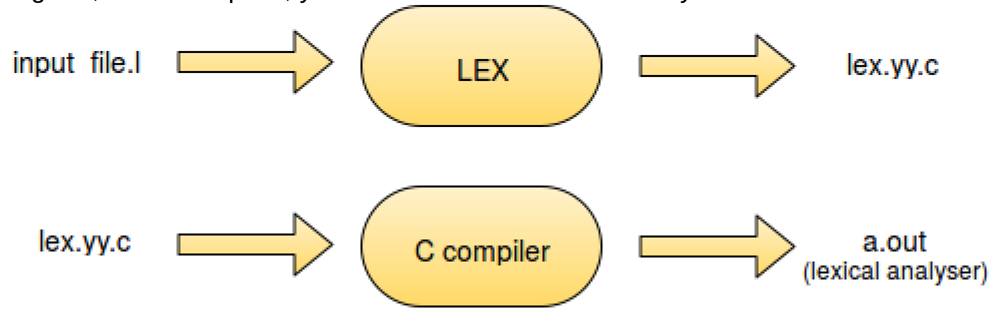
USING LEX

Introduction

LEX is a tool used to generate a lexical analyzer. This document is a tutorial for the use of LEX for **ExpL Compiler** development. Technically, LEX translates a set of regular expression specifications (given as input in input_file.l) into a C implementation of a corresponding finite state machine (lex.yy.c). This C program, when compiled, yields an executable lexical analyzer.



The source ExpL program is fed as the input to the the lexical analyzer which produces a sequence of tokens as output. (Tokens are explained below). Conceptually, a lexical analyzer scans a given source ExpL program and produces an output of tokens.
Each token is specified by a token name. The token name is an abstract symbol representing the kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. For instance integer, boolean, begin, end, if, while etc. are tokens in ExpL.

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

This example demonstrates the specification of a **rule** in LEX. The rule in this example specifies that the lexical analyzer must return the token named ID_TYPE_INTEGER when the pattern “integer” is found in the input file. A rule in a LEX program comprises of a 'pattern' part (specified by a regular expression) and a corresponding (semantic) 'action' part (a sequence of C statements). In the above example, “integer” is the pattern and {return ID_TYPE_INTEGER;} is the corresponding action. The statements in the action part will be executed when the pattern is detected in the input.

The structure of LEX programs

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

```
DECLARATIONS

%%
RULES
%%

AUXILIARY FUNCTIONS
```

2.1 Declarations

The declarations section consists of two parts, **auxiliary declarations** and **regular definitions**.
The auxiliary declarations are copied as such by LEX to the output *lex.yy.c* file. This C code consists of instructions to the C compiler and are not processed by the LEX tool.The auxiliary declarations (which are optional) are written in C language and are enclosed within ' %{ ' and ' %} ' . It is generally used to declare functions, include header files, or define global variables and constants.
LEX allows the use of short-hands and 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.

2.2 Rules

Rules in a LEX program consists of two parts :

1. The pattern to be matched
2. The corresponding action to be executed

The pattern to be matched is specified as a regular expression.

LEX obtains the regular expressions of the symbols 'number' and 'op' from the declarations section and generates code into a function *yylex()* in the *lex.yy.c* file. This function checks the input stream for the first match to one of the patterns specified and executes code in the action part corresponding to the pattern.

2.3 Auxiliary functions

LEX generates C code for the rules specified in the Rules section and places this code into a single function called *yylex()*. (To be discussed in detail later). In addition to this LEX generated code, the programmer may wish to add his own code to the *lex.yy.c* file. The auxiliary functions section allows the programmer to achieve this.

Example:

```
/*Declarations section start here*/
/* Auxiliary declarations start here*/
%{
        #include <stdio.h>
        int global_variable;
%}
/*Auxiliary declarations end & Regular definitions start here*/
  number [0-9]+          //Regular definition
  op     [-|+|*|/|^|=]   //Regular definition
/*Declarations section ends here*/
%%      /* Rules */
  {number}  {printf(" number");}
  {op}      {printf(" operator");}
%%
/* Auxiliary functions */
int main()
{
  yylex();
  return 1;
}
```

The pattern to be matched is specified as a regular expression.
Sample Input/Output for the above example:

```
I: 234
O: number

I: *
O: operator

I: 2+3
O: number operator number
```

LEX obtains the regular expressions of the symbols 'number' and 'op' from the declarations section and generates code into a function *yylex()* in the *lex.yy.c* file. This function checks the input stream for the first match to one of the patterns specified and executes code in the action part corresponding to the pattern.

The yyvariables

The following variables are offered by LEX to aid the programmer in designing sophisticated lexical analyzers. These variables are accessible in the LEX program and are automatically declared by LEX in *lex.yy.c*.

- yyin
- yytext
- yyleng

3.1 yyin
yyin is a variable of the type FILE* and points to the input file. *yyin* is defined by LEX automatically. If the programmer assigns an input file to *yyin* in the auxiliary functions section, then *yyin* is set to point to that file. Otherwise LEX assigns *yyin* to stdin(console input).
Example:

```
        /* Declarations */
        %%
           /* Rules */
        %%
        main(int argc, char* argv[])
        {
                if(argc > 1)
                {
                        FILE *fp = fopen(argv[1], "r");
                        if(fp)
                                yyin = fp;
                }
                yylex();
                return 1;
        }
```

3.2 yytext
yytext is of type *char** and it contains the *lexeme* currently found. A **lexeme** is a sequence of characters in the input stream that matches some pattern in the Rules Section. (In fact, it is the first matching sequence in the input from the position pointed to by *yyin*.) Each invocation of the function *yylex()* results in *yytext* carrying a pointer to the lexeme found in the input stream by *yylex()*. The value of yytext will be overwritten after the next *yylex()* invocation.
Example:

```
%option noyywrap
%{
#include <stdlib.h>
#include <stdio.h>
%}
number [0-9]+

%%
{number} {printf("Found : %d\n",atoi(yytext));}
%%

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

In this case when *yylex()* is called, the input is read from the location given by *yyin* and a string “25” is found as a match to 'number'. This location of this string in the memory is pointed to by *yytext*. The corresponding action in the above rule uses a built-in function *atoi()* to convert the string “25” (of type char*) to the integer 25 (of the type int) and then prints the result on the screen. Note that the header file “stdlib.h” is called in the auxiliary declarations section in order to invoke *atoi()* in the actions part of the rule.

NOTE: The lexeme found by LEX is stored in some memory allocated by LEX which can be accessed through the character pointer yytext.
NOTE: The *%option noyywrap* is used to inform the compiler that the function *yywrap()* has not been defined. We will see what this function does later on.

Sample Input/Output:

```
I: 25
O: Found : 25
```

3.3 yyleng
yyleng is a variable of the type int and it stores the length of the lexeme pointed to by *yytext*.
Example:

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

The yyfunctions

- yylex()
- yywrap()

4.1 yylex()
yylex() is a function of return type int. LEX automatically defines *yylex()* in *lex.yy.c* but does not call it. The programmer must call yylex() in the Auxiliary functions section of the LEX program. LEX generates code for the definition of yylex() according to the rules specified in the Rules section.
NOTE: That yylex() need not necessarily be invoked in the Auxiliary Functions Section of LEX program when used with YACC.

Example:

```
/* Declarations */
%%
{number} {return atoi(yytext);}
%%
int main()
{
        int num = yylex();
        printf("Found: %d",num);
        return 1;
}
```

4.2 yywrap()
LEX declares the function yywrap() of return-type int in the file *lex.yy.c* . LEX does not provide any definition for yywrap(). yylex() makes a call to yywrap() when it encounters the end of input. If yywrap() returns zero (indicating *false*) yylex() assumes there is more input and it continues scanning from the location pointed to by yyin. If yywrap() returns a non-zero value (indicating true), yylex() terminates the scanning process and returns 0 (i.e. “wraps up”). If the programmer wishes to scan more than one input file using the generated lexical analyzer, it can be simply done by setting yyin to a new input file in yywrap() and return 0.
As LEX does not define yywrap() in lex.yy.c file but makes a call to it under yylex(), the programmer must define it in the Auxiliary functions section or provide %option noyywrap in the declarations section. This options removes the call to yywrap() in the lex.yy.c file. Note that, it is **mandatory** to either define yywrap() or indicate the absence using the %option feature. If not, LEX will flag an error