A *lex* tutorial

Victor Eijkhout

August 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. By itself, *lex* is powerful enough to build interesting programs with, as you will see in a few examples.

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;
}
```

In this example, 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`.

2.1 Running *lex*

If you store your *lex* 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.c
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
```

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 **??** for more info.

4 Rules section

The rules section has a number of pattern-action pairs. The patterns are regular expressions (see section **??**, 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.

It is possible to associate one action with more than one pattern:

```
[0-9]+          process_integer();
[0-9]+\.[0-9]*  |
\.[0.9]+        process_real();
```

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++;}
```

Exercise 1. Write an integer postfix calculator in *lex*: expression such as `1 2 +` and `1 2 3 4/*-` should be evaluated to `3` and `-.5` respectively. White space only serves to separate number, but is otherwise optional; the line end denotes the end of an expression. You will probably need the `C` function `int atoi(char*)` which converts strings to ints.

4.2 Context

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

See section **??** for an elaborate example of the use of context.

4.2.1 Left context

Sometimes, using regular expression as we have seen so far is not powerful enough. For example:

```
%%
"/*".*"*/"  ;
.           |
\n          ECHO;
```

works to filter out comments in

```
This line /* has a */ comment
```

but not in

```
This /* line has */ a /* comment */
```

What we want is, after the `/*` string to change the behaviour of *lex* to throw away all characters until `*/` is encountered. In other words, we want *lex* to switch between two states, and there is indeed a state mechanism available.

We can consider states to implement implement a dependence on the left context of a rule, since it changes the behaviour depending on what came earlier. To use a state, a rule is 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 **??**. The initial state of *lex* is `INITIAL`.

Here is the solution to the comment filtering problem:

```
%x COMM

%%

.           |
\n          ECHO;
"/*"        BEGIN COMM;
<COMM>"*/" BEGIN INITIAL;
<COMM>.     |
<COMM>\n    ;

%%
```

We see that the state is defined with `%x COMM` rather than as indicated above with `%s`. This is called an 'exclusive state'. If an exclusive state is active, rules without state prefix will not be matched if there is a match in a rule *with* the prefix of the current state.

4.2.2 Right context

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 context 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 occurrences of the expression.
`<expr>+` One or more occurrences of the expression.
`(<expr1>|<expr2>)` One expression of another.
`[<set>]` A set of characters or ranges, such as `[,.:;]` or `[a-zA-Z]`.
`[^<set>]` The complement of the set, for instance `[^ \t]`.

Exercise 2. It is possible to have `]` and − in a character range. The `]` character has to be first, and − has to be either first or last. Why?

Exercise 3. Write regular expressions that match from the beginning of the line to the first letter 'a'; to the last letter 'a'. Also expressions that match from the first and last 'a' to the end of the line. Include representative input and output in your answer.

6 Remarks

6.1 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.

6.2 Input and output to *lex*

Normally, the executable produced from the *lex* file will read from standard in and write to standard out. However, its exact behaviour is that it has two variables

```
FILE *yyin,*yyout;
```

that are by default set that way. You can open your own files and assign the file pointer to these variables.

6.3 Lex and Yacc

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

6.3.1 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`.

6.3.2 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.

6.3.3 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.

7 Examples

7.1 Text spacing cleanup

(This section illustrates the use of contexts; see section **??**.)

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 context comes in: a closing paren followed by punctuation is allowed, but followed by a letter it is an error to be corrected.

We can solve this problem without using context, but the *lex* code would be longer and more complicated. To see this, consider that we need to deal with spacing before and after a parenthesis. Suppose that there are m cases of material before, and n of material after, to be handled. A *lex* code without context would then likely have $m \times n$ rules. However, using context, we can reduce this to $m + m$.

7.1.1 Right context solution

Let us first try a solution that uses 'right context': 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.

7.1.2 Left context solution

Using left context, 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;}

%%
```

Exercise 4. Write a *lex* parser that analyzes text the way the TEX input processor does with the normal category code values. It should print its output with

- `<space>` denoting any space that is not ignored or skipped, and
- `<cs: command>` for recognizing a control sequence `\command`;
- open and close braces should also be marked as `<{>`, `<}>`.

Here is some sample input:

```
this is {a line} of text.
handle \control sequences \andsuch
with \arg{uments}.
   Aha!
this line has %a comment


x
y%
z

\comm%
and
```

Contents