CIS 343 - Structure of Programming Languages

Nathan Bowman

Slides by (with slight modifications): Ira Woodring

Lexing and Regular Expressions

Lexing (Scanning)

We've learned so far that lexing is the process of tokenizing a string.

As we progress into parsing we will see what we do with these tokens, but for now we are just going to focus on the problem of tokenizing the string.

I want to stop here and make clear what we are about to do, because students often get confused. We are going to run a C program that is going to generate C code.

The C code that is generated is a library that can be used in our own code. It is a lexing library that will be customized to the language we are creating.

This will then be compiled from C into native code.

Lexing (Scanning)

It is perfectly ok to create your own lexer, however there are lexer generators that (in most cases) work very well. We will be using a generator called Flex.

Flex is a newer version of the generator Lex. Lex has been around for decades. While researching Flex and Lex you may see their names used interchangeably in some places.

Lexing (Scanning)

You may also see references to a program called Yacc or Bison. These are the corresponding parser generators for Lex/Flex. We will be talking about them later.

Lexing (Scanning)

Lex/Flex requires us to create a file of definitions for our tokens. We provide those definitions via the use of Regular Expressions (regex).

A regex is a language that defines a pattern.

The language of the regex is characters, literals, and character classes.

Regular Expressions

Regular expressions can be an extremely convenient and powerful way of accomplishing text processing tasks

They can also be a source of great frustration, so be careful

One issue that adds to the confusion around them is that there is no single universally excepted language for regular expressions. Most of them will be very similar to what we study here, but be careful to know the details of whatever language, tool, or library you are using

Regular Expressions

```
Some character: matches that character.
[abc]          : matches the character a, b, or c.
[a-z]          : matches any character a-z.
[^I-M]         : matches any character except I, J, K, L, or M.
regex*         : matches the regex preceeding the * zero or more t
regex+         : matches the regex preceeding the + 1 or more time
regex?         : matches the regex preceeding the ? 0 or 1 times.
regex{3,9}     : matches the regex if it occurs 3-9 times.
regex{7,}      : matches the regex 7 or more times.
```

Special characters:

```
^                : Matches the beginning of a line
$                : Matches the end of a line
.                : Matches any single character except \n
()               : Groupings
\character       : do not evaluate the character; match it
                   literally.  (useful if we want to match
                   a character that is also a regex character.).
```

Regular Expressions

They may also use character classes:

```
[:digit:] or \d   :    class for numbers 0-9.
[:lower:]         :    lower case letters.
[:upper:]         :    upper case letters.
[:space:] or \s   :    white space.
[:alnum:]         :    alphanumeric characters.

Many more.
```

Regular Expressions

So, if we wanted to match a variable name in a string, and our language dictates a variable is two characters, where the first character must be a $ and the second character must be an alphabetical character, we might use the following regex:

```
\$[A-Za-z]
```

Tokens

Recall that a token is a set of one or more lexemes.

For instance, we may have the following tokens defined:

```
[ \t\n]          WHITESPACE
\+               ADD_OP
[0-9]            DIGIT
```

Tokens

So, if you had a sentence in the language such as

```
3 + 9
```

The lexer would output

```
DIGIT WHITESPACE ADD_OP WHITESPACE DIGIT
```

(Though we would probably ignore the WHITESPACE).

Flex

Reminder: You could write C code to perfom lexing (Your book even has an example for a simple language)

However, machines are better at it

Flex will take in a description of what you want and write the C code for you

You then compile the C code into an executable, and, voila! You have a lexer

Flex

A Sample Flex file has three sections:

- **Definitions** - sources any symbols by including header files, or defining any macros.
- **Rules** - The regular expressions and the code to run when they are encountered.
- **Code** - C code that is copied directly into the generated file. This may be a main function, or any other function the rules may need to call when encountered.

The sections are separated with the %% characters.

```
%{
    #include <stdio.h>
%}

%%

[0-9]          { printf("DIGIT\n"); }
[A-Za-z]       { printf("ALPHA_CHAR\n");}
.|\n           ; // Ignore these chars!

%%

int main(int argc, char** argv){
  yylex();    // Start lexing!
  return 0;
}
```

A sample run of this code might look like the following:

```
h3ll0 w0rld.
ALPHA_CHAR
DIGIT
ALPHA_CHAR
ALPHA_CHAR
DIGIT
ALPHA_CHAR
DIGIT
ALPHA_CHAR
ALPHA_CHAR
ALPHA_CHAR
```

Flex

Try it yourself!

You will need to use flex soon, so get familiar with it

- Save the flex file specified above (e.g., `simple.lex`)
- Call flex on the file -- It will output `lex.yy.c`

```
flex simple.lex
```

- Compile generated C code with `-lfl` (links to flex library)

```
clang -lfl lex.yy.c
```

- Run as you normally would (e.g., `./a.out`)

Flex

As cool as this is, it is somewhat worthless...

For instance, we don't even preserve the lexemes we discovered.

Let's fix the file a bit. Flex includes some global variables. One is `yytext`. It holds the currently matched lexeme.

```
%{
    #include <stdio.h>
    void printLexeme();
%}

%%

[0-9]         { printf("DIGIT\t"); printLexeme();}
[A-Za-z]      { printf("ALPHA_CHAR\t"); printLexeme();}
.|\n         ;

%%
void printLexeme(){
        printf("(%s)\n", yytext);
}

int main(int argc, char** argv){
  yylex();
  return 0;
}
```

Now a run might look like this:

```
h3ll0 w0rld.
ALPHA_CHAR    (h)
DIGIT    (3)
ALPHA_CHAR    (l)
ALPHA_CHAR    (l)
DIGIT    (0)
ALPHA_CHAR    (w)
DIGIT    (0)
ALPHA_CHAR    (r)
ALPHA_CHAR    (l)
ALPHA_CHAR    (d)
```

Flex

It is not really the lexer's job to do any more than just output these tokens. However, if we wanted, we could use the lexer to make a more complete program.

For instance, what if we wanted to have a program that calculated a running tally of integers:

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    int total = 0;
%}

%%

[0-9]+          { total += atoi(yytext); printf("= %d\n", total);}
.|\n          ;

%%

int main(int argc, char** argv){
  yylex();
  return 0;
```

And a sample run:

```
12
= 12
124
= 136
1983
= 2119
20394885
= 20397004
a;slj;123
= 20397127
lajsl;fjalm,c1
= 20397128
```

Flex

Important: If the lexer finds more than one pattern to match, it will take the one that matches the most text. If it happens to find more than one match that are the same length it will take the one that comes first in the flex file.

Flex

There are many, many examples of Lex/Flex files online. Additionally, the manuals for Lex and Yacc can be found at http://dinosaur.compilertools.net/.

Let's look at some examples now:

Flex

Sample File: Roman Numerals

From http://home.adelphi.edu/sbloch/class/271/examples/lexyacc/romans.l

```
WS [ \t]+

%%
 int total=0;

I total += 1;
IV total += 4;
V total += 5;
IX total += 9;
X total += 10;
XL total += 40;
L total += 50;
XC total += 90;
C total += 100;
CD total += 400;
D total += 500;
CM total += 900;
M total += 1000;

{WS} |
\n return total;
%%
int main (void) {
   int first, second;

   first = yylex ();
   second = yylex ();

   printf ("%d + %d = %d\n", first, second, first+second);
   return 0;
```

Flex

```
$ ./a.out
VIII
IV
8 + 4 = 12
```

Flex

Sample File: See if you can tell me what it does!

```
int num_lines = 0, num_chars = 0;

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()
{
yylex();
printf( "# of lines = %d, # of chars = %d\n",
        num_lines, num_chars );
}
```

Flex

It implements some of the functionality of the `wc` command (it counts the number of characters and lines in a file!).

https://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html

Flex

Sample File: See if you can tell me what it does!

```
%%
[a-z]    { char ch = yytext[0];
           ch += 3;
           if (ch > 'z') ch -= ('z'+1-'a');
           printf ("%c", ch);
         }

[A-Z]    { char ch = yytext[0];
           ch += 3;
           if (ch > 'Z') ch -= ('Z'+1-'A');
           printf ("%c", ch);
         }
%%
```

Flex

Implements a Caesar Cipher!

(From: http://home.adelphi.edu/sbloch/class/271/examples/lexyacc/caesar1.l)

Flex

Reminder - to compile a lex/flex file we first run the file through lex. In the following examples our flex file is called `my_lang.lex`. Lex will create a file called `lex.yy.c` that we can then compile.

```
flex my_lang.lex
clang lex.yy.cc -lfl
```

(Note! These are the instructions for EOS. If you are on an OSX machine they may be slightly different; for instance the clang flag may be "-ll").

Flex

I suggest you practice with flex on your own a bit to get a feel for it.

Some good warm-up exercises are to create a lex file that scans for:

Names - Must start with a capital letter and only include lowercase letters afterward.

Phone Numbers - Must be of the form (xxx)xxx-xxxx, where x is a digit.

Day of the Week - Must be the complete day of the week name (note! They all in in 'day'...)