Handout 7, Lex (5/30/2001)

Lex is a venerable Unix tool that generates scanners. Input to lex is a text file that specifies the scanner; more precisely: specifying tokens, a yet to be made scanner must recognize, and what to do with each recognized pattern. The specification is partitioned into three sections: a definition section, rules section, and the function section. The command to run lex is: lex switches file_names. Switches are optional. When more than one file name is specified, they are all concatenated and used as one single input to lex. When no input file is specified, then input comes from stdin.

The default action of the generated scanner is to read input from stdin, and produce output onto stdout. The shortest possible lex program %% will do just that.

Table 0 defines lex switches. The lex commands is:
lex [-cntv] [-e|-w] [-V -Q [y|n]] [ file . . . ]
Example:
lex in.l and then: cc lex.yy.c -ll

<table>
<thead>
<tr>
<th>Switch</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>C-language action; is default</td>
</tr>
<tr>
<td>-e</td>
<td>Scanner can handle European language EUC characters, exclusive with -w</td>
</tr>
<tr>
<td>-f</td>
<td>No packing</td>
</tr>
<tr>
<td>-n</td>
<td>Suppress table size statistics after scanner generation</td>
</tr>
<tr>
<td>-t</td>
<td>Instead of generating C source program in file lex.yy.c, emit C source onto stdout. From there it may be redirected, or piped into more for viewing.</td>
</tr>
<tr>
<td>-v</td>
<td>Write summary of lex table sizes to stderr;</td>
</tr>
<tr>
<td>-w</td>
<td>Scanner cannot handle EUC characters; exclusive with -e</td>
</tr>
<tr>
<td>-V</td>
<td>Print version information onto stderr;</td>
</tr>
<tr>
<td>-Q[n</td>
<td>y]</td>
</tr>
</tbody>
</table>

The definition section holds C language definitions, for example external objects, or objects unique to functions that constitute the scanner. The rules section is composed of patterns and actions; we'll explain patterns below. When the pattern defined is matched with the input, the associated action is executed. The function section holds any C text you wish, including executable statements in C function bodies. Sections are separated from one another by the %%% meta operators. When only a single %%% is given, this means, the function section is empty. When no %%% is given, the specification is erroneous.

When 2 or more patterns start matching the same source input, the longest one is accepted. If two or more matching ones are of the same length, the pattern defined first is taken.

Definition Section: Any line in the definition section (actually anywhere in a lex program) starting with a blank ' ' is considered a snippet of C to be copied into the external definition area of the resulting lex.yy.c source program. Also, text enclosed in the %{ and %} meta brackets is considered C source text and is copied into lex.yy.c.

Any line in the definition section not starting with a blank (and not enclosed between the %{ and %} meta operators) is a definition. The format of definitions is: name substitute. Name must
match the C lexical rules for identifier. Such a name may be referenced in the rules section, enclosed in { and } braces, for example: {name1}. In that case, the matching substitute replaces the complete string {name1} in a rule. The { and } braces are syntactic means to tell lex: Here is a name, defined in the definition section!

**States:** Any line in the definition section starting with % and followed by an identifier starting with s or S defines a state (aka start condition). For example, %state1 in the definition section defines a lex state, named state1. States can be used in the rules of the rules section to make actions conditional. Any line in the definition section starting with % and followed by an identifier starting with x or X defines an exclusive state (aka exclusive start condition). For example, %x ex1 defines a lex exclusive state, named ex1. When the generated scanner is in a %s state, patterns with no state will also be considered. When the scanner is in an exclusive %x state, then only rules associated with an exclusive state will be considered. Mostly you'll use the %s form.

**Pointer or Array:** The predefined words %array and %pointer specify, how yytext is defined in C. The default, matching %array, is extern char yytext[]; . This can be overridden with %pointer to mean: extern char * yytext; .

**Rules Section:** The 0 or more rules in the rules section each consist of a pattern (an Extended Regular Expression, aka ERE), followed by an associated action, expressed as a C program snippet. The two are separated from one another by 1 or more blanks. For example, the rule below is meant to convert uppercase letters to lowercase:

```
[A-Z] putchar( yytext[ 0 ] + 'a' - 'A' );
```

When an action is not specified, lex does not have to detect this and the result is undefined. Note that the ; in place of the C program snippet is a complete action: it specifies the empty statement, doing nothing. This will effectively skip the matched input at compilation time (when the scanner reads source). Also note that when an input character (or string) is not matched, it will be output by the scanner. Another example:

```
[abcx-z] { return SPECIAL_L; }
```

When the input is any of the characters a, b, c, x, y, or z, then the value SPECIAL_L is returned by the scanner specified above.

**Function Section:** This section includes 0 or more C function definitions with complete bodies, that can be used in the scanner. For example, these can be functions that are callable from the C snippets specified as actions when patterns are matched.

Table 1 defines the lex meta operators used to build patterns of character strings:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%</td>
<td>Separate one section from another, at most 3 sections, at most two %%</td>
</tr>
</tbody>
</table>
Table 2 shows a few sample patterns and their meaning:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matching strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a-z]</td>
<td>All lower case letters, exactly one of them</td>
</tr>
<tr>
<td>[A-Z]+</td>
<td>One or more upper case letters</td>
</tr>
<tr>
<td>[0-9]*</td>
<td>Zero or more digits</td>
</tr>
<tr>
<td>-?[0-9]+</td>
<td>Decimal number with 1 or more digits, optionally signed with -</td>
</tr>
<tr>
<td>[a-zA-Z][0-9]*</td>
<td>A letter, followed by 0 or more letters or digits</td>
</tr>
<tr>
<td>&quot;Hello&quot;</td>
<td>The string &quot;Hello&quot;</td>
</tr>
<tr>
<td>^mom</td>
<td>The string or substring &quot;mom&quot; at the beginning of a line</td>
</tr>
<tr>
<td>-?((0-9)+)</td>
<td>Optionally signed integer or float literal. Float need not have a</td>
</tr>
<tr>
<td></td>
<td>leading digit, for example - .99. But -0.99 also o.k.</td>
</tr>
</tbody>
</table>

Table 3 lists selected external lex C objects, used by programs linked with lex, e.g. yacc:

<table>
<thead>
<tr>
<th>lex C object</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>yylex()</td>
<td>The scanner function, callable by yacc</td>
</tr>
<tr>
<td>yytext</td>
<td>Char array holding scanned input, ended by null; length can be set</td>
</tr>
<tr>
<td>lex.yy.c</td>
<td>C program that is the scanner, generated by lex</td>
</tr>
<tr>
<td>ECHO</td>
<td>Macro that copies source matched to standard out</td>
</tr>
<tr>
<td>y.tab.h</td>
<td>Include file used by lex, generated by yacc</td>
</tr>
<tr>
<td>yylval</td>
<td>Integer object of yacc, frequently assigned by lex</td>
</tr>
</tbody>
</table>
lex Sample 1: The lex program below scans integer literals, returns a NUMBER token for each, and filters out (skips) the rest of the source coming from stdin:

```
%{
    /*
    * The definition section:
    * PSU CS 302 Spring 2001
    * Herb Mayer 4/26/2001
    */
    #include "y.tab.h"
    extern int yylval;
%
[0-9]+ { yylval = atoi( yytext ); return NUMBER; }
.\n    { return yytext[ 0 ]; }
%
/*
* The Function Section: empty
*/
```

The simple lex source shows the 3 sections separated by %% from one another. The definition section includes a C style comment, #includes a file with #defines generated by yacc, and specifies a C integer object named yylval defined elsewhere; it is external to lex. It filters out decimal numbers, one digit or more constituting a token, and returns the converted integer value to its caller, presumably a parser generated by yacc. All other input, expressed by the . meta character, and the newline are silently swallowed (not copied to stdout), and returned to the caller of yylex(). Note that the meta character . does not include (not match on) the newline.

lex Sample 2: The lex program below converts upper- to lowercase characters:

```
%%
[A-Z]    putchar( yytext[ 0 ] + 'a' - 'A' );
```

A scanner generated with Sample 2 reads input, copies all input to stdout, except that uppercase letters are all converted to lowercase letters. The rest of the output looks just like the input. Note that the second %% meta operator is superfluous.

lex Sample 3: The lex program below converts to lowercase and reduces white space:

```
%%
[A-Z]    putchar( yytext[ 0 ] + 'a' - 'A' );
[ ]+\$ ;
[ ]+    putchar( ' ' );
```
A scanner generated with Sample 3 reads input, copies it to `stdout`, except that uppercase letters are all converted to lowercase. Also, blanks at the end of a text line are discarded, and multiple blanks inside other text are all replaced by a single blank each.

**Lex macros for I/O:** There are a few macros defined in lex that can be used in the *action* portion of a *rule*, listed in Table 4:

<table>
<thead>
<tr>
<th>Table 4: Lex I/O intrinsics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input()</strong></td>
</tr>
<tr>
<td><strong>output( char )</strong></td>
</tr>
<tr>
<td><strong>unput( char )</strong></td>
</tr>
<tr>
<td><strong>ECHO</strong></td>
</tr>
</tbody>
</table>

We’ll see an example of the use of `input()` and `unput( char )` further on.

**lex Sample 4:** The lex program below filters out C style /* ... */ comments.

```c
%%
"/**" { again: // loops if EOF before */
while( input() != '*' );
// last input() was '***', what is next? '/#' perhaps?
switch( input() ) {
  case '/': break; // comment was skipped, /* ... */
  case '***': unput( '***' ); // stuff back, could be /** then /**/
  default: goto again; // yeah, we know, gotos are ugly 😞
} //end switch
} // end single C action, matching pattern "/**"
```

A scanner generated with Sample 4 reads any text input, copies it back onto `stdout`, except that all text between /* and */, including the /* and */ delimiters have magically disappeared. However, the program loops, if a comment is incomplete such that the end of the file is found after a /* comment started, but the ending */ is not provided in the source.

**Trailing context / lex meta operator:**

**The / trailing context operator:** An ERE of the form `pattern1/pattern2` causes `pattern1` in the *rules section* to match only, if it is followed by a source string that matches `pattern2`. Only the source strings matching `pattern1` are actually consumed; `pattern2` remains untouched. Thus, `pattern2` or a portion thereof may be read in a subsequent lex activity.

Confusion can arise, and lex is not defined in some of these cases, if the end of `pattern1` matches the start of `pattern2`. For example: given an ERE pattern a*b/ccc and input aaabccc, yytext will hold aaab. However, given the pattern x*/xy and input xxxxy, the token xxx and not xx is returned.
by some lexes, since \textit{xxx} matches the longest of \textit{x*}. Similarly: for the pattern with trailing context \textit{ab*/bc} and input \textit{abbbc} the length of the recognized match is undefined. It is either \textit{abb} or \textit{abbb}. But for pattern \textit{ab/bc} and input \textit{abbc} the matched string is clearly defined: \textit{ab}.

\textbf{Sample of Lex State:}

\textbf{lex Sample 5:} The same below shows a convenient use of states, arbitrarily using exclusive states. The \texttt{%s} state would have worked the same:

\begin{verbatim}
lex Sample 5

%x CMT
%%
"/\*"  { printf("/\* "); BEGIN CMT; }
<CMT>.  putchar(yytext[ 0 ]);<CMT>\n    putchar(yytext[ 0 ]);<CMT>"*/"  { printf("*/ "); BEGIN INITIAL; }
   ;
\n%%

\end{verbatim}

When the scanner generated by \textit{Sample 5} reads C source input, it generates output solely consisting of the old C-style comments, plus a blank character after each complete /\* and */ comment. Note the way the state CMT is initialized to \texttt{on}, and turned \texttt{off}. The former is accomplished by \texttt{BEGIN CMT}, the latter by \texttt{BEGIN INITIAL}. The \texttt{INITIAL} is a predefined command resetting all switches.

\textbf{lex Sample 6:} The lex program below is intended to filter out C style /\* and */ comments. But it will fail. We leave it as an exercise to the reader, why this lex pattern will not filter just all /\* and */ comments as desired ☺

\begin{verbatim}
lex Sample 6

%%
"/\*" (. \| \n) */" ;

\end{verbatim}

Note that the pattern (. \| \n) will also match, at least some times, the ' * ' and ' / ' characters. Which match case is excluded?

\textbf{lex Sample 7:} The next small lex program filters out all ANSI C-style comments, and copies all other characters of an ASCII text file onto standard output:

\begin{verbatim}
lex Sample 7

%%
"/\*" . */ ;

\end{verbatim}
The lex source in Example 7 scans for strings that start with //, exactly the characters for the beginning of a new-style C comment. From the first pair of // on any text line until the end of that line, all text is skipped. All other characters of the input file are copied back to standard output.

**Caution:**

There exist numerous inconsistencies in the different implementations of lex on different systems, aside from character set differences. Major causes for differences include:

- Lex response on illegal input, e.g. for a missing action in a rule
- Trailing context match, if `pattern1's` end matches `pattern2's` start
- Type of `yytext`
- Lex command line switches