A Lex Tutorial

Victor Eijkhout

July 2004

1 Introduction

The unix utility *lex* parses a file of characters. It uses regular expression matching; typically it is used to 'tokenize' the contents of the file. In that context, it is often used together with the *yacc* utility. However, there are many other applications possible.

2 Structure of a *lex* file

A *lex* file looks like

```
   ...definitions...
%%
   ...rules...
%%
   ...code...
```

Here is a simple example:

```
%{
 int charcount=0,linecount=0;
%}

%%

. charcount++;
\n {linecount++; charcount++;}

%%
int main()
{
  yylex();
  printf("There were %d characters in %d lines\n",
         charcount,linecount);
  return 0;
}
```

If you store this code in a file `count.l`, you can build an executable from it by

```
lex -t count.l > count.c
cc -c -o count.o count.l
cc -o counter count.o -ll
```

You see that the *lex* file is first turned into a normal C file, which is then compiled and linked.

If you use the *make* utility (highly recommended!) you can save a few steps because *make* knows about *lex*:

```
counter: count.o
        cc -o counter count.o -ll
```

In the example you just saw, all three sections are present:

definitions All code between `%{` and `%}` is copied to the beginning of the resulting C file.

rules A number of combinations of pattern and action: if the action is more than a single command it needs to be in braces.

code This can be very elaborate, but the main ingredient is the call to `yylex`, the lexical analyser. If the code segment is left out, a default main is used which only calls `yylex`.

3 Definitions section

There are three things that can go in the definitions section:

C code Any indented code between `%{` and `%}` is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

definitions A definition is very much like a `#define` cpp directive. For example

```
letter [a-zA-Z]
digit [0-9]
punct [,.:;!?]
nonblank [^ \t]
```

These definitions can be used in the rules section: one could start a rule

```
{letter}+ {...
```

state definitions If a rule depends on context, it's possible to introduce states and incorporate those in the rules. A state definition looks like `%s STATE`, and by default a state `INITIAL` is already given. See section 4.2 for more info.

4 Rules section

The rules section has a number of pattern-action pairs. The patterns are regular expressions (see section 5, and the actions are either a single C command, or a sequence enclosed in braces.

If more than one rule matches the input, the longer match is taken. If two matches are the same length, the earlier one in the list is taken.

4.1 Matched text

When a rule matches part of the input, the matched text is available to the programmer as a variable `char* yytext` of length `int yyleng`.

To extend the example from the introduction to be able to count words, we would write

```
%{
 int charcount=0,linecount=0,wordcount=0;
%}
letter [^ \t\n]

%%

{letter}+ {wordcount++; charcount+=yyleng;}
. charcount++;
\n {linecount++; charcount++;}
```

4.2 State

If the application of a rule depends on context, there are a couple of ways of dealing with this. We distinguish between ‘left state’ and ‘right state’, basically letting a rule depend on what comes before or after the matched token.

See section 8.1 for a good example of the use of state.

4.2.1 Left state

A rule can be prefixed as

```
<STATE>(some pattern) {...
```

meaning that the rule will only be evaluated if the specified state holds. Switching between states is done in the action part of the rule:

```
<STATE>(some pattern) {some action; BEGIN OTHERSTATE;}
```

where all state names have been defined with `%s SOMESTATE` statements, as described in section 3. The initial state of *lex* is `INITIAL`.

4.2.2 Right state

It is also possible to let a rule depend on what follows the matched text. For instance

```
abc/de {some action}
```

means ‘match `abc` but only when followed by `de`. This is different from matching on `abcde` because the `de` tokens are still in the input stream, and they will be submitted to matching next.

It is in fact possible to match on the longer string; in that case the command

```
abcde {yyless(3); .....}
```

pushes back everything after the first 3 characters. The difference with the slash approach is that now the right state tokens are actually in `yytext` so they can be inspected.

5 Regular expressions

Many Unix utilities have regular expressions of some sort, but unfortunately they don’t all have the same power. Here are the basics:

`.` Match any character except newlines.
`\n` A newline character.

`\t` A tab character.
`^` The beginning of the line.
`$` The end of the line.
`<expr>*` Zero or more occurences of the expression.
`<expr>+` One or more occurences of the expression.
`(<expr1>|<expr2>)` One expression of another.
`[<set>]` A set of character or ranges, such as `[a-zA-Z]`.
`[^<set>]` The complement of the set, for instance `[^ \t]`.

6 User code section

If the *lex* program is to be used on its own, this section will contain a `main` program. If you leave this section empty you will get the default main:

```
int main()
{
  yylex();
  return 0;
}
```

where `yylex` is the parser that is built from the rules.

7 Lex and Yacc

The integration of *lex* and *yacc* will be discussed in the *yacc*tutorial; here are just a few general comments.

7.1 Definition section

In the section of literal C code, you will most likely have an include statement:

```
#include "mylexyaccprog.h"
```

as well as prototypes of *yacc* routines such as `yyerror` that you may be using. In some *yacc* implementations declarations like

```
extern int yylval;
```

are put in the `.h` file that the *yacc* program generates. If this is not the case, you need to include that here too if you use `yylval`.

7.2 Rules section

If you *lex*programmer is supplying a tokenizer, the *yacc* program will repeatedly call the `yylex` routine. The rules will probably function by calling `return` everytime they have constructed a token.

7.3 User code section

If the *lex* program is used coupled to a *yacc* program, you obviously do not want a main program: that one will be in the *yacc* code. In that case, leave this section empty; thanks to some cleverness you will not get the default main if the compiled *lex* and *yacc* programs are linked together.

8 Examples

8.1 Text spacing cleanup

(This section illustrates the use of states; see section 4.2.)

Suppose we want to clean up sloppy spacing and punctuation in typed text. For example, in this text:

```
This    text (all of it  )has occasional lapses , in
 punctuation(sometimes pretty bad) ,( sometimes not so).


(Ha!) Is this: fun? Or what!
```

We have

- Multiple consecutive blank lines: those should be compacted.
- Multiple consecutive spaces, also to be compacted.
- Space before punctuation and after opening parentheses, and
- Missing spaces before opening and after closing parentheses.

That last item is a good illustration of where state comes in: a closing paren followed by punctuation is allowed, but followed by a letter it is an error to be corrected.

8.1.1 Right state solution

Let us first try a solution that uses ‘right state’: it basically describes all cases and corrects the spacing.

```
punct [,.;:!?]
text [a-zA-Z]

%%

")"" "+/{punct}       {printf(")");}
")"/{text}            {printf(") ");}
{text}+" "+/")"       {while (yytext[yyleng-1]==’ ’) yyleng--; ECHO;}

({punct}|{text}+)/"(" {ECHO; printf(" ");}
"("" "+/{text}        {while (yytext[yyleng-1]==’ ’) yyleng--; ECHO;}

{text}+" "+/{punct}   {while (yytext[yyleng-1]==’ ’) yyleng--; ECHO;}

^" "+                 ;
" "+                  {printf(" ");}
.                     {ECHO;}
\n/\n\n               ;
\n                    {ECHO;}
```

In the cases where we match superfluous white space, we manipulate `yyleng` to remove the spaces.

8.1.2 Left state solution

One problem with the right state approach is that generalizing it may become unwieldy: the number of rules potentially grows as the product of the number of categories that we

recognise before and after the spaces. A solution that only grows as the sum of these can be found by using ‘left state’.

Using left state, we implement a finite state automaton in *lex*, and specify how to treat spacing in the various state transitions. Somewhat surprisingly, we discard spaces entirely, and reinsert them when appropriate.

We recognise that there are four categories, corresponding to having just encountered an open or close parenthesis, text, or punctuation. The rules for punctuation and closing parentheses are easy, since we discard spaces: these symbols are inserted regardless the state. For text and opening parentheses we need to write rules for the various states.

```
punct [,.;:!?]
text [a-zA-Z]

%s OPEN
%s CLOSE
%s TEXT
%s PUNCT

%%

" "+ ;

<INITIAL>"(" {ECHO; BEGIN OPEN;}
<TEXT>"(" {printf(" "); ECHO; BEGIN OPEN;}
<PUNCT>"(" {printf(" "); ECHO; BEGIN OPEN;}

")" {ECHO ; BEGIN CLOSE;}

<INITIAL>{text}+ {ECHO; BEGIN TEXT;}
<OPEN>{text}+ {ECHO; BEGIN TEXT;}
<CLOSE>{text}+ {printf(" "); ECHO; BEGIN TEXT;}
<TEXT>{text}+  {printf(" "); ECHO; BEGIN TEXT;}
<PUNCT>{text}+ {printf(" "); ECHO; BEGIN TEXT;}

{punct}+ {ECHO; BEGIN PUNCT;}

\n {ECHO; BEGIN INITIAL;}

%%
```