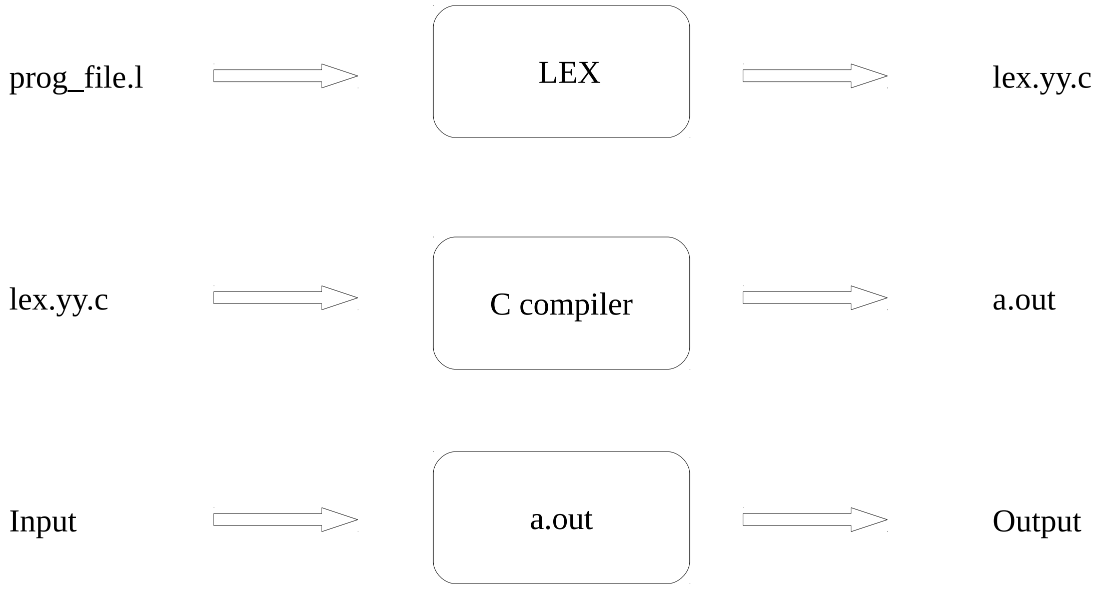
1 Introduction to LEX

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 LEX programs (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* i.e. pointer to a character 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 which 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.

Example :

```
{number} {
              yylval=yyleng;
              return NUMBER;
         }
```

Here, yylval has been used to return the the number of digits in the lexeme found.

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 in case, yylex() stops scanning because of return statement, and if yylex() is called again, it simply resumes scanning from where it stopped (in case of an input file).
- During input by user via the terminal, i.e. command line input, there is no eof.

A parser repetitively calls the yylex() function to return a sequence of tokens.
The rules specified in the LEX program is used to build the code of yylex() in the lex.yy.c file.

5.2 yywrap()

yywrap() is a function called by yylex() when the scanner reaches the eof. It must be manually defined in the auxiliary functions section. It is useful if in case the programmer wishes to arrange for more input for scanning on reaching the eof. In this case yywrap() must be set to return 0. But if the user does not wish to arrange for more input and hence, stop the scanning process and wrap up on normal eof, then yywrap() must be defined to return 1.

LEX also allows the programmer to avoid defining yywrap() by using :

```
%option noyywrap
```

in the declarations section.

Note that, it is **mandatory** to either define yywrap or indicate the absence through `%option` feature. If not, the LEX compiler will flag an error.

6 A 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;
}
```

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.