

**TOPIC – IMPLEMENTATION OF TOY COMPILER**

**PRESENTED BY**

1. **ROHAN BHOWMICK(15BCE1032)**
2. **SURYANSH BHARDWAJ(15BCE1047)**
3. **RITWIK KALA(15BCE1114)**
4. **YASHASHVI ASTHANA(15BCE1161)**

**ABSTRACT**

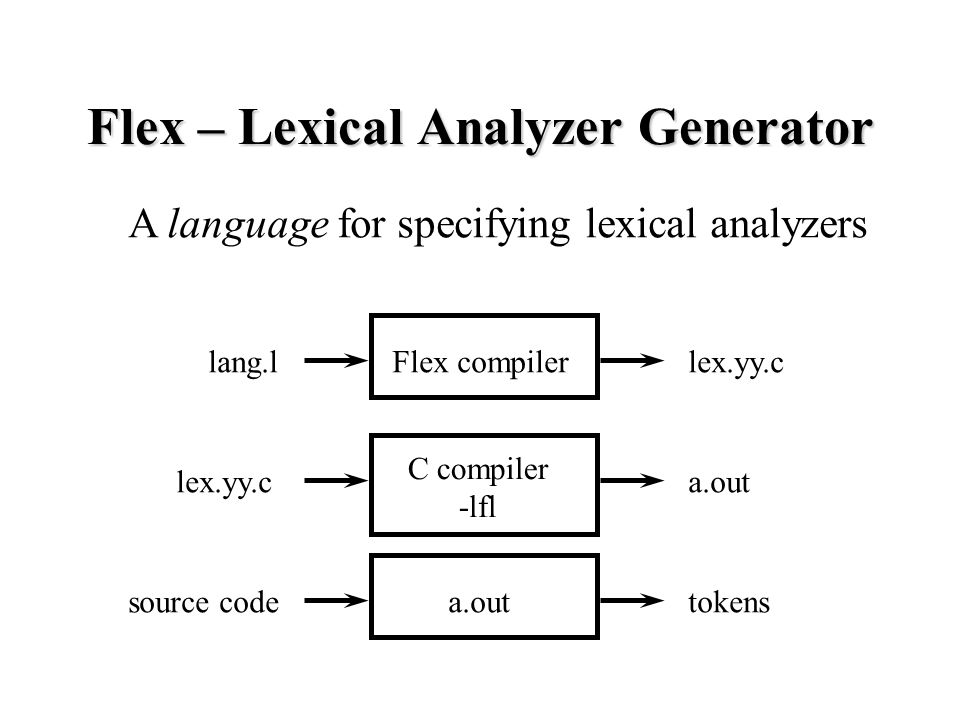
**The main aim of our project is to create a compiler which performs basic mathematical operations like addition, subtraction, finding sum of n numbers, factorial of a number , fibonacci series etc.**

**It consists of 2 files**

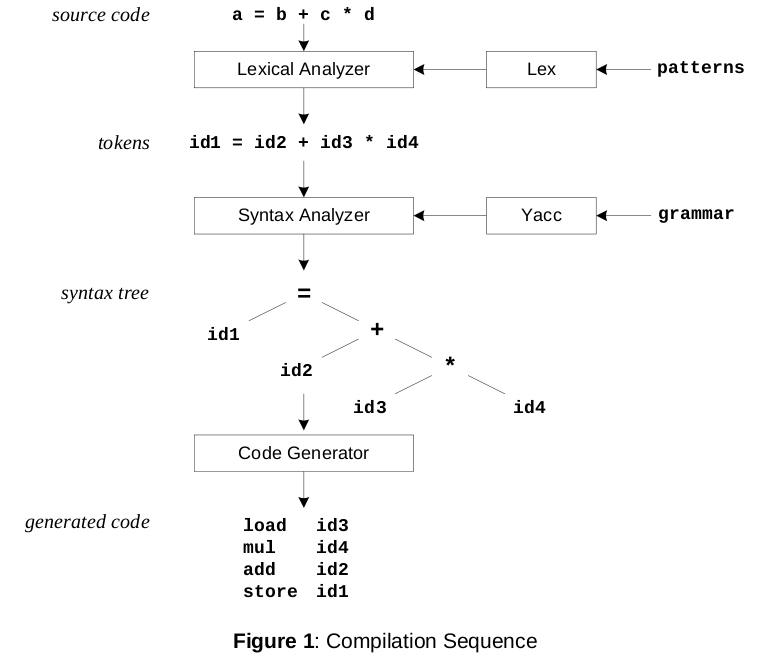
1. **Lex file –** it is used to identify the keywords and user defined variables and define a token for each of them
2. **Bison file –** it is mainly used to define the grammar of our language. We have used c++ to define the mathematical operation

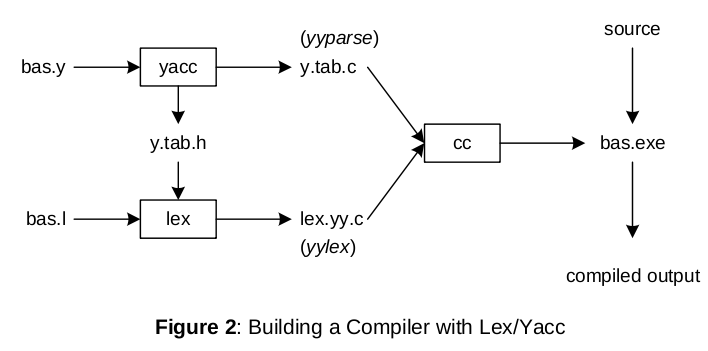
**MODULES**

1. **LEXICAL ANALYSIS USING FLEX -** The first part of that process is often called lexical analysis, particularly for such languages as C. A good tool for creating lexical analyzers is flex. It takes a specification file and creates an analyzer, usually called lex.yy.c.



1. **YACC** - yet another compiler compiler Parses and does semantic processing on the stream of tokens produced by lex.

****

****

**YACC FILES**

First Part includes

1. C declarations

2. Yacc declarations for example,

3. %start

4. %token

5. %union

6. %type

**Second Part includes**

1. Production rules

2. 2 sides- left and right side

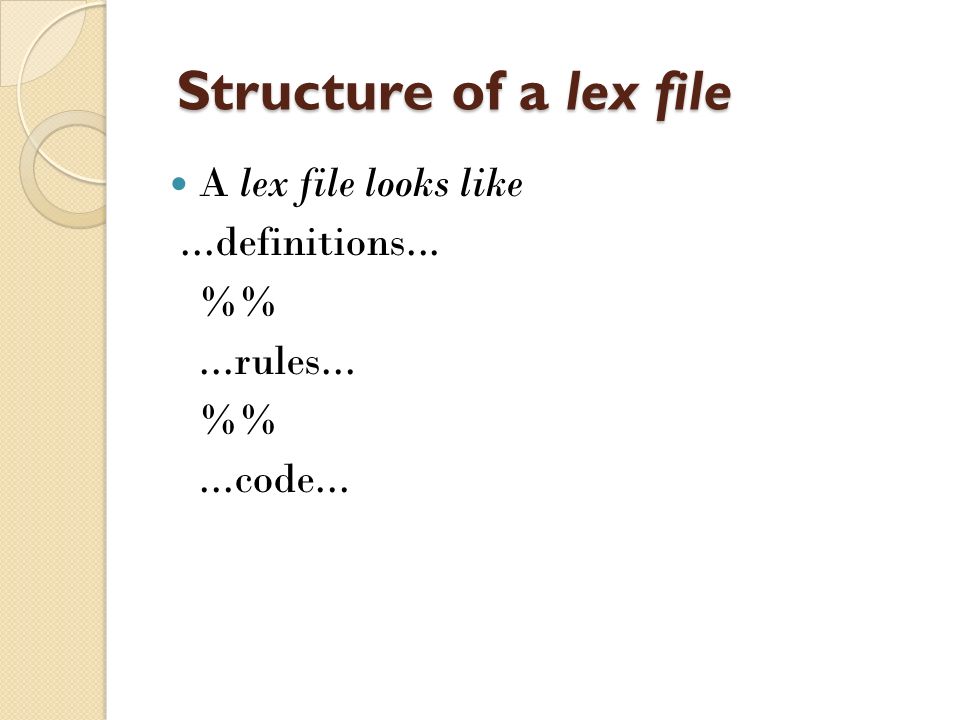
3. Left hand side is followed by a colon

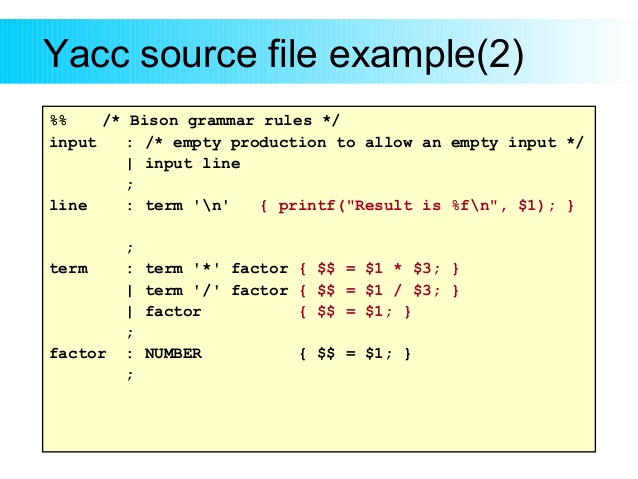
4. Multiple right hand sides may follow separated by a ‘|’

5. Actions associated with a rule are entered in braces

6. $1,$2,….$n can refer to the values associated with symbols

7. $$ refers to the value at left





**FOR OUR LANGUAGE WE HAVE DEFINED A LEX FILE WHICH CONTAINS THE TOKENS WHICH ARE ALL NEEDED TO PERFORM THE MATHEMATICAL OPERATION**

%{

#include "y.tab.h"

%}

%%

"print" {return print;}

"sum" {return sum;}

"fib" {return fib;}

"exit" {return exit\_command;}

[a-zA-Z] {yylval.id = yytext[0]; return identifier;}

[0-9]+(\.[0-9][0-9]?)? {yylval.num = atoi(yytext); return number;}

[ \t\n] ;

[-?!&/\*+=;] {return yytext[0];}

. {ECHO; yyerror ("unexpected character");}

%%

int yywrap (void) {return 1;}

**WE HAVE CREATED A BISON FILE WHICH CONTAINS MY GRAMMAR AND ALL THE MATHEMATICAL FUNCTIONS**

%{

void yyerror (char \*s); /\*parser is going to call when there is some error \*/

#include <suryansh.h>

int symbols[52];

int symbolVal(char symbol); /\* reads the value of symbol \*/

void updateSymbolVal(char symbol,int val); /\* updates the value of symbol \*/

%}

%union {int num; char id;} /\* Yacc definitions \*/ /\* different types my analyzer can return \*/

%start line /\* which production is my starting rule \*/

%token print /\* print token is present \*/

%token sum

%token fib

%token exit\_command

%token <num> number

%token <id> identifier

%type <num> line exp term

%type <id> assignment

%%

/\* descriptions of expected inputs corresponding actions (in C) \*/

line : assignment ';' {;}

| exit\_command ';' {exit(EXIT\_SUCCESS);}

| print exp ';' {printf("Printing %d\n", $2);}

| sum exp ';' {printf("Sum is %d\n",SUM($2));}

| fib exp ';' {int c,i=0;for ( c = 1 ; c <= $2 ; c++ )

{

printf("Printing Fibonacci series:%d\n", Fibonacci(i));

i++;

}}

| line assignment ';' {;}

| line print exp ';' {printf("Printing %d\n", $3);}

| line exit\_command ';' {exit(EXIT\_SUCCESS);}

| line sum exp ';' {printf("Sum is %d\n",SUM($3));}

| line fib exp ';' {int c,i=0;for ( c = 1 ; c <= $3 ; c++ )

{

printf("Printing Fibonacci series:%d\n", Fibonacci(i));

i++;

}}

;

assignment : identifier '=' exp {updateSymbolVal($1,$3); }

;

exp : term {$$ = $1;}

| exp '+' term {$$ = $1 + $3;}

| exp '-' term {$$ = $1 - $3;}

| exp '\*' term {$$ = $1 \* $3;}

| exp '/' term {$$ = $1 / $3;}

| exp '&' term {$$ = pow($1,$3);} /\*Power \*/

| exp '?' {$$ = sqrt($1);} /\*Square Root \*/

| exp '!' {$$ = fact($1);}

;

term : number {$$ = $1;}

| identifier {$$ = symbolVal($1);}

;

%% /\* C code \*/

int computeSymbolIndex(char token)

{

int idx = -1;

if(islower(token)) {

idx = token - 'a' + 26;

} else if(isupper(token)) {

idx = token - 'A';

}

return idx;

}

/\* returns the value of a given symbol \*/

int symbolVal(char symbol)

{

int bucket = computeSymbolIndex(symbol);

return symbols[bucket];

}

/\* updates the value of a given symbol \*/

void updateSymbolVal(char symbol,int val)

{

int bucket = computeSymbolIndex(symbol);

symbols[bucket] = val;

}

/\*Function for factorial\*/

int fact(int num){

if(num == 0 || num == 1){ return 1;}

else {

return(num \* fact(num - 1));

}

}

/\*Sum of n numbers\*/

int SUM(int num)

{

int i,su=0;

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

{

su = su + i;

}

return su;

}

/\*Function for fibonacci\*/

int Fibonacci(int n)

{

if ( n == 0 )

return 0;

else if ( n == 1 )

return 1;

else

return ( Fibonacci(n-1) + Fibonacci(n-2) );

}

int main (void) {

/\* initialize symbol table \*/

int i;

for(i=0; i<52; i++) {

symbols[i] = 0;

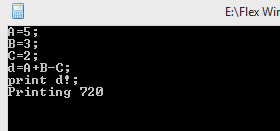
}

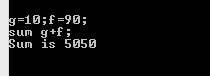
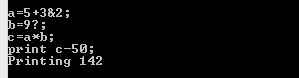
return yyparse ( );

}

void yyerror (char \*s) {fprintf (stderr, "%s\n", s);}

**OUTPUT**

****

****