LEX – TOOL FOR LEXICAL ANALYZER GENERATOR

SHEDULED FOR 20.12.07 3rd HOUR

lex is a lexical analyzer generator. You specify the scanner you want in the form of patterns to match and actions to apply for each token. **lex** takes your specification and generates a combined NFA to recognize all your patterns, converts it to an equivalent DFA, minimizes the automaton as much as possible, and generates C code that will implement it. **lex** is the original scanner generator designed by Lesk and Schmidt.

How It Works

lex is designed for use with C code and generates a scanner written in C. The scanner is specified using regular expressions for patterns and C code for the actions. The secification files are traditionally identified by their **.l** extension. You invoke **lex** on a **.l** file and it creates **lex.yy.c**, The file provides an **extern** function **yylex()** that will scan one token.You compile that C file normally, link with the **lex** library, and you have built a scanner! The scanner reads from **stdin** and writes to **stdout** by default.

% lex myFile.l creates **lex.yy.c** containing C code for scanner
% cc lex.yy.c compiles scanner, links with **lex** lib
% ./a.out executes scanner, that will read from **stdin**

Linking with the **lex** library provides a simple **main** that will repeatedly calls **yylex** until it reaches **EOF**.

A lex Input File

Your input file is organized as follows:
%{
Declarations
%}
Definitions

%%

Rules

%%

User subroutines

The optional **Declarations** and **User subroutines** sections are used for ordinary C code The optional **Definitions** section is where you specify options for the scanner and can set up definitions to give names to regular expressions as a simple substitution mechanism that allows for more readable entries in the **Rules** section that follows. The required **Rules** section is where you specified the patterns that identify your tokens and the action to perform upon recognizing each token.

lex Rules

A rule has a regular expression (called the *pattern*) and an associated set of C statements (called the *action*). The idea is that whenever the scanner reads an input sequence that matches a pattern, it executes the action to process it.In specifying patterns, **lex** supports a fairly rich set of conveniences (character classes, specific repetition, etc.) beyond our formal language definition of a regular expression. These features don't add expressive power, but simply allow you to construct complicated patterns more succinctly. The table below shows some operators to give you an idea of what is available. For more details, see the web or man pages.

Character classes **[0-9].** This means alternation of the characters in the range listed (in this case: **0|1|2|3|4|5|6|7|8|9**). More than one range may be specified, e.g. **[0-9A-Za-z]** as well as specifying individual characters, as with **[aeiou0-9]**. Character exclusion **^** The first character in a character class may be **^** to indicate the complement of the set of characters specified. For example, **[^0-9]** matches any non digit character. Arbitrary character **.** The period matches any single character except newline. Single repetition **x?** This means 0 or 1 occurrence of **x**. Non-zero repetition **x+** This means **x** repeated one or more times; equivalent to **xx***. Specified repetition **x{n,m} x** repeated between **n** and **m** times.

As the scanner reads characters from the file, it will gather them until it forms the longest possible match for any of the available patterns. If two or more patterns match an equally long sequence, the pattern listed first in the file is used. The code that you include in the actions depends on what processing you are trying to do with each token. Perhaps the only action necessary is to print the matching token, add it to a table, or perhaps ignore it in the case of white space or comments. For a scanner designed to be used by a compiler, the action will usually record the token attributes and return a code that identifies the token type.

lex Global Variables

The token-grabbing function **yylex** takes no arguments and returns an integer. Often more information is needed about the token just read than that one integer code. The usual way information about the token is communicated back to the caller is by having the scanner set the contents of a global variable which can be read by the caller. Here are the specific global variables used:

• **yytext** is a null-terminated string containing the text of the lexeme just recognized as a token. This global variable is declared and managed in the **lex.yy.c** file. Do not modify its contents.

• **yyleng** is an integer holding the length of the lexeme stored in **yytext**. This global variable is declared and managed in the **lex.yy.c** file. Do not modify its contents.

• **yylval** is the global variable used to store attributes about the token, e.g. for an integer lexeme it might store the value, for a string literal, the pointer to its characters and so on.

• **yylloc** is the global variable that is used to store the location (line and column) of the token.

Example 1

Here is a simple and complete specification for a scanner that recognizes the identifiers (variables) and arithmetic operators.

```
%{
%}
identifier [a-z A-Z][a-z A-Z 0-9]*
%%
{identifier} {printf("%s identifier",yytext);}
\+|\-|\*|\/|% {printf("%s arithmetic operators",yytext);}
%%
int main(int argc, char **argv)
{
  FILE *fp;
  fp=fopen(argv[1],"r");
  yyin=fp;
  yylex();
  return 0;
}
int yywrap()
{
   return 1;
 }
```