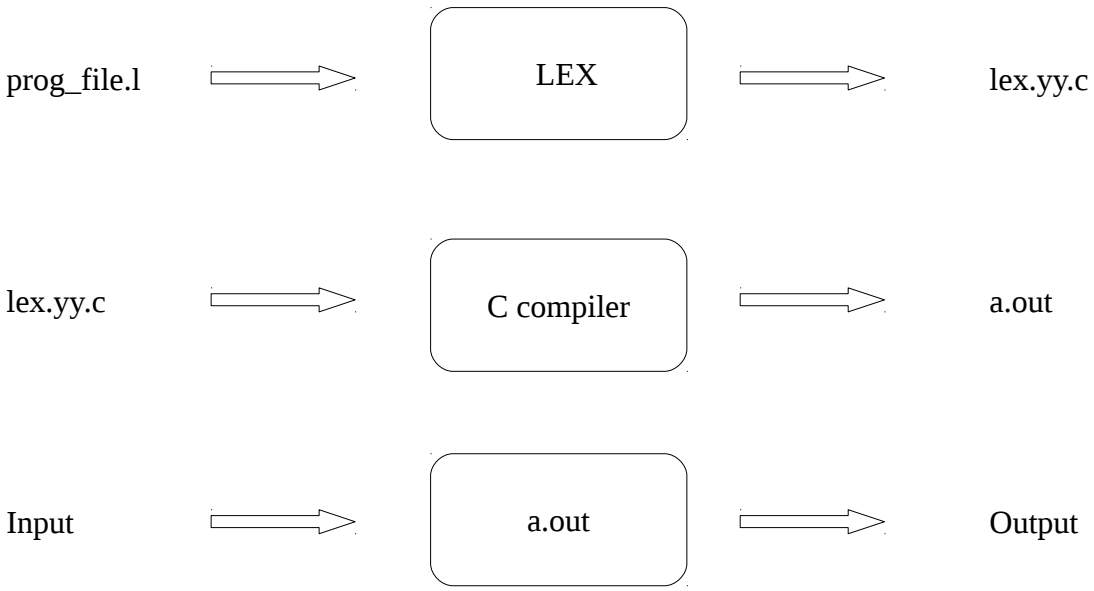
1 Introduction to LEX

LEX is a tool used to generate a lexical analyzer. This documentation covers all aspects regarding the use of LEX for SILC development. Technically, LEX is a compiler which compiles a LEX program (prog_file.l) and generates a C file (lex.yy.c) which is in turn compiled using a C compiler to generate an executable file, which is the generated lexical analyzer.



The source program is fed as the input to the the lexical analyzer which produces an output of a sequence of tokens.

2 Generating a lexical analyzer using LEX

Conceptually, a lexical analyzer scans a given source program and produces an output of tokens for each lexeme found. A lexeme is a sequence of characters in the source program that matches a pattern that has been specified in the lex program. A token is a single element of a specific programming language (used in the source program) that is recognized by the compiler. This documentation focuses on how this is achieved practically using LEX. The LEX tool is used to find a certain pattern in the input stream and execute a corresponding action associated with it, as specified in the LEX program. This ability of LEX is used to return tokens to a parser.

Example :

```
“integer”          {return ID_TYPE_INTEGER;}
```

This rule tells the compiler that the identifier to follow the pattern “integer” is of the type integer. Here the string “integer” is the pattern, the return statement is the action and ID_TYPE_INTEGER is the token returned . A rule in a lex program comprises of a pattern to be matched and a corresponding action.

The reason LEX is used instead of cascading a series of if-else statements with stcmp() conditions in an attempt to manually write a lexical analyzer in C , is because :

- LEX lets us handle character sequences as abstract entities
- Ease of handling errors
- Speed and efficiency

3 The LEX program

A LEX program consists of three sections : Declarations, Rules and Auxiliary functions.

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

3.1 Declarations

The declarations section is made up of regular definitions and auxiliary declarations (in C language). LEX allows the use of shorthand / extensions to regular expressions for the regular definitions.

A regular definition in LEX is of the form :

D R

where D is the symbol representing the regular expression R. The optional auxiliary declaration in C language is enclosed within ' %{ ' and ' %} ' . It is generally used to declare functions, include header files, or define global variables and constants.

Example :

```
%{
     /*My lexical analyzer's declaration section*/
     #include<stdio.h>
%}
number    [0-9]+
op        [-|+|*|/|^|=]
%%
```

3.2 Rules

Rules in a lex program consists of two parts :

i. The pattern to be matched
ii. The corresponding action to be executed

The pattern match to be found is specified as a regular expression . The reason we use regular expressions is because they are simple, handy and do the perfect job of representing a set of strings.

Example:

```
{number} {printf(“ This is a number”);}
{op}     {printf(“ This is an operator”);}
%%
```

Sample Input/Output

```
I: 234
O: This is a number

I: *
O: This is an operator
```

```
I: 2+3
O: This is a number This is an operator This is a number
```

The LEX compiler obtains the regular expressions of the symbols `number` and `op` from the declarations section and if a match is found in the input stream according to the specification, executes the corresponding action.

3.3 Auxiliary functions

In order to avoid huge amounts of code in the actions section, LEX provides auxiliary functions section. This section may be omitted in case of use with a parser. The user must define the main function in this section for using the lexical analyzer independent of the parser. The C code in this section and the declarations is copied as such to the lex.yy.c file.

Example :

```
int main()
{
     yylex();
     return 1;
}
```

4 The yyvariables

The following variables are offered by lex and are of great use while constructing a lexical analyzer for a compiler.

- yytext
- yyleng
- yylval

4.1 yytext

yytext is of the type char* and it contains the lexeme currently found. yytext points to the beginning of the input stream in the input buffer and is valid only within the actions section, i.e. It is available only when a pattern match is found.

Example:

```
{number} {printf("Number : %d",atoi(yytext));}
```

Sample Input/Output

```
I: 25
O: Number : 25
```

4.2 yyleng

yyleng is of the type int and it stores the length of the string given by yytext, i.e. The length of the string in the buffer currently matched. It is similar in nature w.r.t. availability.

Example:

```
{number} printf("Number of digits =  %d",yyleng);
```

Sample Input/Output

```
I: 1234
O: Number of digits = 4
```

4.3 yylval

yylval is a global variable of the type int. This variable is accessible by the parser, hence it helps in sharing of data between the lexical analyzer and the parser. It is used to return other additional information about the lexeme found to the parser i.e. yylval is used to return an attribute in addition to the token name to the parser.

Example :

```
{number} {
              yylval=atoi(yytext);
              return NUMBER;
          }
```

In the above example, the lexical analyzer tells the parser that the lexeme found is of the token type `NUMBER` and the value of the token is obtained by the parser by using yylval.

To return an attribute of a type other than int, yylval maybe overridden by a user defined yylval in the auxiliary declarations section. In order to return multiple attribute values for a token, it may be declared to be of the type `union`.

Example:

```
%{
     #include<stdio.h>
     typedef union
     {
          int value;
          int number_of_digits;
     }YYSTYPE;
     YYSTYPE yylval;
%}

number    [0-9]+

%%

{number} {
              yylval.value = atoi(yytext);
              yylval.number_of_digits = yyleng;
              return NUMBER;
          }
```

NOTE :

- The reason union is preferred over `struct` is lesser memory requirement by `union`. Creating a `struct` for each token might lead to wastage of huge amounts of memory, while on the other hand, values are simply overwritten in case of the type `union` for each consecutive lexeme found.

- The description of yylval here has been provided only for the understanding of handling of attributes. It is not declared by default in lex.yy.c by LEX. Hence, yylval must be defined manually to handle attributes independent of a parser.

5 The yyfunctions

The following functions in lex are of significant importance while constructing a lexical analyzer for a compiler.

- yylex()
- yywrap()

5.1 yylex()

yylex() is the scanning routine function that returns the token that has been obtained by the scanning process. It's return type is int. yylex() continues scanning either till EOF is reached or one if actions executes a return statement.

NOTE :

- If yylex() is called more than once, it simply starts scanning from the position in the file where it had ceased in the previous call.
- The carriage return CR marks the end of input instead of EOF when the input is read from the terminal.

A parser repetitively calls the yylex() function to consecutively read tokens from the input. The rules specified in the LEX program is used by LEX to build the code of yylex() in the lex.yy.c file.

5.2 yywrap()

yywrap() is a function called by the scanner when it encounters an EOF. It's return type is int. If yywrap() returns a non-zero value (indicating true), the scanner 'wraps up' i.e. stops scanning.

If yywrap() returns zero (indicating false), the scanner assumes that there is more input to be scanned and continues scanning. This function is useful if the programmer wishes to arrange for more input on completion of scanning an input file.

Example :

If the programmer wishes to scan more than one input file using the generated lexical analyzer, it can be simply done by setting `yyin` (the input file pointer) to a new input file in yywrap()'s definition.

```
int yywrap()
{
     FILE *newfile_pointer;
     newfile_pointer = fopen("prog_file_2.l");
     if(yyin != newfile_pointer)
     {
          yyin = newfile_pointer;
          return 0;
     }
     else
          return 1;
}
```

This definition of `yywrap()` sets the input file pointer to `prog_file_2.l` and returns 0 if the scanner finished scanning the first input file. As a result, the scanner continues scanning in `prog_file_2.l` . When the scanner calls yywrap() on EOF of `prog_file_2.l` then it returns 0, and the scanner stops scanning.

LEX also allows the programmer to avoid defining yywrap() by using `%option noyywrap` in the declarations section. This options removes the call to yywrap(). Note that, it is **mandatory** to either define yywrap() or indicate the absence using the `%option` feature. If not, the LEX compiler will flag an error.

6 The Even-Odd Program

```
%{
/*
1.Request input of an even and an odd number
2.indicate input characteristic : Even/Odd [digit_length]
3.check for input's correctness and print result
*/

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

int number_1;
int number_2;
%}


number_sequence [0-9]*

%%

{number_sequence}[0|2|4|6|8]         {
                              printf("Even number [%d]",yyleng);
                              return atoi(yytext);
                         }

{number_sequence}[1|3|5|7|9]         {
                              printf("Odd number [%d]",yyleng);
                              return atoi(yytext);
                         }
%%

int yywrap
{
     return 1;
}

int main()
{
     printf("\nInput an even number and an odd number\n");
     number_1 = yylex();
     number_2 = yylex();
     int diff = number_1 - number_2;
     if(diff%2!=0)
          printf("\nYour inputs were checked for correctness,
\nResult : Correct\n");
     else
          printf("\nYour inputs were checked for correctness,
\nResult : You do not know how to read\n");
     return 1;
}
```

7 Disambiguation rules

LEX uses two important disambiguation rules in selecting the right action to execute in case of a conflict :

- Order of occurrence is assigned as the pattern's matching priority.
- “Longest match” is preferred.

Example :

```
“break”                  { return BREAK; }
[a-zA-Z][a-zA-Z0-9]*     { return IDENTIFIER; }
```

Here, `break` is matched by both the regular expressions, but `break` is a keyword and not an identifier hence it is ordered in such a manner that LEX uses its first disambiguation rule to execute `return BREAK;`

Example :

```
“-”       {return MINUS;}
“--”      {return DECREMENT;}
```

In case of an `--` input to the lexical analyzer, note that LEX does not return two `MINUS` tokens, but instead returns a `DECREMENT` token, by the second disambiguation rule.

8 Pattern matching by LEX

LEX converts all its regular expressions into a finite state machine which it uses to accept or reject a string in the input stream. The corresponding action is executed when the machine is in accept state. The LEX compiler stores information about the constructed finite state machine in the form of a decision table (transition table) in the lex.yy.c file. Also the corresponding actions and the information regarding when they are to be executed is stored in the lex.yy.c file.

A `transition(current_state,input_char)` function is used to access the decision table. LEX makes it's decision table visible if we compile the program with the -T flag. The finite state machine used by LEX is deterministic in nature i.e. it is a DFA. The simulation of the constructed DFA is done in the lex.yy.c file. Hence, a LEX compiler constructs a DFA according to the specifications of the regular expression in the LEX program, and generates a simulation algorithm (to simulate the DFA) and a matching `switch-case` algorithm (to match and execute the appropriate action if the DFA enters an accept state).

9 The Token simulator program

```
%{
/* Scan and return a token for identifiers of the format :
          (string)(number)
     Note : strings are not case sensitive
     examples : a0 , A1 , ab2 , AB4 , aBc5
*/
#include<stdio.h>

#define ID 1  //Identifier token
#define ER 2  //Error token

%}
low_case [a-z]
upp_case [A-Z]
number   [0-9]

%option noyywrap

%%
({low_case}|{upp_case})({low_case}|{upp_case})*({number})
                                        return ID;
(.)*                                    return ER;

%%
int main()
{
     int token = yylex();
     if(token==ID)
          printf("Acceptable\n");
     else if(token==ER)
          printf("Unacceptable\n");
     return 1;
}
```

In this program, the main() function obtains the tokens (in place of a parser) and checks if the input contains a valid identifier.

Sample Input/Output:

```
I: Var9
O: Acceptable
```

When `Var9` is provided as the input, the DFA constructed by LEX accepts the string, and the corresponding action `return ID` is found and executed. As a result yylex() returns the token `ID`, and the main() function prints `Acceptable` on the screen.

10 Construction of a DFA from a regular expression

The construction of a DFA from a regular expression takes place in two steps.

- Constructing a syntax tree from the regular expression
- Converting the syntax tree into a DFA

10.1 The intermediate syntax tree

Consider the first rule in the token simulator program in section 9. It consists of the following regular expression :

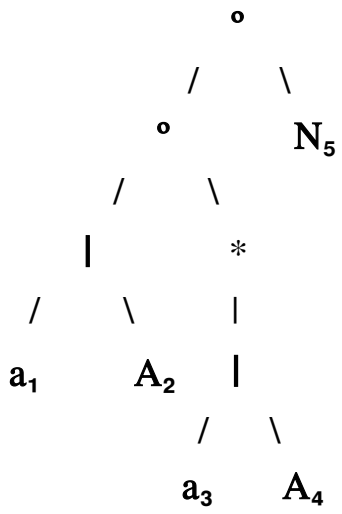
`({low_case}|{upp_case})({low_case}|{upp_case})*({number})`

For convenience in representation , it has been represented as :

(a | A) (a | A)* (N)

where, 'a' represents `{low_case}`, 'A' represents `{upp_case}` and 'N' represents `{number}`.

The syntax tree constructed for the above regular expression would look like :

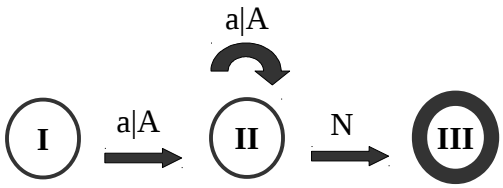


In the above figure ° represents the 'cat' (concatenation) operator, * represents the 'star' operator (a unary operator) and | represents the 'or' operator. In the syntax tree the inner nodes are operators while the leaves are the operands. The subscript assigned to every leaf is called the position of the leaf.

NOTE:
This syntax tree is an intermediate data structure. There will be no traces of this in lex.yy.c file, because it is only used in the construction of the DFA.

10.2 The constructed DFA

The DFA obtained for the above syntax tree would look like :



I, **II** and **III** denote the three states of the machine where **I** is the start state and **III** is the accept state. This DFA represents the regular expression provided as a specification (i.e. pattern to be matched) in the first rule of the token simulator program. The constructed DFA is simulated using a simulation algorithm.

11 The DFA simulation algorithm

The working of the constructed DFA is simulated using the following algorithm :

```
DFA_simulator()
     current_state = start_state
     c = get_next_char()
     while(c != EOF)
          current_sate = transition(current_state , c)
          c = get_next_char()
          if(current_state ∈ Final_states)
               /*ACCEPT*/
          else
               /*REJECT*/
```

The `transition()` function access the decision table which is generally a two dimensional matrix. The information about all the transitions made by the DFA can be obtained from the decision table through the `transition()` function.