



# CD LAB RECORD



**Sharath Shankaran D**

**CSE B 101**

Date: 23/09/2020

## Introduction

### Compiler:

Compiler is a software which converts a program written in high level language (Source Language) to low level language (Object/Target/Machine Language).

- **Cross Compiler** that runs on a machine 'A' and produces a code for another machine 'B'. It is capable of creating code for a platform other than the one on which the compiler is running.
- **Source-to-source Compiler** or trans-compiler or transpiler is a compiler that translates source code written in one programming language into source code of another programming language.

### Pre-Processor:

A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers. The amount and kind of processing done depends on the nature of the preprocessor; some preprocessors are only capable of performing relatively simple textual substitutions and macro expansions, while others have the power of full-fledged programming languages. Lexical preprocessors are the lowest-level of preprocessors as they only require lexical analysis, that is, they operate on the source text, prior to any parsing, by performing simple substitution of tokenized character sequences for other tokenized character sequences, according to user-defined rules. They typically perform macro substitution, textual inclusion of other files, and conditional compilation or inclusion.

### Interpreter:

An interpreter is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program. An interpreter generally uses one of the following strategies for program execution:

1. Parse the source code and perform its behavior directly;
2. Translate source code into some efficient intermediate representation and immediately execute this;
3. Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.

### Assembler:

Assembler is a program for converting instructions written in low-level assembly code into relocatable machine code and generating along information for the loader. It generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce

machine code. If assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler.

### **Linker:**

A **linker** or link editor is a computer system program that takes one or more object files (generated by a compiler or an assembler) and combines them into a single executable file, library file, or another 'object' file. Computer programs typically are composed of several parts or modules; these parts/modules need not all be contained within a single object file, and in such cases refer to each other by means of symbols as addresses into other modules, which are mapped into memory addresses when linked for execution. For most compilers, each object file is the result of compiling one input source code file. When a program comprises multiple object files, the linker combines these files into a unified executable program, resolving the symbols as it goes along.

### **Loader:**

A **loader** is the part of an operating system that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution. Loading a program involves reading the contents of the executable file containing the program instructions into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

### **Phases of Compiler:**

1. **Lexical Analysis:** The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as: **<token-name, attribute-value>**
2. **Syntax Analysis:** The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.
3. **Semantic Analysis:** Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.
4. **Intermediate Code Generation:** After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

5. **Code Optimization:** The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).
6. **Code Generation:** In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

**Symbol Table:** It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management

Program 1

Date: 23/09/2020

## **Vowels and Consonants**

Aim: To write a Lex program to count the occurrences of vowels and consonants in an input string.

### Algorithm

#### Definition Section

1. The required headers are included here.
2. Variables are declared and initialized (vowel – to count vowels and consonant- to count consonants).

#### Rule Section

3. We define regular expression required to count occurrence of vowels [aeiouAEIOU].
4. On each occurrence in 4 vowel is incremented by one.
5. Then regular expression for consonants is defined [a-zA-Z].
6. On occurrence of 5 consonant is incremented by one.

#### Main Section/Function

7. User is asked for an input.
8. The input is read by using yylex().
9. The counts of vowels and consonants are printed out separately.

### Result

The program was successfully compiled and executed and a result was obtained.

## Program

```
vowels - Notepad
File Edit Format View Help
/* Lex program to count number of vowels and consonants in an input string */

/* Declaration section */

%{
    #include <stdio.h>
    int vowel = 0;
    int consonant = 0;
}%

/* Defined section */

%%
[AEIOUaeiou] {vowel++;}
[a-zA-Z] {consonant++;}
%%

int yywrap() { return 1; }

/* Main function */

int main()
{
    printf("Enter three string of vowels and constants");
    yylex();
    printf("Number of vowels are: %d\n", vowel);
    printf("Number of consonants are: %d\n", consonant);
    return 0;
}
```

## Output

```
Command Prompt
Microsoft Windows [Version 10.0.18362.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Sharath Shankaran>cd Desktop\flex_code\vowels_and_consonants
C:\Users\Sharath Shankaran\Desktop\flex_code\vowels_and_consonants>flex vowels.l
C:\Users\Sharath Shankaran\Desktop\flex_code\vowels_and_consonants>gcc lex.yy.c
C:\Users\Sharath Shankaran\Desktop\flex_code\vowels_and_consonants>a.exe
Enter three string of vowels and constants The quick brown fox jumps over the lazy dog
^Z
Number of vowels are: 11
Number of consonants are: 24
C:\Users\Sharath Shankaran\Desktop\flex_code\vowels_and_consonants>
```

Program 2

Date: 30/09/2020

## **Counting spaces, words, letters and lines from the input file**

Aim: To write a Lex program to count the number of spaces, letters, words and lines in an input file.

### Algorithm

#### Definition Section

1. Required headers are included.
2. Variables are declared and initialized.

#### Rule Section

3. Regular expression to count lines (`[\n]`).Incremented whenever a newline character is encountered.
4. Regular expression to count letters (`[a-zA-Z]`).Incremented whenever letters are encountered.
5. Regular expression to count spaces (`[ ]`).Incremented whenever a blank space is encountered.
6. Regular expression to count sentences (`"."`).Increment sentence count whenever a full stop is encountered.
7. Regular expression to count other characters (`.`).Incremented when special characters are encountered.

#### Main Section

8. An example text file is opened in read mode with `fopen()` and the File pointer is assigned to `yyin()`.
9. The words, spaces, lines, letters, sentences are displayed.
10. Stop

## Result

The program was successfully compiled and executed and a result was obtained.

## Program:

```
count - Notepad
File Edit Format View Help
%{
#include<stdio.h>
int wd_count = 0,s_count = 0,sp_count = 0,l_count = 0,ch_count = 0,let_count = 0;
}%
%%
[\\n] {l_count++;}
[a-zA-Z] {let_count++;}
[ ] {sp_count++;}
"-' {s_count++;}
. {ch_count++;}
%%
int yywrap(){}
int main()
{
yyin = fopen("example.txt","r");
yylex();
printf("Words :%d\\n",sp_count+s_count);
printf("Sentences :%d\\n",s_count);
printf("Lines :%d\\n",l_count);
printf("Spaces :%d\\n",sp_count);
printf("Symbol :%d\\n",ch_count);
printf("Letters :%d\\n",let_count);
}
}
```

## Input:

```
example - Notepad
File Edit Format View Help
My name is Sharath Shankaran D.
IPL is great.
RCB is the best in the game.
I am an engineering student in final year.
Some other symbols to count l@#$%.
```

## Output:

```
Command Prompt
Microsoft Windows [Version 10.0.18362.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Sharath Shankaran>cd Desktop\flex_code\Count
C:\Users\Sharath Shankaran\Desktop\flex_code\Count>flex count.l
C:\Users\Sharath Shankaran\Desktop\flex_code\Count>gcc lex.yy.c
C:\Users\Sharath Shankaran\Desktop\flex_code\Count>a.exe
Words :30
Sentences :5
Lines :5
Spaces :25
Symbol :5
Letters :113
C:\Users\Sharath Shankaran\Desktop\flex_code\Count>
```



Program 3

Date: 21/10/2020

## Identifying Keywords and capitalizing them

Aim: To write a Lex program to identify keywords from a C file and capitalize them.

Algorithm:

Definition Section

1. Required headers are included.
2. Required variables are declared.

Rule Section

3. A regular expression for keywords [**keyword**] with different possible keywords is defined.
4. Using **yylength()** to read each character from the keyword that is read and using the **ctype** function **toupper()** each of the characters is converted to capital case.
5. The result is displayed as they are converted.


Main Section

6. A C file is opened with read mode using **yyin()**.
7. **yylex()** makes sure the rule section is mapped.
8. Output is generated.

### Result:

The program was successfully compiled and executed and a result was obtained.

### Program:

 cap - Notepad

File Edit Format View Help

```
%{#include<stdio.h>
#include<ctype.h>
int i;
%}
keyword main|int|scanf|printf|if|else|float|char|return|
%%
{keyword} {
    for(i=0;i<yyleng;i++)
        printf("%c",toupper(yytext[i]));
}
%%

void main()
{
    yyin=fopen("ex.c","r");
    yylex();
}

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

### Output:

Command Prompt

```
C:\Users\Sharath Shankaran\Desktop\flex_code\Capitalize>flex cap.1
C:\Users\Sharath Shankaran\Desktop\flex_code\Capitalize>gcc lex.yy.c
C:\Users\Sharath Shankaran\Desktop\flex_code\Capitalize>a.exe
INT MAIN()
{
    FLOAT marks;
    CHAR grade;

    PRINTF("Enter marks: ");
    SCANF("%f", &marks);

    IF(marks >= 90)
    {
        grade = 'A';
    }
    ELSE IF(marks >= 80 && marks < 90)
    {
        grade = 'B';
    }
    ELSE IF(marks >= 70 && marks < 80)
    {
        grade = 'C';
    }
    ELSE IF(marks >= 60 && marks < 70)
    {
        grade = 'D';
    }
    ELSE IF(marks >= 50 && marks < 60)
    {
        grade = 'E';
    }
    ELSE
    {
        grade = 'F';
    }

    PRINTF("Your grade is %c", grade);

    RETURN 0;
}
C:\Users\Sharath Shankaran\Desktop\flex_code\Capitalize>
```

### The C File:

ex - Notepad

File Edit Format View Help

```
int main()
{
    float marks;
    char grade;

    printf("Enter marks: ");
    scanf("%f", &marks);

    if(marks >= 90)
    {
        grade = 'A';
    }
    else if(marks >= 80 && marks < 90)
    {
        grade = 'B';
    }
    else if(marks >= 70 && marks < 80)
    {
        grade = 'C';
    }
    else if(marks >= 60 && marks < 70)
    {
        grade = 'D';
    }
    else if(marks >= 50 && marks < 60)
    {
        grade = 'E';
    }
    else
    {
        grade = 'F';
    }

    printf("Your grade is %c", grade);

    return 0;
}
```

## Program 4

Date: 28/10/2020

### Infix Calculator

Aim: To write a Lex program to take input from the user in the form of infix expression and calculate the result.

#### Algorithm:

##### Definition Section

1. Required headers are included.
2. Required variables are declared.

##### Rule Section

3. Regular expressions for operations `[+]`, `[-]`, `[*]`, `[/]` is defined with each given an integer value so that that can be used to calculate what the expression wants.
4. Using `[0-9]` + to read input digits of expression, a flag is used to control flow also `atoi()` is used to convert ASCII to integer values such that the result of the expression is not in ASCII.
5. For each case of operation the corresponding value is calculated and saved into a variable

##### Main Section

6. User is asked to enter an infix expression.
7. `yylex()` makes sure the rule section is mapped.
8. Output is generated.

Result: The program was successfully compiled and executed and a result was obtained.

## Program:

```
infix - Notepad
File Edit Format View Help
%{
#include<stdio.h.>
int flag = 0, a = 0, b = 0, op, c = 0;
}%
%%
[+] {op = 1;}
[-] {op = 2;}
[*] {op = 3;}
[/] {op = 4;}
[0-9]+ {if(flag == 0) {a=atoi(yytext); flag = 1;}
      else {
          b = atoi(yytext);
          if(op == 1) {c = a+b; a=c;}
          if(op == 2) {c = a-b; a=c;}
          if(op == 3) {c = a*b; a=c;}
          if(op == 4) {c = a/b; a=c;}
      }
}
}%
int yywrap(){}
int main()
{
printf("Enter the input :");
yylex();
printf("Answer = %d", c);
return 0;
}
```

## Output:

```
cmd Command Prompt

C:\Users\Sharath Shankaran\Desktop\flex_code\infix>a.exe
Enter the input :33*4

^Z
Answer = 132
C:\Users\Sharath Shankaran\Desktop\flex_code\infix>a.exe
Enter the input :33+4

^Z
Answer = 37
C:\Users\Sharath Shankaran\Desktop\flex_code\infix>a.exe
Enter the input :33-45

^Z
Answer = -12
C:\Users\Sharath Shankaran\Desktop\flex_code\infix>a.exe
Enter the input :256/16

^Z
Answer = 16
C:\Users\Sharath Shankaran\Desktop\flex_code\infix>
```

## **Counting Keywords, Identifiers, Comments, Operators and Preprocessor Directives in a C program file.**

Aim: To write a Lex program to count keywords, Identifiers, comments, operators and preprocessor directives in a C program file.

Algorithm:

### Definition Section

1. Required headers are included.
2. Required variables are declared and initialized.

### Rule Section

3. Regular Expression for  
Keywords [**main|int|scanf|printf|else|return|char|float|if|else**],  
operators [**"+"|"-"|"\*"|"/"|"="|">="|"<="|"&&"|"<"|">"**],  
preprocessors [**#**], comments are written.
4. Whenever any of the above are encountered their respective counts are increased.
5. For identifiers the rest of the words are considered and the count is incremented accordingly.
6. All these when read are written into an output file with **fprintf()** function.
7. Semicolons and the rest are copied to the output file.

### Main Section

8. A C program file is opened in read mode with **yyin()**.
9. **yyout()** writes the output to an output file of user's choice.
10. Finally the count of keywords, Identifiers, comments, operators and preprocessor directives in the C program file is also displayed.
11. End

## Result

The program was successfully compiled and executed and a result was obtained.

## Program:


```
keyword - Notepad
File Edit Format View Help
%{#include<stdio.h>
int pre =0,num=0,key=0,op=0,id=0,scmt=0,mcmt=0;
%}
keyword main|int|scanf|printf|else|return|char|float|if|else
operators "+"|"-"|"*"|"/"|"="|">="|"<="|"&&"|"<"|">"
start \\\"
end \*\"
%%
[#] {fprintf(yyout,"%s",yytext);pre++;}
[0-9]* {fprintf(yyout,"%s",yytext);num++;}
{keyword} {fprintf(yyout,"%s",yytext);key++;}
{operators} {fprintf(yyout,"%s",yytext);op++;}
[a-zA-Z][a-zA-Z0-9]* {fprintf(yyout,"%s",yytext);id++;}

\\\"/(.*) {scmt++;}
{start}.*{end} {mcmt++;}

[\\n \\t ] {fprintf(yyout,"%s",yytext);}
[" "] {;}
%%
int yywrap()
{
return 1;
}
int main()
{
yyin=fopen("cfile.c","r");
yyout=fopen("output.txt","w");
yylex();
printf("no. of keywords : %d \\n",key);
printf("no. of operators: %d \\n",op);
printf("no. of comments: %d \\n",(mcmt+scmt));
printf("no. of identifiers :%d \\n",id);
printf("no. of numbers : %d \\n",num);
printf("no. of preprocessor directives : %d\\n",pre);
return 0;
}
```



## C file:

 cfile - Notepad

File Edit Format View Help

//A C program to find the grade of a student

```
#include<stdio.h>

int main()
{
    float marks;
    char grade;

    printf("Enter marks: ");
    scanf("%f", &marks);

    if(marks >= 90)
    {
        grade = 'A';
    }
    else if(marks >= 80 && marks < 90)
    {
        grade = 'B';
    }
    else if(marks >= 70 && marks < 80)
    {
        grade = 'C';
    }
    else if(marks >= 60 && marks < 70)
    {
        grade = 'D';
    }
    else if(marks >= 50 && marks < 60)
    {
        grade = 'E';
    }
    else
    {
        grade = 'F';
    }

    printf("Your grade is %c", grade);

    return 0;
}
```

## Output:

```
Command Prompt

C:\Users\Sharath Shankaran>cd Desktop/flex_code/Keywords
C:\Users\Sharath Shankaran\Desktop\flex_code\Keywords>flex keyword.1
C:\Users\Sharath Shankaran\Desktop\flex_code\Keywords>gcc lex.yy.c
C:\Users\Sharath Shankaran\Desktop\flex_code\Keywords>a.exe
no. of keywords : 18
no. of operators: 21
no. of comments: 1
no. of identifiers :35
no. of numbers : 10
no. of preprocessor directives : 1
C:\Users\Sharath Shankaran\Desktop\flex_code\Keywords>
```

## Output File:

```
output - Notepad
File Edit Format View Help

#include<stdio.h>

int main()
{
    float marks;
    char grade;

    printf(Enter marks: );
    scanf(%f, &marks);

    if(marks >= 90)
    {
        grade = 'A';
    }
    else if(marks >= 80 && marks < 90)
    {
        grade = 'B';
    }
    else if(marks >= 70 && marks < 80)
    {
        grade = 'C';
    }
    else if(marks >= 60 && marks < 70)
    {
        grade = 'D';
    }
    else if(marks >= 50 && marks < 60)
    {
        grade = 'E';
    }
    else
    {
        grade = 'F';
    }

    printf(Your grade is %c, grade);

    return 0;
}
```

Program 6

Date: 17/11/2020

## **Identifier Validation**

Aim: To write a Yacc program to check whether the entered string is a valid identifier or not.

### Algorithm:

1. Start
2. Declare tokens for A B NL.
3. Set grammar as start->S NL, S->A|AC, and C -> AC|A|B|BC.
4. Set rules as 5 to 8.
5. If the string contains an alphabet or its combinations return A.
6. If the string contains a number or its combinations return B.
7. If the string is having “\_” return A
8. If ‘\n’ is found, return NL.
9. Get input string.
10. Read input character by character.
11. If the string follows the grammar print “Valid identifier”.
12. Else print “Invalid identifier”.
13. Stop

### Result

The program was successfully compiled and executed and a result was obtained.

## Program:

### Lex file:

```
lex_part - Notepad
File Edit Format View Help
%{
#include "y.tab.h"
%}
%%
[a-zA-Z_]+ {return letter;}
[0-9]+ {return digit;}
\n {return NL;}
. {return yytext[0];}
%%
```

### Yacc file:

```
yacc_part.y - Notepad
File Edit Format View Help
%{
#include<stdio.h>
#include<stdlib.h>
int yylex();
%}

%token digit letter NL;

%%
start: S NL      {printf("Valid Identifier\n"); exit(0);};

S : letter C|letter;
C : letter C|letter|digit|digit C;
%%

int yyerror()
{
printf("Invalid Identifier\n");
exit(0);
}

int main()
{
printf("Enter the string:\t");
yyparse();
return 1;
}

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

## Output:

cmd Command Prompt

```
C:\Users\Sharath Shankaran\Desktop\flex_code\Identifier>flex lex_part.1
C:\Users\Sharath Shankaran\Desktop\flex_code\Identifier>bison -dy yacc_part.y
C:\Users\Sharath Shankaran\Desktop\flex_code\Identifier>gcc lex.yy.c y.tab.c
y.tab.c: In function 'yyparse':
y.tab.c:1356:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
yyerror (YY_("syntax error"));
^~~~~~
yyerrok
C:\Users\Sharath Shankaran\Desktop\flex_code\Identifier>a.exe
Enter the string: _abxcgx
Valid Identifier
C:\Users\Sharath Shankaran\Desktop\flex_code\Identifier>a.exe
Enter the string: 123abd
Invalid Identifier
C:\Users\Sharath Shankaran\Desktop\flex_code\Identifier>a.exe
Enter the string: abcd-56
Invalid Identifier
C:\Users\Sharath Shankaran\Desktop\flex_code\Identifier>a.exe
Enter the string: a_123
Valid Identifier
```

Program 7

Date: 02/12/2020

## Arithmetic Expression Validation

Aim: To write a Yacc program to check if the given arithmetic expression is valid or not.

Algorithm:

Lex Program

Definition Section:

1. `#include "y.tab.h" //header file needed in the program`

Rule Section:

2. `[a-zA-Z] //return ID token to parser`
3. `[0-9]+ //return number token to parser`
4. `[\t] //ignore \t tab spaces`
5. `\n //end of expression, so return 0 to terminate lexer execution`
6. `. //any other character is returned as it is`

`yywrap()` function is responsible for checking whether there are any more inputs. If no, then `yywrap()` tells the program to end.

Yacc Program

Definition Section:

1. `#include<stdio.h> ,#include <stdlib.h> //header file needed in the program`
2. `%token ID NUMBER, %left '+' '-' , %left '*' '/' //tokens which are getting returned from lexer.`

### Rule Section:

3. `expr '+' expr , expr '-' expr , expr '*' expr , expr '/' expr , '(' expr ')'` , NUMBER , ID  
// the above mentioned expressions are grammar defined in multiple ways. If input exactly satisfies this grammar, then we return to `main()` successfully, and print "Valid" in the `main()`. Else, `yyerror` gets called, and the message "Invalid" gets printed.

### Result

The program was successfully compiled and executed and a result was obtained.

### Program

#### Lex File

```
p7 - Notepad
File Edit Format View Help
%{
#include "y.tab.h"
%}
%option noyywrap
%%
[a-zA-Z] {return ID;}
[0-9]+ {return NUMBER;}
[ \t] {;}
\n {return 0;}
. { return yytext[0];}
%%
```

## Yacc File

```
p7 - Notepad
File Edit Format View Help

%{
int yylex();
void yyerror();
#include<stdio.h>
#include <stdlib.h>
%}

%token ID NUMBER
%left '+' '-'
%left '*' '/'
%%
stmt:expr
;
expr:
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '(' expr ')'
| NUMBER
| ID
;

%%
void main()
{
printf("enter expr : \n");
yyparse();
printf("valid exp");
exit(0);
}
void yyerror()
{
printf("invalid exp");
exit(0);
}
```

## Output

```
C:\Windows\System32\cmd.exe

C:\Users\Sharath Shankaran\Desktop\flex_code\Program7>a.exe
enter expr :
(a+b)*(c-d)
valid exp
C:\Users\Sharath Shankaran\Desktop\flex_code\Program7>a.exe
enter expr :
(a+b+c+d)/e
valid exp
C:\Users\Sharath Shankaran\Desktop\flex_code\Program7>a.exe
enter expr :
a+b-
invalid exp
C:\Users\Sharath Shankaran\Desktop\flex_code\Program7>
```



Program 8

DATE: 07/12/2020

## Infix Calculator

Aim: To write Lex and Yacc program to implement infix calculator

### Algorithm

1. Starts
2. Declare tokens for NUMBER
3. Set grammar as start  $\rightarrow E$ , print \$\$ or
4. E:  $E+E$  then return  $$$= \$1+ \$2$  or
5. E:  $E-E$  then return  $$$= \$1- \$2$  or
6. E:  $E * E$  then return  $$$= \$1 * \$2$  or
7. E:  $E / E$  then return  $$$= \$1 / \$2$  or
8. E:  $E \% E$  then return  $$$= \$1 \% \$2$  or
9. E: (E) then return  $$$= \$2$  or
10. E: NUMBER then return  $$$= \$1$
11. Set rules as if [0-9] is found return NUMBER
12. Get input expression
13. Read input character by character
14. If the string follows the grammar
15. Print "Valid expression".
16. Evaluate it and print the result
17. Else print "Invalid expression"
18. Stop

### Result

The program executed correctly and output was verified.

## **Program:**

Lex file:



p8 - Notepad

File Edit Format View Help

---

```
%{
#include<stdio.h>
#include"y.tab.h"
extern int yylval;
}%

%%
[0-9]+ {yylval = atoi(yytext); return NUMBER;}
[\\t];
[\\n] return 0;
. return yytext[0];
%%

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

## Yacc File

 p8 - Notepad

File Edit Format View Help

```
%{
#include<stdio.h>
int yylex();
void yyerror();
int flag = 0;
%}

%token NUMBER

%left '+' '-'
%left '*' '/'
%left '(' ')'
%%

ArithmeticExpression:E{printf("\nResult:\t%d",$$);return 0;};

E : E '+' E {$$=$1+$3;}
   | E '-' E {$$=$1-$3;}
   | E '*' E {$$=$1*$3;}
   | E '/' E {$$=$1/$3;}
   | '(' E ')' {$$=$2;}
   | NUMBER {$$=$1;}
   | '-' E {$$=-$2;}
;

%%

void main()
{
printf("Enter the expression:\n");
yyparse();
if (flag==0){printf("\nValid");}
}
void yyerror()
{
printf("\nInvalid");
flag=1;
}
```

## OUTPUT

C:\Windows\System32\cmd.exe

```
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 8>flex p8.l
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 8>bison -dy p8.y
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 8>gcc lex.yy.c y.tab.c
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 8>a.exe
Enter the expression:
23
Result: 23
Valid
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 8>a.exe
Enter the expression:
-023
Result: -23
Valid
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 8>a.exe
Enter the expression:
0/7
Result: 0
Valid
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 8>a.exe
Enter the expression:
7-0/89*6
Result: 7
Valid
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 8>a.exe
Enter the expression:
2-3-/8
Invalid
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 8>
```

Program 9

Date: 9/12/2020

## **Intermediate Code Generator**

Aim: To write a Yacc program that generates the Intermediate Code for the given expression.

### **Algorithm**

#### **L file**

1. Start
2. Include all the required header files.
3. Declare an external variable yylval to send to Yacc.
4. ALPHA is assigned the regex for the alphabets.
5. DIGIT is assigned regex for the numerical values
6. When ALPHA is encountered returns ID
7. When DIGIT is encountered NUM is returned.
8. For newline [\n] and space [\t] it exits.

#### **Y file**

9. Start
10. Include all the required headers.
11. Functions codegen(), codegen\_umin(), codegen\_assign() prototypes are written.
12. A stack is created (of type char).
13. Operator associativity is defined with “%left”.
14. Grammar for the arithmetic expression is defined with all relevant operations.
15. The stack top will have the expression and the three address notation variable (t).
16. The characters t and i [0] are concatenated and printed along with the required expression.
17. Final assignment is done with popping out the last element in the stack top.
18. Stop

Result: The program was successfully compiled and executed and a result was obtained.

## Program

### Lex file



p9 - Notepad

File Edit Format View Help

```
%{
#include <stdio.h>
#include "y.tab.h"
extern int yylval;
}%
ALPHA [A-Za-z]
DIGIT [0-9]
%%
{ALPHA}({ALPHA}|{DIGIT})* return ID;
{DIGIT}+ {
yylval=atoi(yytext);
return NUM;
}
[\n \t] {yyterminate();}
. {return yytext[0];}
%%
int yywrap()
{
return 1;
}
```

## Yacc file

```
p9 - Notepad
File Edit Format View Help
%token NUM ID
%right '='
%left '+' '-'
%left '*' '/'
%left UMINUS
%%
S : ID{push();} '=' {push();} E{codegen_assign();};
E : E '+' {push();} T{codegen();}
  | E '-' {push();} T{codegen();}
  | T ;
T : T '*' {push();} F{codegen();}
  | T '/' {push();} F{codegen();}
  | F ;
F : '(' E ')'
  | '-' {push();} F {codegen_umin();}
  | ID{push();}
  | NUM{push();};
%%
int main(){
printf("Enter the expression : ");
yyvsparse();return 0;
}
int push(){
strcpy (st[++top],yytext);
return 0;}
int codegen(){
strcpy(temp,"t");
strcat(temp,i_);
printf( "%s = %s %s %s\n" ,temp,st[top- 2 ],st[top- 1 ],st[top]);
top-= 2 ;
strcpy(st[top],temp);
i_ [ 0 ]++;
return 0;}
void codegen_umin(){
strcpy(temp, "t" );
strcat(temp, i_ );
printf( "%s = -%s\n" ,temp,st[top]);
top--;
strcpy(st[top],temp);
i_ [ 0 ]++;
}
void codegen_assign(){
printf( "%s = %s \n" ,st[top- 2 ],st[top]);
top-= 2 ;
}
void yyerror()
{
printf("\nError\n\n");}
```

## Output

```
C:\Windows\System32\cmd.exe

C:\Users\Sharath Shankaran\Desktop\flex_code\Program 9>flex p9.1
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 9>bison -dy p9.y
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 9>gcc lex.yy.c y.tab.c
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 9>a.exe
Enter the expression : a=(b+c)*(d/e)+f
t0 = b + c
t1 = d / e
t2 = t0 * t1
t3 = t2 + f
a = t3

C:\Users\Sharath Shankaran\Desktop\flex_code\Program 9>a.exe
Enter the expression : a=b--c
t0 = -c
t1 = b - t0
a = t1

C:\Users\Sharath Shankaran\Desktop\flex_code\Program 9>a.exe
Enter the expression : a=c*d-b+
t0 = c * d
t1 = t0 - b

Error

C:\Users\Sharath Shankaran\Desktop\flex_code\Program 9>
```

## **First and Follow of a Grammar**

Aim: To write a C program to find the first and follow of a grammar.

### **Algorithm**

[NOTE: epsilon is taken as #]

1. Start
2. Perform the steps 3-5 to compute FIRST of a Non Terminal
3. If  $x$  is a terminal, then  $\text{FIRST}(x) = \{x\}$
4. If  $X \rightarrow \epsilon$ , is a production rule, then add  $\epsilon$  to  $\text{FIRST}(x)$ .
5. If  $X \rightarrow Y_1 Y_2 Y_3 \dots Y_n$  is a production,
6.  $\text{FIRST}(X) = \text{FIRST}(Y_1)$
7. If  $\text{FIRST}(Y_1)$  contains  $\epsilon$  then  $\text{FIRST}(X) = \{\text{FIRST}(Y_1) - \epsilon\} \cup \{\text{FIRST}(Y_2)\}$
8. If  $\text{FIRST}(Y_i)$  contains  $\epsilon$  for all  $i = 1$  to  $n$ , then add  $\epsilon$  to  $\text{FIRST}(X)$ .
9. Perform the steps 7-10 to compute FIRST of a Non Terminal
10.  $\text{FOLLOW}(S) = \{\$ \}$  where  $S$  is the starting Non-Terminal
11. If  $A \rightarrow pBq$  is a production, where  $p$ ,  $B$  and  $q$  are any grammar symbols,
12. Everything in  $\text{FIRST}(q)$  except  $\epsilon$  is in  $\text{FOLLOW}(B)$ .
13. If  $A \rightarrow pB$  is a production,
14. Then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ .
15. If  $A \rightarrow pBq$  is a production and  $\text{FIRST}(q)$  contains  $\epsilon$ ,
16. Then  $\text{FOLLOW}(B)$  contains  $\{\text{FIRST}(q) - \epsilon\} \cup \text{FOLLOW}(A)$
17. Print Computed Arrays for each Non Terminal
18. End



## Result

The program executed correctly and output was verified.

## Program

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
void followfirst(char, int, int);
void follow(char c);
void findfirst(char, int, int);
int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10];
char f[10], first[10];
int k,e;
char ck;
int main(int argc, char **argv)
{
    int jm = 0, km = 0, i, choice, nop;
    char c, ch;
    printf("Enter the number of productions:");
```

```

scanf("%d", &nop);
count=nop;
printf("Enter Productions:\n");
for(i=0;i<nop;i++){
scanf("%s",production[i]);
}
int kay;
char done[count];
int ptr = -1;
for(k = 0; k < count; k++)
{
for(kay = 0; kay < 100; kay++) {
calc_first[k][kay] = '!'; }
}
int point1 = 0, point2, xxx;
for(k = 0; k < count; k++)
{
c = production[k][0];
point2 = 0;
xxx = 0;
for(kay = 0; kay <= ptr; kay++)
if(c == done[kay])
xxx = 1;
if (xxx == 1)

```

```
continue;
findfirst(c, 0, 0);
ptr += 1;
done[ptr] = c;
printf("\n First(%c) = { ", c);
calc_first[point1][point2++] = c;
for(i = 0 + jm; i < n; i++) {
int lark = 0, chk = 0;
for(lark = 0; lark < point2; lark++) {
if (first[i] == calc_first[point1][lark])
{
chk = 1;
break;
}
}
if(chk == 0)
{
printf("%c, ", first[i]);
calc_first[point1][point2++] = first[i];
}
}
printf("}\n");
jm = n;
point1++;
```

```

}
printf("\n*****\n\n");
char donee[count];
ptr = -1;
for(k = 0; k < count; k++) {
    for(kay = 0; kay < 100; kay++) {
        calc_follow[k][kay] = '!';
    }
}
point1 = 0;
int land = 0;
for(e = 0; e < count; e++)
{
    ck = production[e][0];
    point2 = 0;
    xxx = 0;
    for(kay = 0; kay <= ptr; kay++)
        if(ck == donee[kay])
            xxx = 1;
    if (xxx == 1)
        continue;
    land += 1;
    follow(ck);
    ptr += 1;
}

```

```

donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;
for(i = 0 + km; i < m; i++) {
int lark = 0, chk = 0;
for(lark = 0; lark < point2; lark++)
{
if (f[i] == calc_follow[point1][lark])
{
chk = 1;
break;
}
}
if(chk == 0)
{
printf("%c, ", f[i]);
calc_follow[point1][point2++] = f[i];
}
}
printf(" }\n\n");
km = m;
point1++;
}
}

```

```

void follow(char c)
{
    int i, j;
    if(production[0][0] == c) {
        f[m++] = '$';
    }
    for(i = 0; i < 10; i++)
    {
        for(j = 2; j < 10; j++)
        {
            if(production[i][j] == c)
            {
                if(production[i][j+1] != '\0')
                {
                    followfirst(production[i][j+1], i, (j+2));
                }
                if(production[i][j+1] == '\0' && c != production[i][0])
                {
                    follow(production[i][0]);
                }
            }
        }
    }
}

```

```

void findfirst(char c, int q1, int q2)
{
    int j;
    if(!(isupper(c))) {
        first[n++] = c;
    }
    for(j = 0; j < count; j++)
    {
        if(production[j][0] == c)
        {
            if(production[j][2] == '#')
            {
                if(production[q1][q2] == '\0')
                first[n++] = '#';
                else if(production[q1][q2] != '\0'
                && (q1 != 0 || q2 != 0))
                {
                    findfirst(production[q1][q2], q1, (q2+1));
                }
            }
            else
            first[n++] = '#';
        }
        else if(!isupper(production[j][2]))
        {

```

```

first[n++] = production[j][2];
}
else
{
    findfirst(production[j][2], j, 3);
}
}
}
}
}
void followfirst(char c, int c1, int c2)
{
    int k;
    if(!(isupper(c)))
        f[m++] = c;
    else
    {
        int i = 0, j = 1;
        for(i = 0; i < count; i++)
        {
            if(calc_first[i][0] == c)
                break;
        }
        while(calc_first[i][j] != '!')
        {

```



```
if(calc_first[i][j] != '#')
{
f[m++] = calc_first[i][j];
}
else
{
if(production[c1][c2] == '\0')
{
follow(production[c1][0]);
}
else
{
followfirst(production[c1][c2], c1, c2+1);
}
}
j++;
}
}
}
```

## OUTPUT

```
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 10>a.exe
Enter the number of productions:4
Enter Productions:
S=ABC
A=aA
B=aB
C=c

First(S) = { a, }

First(A) = { a, }

First(B) = { a, }

First(C) = { c, }

*****

Follow(S) = { $, }

Follow(A) = { a, }

Follow(B) = { c, }

Follow(C) = { $, }
```

Program 11

DATE: 20/12/2020

## **OPERATOR PRECEDENCE PARSER**

Aim: To write a C program to implement operator precedence parser.

### Algorithm

1. Start
2. Get the language to be checked
3. Set p to point to the first symbol of w\$
4. Repeat 5 and 6 forever
5. If ( \$ is on top of the stack and p points to \$ ) then return
6. Else do following
  - a) let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;
  - c) If (  $a < \cdot b$  or  $a = \cdot b$  ) then do ( SHIFT)
    - push b onto the stack
    - advance p to the next input symbol
  - d) Else if (  $a \cdot b$  ) then do ( REDUCE)
    - repeat pop stack
    - until ( the top of stack terminal is related by  $<$  to the terminal most recently popped )
  - e) Else give error message
7. Stop

### Result

The program executed correctly and output was verified.

## Program

```
#include<stdio.h>

#include<string.h>

char *input;

int i=0;

char lasthandle[6],stack[50],handles[][5]={")E(","E*E","E+E","i","E^E"};

//(E) becomes )E( when pushed to stack

int top=0,l;

char prec[9][9]={

    /*input*/

    /*stack + - * / ^ i ( ) $ */

    /* + */ '>','>','<','<','<','<','<','<','>','>',

    /* - */ '>','>','<','<','<','<','<','<','>','>',

    /* * */ '>','>','>','>','<','<','<','<','>','>',

    /* / */ '>','>','>','>','<','<','<','<','>','>',

    /* ^ */ '>','>','>','>','<','<','<','<','>','>',

    /* i */ '>','>','>','>','>','e','e','>','>',

    /* ( */ '<','<','<','<','<','<','<','<','>','e',

    /* ) */ '>','>','>','>','>','e','e','>','>',

    /* $ */ '<','<','<','<','<','<','<','<','<','>',

};

int getindex(char c)

{

    switch(c)
```

```

{
case '+':return 0;
case '-':return 1;
case '*':return 2;
case '/':return 3;
case '^':return 4;
case 'i':return 5;
case '(':return 6;
case ')':return 7;
case '$':return 8;
}
}

int shift()
{
stack[++top]=*(input+i++);
stack[top+1]='\0';
}

int reduce()
{
int i,len,found,t;
for(i=0;i<5;i++)//selecting handles
{
len=strlen(handles[i]);
if(stack[top]==handles[i][0]&&top+1>=len)
{

```

```

found=1;
for(t=0;t<len;t++)
{
if(stack[top-t]!=handles[i][t])
{
found=0;
break;
}
}
if(found==1)
{
stack[top-t+1]='E';
top=top-t+1;
strcpy(lasthandle,handles[i]);
stack[top+1]='\0';
return 1;//successful reduction
}
}
}
return 0;
}

void dispstack()
{
int j;
for(j=0;j<=top;j++)

```

```

    printf("%c",stack[j]);
}
void dispinput()
{
    int j;
    for(j=i;j<l;j++)
        printf("%c",*(input+j));
}
void main()
{
    int j;
    input=(char*)malloc(50*sizeof(char));
    printf("\nEnter the string\n");
    scanf("%s",input);
    input=strcat(input,"$");
    l=strlen(input);
    strcpy(stack,"$");
    printf("\nSTACK\tINPUT\tACTION");
    while(i<=l)
    {
        shift();
        printf("\n");
        dispstack();
        printf("\t");
        dispinput();
    }
}

```

```
printf("\tShift");
if(prec[getIndex(stack[top])][getIndex(input[i])]=='>')
{
while(reduce())
{
printf("\n");
dispstack();
printf("\t");
dispinput();
printf("\tReduced: E->%s",lasthandle);
}
}
}
if(strcmp(stack,"$E$")==0)
printf("\nAccepted;");
else
printf("\nNot Accepted;");
}
```



## Output

```
C:\Windows\System32\cmd.exe
C:\Users\Sharath Shankaran\Desktop\flex_code\Program 11>a.exe

Enter the string
(i+i)*i

STACK   INPUT   ACTION
$(      i+i)*i$ Shift
$(i     +i)*i$ Shift
$(E     +i)*i$ Reduced: E->i
$(E+    i)*i$ Shift
$(E+i   )*i$ Shift
$(E+E   )*i$ Reduced: E->i
$(E     )*i$ Reduced: E->E+E
$(E)    *i$ Shift
$E      *i$ Reduced: E->)E(
$E*     i$ Shift
$E*i    $ Shift
$E*E    $ Reduced: E->i
$E      $ Reduced: E->E*E
$E$     Shift
$E$     Shift
Accepted;
```