PROJECT REPORT

COMPILER DESIGN

COURSE PROJECT

HybridCalC

Under the Guidance of

Prof. Manish Kamboj
Assistant Professor, CSE
Punjab Engineering College, Chandigarh

Submitted By

Lovedeep Singh, SID-16103104
Kanishk Gautam, SID-16103118

Department of Computer Science and Engineering
Punjab Engineering College, Chandigarh
(Deemed University)

DECLARATION

We hereby declare that the project work enclosed is an authentic record of our own work carried as requirements of the course project of Compiler Design under the guidance Dr. Manish Kamboj, Assistant Professor, CSE during Jan 2020 to May 2020.

Date: ________________

Certified that the above statement made by the student is correct to the best of our knowledge and belief.

Dr. Manish Kamboj
Assistant Professor
Computer Science and Engineering
Punjab Engineering College, Chandigarh

ACKNOWLEDGEMENT

Training is an agglomeration of theoretical, practical and technical concepts that enhances our skills in the field of technology. Training under renowned and knowledgeable mentors can yield prolific results wherein the cachet and technical skills are imparted.

We would like to express our deep sense of gratitude towards Dr. Manish Kamboj for providing us an opportunity to work in a challenging project.

We thank profusely all the colleagues for their kind help, friendly nature, timely suggestions and co-operation throughout the first phase of our Major Project.

(Lovedeep Singh, SID-16103104

Kanishk Gautam, SID-16103118)

ABSTRACT

Programming is an old discipline in Computer Science that has primitive importance. Today, we have come forward with a large number of programming languages each evolving from the experiences of past programmers and earlier languages. In older times, we used to code directly in Assembly language and Assembler would convert that into machine language that was then fed to the appropriate hardware to get the desired output. Today, we have added another layer on the top of Assembler that is of Compiler. Now we code in English like languages and it is then the job of the compiler to convert this High Level Language to lower level Assembly language.

S No.	Table of Contents	Page No.
I	List of Figures	6
II	List of Tables	7
Chapter 1	Basics	8
	1.1 Scanner (Lexical Analyzer)	8
	1.2 Parser (Syntax Analysis)	9
	1.3 Semantic Analysis and Code Generation	9
	1.4 Machine Independent Code improvement (optional)	9
	1.5 Target Code Generation	9
	1.6 Machine-specific code improvement (optional)	9
	1.7 Compiler	10
Chapter 2	Work	12
	2.1 Lex	12
	2.2 YACC	15
	2.3 Codes and output	15
	2.4 Hybrid CalC	25
	2.4.1 HybridCalCL – the lex file	25
	2.4.2 HybridCalCG – the yacc file	27
	2.4.3 Other phases	30
	2.4.4 Building and Running the HybridCalC	31
3	References	33

Chapter-1 BASICS

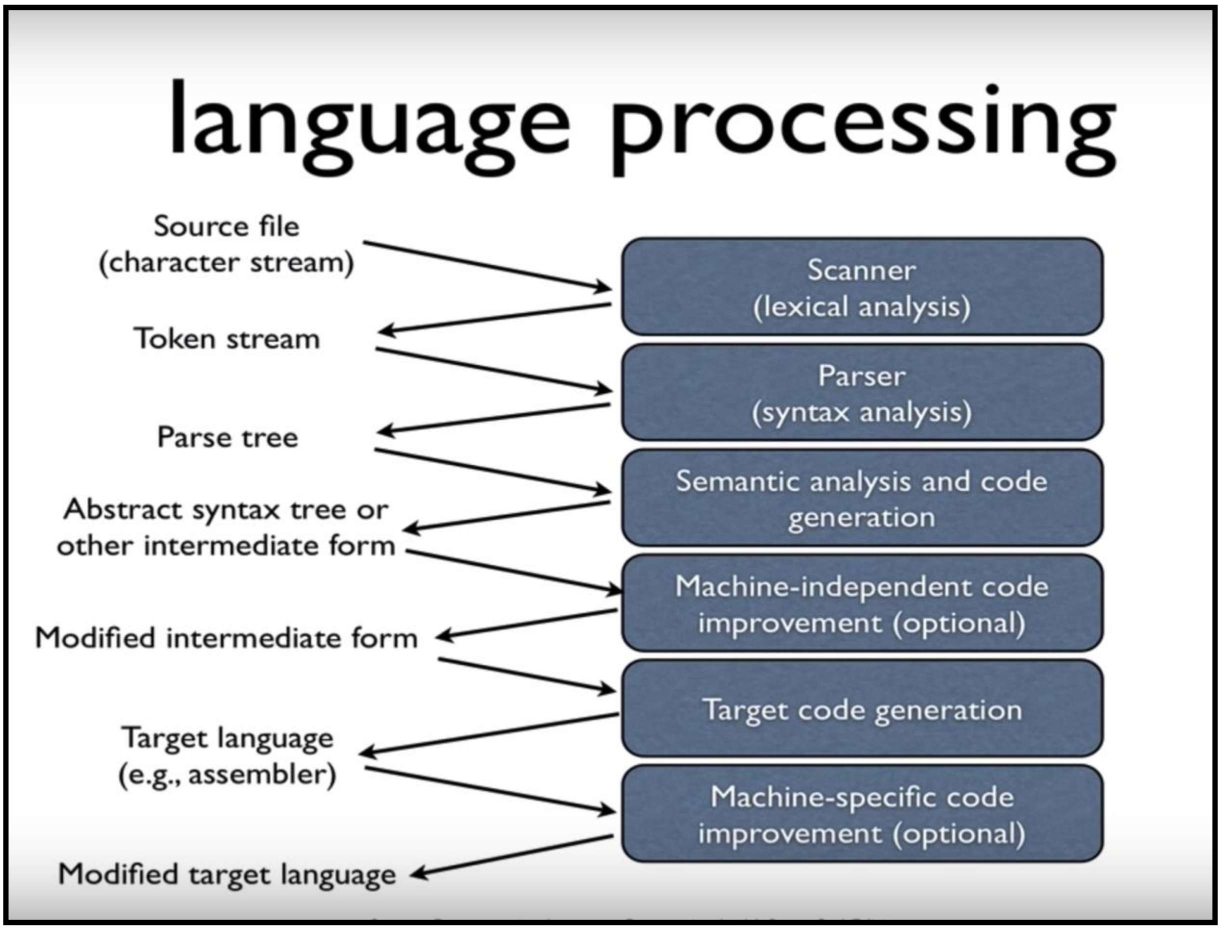

Figure 1.1 Language Processing Stack

Machines are comfortable with binary; humans are comfortable with languages like English. This makes a gap in between that is filled through a language processing unit or the language processor.

Here we will briefly discuss the details of each module and discuss the open source implementations of these if available.

1.1 Scanner (Lexical Analyzer)

This is the first phase in the language processing stack. Here, the input source code is split up into a number of tokens based upon some predefined rules. Regular expressions are used extensively to define the tokens. Flex is a unix open source facility for implementing a lexical analyser.

1.2 Parser (Syntax Analysis)

This is the second phase in the language processing stack. Here, the Parser takes stream of tokens as input and builds a parse tree. It syntactically verifies the input and adds information regarding attributes like type, scope, dimension, line of

referent and line of use to symbol table.

1.3 Semantic Analysis and Code Generation

It will verify whether the parse tree is meaningful or not. It uses available information to check for the semantics. The next part is also called as intermediate code generation phase. It will take the parse tree and generate three address code. There are various intermediate codes and three address code is a very popular one

1.4 Machine Independent Code improvement (optional)

The output of the intermediate code generation is given to code optimiser. It makes the program less computation intensive by eliminating redundant computations and reduces the size of the program.

1.5 Target Code Generation

It takes the output from the code optimiser as input and generated the assembly code.

1.6 Machine-specific code improvement (optional)

It is an optional phase; we can carry out further optimizations to best suit the assembler. These optimizations are specific to the machine.

1.7 Compiler

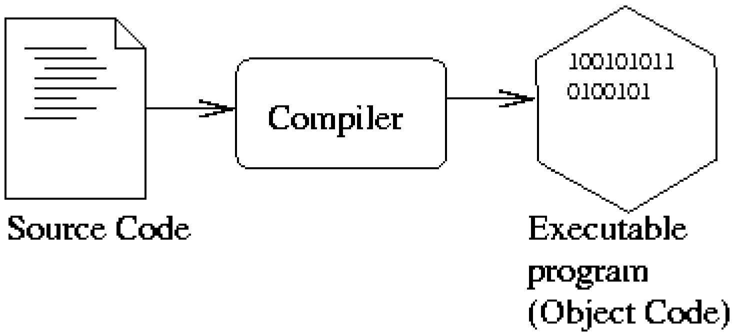

Figure 1.2 Compiler High Level Design

Compiler: A compiler is a computer program that translates computer code written in one programming language into another language. The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language to create an executable program. (Wikipedia)

Table 1.1: Phases in a Compiler

Phase	Role
Lexical Analysis	Tokenises the given program into a stream of tokens. Creates new entries for each new identifier in Symbol Table. We will use Lex to build it.
Syntax Analysis	The stream of tokens will be given to syntax analyser, it is also called as Parser. It will take the tokens and convert it into a parse tree. It adds information regarding attributes like type, scope, dimension, line of referent and line of use to symbol table. We will use Yacc to build this.
Semantic Analysis	It will verify whether the parse tree is meaningful or not. It uses available information to check for the semantics.
Intermediate Code Generation	It will take the parse tree and generate three address code. There are various intermediate codes and three address code is a very popular one.
Code Optimisation	The output of the intermediate code generation is given to code optimiser. It makes the program less computation intensive by eliminating redundant computations and reduces the size of the program.
Target Code Generation	This is the final phase. It takes the output from the code optimiser as input and generated the assembly code.

Here in this project, we experiment with the subset of C-language. The compiler will be fed with an input made with a subset of C language. The compiler will translate the input into an equivalent program in some other language. It will also detect syntax errors during the complete process. The input is generally called as the HLL (High Level Language) and the output as the Assembly Language. The first four phases are constant across different compilers and the last two phases can be changed to make a compiler for a different assembly environment. The first four phase are also called as the front end or the Analysis phase and the last two phases are called as the back end or the Synthesis phase.

In this chapter we discussed about the basics of Language Processing Stack, discussing brief details of each phase and its available open source implementations if any.

In the next chapter, we discuss the present work done till now.

<u>**Chapter-2 WORK**</u>

At present, we are working on LEX and YACC. These play part in the first two phases of the language processing stack.

2.1 LEX

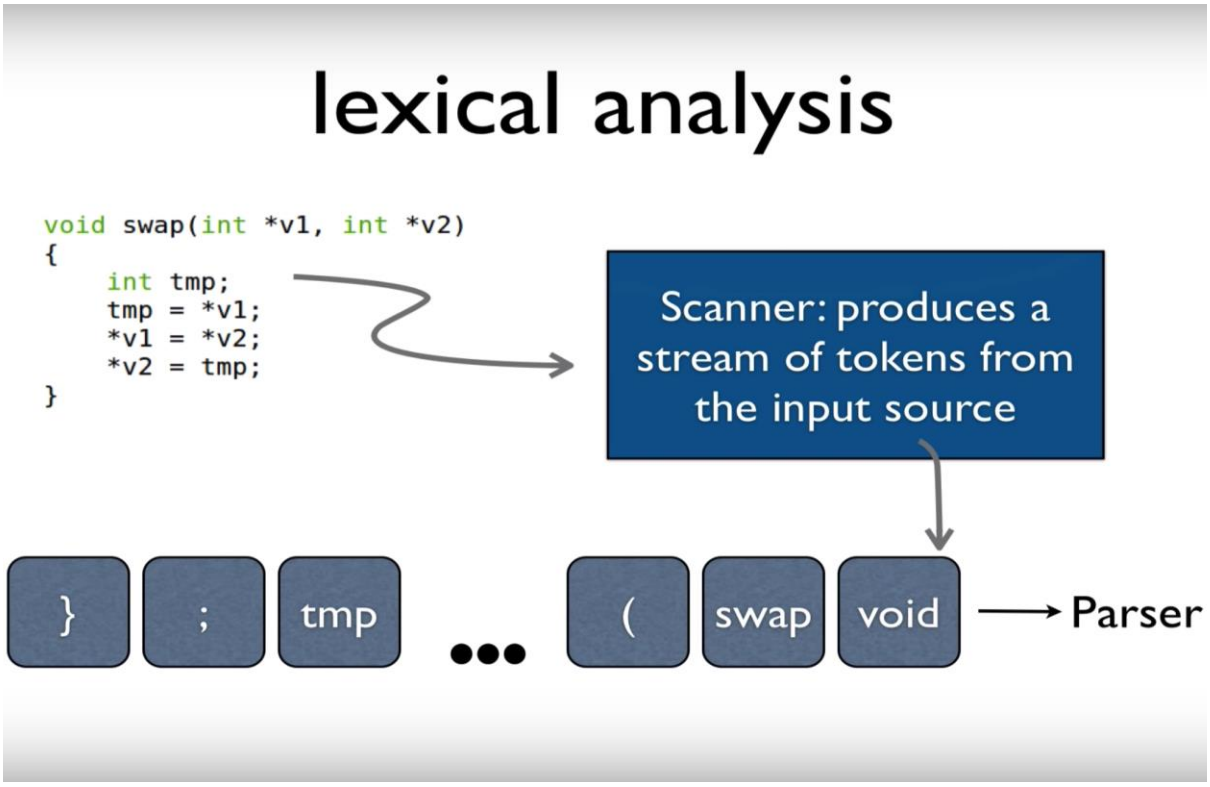

Figure 2.1 Lexical Analysis Illustration

LEX is a computer software to implement the Lexical analysis phase in the language processing stack. It read the code character by character and looks for identifiers: letter(letter|digit)*

Lex is a scanner generator. The output is a table-driven scanner in lex.yy.c. The input is a set of regular expressions and associated actions(written in C).

Flex is an open source implementation of the original UNIX lex utility.

Note the following symbols:

- Repetition is expressed by *
- Alternation is expressed by |
- Concatenation as it is

Regular Expressions: It is an important manner by which we can specify patterns in a subtle and short notation. We use the following symbols in the regular expressions: [a-z], [A-Z], [0-9], [+|-].

Table 2.1: Pattern Matching Primitives

<table>
<tr><th>Metacharacter</th><th>Matches</th></tr>
<tr><td>.</td><td>any character except newline</td></tr>
<tr><td>\n</td><td>newline</td></tr>
<tr><td>*</td><td>zero or more copies of the preceding expression</td></tr>
<tr><td>+</td><td>one or more copies of the preceding expression</td></tr>
<tr><td>?</td><td>zero or one copy of the preceding expression</td></tr>
<tr><td>^</td><td>beginning of line</td></tr>
<tr><td>$</td><td>end of line</td></tr>
<tr><td>a|b</td><td>a or b</td></tr>
<tr><td>(ab)+</td><td>one or more copies of ab (grouping)</td></tr>
<tr><td>"a+b"</td><td>literal "a+b" (C escapes still work)</td></tr>
<tr><td>[]</td><td>character class</td></tr>
</table>

Table 2.2: Pattern Matching Examples

<table>
<tr><th>Expression</th><th>Matches</th></tr>
<tr><td>abc</td><td>abc</td></tr>
<tr><td>abc*</td><td>ab abc abcc abccc ...</td></tr>
<tr><td>abc+</td><td>abc abcc abccc ...</td></tr>
<tr><td>a(bc)+</td><td>abc abcbc abcbcbc ...</td></tr>
<tr><td>a(bc)?</td><td>a abc</td></tr>
<tr><td>[abc]</td><td>one of: a, b, c</td></tr>
<tr><td>[a-z]</td><td>any letter, a-z</td></tr>
<tr><td>[a\-z]</td><td>one of: a, -, z</td></tr>
<tr><td>[-az]</td><td>one of: -, a, z</td></tr>
<tr><td>[A-Za-z0-9]+</td><td>one or more alphanumeric characters</td></tr>
<tr><td>[\t\n]+</td><td>whitespace</td></tr>
<tr><td>[^ab]</td><td>anything except: a, b</td></tr>
<tr><td>[a^b]</td><td>one of: a, ^, b</td></tr>
<tr><td>[a|b]</td><td>one of: a, |, b</td></tr>
<tr><td>a|b</td><td>one of: a, b</td></tr>
</table>

These are used extensively in preparing the Lex file. The Lex file has l as extension. The syntax for Lex file goes as follows:

Notes: Two special operators are hyphen ("-") and circumflex ("^"). Hyphen is used for range of characters. The circumflex negates the expression. In case of a clash, longest matching pattern wins. In case of equal lengths, first matching pattern wins.

Lex input format:

```
FIRST SECTION ... definitions ... (optional)

%%

 Pattern matching rules ... rules ... (consist of pattern and action)

 %%

 THIRD SECTION ... subroutines ... (optional)
```

So, it contains three sections with %% being the separator between the sections.

Table 2.3: Lex Predefined variables

Name	Function
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start condition
ECHO	write matched string

There must be a whitespace between associated terms and defining expressions. Whenever there is a match, the associated C code in the action part is executed.

2.2 YACC

YACC stands for yet another compiler compiler. In YACC, the Grammars for yacc are described using a variant of Backus Naur Form (BNF). A BNF grammar is used to express context-free languages. BNF can represent most constructs in Modern Programming Languages.

Although we avoid ambiguous grammars, yacc takes default action in case of a conflict. The conflict can be either shift-reduce or reduce-reduce. Yacc shifts for shift-reduce conflict and it uses the first rule for reduce-reduce conflict. A warning is issued whenever there is a conflict. These warnings can be avoided by making the grammar unambiguous.

```
FIRST SECTION ... definitions ... (optional)

%%

 Pattern matching rules ... rules ... (consist of pattern and action)

 %%

 THIRD SECTION ... subroutines ... (optional)
```

Input to yacc is divided into three sections. The C code is contained in a section enclosed by "%{" and "%}".

2.3 Codes and Output

1. Simplest scanner sc1.l

```
%%
"hello world"              printf("GOODBYE!\n');
.                    ;
%%
```

We can see how this works, whenever it encounters the string "hello world", this will simply display GOODBYE. For any other character it simply skips it and does nothing.

```
$ lex sc.l
$ cc lex.yy.c -ll
```

```
$ ./a.out
hello world
GOODBYE!
$
```

2. Example 2 scanner sc2.l

```
%%
“hi”                  printf(“HI THERE!\n’);

“hello”               printf(“HELLO WORLD!\n’);

.                     ;
%%
```

In this one, as the scanner works from top to bottom, if it encounters “hi”, it is going to display “HI THERE!”, if it encounters “hello”, it will output “HELLO WORLD!” on the terminal screen. If it is neither of the above two cases, it will bypass the character.

Lets see it run

```
$ lex sc.l
$ cc lex.yy.c -ll
$ ./a.out
hello
HELLO WORLD!
$
```

3. Example 3 sc3.l

```
%%
.
\n
```

This is another trivial example. In this as we can see it simply matches all the program character by character and skips everything including new line character. It simply scans all the input and produces no output to display.

4. Example 4 sc4.l

```
%{
int yylineno;
 %}
%%
^(.*)\n             printf("%4d\t%s", ++yylineno, yytext);
%%
 int main(int argc, char *argv[])
{
yyin = fopen(argv[1], "r");
yylex(); fclose(yyin);
 }
```

This is a classic example which adds the number of the line in the beginning of each line.

5. We made a simple calculator using lex and yacc. The code is found below

calc.l

```
%{

#include "y.tab.h"
void yyerror (char *s);
int yylex();
%}
%%
"print"                         {return print;}
"exit"                          {return exit_command;}
[a-zA-Z]     {
               yylval.id = yytext[0];
               return identifier;
               }
[0-9]+           {
               yylval.num = atoi(yytext);
               return number;
               }
[ \t\n]          ;// do nothing
[-+=;]         {
               return yytext[0];
               }
.              {ECHO; // other than above, things if we get anything
               yyerror("this character is not expected");
               }
%%
int yywrap (void) {return 1;}
```

calc.y

```
%{
void yyerror (char *s);
int yylex();
#include <stdio.h>     /* C declarations used in actions */
#include <stdlib.h>
#include <ctype.h>
int symbols[52];
int symbolVal(char symbol);
void updateSymbolVal(char symbol, int val);
%}

%union {int num; char id;}        /* Yacc definitions */
%start line
%token print
%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("we have, printing the value
%c=%d\n", $$, $2);}
                | line assignment ';'     {;}
                | line print exp ';' {printf("we have, printing the value %c=%d\n", $$, $3);}
                | line exit_command ';'  {exit(EXIT_SUCCESS);}
        ;

assignment : identifier '=' exp  { updateSymbolVal($1,$3); }
                        ;
exp     : term              {$$ = $1;}
        | exp '+' term       {$$ = $1 + $3;}
        | exp '-' term       {$$ = $1 - $3;}
        ;
term            : number            {$$ = $1;}
                | identifier                {$$ = symbolVal($1);}
        ;

%%              /* C code */

/* this is simple calculation of the symbol table index */
int computeSymbolIndex(char token)
{
        int idx = -1;
        if(islower(token)) {
                idx = token - 'a' + 26;
        } else if(isupper(token)) {
                idx = token - 'A';
        }
        return idx;
}

/* this function will simply return the value of the passed symbol */
int symbolVal(char symbol)
{
        int bucket = computeSymbolIndex(symbol);
        return symbols[bucket];
}
```

```
/* sets the value in the symbol table */
void updateSymbolVal(char symbol, int val)
{
        int bucket = computeSymbolIndex(symbol);
        symbols[bucket] = val;
}

int main (void) {
        /* here we first initialize the symbol table */
        int i;
        for(i=0; i<52; i++) {
                symbols[i] = 0;
        }

        return yyparse ( );
}

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

Here clearly, we used an array for symbol table with a capacity of 52, so we cannot have more than 52 variables in consideration at a time.

Now what happens is depicted in the figure below

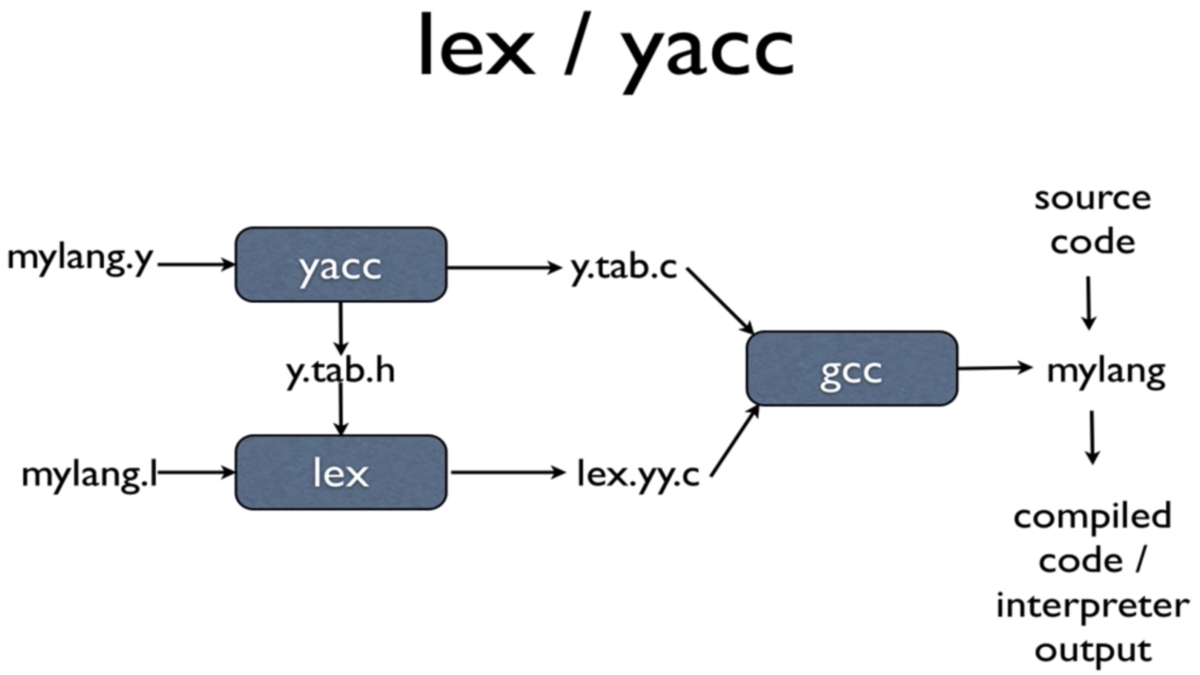

Figure 2.2 Lex and Yacc working together

The utility itself generates ytab.h and ytab.c

MAKEFILE

```
calc: lex.yy.c y.tab.c
	gcc -g lex.yy.c y.tab.c -o calc

lex.yy.c: y.tab.c calc.l
	lex calc.l

y.tab.c: calc.y
	yacc -d calc.y

clean:
	rm -rf lex.yy.c y.tab.c y.tab.h calc calc.dSYM
```

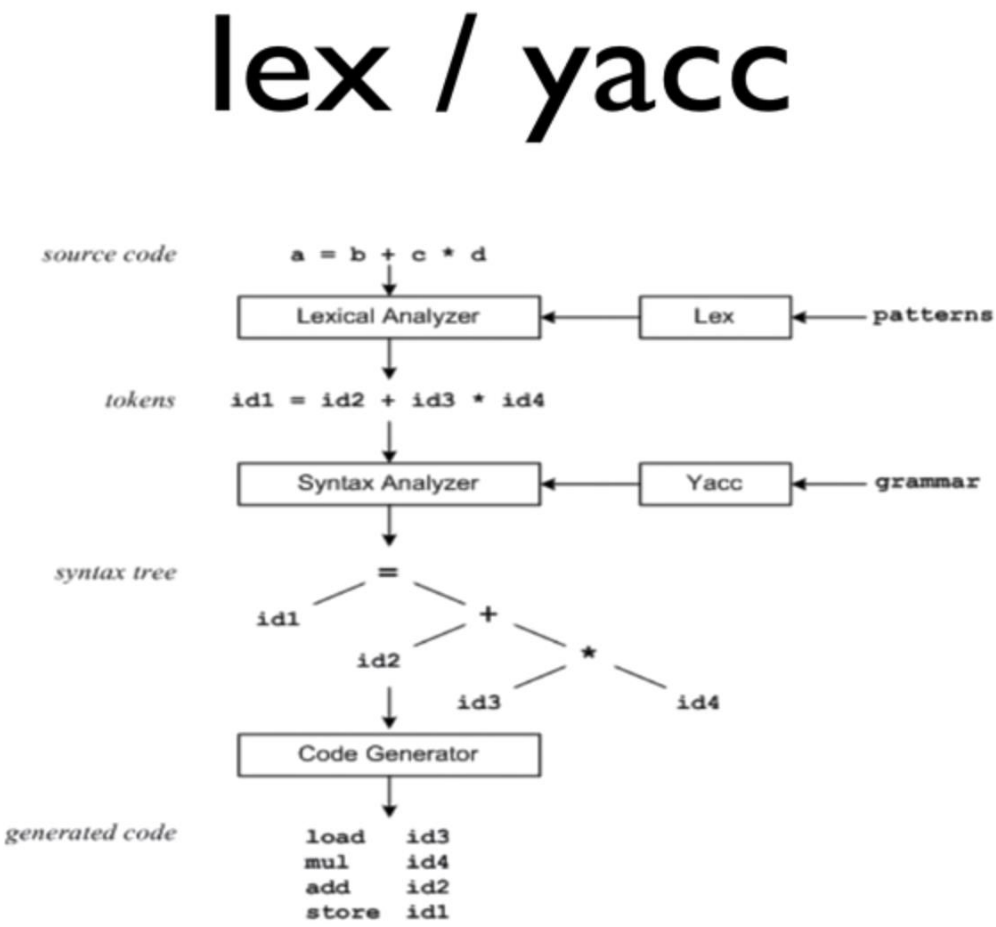

Figure 2.3 How it works

SAMPLE RUN SCREENSHOTS ARE SHOWN BELOW:

```
(base) Lovedeeps-MacBook-Pro:yacc-tutorial-master lovedeepsingh$ lex calc.l
(base) Lovedeeps-MacBook-Pro:yacc-tutorial-master lovedeepsingh$ yacc -d calc.y
(base) Lovedeeps-MacBook-Pro:yacc-tutorial-master lovedeepsingh$ gcc lex.yy.c y.
tab.c -o calc
(base) Lovedeeps-MacBook-Pro:yacc-tutorial-master lovedeepsingh$ ./calc
a=234;
print a;
the value of a is 234
a=3+7;
print a;
the value of a is 10
a=a+19;
print a;
the value of a is 29
```

Figure 2.4 Successful working on + operator

```
a=2;
b=3;c=7;
a=a+b+c;
print a;
the value of a is 12
a=a-2;
print a;
the value of a is 10
a=265-234;
print a;
the value of a is 31
exit;
```

Figure 2.5 Successful working on - operator

```
a=234;
b=234;
c=a+b;
c=c*2;
this character is not expected as is not part of the language
syntax error
```

Figure 2.6 Since * is not defined and we had an error message, that error message is being shown

```
(base) Lovedeeps-MacBook-Pro:yacc-tutorial-master lovedeepsingh$ ./calc
aa=23;
syntax error
```

Figure 2.7 We used [a-zA-Z] that allows only single alphabets as variables.

```
a=b+C+D;
print a;
the value of a is 0
exit;
```

Figure 2.8 By default, as we defined in our yacc, value assigned to variable is 0.

2.4 HybridCalC

HybridCalC is a hybrid calculator capable of doing all the equations as done by a modern scientific calculator. It is designed using lex and yacc. After generating the appropriate files, it is compiled using gcc and the output is given a.out. We can run a.out to simulate the calculator and perform calculations on the same. It supports the following functions:

<table>
<tr><td>Addition</td><td>+</td><td>Binary</td><td>Unary</td></tr>
<tr><td>Subtraction</td><td>-</td><td>Binary</td><td>Unary</td></tr>
<tr><td>Increment</td><td>++</td><td>Not Binary</td><td>Unary, pre and post</td></tr>
<tr><td>Decrement</td><td>--</td><td>Not Binary</td><td>Unary, pre and post</td></tr>
<tr><td>Logical ‘OR’</td><td>||</td><td>Binary</td><td>Not Unary</td></tr>
<tr><td>Logical ‘AND’</td><td>&&</td><td>Binary</td><td>Not Unary</td></tr>
<tr><td>Bitwise ‘OR’</td><td>|</td><td>Binary</td><td>Not Unary</td></tr>
<tr><td>Bitwise ‘AND’</td><td>&</td><td>Binary</td><td>Not Unary</td></tr>
<tr><td>Left Shift</td><td><<</td><td>Not Binary</td><td>Unary</td></tr>
<tr><td>Right Shift</td><td>>></td><td>Not Binary</td><td>Unary</td></tr>
<tr><td>Multiply</td><td>*</td><td>Binary</td><td>Not Unary</td></tr>
<tr><td>Divide</td><td>/</td><td>Binary</td><td>Not Unary</td></tr>
<tr><td>Bitwise XOR</td><td>^^</td><td>Binary</td><td>Not Unary</td></tr>
<tr><td>Logarithm</td><td>log()</td><td>Not Binary</td><td>Unary</td></tr>
<tr><td>Sine</td><td>sin()</td><td>Not Binary</td><td>Unary</td></tr>
<tr><td>Cosine</td><td>cos()</td><td>Not Binary</td><td>Unary</td></tr>
<tr><td>Tangent</td><td>tan()</td><td>Not Binary</td><td>Unary</td></tr>
<tr><td>ArcSine</td><td>asin()</td><td>Not Binary</td><td>Unary</td></tr>
<tr><td>ArcCosine</td><td>acos()</td><td>Not Binary</td><td>Unary</td></tr>
</table>

ArcTangent	atan()	Not Binary	Unary
HyperbolicSine	sinh()	Not Binary	Unary
HyperbolicCosine	cosh()	Not Binary	Unary
HyperbolicTangent	tanh()	Not Binary	Unary
Ceiling	ceil()	Not Binary	Unary
Floor	floor()	Not Binary	Unary
Absolute	abs()	Not Binary	Unary
SquareRoot	sqrt()	Not Binary	Unary
Factorial	fact()	Not Binary	Unary
ConversionToDecimal	bin_to_dec()	Not Binary	Unary
π = 3.14	pi	Not Binary	Not Unary
RemainderModuli	%	Binary	Not Unary

Table 2.4 Functions supported by HybridCalC

2.4.1 HybridCalCL – the lex file

```
value [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ \t]   { ; }
log10     return sys_log;
fact      return sys_fact;
bin_to_dec   return sys_bin_to_dec;
pi       return sys_pi;
sin      return sys_sin;
cos      return sys_cos;
tan      return sys_tan;
sinh     return sys_sinh;
cosh     return sys_cosh;
tanh     return sys_tanh;
asin     return sys_asin;
acos     return sys_acos;
atan     return sys_atan;
xor      return sys_xor;
```

```
or      return sys_or;
ceil    return sys_ceil;
floor   return sys_floor;
abs     return sys_abs;
{value}   { yylval=atof(yytext);return sys_value; }
"<<"    return sys_leftshift;
">>"    return sys_rightshift;
"++"    return sys_inc;
"--"    return sys_dec;
"+"     return sys_plus;
"-"     return sys_minus;
"~"     return sys_unaryminus;
"/"     return sys_div;
"*"     return sys_mul;
"^"     return sys_pow;
sqrt    return sys_sqrt;
"("     return sys_openbracket;
")"     return sys_closebracket;
"%"     return sys_mod;
"^^"    return sys_xor;
"="     return sys_assign;
"&&"    return sys_land;
"||"    return sys_or;
"|"     return sys_ior;
"&"     return sys_land;
\n|. {return yytext[0];}
```

2.4.2 HybridCalCG – the yacc file

```
%{

#include <stddef.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include<stdio.h>
#include <alloca.h>
#define YYSTYPE double
long int bin_to_dec(long int val){
    long int left,s=0,pw=0;
    while(val>0){
         left = val%10;
         val/=10;
         s = s + left * pow(2,pw);
         pw++;
    }
    return s;
```

```
}
float fact(int v){
    float r = 1;
    int i;
    for (i = 1; i <= v; i++)
        r = r * i;
    return r;
}

%}

%token value sys_mod sys_rightshift sys_leftshift sys_pi
%token sys_plus sys_minus sys_div sys_mul sys_pow sys_sqrt
sys_openbracket sys_closebracket sys_unaryminus
%token sys_asin sys_acos sys_atan sys_sin sys_sinh sys_cos
sys_cosh sys_tan sys_tanh sys_inc sys_dec sys_land sys_or
sys_xor sys_assign sys_ior sys_and  sys_ceil sys_floor
sys_abs sys_fact sys_bin_to_dec
%left sys_plus sys_minus sys_mul sys_div sys_unaryminus
sys_land sys_or sys_xor sys_and sys_ior sys_log

%%

L    :    L E '\n'  { printf("%g\n", $2); }
     |    L '\n'
     |
     ;
E:   LOR
        ;
LOR: LAND
        | LOR sys_or LAND
          { $$ = (int) $1 || (int) $3; }
        ;
LAND: or
        | LAND sys_land or
          { $$ = (int) $1 && (int) $3; }
        ;
or: or1
        | or sys_ior or1
          { $$ = (int) $1 | (int) $3; }
        ;
or1: and
        | or1 sys_xor and
          { $$ = (int) $1 ^ (int) $3; }
        ;
and: shift
        | and sys_and shift
          { $$ = (int) $1 & (int) $3; }
        ;
shift: pow
        | shift sys_leftshift pow
```

```
            { $$ = (int) $1 << (int) $3; }
          | shift sys_rightshift pow
            { $$ = (int) $1 >>(int) $3; }

          ;
pow: add
          | pow sys_pow add { $$ = pow($1,$3); }
      | sys_sqrt sys_openbracket E sys_closebracket { $$ =
sqrt($3) ; }
          ;
add: mul
          | add sys_plus mul  { $$ = $1 + $3;}
          | add sys_minus mul { $$ = $1 - $3; }
          ;
mul: unary
          | mul sys_mul unary { $$ = $1 * $3; }
          | mul sys_div unary { $$ = $1 / $3; }
          | mul sys_mod unary { $$ = fmod($1,$3); }
          ;
unary: post
          | sys_minus primary %prec sys_unaryminus { $$ = -$2;
}
          | sys_inc unary { $$ = $2+1; }
          | sys_dec unary { $$ = $2-1; }
          | sys_log unary { $$ = log($2); }
          ;
post    : primary
          | post sys_inc { $$ = $1+1; }
          | post sys_dec { $$ = $1-1; }
          ;
primary:
           sys_pi { $$ = M_PI; }
          | sys_openbracket E sys_closebracket { $$ = $2; }
          | function
          ;
function: sys_sin sys_openbracket E sys_closebracket
                { $$ = (cos($3)*tan($3)); }
          | sys_cos sys_openbracket E sys_closebracket
                { $$ = cos($3); }
          | sys_sinh sys_openbracket E sys_closebracket
                { $$ = sinh($3); }
          | sys_asin sys_openbracket E sys_closebracket
                { $$ = asin($3); }
          | sys_acos sys_openbracket E sys_closebracket
                { $$ = acos($3); }
          | sys_atan sys_openbracket E sys_closebracket
                { $$ = atan($3);}
          | sys_tan sys_openbracket E sys_closebracket
                { $$ = tan($3);}
          | sys_cosh sys_openbracket E sys_closebracket
                { $$ = cosh($3);}
```

```
        | sys_tanh sys_openbracket E sys_closebracket
               { $$ = tanh($3);}
     | sys_ceil sys_openbracket E sys_closebracket
     { $$ = ceil($3);}
     | sys_floor sys_openbracket E sys_closebracket
          { $$ = floor($3);}
     | sys_abs sys_openbracket E sys_closebracket
          { $$ = abs($3);}
     | sys_fact sys_openbracket E sys_closebracket
          { $$ = fact((int)$3);}
     | sys_bin_to_dec sys_openbracket E sys_closebracket
          { $$ = bin_to_dec((float)$3);}
     | value
        ;

%%

#include <stdio.h>
#include <ctype.h>
#include "lex.yy.c"
#include <string.h>

char *progname;
yyerror( s )
char *s;
{
  warning( s , ( char * )0 );
  yyparse();
}
warning( s , t )
char *s , *t;
{
  fprintf( stderr ,"%s: %s\n" , progname , s );
  if ( t )
    fprintf( stderr , " %s\n" , t );
}
```

2.4.3 Other phases

The other phases are handled internally by the gcc. Once we compile the output file y.tab.c, gcc handles the internal phases of code generation, optimization and finally the machine level code is generated. The finally binary file is in a.out. We can run it using ./a.out in the same directory it is located from the terminal.

2.4.4 Building and Running the HybridCalC

It is very simple to build the HybridCalC and run it using the following commands step by step. These commands step by step have been written in run.sh file which can be run using a single command and internally runs these steps in order. We need to change the permissions for run.sh to make it executable, simply run chmod +x run.sh command from the terminal. The basics remain same which have been explained in the initial tutorials of this report.

run.sh file

```
lex HybridCalCL.l
yacc HybridCalCG.y
cc y.tab.c -ly -ll -lm
./a.out
```

```
2+3-1/2*22+12*3-321+22*234-sin(pi/2)-log(10000)
4846.79
2+3-22/2+22/2-44*3/2+21
-40
asin(sin(pi/2))
1.5708
log(sin(pi/2))
0
log(log(log(10^10^10)))
1.69363
log(100)
4.60517
log(2.718)
0.999896
```

Figure 2.9 HybridCalC simulation

```
tan(0)
0
tan(atan(1))/pi*4
1.27324
atan(tan(pi/4))/pi*4
1
2^3
8
log(2.718^22)
21.9977
1/2/3/4/5/6
0.00138889
1/2*2
1
1/2+1/2+1/3+1/3+1/3+1/4+1/2+1/2+1/4+1/2
4
1^222
1
1^2/2^1/2^1
1
```

Figure 2.10 HybridCalC simulation

```
2 ^^ 2
0
2 ^^ 2 ^^ 2 ^^ 1 ^^ 1 ^^ 23 ^^ 23 ^^ 23 ^^ 23 ^^ 122 ^^ 2
122
2 & 2 & 0
0
```

Figure 2.11 HybridCalC simulation

```
(2^23)/(2^20)
8
(1+2^3-3-3)*2/3*3/2*(234+2)/123
0.0710629
log(23)*tan(232)*sin(123)*asin(.5)*23*32/21*(12+2+2/2*3)
232.978
2|2|2|2|3|7|9|123|213
255
```

Figure 2.12 HybridCalC simulation

```
2323233 % 3
0
3333 % 3
0
21212 % 3
2
2 % 3*2*2 / (9*7*6)
0.021164
```

Figure 2.13 HybridCalC simulation

```
122 ^^ 212
174
pi * 3^2
88.8264
pi * 3 * 2/pi
6
```

Figure 2.14 HybridCalC simulation

References:

1. Wikipedia
2. COMPILER DESIGN IN c Allen I. Holub - Allen Holub
3. Principles of Compiler Design Textbook by Alfred Aho and Jeffrey Ullman
4. Tutorials point
5. Lex and Yacc tutorial by Tom Niemann Portland, Oregon
6. Youtube tutorials by Jonathon R.Elgesma