

(f) `lex` In A Nutshell

Handout written by Julie Zelenski.

`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, `flex` is Paxson's improved version. We will use `flex` in the course, but almost all features we use are also present in the original `lex`.

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 specification files are traditionally identified by their `.l` extension. You invoke `lex` on a `.l` file and it creates `lex.yy.c`, a source file containing a wad of unrecognizable C code that implements a FA encoding all your rules and including the code for the actions you specified. 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
% gcc -o scan lex.yy.c -llex  compiles scanner, links with lex lib
% scan                   executes scanner, will read from stdin
```

Linking with the `lex` library provides a simple `main` that will repeatedly calls `yylex` until it reaches `EOF`. You can also compile and link the scanner into your project and use your own `main` to control when tokens are scanned. The `Makefiles` we provide for the projects will execute the compilation steps for you, but it is worthwhile to understand the steps required.

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 that you want copied verbatim to the generated C file, declarations are copied to the top of the file, user subroutines to the bottom. 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.

1ex 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, `1ex` 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	<code>[0-9]</code>	This means alternation of the characters in the range listed (in this case: <code>0 1 2 3 4 5 6 7 8 9</code>). More than one range may be specified, e.g. <code>[0-9A-Za-z]</code> as well as specifying individual characters, as with <code>[aeiou0-9]</code> .
Character exclusion	<code>^</code>	The first character in a character class may be <code>^</code> to indicate the complement of the set of characters specified. For example, <code>[^0-9]</code> matches any non-digit character.
Arbitrary character	<code>.</code>	The period matches any single character except newline.
Single repetition	<code>x?</code>	This means 0 or 1 occurrence of <code>x</code> .
Non-zero repetition	<code>x+</code>	This means <code>x</code> repeated one or more times; equivalent to <code>xx*</code> .
Specified repetition	<code>x{n,m}</code>	<code>x</code> repeated between <code>n</code> and <code>m</code> times.
Beginning of line	<code>^x</code>	Match <code>x</code> at beginning of line only.
End of line	<code>x\$</code>	Match <code>x</code> at end of line only.
Context-sensitivity	<code>ab/cd</code>	Match <code>ab</code> but only when followed by <code>cd</code> . The lookahead characters are left in the input stream to be read for the next token.

Literal strings	" x "	This means x even if x would normally have special meaning. Thus, " x* " may be used to match x followed by an asterisk. You can turn off the special meaning of just one character by preceding it with a backslash, .e.g. <code>\.</code> matches exactly the period character and nothing more.
Definitions	{ name }	Replace with the earlier defined pattern called name . This kind of substitution allows you to re-use pattern pieces and define more readable patterns.

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. The buffer is overwritten with each subsequent token, so you must make your own copy of a lexeme you need to store more permanently.
- `yylen` 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.
- `yyval` 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. This variable is declared to be of type `YYSTYPE`, and is usually a union of all the various fields needed for different token types. If you are using a parser generator (such as `yacc` or `bison`), it will define this type for you, otherwise, you must provide the definition yourself. Your scanner actions should appropriately set the contents of the variable for each token.
- `yyloc` is the global variable that is used to store the location (line and column) of the token. This variable is declared to be of type `YYLTYPE`. Again, the parser

generator can provide this or it may be your responsibility. Your scanner actions should appropriately set the contents of the variable for each token.

Example 1

Here is a simple and complete specification for a scanner that capitalizes all lowercase letters and echo all others unchanged:

```
%%
[a-z] putchar(toupper(yytext[0]));
.      ECHO;
```

The first %% marks the beginning of the rules section, the only section required in the input file. The pattern for the first rule matches any single lowercase letter and the associated action prints the letter after uppercasing it. The second rule matches any remaining character and uses the standard action `ECHO` (which just prints the character unchanged).

```
% lex uppercase.l
% gcc -o uppercase lex.yy.c -ll
% uppercase
... at this point anything you type, the scanner echos in UPPPERCASE
```

Example 2

The following `lex` input file has all three sections: a definitions section (when you can define substitutions, set up global variables, etc.), the rules section, and the user subroutines section (where you can define helper functions). This function includes its own `main` rather than using the one supplied by the `lex` library. What does this program do?

```
%{
    int numchar = 0, numword = 0, numline = 0;
}%
%%
\n {numline++; numchar++;}
[^ \t\n]+ {numword++; numchar+= yyleng;}
. {numchar++;}
%%
main()
{
    yylex();
    printf("%d\t%d\t%d\n", numchar, numword, numline);
}
```

To build and run this program:

```
% lex count.l
% gcc -o count lex.yy.c -ll
% count <count.l
% 216    32    17
```

Example 3

The following shows an excerpt of a scanner configured for use in a compiler. The global variable `yylval` is being used to store token attributes and pre-defined token codes are returned from each action to inform the compiler of the token type just scanned. There's obviously a lot more that needs to be here, but that's what you get to do for your first programming project!

```
%%
[+>]      { return yytext[0]; /* use ASCII code for single-char token */}
"for"     { return T_For; }
[0-9]+    { yyval.integerConstant = atoi(yytext);
            return T_IntConstant; }
[a-z]+    { yyval.identifier = strdup(yytext);
            return T_Identifier; }
```

Bibliography

T. Mason, D. Brown, lex & yacc. Sebastopol, CA: O'Reilly & Associates, 1990.