:

- Keywords are reserved words that have a special meaning in the programming language. Examples include **if**, **else**, **while**, **for**, **int**, **float**, **return**, and others.

2. Identifiers:

- Identifiers are names given to variables, functions, or other user-defined entities in the program. They typically consist of letters, digits, and underscores, with certain rules regarding their formation (e.g., cannot start with a digit).

3. Literals:

- Literals represent constant values used in the program. There are different types of literals:
 - Integer literals: Examples include **42**, **-15**, and **0**.
 - Floating-point literals: Examples include **3.14**, **-0.001**, and **2.0e10**.
 - Character literals: Examples include **'A'**, **'1'**, and **'\n'**.
 - String literals: Examples include **"Hello, World!"** and **"abc"**.

4. Operators:

- Operators perform operations on operands. Examples include **+**, **-**, *****, **/** (arithmetic operators), **==**, **!=** (relational operators), **&&**, **||** (logical operators), and **=** (assignment operator).

5. Symbols:

- Symbols include punctuation marks and other special characters used in the program. Examples include **,** (comma), **;** (semicolon), **(** (left parenthesis), **)** (right parenthesis), **{** (left curly brace), **}** (right curly brace), **[** (left square bracket), **]** (right square bracket), and others.

6. Comments:

- Comments are not typically considered tokens, but they are important for code documentation. In most programming languages, comments start with a specific symbol (e.g., **//** for single-line comments or **/*** and ***/** for multi-line comments) and are ignored during lexical analysis.

LEXICAL ANALYZER→

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

int isKeyword (char *str)
{
    int i;
    char keywords[][10] =
{"void","main","if","else","while","for","int","float","char","break","return"};
    int numKeywords = sizeof (keywords) / sizeof (keywords[0]);
    for (i=0;i<numKeywords;i++)
    {
        if (strcmp(str, keywords[i]) == 0)
        {
            return 1;
        }
    }
    return 0;
}
```

```
void lexicalAnalyzer (char *input)
{
    char *token = strtok(input, " ");
    while (token != NULL)
    {
        if (isKeyword(token))
        {
            printf("Lexeme: %s, Token: Keyword\n", token);
        }
        else if (isalpha(token[0]))
        {
            printf("Lexeme: %s, Token: Identifier\n", token);
        }
        else if (isdigit(token[0]))
        {
            printf("Lexeme: %s, Token: Constant\n", token);
        }
        else
        {
            printf("Lexeme: %s, Token: Operator\n", token);
        }
        token = strtok(NULL, " ");
    }
}
```

```
int main()
{
    char input[250];
    printf("Enter the String:\n");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = '\0';
    lexicalAnalyzer(input);
    return 0;
}
```

Output

```
/tmp/M24bxkesGa.o
Enter the String:
void main () { if ( a > b ) }
Lexeme: void, Token: Keyword
Lexeme: main, Token: Keyword
Lexeme: (), Token: Operator
Lexeme: {, Token: Operator
Lexeme: if, Token: Keyword
Lexeme: (, Token: Operator
Lexeme: a, Token: Identifier
Lexeme: >, Token: Operator
Lexeme: b, Token: Identifier
Lexeme: ), Token: Operator
Lexeme: }, Token: Operator
```

What is Grammar?

Grammar is a set of rules that define the structure of valid strings in a language. It is a formal system used to describe the syntax of a language. $G = (V, T, P, S)$ Where,

- **G:** This typically represents the grammar itself.
- **V:** Represents a set of variables or non-terminal symbols in the grammar. Non-terminals are placeholders for patterns in the language.
- **T:** Represents a set of terminals or terminal symbols. Terminals are the actual symbols that appear in valid strings. Terminals are the basic building blocks of the language.
- **P:** Represents a set of production rules. Each production rule defines how a non-terminal can be replaced by a sequence of terminals and/or non-terminals.
- **S:** Represents the start symbol from which the derivation of valid strings begins.

What is LEFT RECURSION?

A grammar is said to be in **left recursive** if it has the production rules of the form $A \rightarrow A\alpha \mid \beta$.

In the production rule above, the variable in the left side occurs at the first position on the right side production, due to which the left recursion occurs.

If we have a left recursion in our grammar, it leads to infinite recursion, due to which we cannot generate the given string.

In other words, a grammar production is said to have left recursion if the leftmost variable of its Right Hand Side is the same as the variable of its Left Hand Side. A grammar containing a production of having left recursion is called **Left Recursive Grammar**.

Eliminating Left Recursion

A left recursive production can be eliminated by rewriting the offending productions.

Consider a nonterminal **A** with two productions $A \rightarrow A\alpha \mid \beta$ where α and β are sequences of terminals and non-terminals that do not start with **A**.

For example, in $\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{term}$

nonterminal **A** = *expr*, string = *+term*, and string = *term*.

The nonterminal **A** and its productions are said to be left recursive because the production

$A \rightarrow A\alpha$ has **A** itself as the leftmost symbol on the right side.

The left-recursive pair of productions $A \rightarrow A\alpha \mid \beta$ can be replaced by the non-left-recursive productions:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

Why do we eliminate left recursion?

Left recursion often poses problems for parsers, either because it leads them into infinite recursion (loop) or because they expect rules in a normal form that forbids it.

Top-down parsing methods cannot handle left recursive grammars, so a transformation is needed to eliminate left recursion.

Therefore, grammar is often pre-processed to eliminate the left recursion.

Algorithm:

1. **Identify Left Recursive Productions:** For each non-terminal **A** in the grammar, check if there exists a production $A \rightarrow A\alpha$, where α is a sequence of terminals and/or non-terminals.
2. **Create New Non-terminals:** For each left-recursive non-terminal **A**, introduce a new non-terminal **A'** that is not present in the original grammar.
3. **Replace Left-Recursive Productions:** Replace each left-recursive production $A \rightarrow A\alpha$ with the following two productions:
 - $A \rightarrow \beta A'$
 - $A' \rightarrow \alpha A' \mid \epsilon$ (where ϵ represents an empty string)

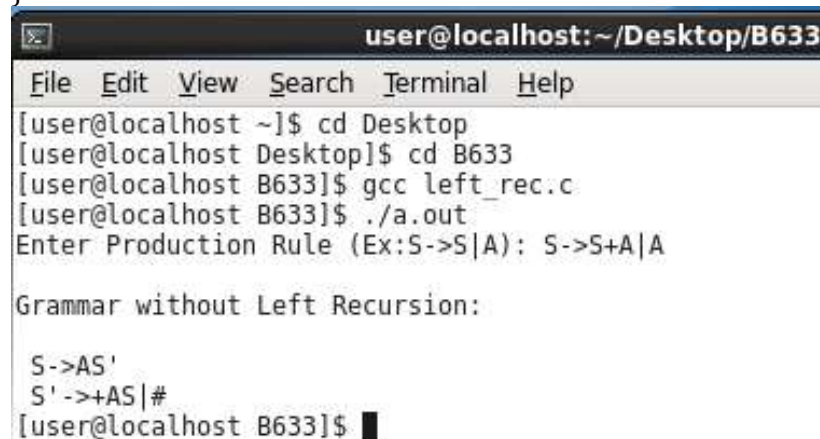
Here, β is the remaining part of the original production that doesn't lead to left recursion.

4. **Update Original Non-terminals:** Replace the original non-recursive production $A \rightarrow \beta$ with $A \rightarrow \beta A'$.

ELIMINATE LEFT RECURSION→

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define size 20

int main()
{
    char pro[size], alpha[size], beta[size];
    int nt, i, j, index = 3; //nt-non terminal
    printf("Enter Production Rule (Ex:S->S|A): ");
    scanf("%s",pro);
    nt = pro[0];
    if(nt == pro[index]) //Checking if Grammar is Left Recursive
    {
        //Getting Alpha
        for(i = ++index,j = 0;pro[i]!='|'; i++,j++){
            alpha[j] = pro[i];
            //Checking if there is NO Vertical Bar (|)
            if(pro[i+j] == 0){
                printf("This Grammar can't be reduced.\n");
                exit(0);
            }
        }
        alpha[j] = '\0'; //String ending NULL character
        //Checking if there is a character after Vertical-Bar(|)
        if(pro[++i] != 0){
            //Getting Beta
            for(j = i,i = 0;pro[j] != '\0';i++,j++){
                beta[i] = pro[j];
            }
            beta[i] = '\0'; //String ending NULL character
            // Showing Output without LEFT Recursion
            printf("\nGrammar without Left Recursion: \n\n");
            printf(" %c->%s%c\n", nt,beta,nt);
            printf(" %c'->%s%c|#\n", nt,alpha,nt);
        }
        else
            printf("This Grammar can't be reduced.\n");
    }
    else
        printf("\n This Grammar is not Left Recursive.");
}
```



The screenshot shows a terminal window with the title bar "user@localhost: ~/Desktop/B633". The menu bar includes "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal output shows the user navigating to the Desktop and then to the B633 directory, compiling the program "left_rec.c" with gcc, and running the resulting executable "a.out". The program prompts the user to enter a production rule, and the user enters "S->S+A|A". The program then outputs the grammar without left recursion, showing the original rule "S->AS'" and the modified rule "S' ->+AS|#".

```
user@localhost: ~/Desktop/B633
File Edit View Search Terminal Help
[user@localhost ~]$ cd Desktop
[user@localhost Desktop]$ cd B633
[user@localhost B633]$ gcc left_rec.c
[user@localhost B633]$ ./a.out
Enter Production Rule (Ex:S->S|A): S->S+A|A

Grammar without Left Recursion:

S->AS'
S' ->+AS|#
[user@localhost B633]$
```

SR (Shift-Reduce) Parser

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom-up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up).

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.

A String $\xrightarrow{\text{reduce to}}$ the starting symbol

- Shift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will be replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

A more general form of the shift-reduce parser is the LR parser.

This parser requires some data structures i.e.

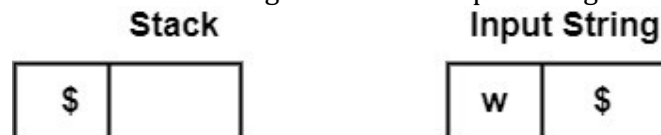
- An input buffer for storing the input string.
- A stack for storing and accessing the production rules.

Basic Operations →

- **Shift:** This involves moving symbols from the input buffer onto the stack.
- **Reduce:** If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of a production rule is popped out of a stack and LHS of a production rule is pushed onto the stack.
- **Accept:** If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accepted action is obtained, it means successful parsing is done.
- **Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Working→

Insert \$ at the bottom of the stack and the right end of the input string in Input Buffer.



- **Shift** – Parser shifts zero or more input symbols onto the stack until the handle is on top of the stack.
- **Reduce** – Parser reduce or replace the handle on top of the stack to the left side of production, i.e., R.H.S. of production is popped, and L.H.S is pushed.
- **Accept** – Step 3 and Step 4 will be repeated until it has detected an error or until the stack includes start symbol (S) and input Buffer is empty, i.e., it contains \$



Example →

Consider the following Grammar

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

Check whether **Input string "ccdd"** is accepted or not accepted using Shift-Reduce parsing.

Solution:

$S \Rightarrow^{rm} CC$
 $\Rightarrow^{rm} Cd$
 $\Rightarrow^{rm} cCd$
 $\Rightarrow^{rm} ccCd$
 $\Rightarrow^{rm} ccdd$

↑
Bottom-Up Parsing

Stack	Input String	Action
\$	ccdd\$	Shift
\$ c	cdd\$	Shift
\$ cc	dd\$	Shift
\$ ccd	d\$	Reduce by $C \rightarrow id$
\$ ccC	d\$	Reduce by $C \rightarrow cC$
\$cC	d\$	Reduce by $C \rightarrow cC$
\$C	d\$	Shift
\$Cd	\$	Reduce by $C \rightarrow d$
\$CC	\$	Reduce by $S \rightarrow CC$
\$S	\$	Accept

Algorithm:

1. Start
2. Initialize an empty stack and push the initial state (usually the start symbol) onto the stack.
3. Get input expression and store it in the input buffer
4. Read the data from the input buffer one at a time
5. Using stack and PUSH & POP operation SHIFT & REDUCE symbols with respect to production rules available.
6. Parsing Loop
 - a. **Shift Operation:**
If the current stack state and the current input symbol allow a shift operation, perform a shift. Push the current input symbol onto the stack and read the next token from the input stream.
 - b. **Reduce Operation:**
If the current stack state and the top of the stack allow a reduce operation, perform a reduce. Pop symbols from the stack based on a production rule and push the corresponding non-terminal onto the stack.
7. Repeat until the input is empty (i.e. Continue the process till Symbol shift and production rule reduce reaches the start symbol.)
8. Display the stack implementation table with corresponding Stack Actions with input symbol.
9. Stop

SR PARSER→

```
#include<stdio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main()
{
    puts("GRAMMAR is \n E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("Enter input string ");
    scanf("%s",a);
    c=strlen(a);
    strcpy(act,"SHIFT->");
    puts("stack \t\t input \t\t action");
    for(k=0,i=0; j<c; k++,i++,j++)
    {
        if(a[j]=='i' && a[j+1]=='d')
        {
            stk[i]=a[j];
            stk[i+1]=a[j+1];
            stk[i+2]='\0';
            a[j]=' ';
            a[j+1]=' ';
            printf("\n%s\t\t%s\t\t%s",stk,a,act);
            check();
        }
        else
        {
            stk[i]=a[j];
            stk[i+1]='\0';
            a[j]=' ';
            printf("\n%s\t\t%s\t\t%ssymbols",stk,a,act);
            check();
        }
    }
}

void check()
{
    strcpy(ac,"REDUCE TO E");
    for(z=0; z<c; z++)
    if(stk[z]=='i' && stk[z+1]=='d')
    {
        stk[z]='E';
        stk[z+1]='\0';
        printf("\n%s\t\t%s\t\t%s",stk,a,ac);
        j++;
    }
}
```



```

for(z=0; z<c; z++)
    if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
    {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+2]='\0';
        printf("\n$%s\t\t%s$\t\t%s",stk,a,ac);
        i=i-2;
    }
for(z=0; z<c; z++)
    if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
    {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+1]='\0';
        printf("\n$%s\t\t%s$\t\t%s",stk,a,ac);
        i=i-2;
    }
for(z=0; z<c; z++)
    if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
    {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+1]='\0';
        printf("\n$%s\t\t%s$\t\t%s",stk,a,ac);
        i=i-2;
    }
}

```

Output:

```

/tmp/P6z090Dc4x.o
GRAMMAR is
E->E+E
E->E*E
E->(E)
E->id
Enter input string
id+id+id*id
stack      input      action

$id      +id+id*id$      SHIFT->id
$E      +id+id*id$      REDUCE TO E
$E+      id+id*id$      SHIFT->symbols
$E+id      +id*id$      SHIFT->id
$E+E      +id*id$      REDUCE TO E
$E      +id*id$      REDUCE TO E
$E+      id*id$      SHIFT->symbols
$E+id      *id$      SHIFT->id
$E+E      *id$      REDUCE TO E
$E      *id$      REDUCE TO E
$E*      id$      SHIFT->symbols
$E*id      $      SHIFT->id
$E*E      $      REDUCE TO E
$E      $      REDUCE TO E

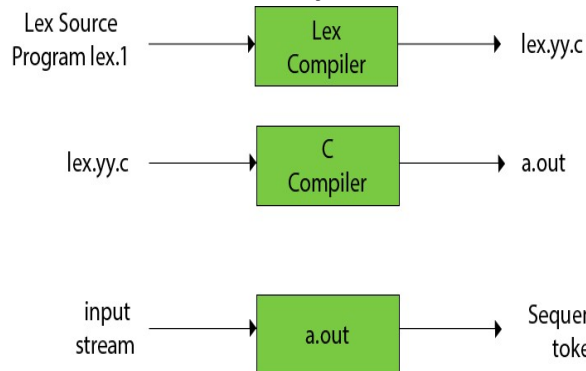
```

LEX → Lexical Analyzer Generator

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- Lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows →

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format → A Lex program is separated into three sections by %% delimiters.

1. %{ #include<headers>
2. %}
3. { definitions }
4. %%
5. { rules }
6. %%
7. { user subroutines }

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form $p_1 \{action_1\} p_2 \{action_2\} \dots p_n \{action_n\}$.

(Where p_i describes the regular expression and **action₁** describes the actions what action the lexical analyzer should take when pattern p_i matches a lexeme.)

User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

LEX → Yet Another Compiler-Compiler

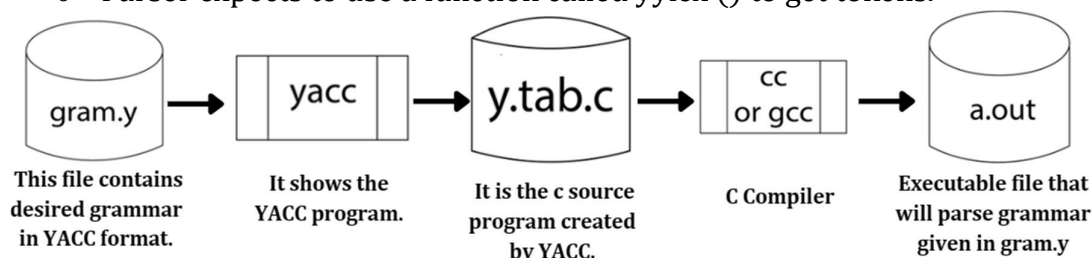
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

These are some points about YACC:

Input: A CFG- file.y

Output: A parser y.tab.c (yacc)

- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.



Basic Operational Sequence

<https://faun.pub/introduction-to-lex-and-yacc-b9bafab67447>

LEX & YACC TOOLS → CHECK IF IDENTIFIER OR NOT ?

```
E10a.l (~) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
E10a.l X
%{
#include<stdio.h>
%}
digit [0-9]
letter [a-z]
%%
{letter}({letter}|{digit})* {printf("\n Identifier Recognized!");}
.+ {printf("\n Entered Input is not an Identifier");}
%%
void main()
{
    printf("\n Enter Input to Recognize the if it's Identifier or not:");
    yylex();
}
```

Output:

```
user@localhost:~
File Edit View Search Terminal Help
[user@localhost ~]$ gedit E10a.l
[user@localhost ~]$ ls
a.out      E10a.l      ICG~       P1.java    Public
Desktop    E10a.l~     ICG.c      pgm5.l     Templates
Documents  Grandparent.java~ lex.yy.c   pgm5.l~    Unsaved Document 1
Downloads  ICG         Music      Pictures    Videos
[user@localhost ~]$ lex E10a.l
[user@localhost ~]$ ls
a.out      E10a.l      ICG~       P1.java    Public
Desktop    E10a.l~     ICG.c      pgm5.l     Templates
Documents  Grandparent.java~ lex.yy.c   pgm5.l~    Unsaved Document 1
Downloads  ICG         Music      Pictures    Videos
[user@localhost ~]$ gcc lex.yy.c -ll
[user@localhost ~]$ ./a.out

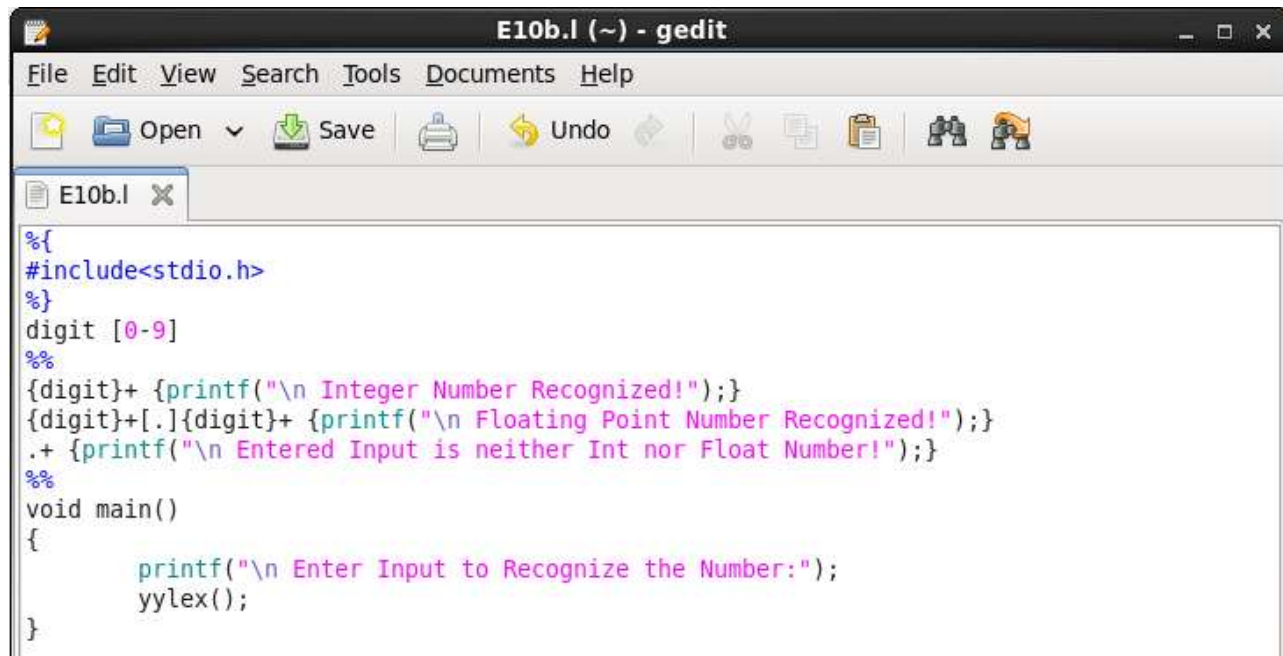
Enter Input to Recognize the if it's Identifier or not:a123

Identifier Recognized!
^C
[user@localhost ~]$ ./a.out

Enter Input to Recognize the if it's Identifier or not:123

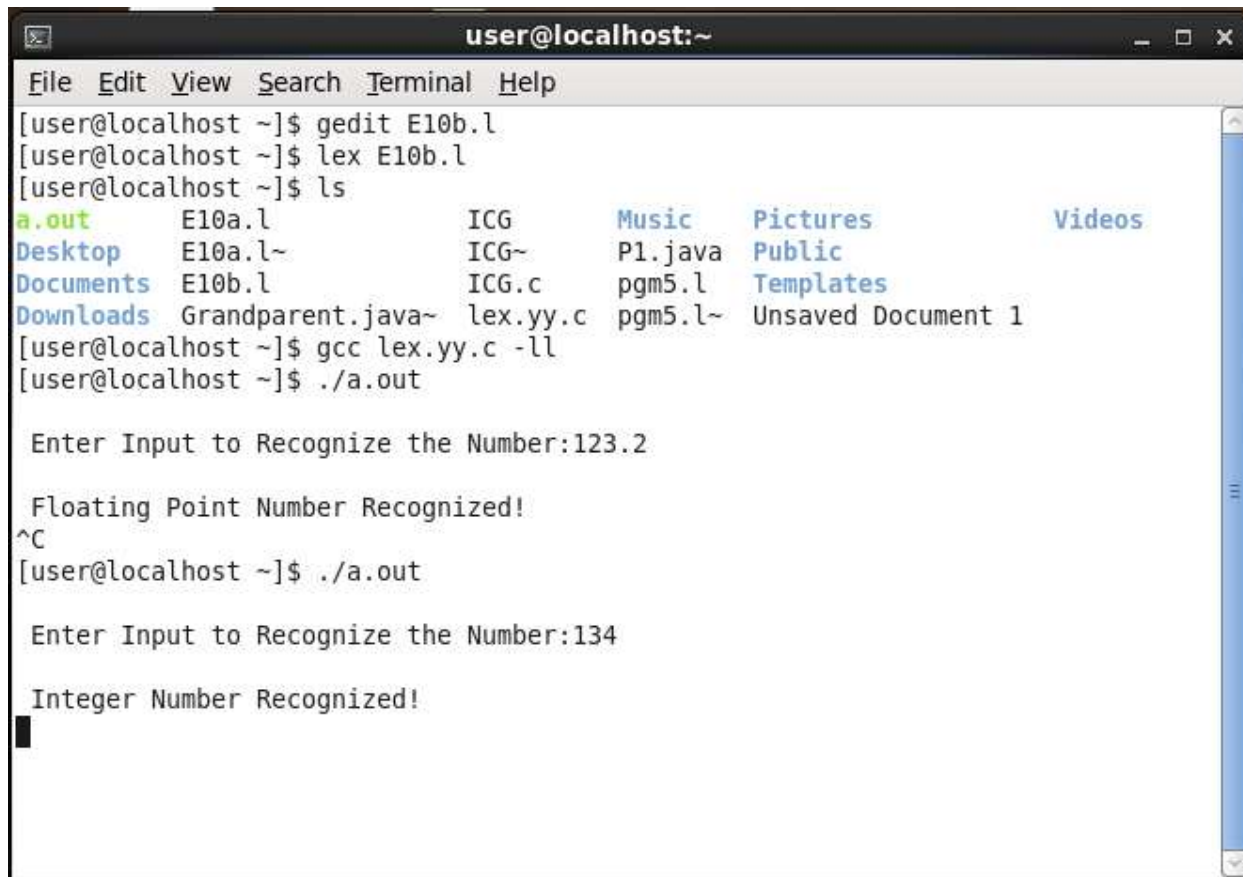
Entered Input is not an Identifier
```

LEX & YACC TOOLS → CHECK IF INTEGER OR FLOATING POINT NUMBER



```
E10b.l (~) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
E10b.l
%{
#include<stdio.h>
%}
digit [0-9]
%%
{digit}+ {printf("\n Integer Number Recognized!");}
{digit}+[.]{digit}+ {printf("\n Floating Point Number Recognized!");}
.+ {printf("\n Entered Input is neither Int nor Float Number!");}
%%
void main()
{
    printf("\n Enter Input to Recognize the Number:");
    yylex();
}
```

Output:



```
user@localhost:~
File Edit View Search Terminal Help
[user@localhost ~]$ gedit E10b.l
[user@localhost ~]$ lex E10b.l
[user@localhost ~]$ ls
a.out      E10a.l      ICG      Music      Pictures      Videos
Desktop    E10a.l~     ICG~     P1.java    Public
Documents  E10b.l      ICG.c    pgm5.l     Templates
Downloads  Grandparent.java~ lex.yy.c  pgm5.l~    Unsaved Document 1
[user@localhost ~]$ gcc lex.yy.c -ll
[user@localhost ~]$ ./a.out

Enter Input to Recognize the Number:123.2

Floating Point Number Recognized!
^C
[user@localhost ~]$ ./a.out

Enter Input to Recognize the Number:134

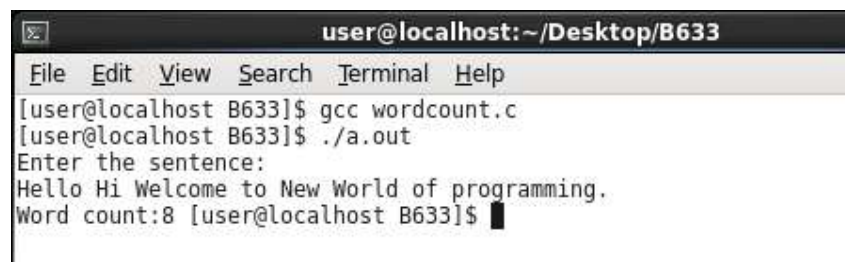
Integer Number Recognized!
```

FIND NO. OF WORDS IN A SENTENCE→

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

void main()
{
    char s[100];
    printf("Enter the sentence:\n");
    //scanf("%[^\\n]s",s);
    int i=0;
    int count = 0;
    while(1) {
        scanf("%c",&s[i]);
        if ( s[0] == '.' || s[0] == ' ' && s[1] == '.')
        {
            count = -1;
            break;
        }
        if(s[i] == ' ')
        {
            count=count+1;
        }
        else if(s[i]== '.')
        {
            break;
        }
        i=i+1;
    }
    int count1 = count+1;
    printf("Word count:%d ", count1);
}
```

Output:



```
user@localhost:~/Desktop/B633
File Edit View Search Terminal Help
[user@localhost B633]$ gcc wordcount.c
[user@localhost B633]$ ./a.out
Enter the sentence:
Hello Hi Welcome to New World of programming.
Word count:8 [user@localhost B633]$
```

FIND NO. OF CHARACTERS IN A WORD→

CODE #1

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

```
int main() {
    char word[100];
    int count = 0;

    // Input word from user
    printf("Enter a word: ");
    scanf("%s", word);

    // Count characters in the word
    count = strlen(word);

    // Output the result
    printf("The word \"%s\" has %d characters.\n", word, count);

    return 0;
}
```

Output

```
/tmp/IBW2LNw8ht.o
Enter a word: archeologist
The word "archeologist" has 12 characters.
```

CODE #2

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

```
int main() {
    char word[100];
    int count = 0;

    // Input word from the user
    printf("Enter a word: ");
    scanf("%s", word);

    // Count characters in the word
    for(int i = 0; word[i] != '\0'; i++) {
        count++;
    }

    // Output the result
    printf("The number of characters in the word is: %d\n", count);

    return 0;
}
```

Output

```
/tmp/h2jLaoNt39.o
Enter a word: conundrum
The number of characters in the word is: 9
```

SINGLE PASS ASSEMBLER→