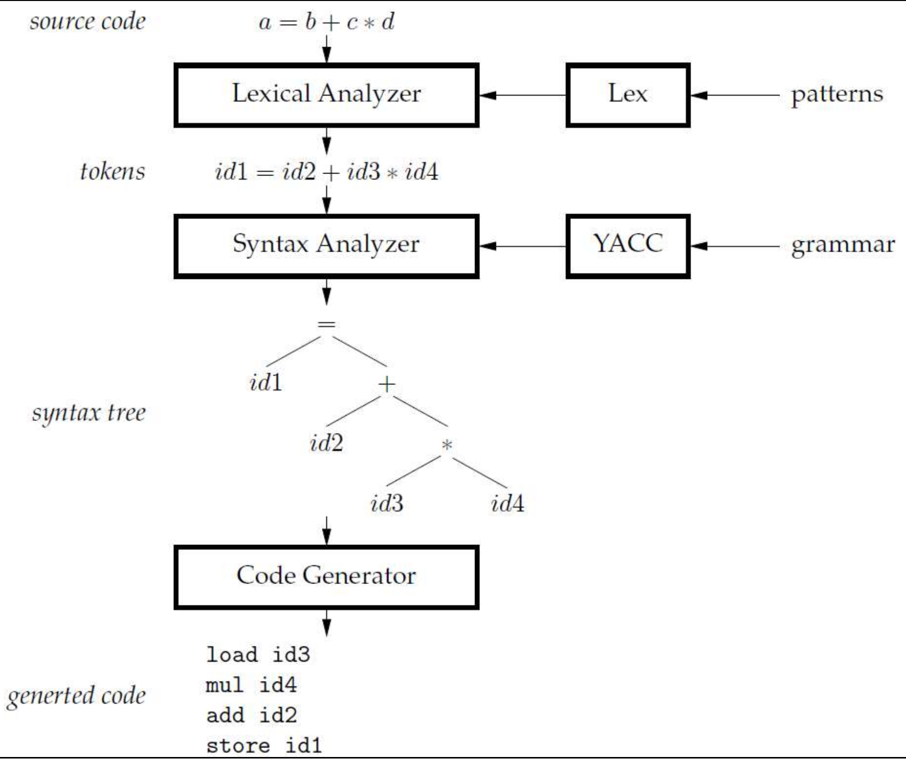
LEX Programming

Lex and The Big Picture

- A **Scanner** is the first stage in compilation is lexical analysis
    - the process through which a input stream of text is converted in a output stream of tokens.
- The program Lex ("Lexer")
    - Generates C code for the Scanner
    - is a program that takes a stream of characters as its input, and whenever it sees a group of characters that match a key (or reg exp), takes a certain action.
    - we have neatly 'Tokenized' the input meaning each part of the configuration file has been matched, and converted into a token.
    - accepts a text file that is written in a specific format and generates C code that can be incorporated in your application.

Where Lex fits in



Programming in Lex

- Files are in .l extension
- Lex is already on GL
- looks a lot like C, because it uses C!!!

The physical file (".l") breakdown

- ***overall no tabs to indent line unless in main!!!***

Lex Example

```
/* Program header, name, description, email, etc… */

/* declarations */
%{
#include <stdio.h> // needed for "printf" below
%}
/* rules */
%%
stop    printf("Stop command received\n");  /* literal "stop" in input */
start   printf("Start command received\n");
%%
/* code (main) */
int main()
{
     yylex();
     return 0;
}
```

- definitions
  - includes
    - for C library files
    - you will see some examples with #include <stdio.h>
      - used for "print" command to print to the screen (in C)
  - declaration of variables
    - counter (count the # of tokens)
    - "global"
  - definition and <SYMBOLS>
    - <SYMBOLS> just like in Grammars

    - programmer created C functions with bodies
- Lex macros
    - just like in a Grammar
    - not required
    - form is
        - <SYMBOL> then the Reg Ex. to find it
        - replaces some below
    - not in a lot of examples (if looking on the web)
    - helpful though!!
- rules
    - A number of combinations of pattern and action
        - patterns
            - Reg Ex.
        - actions
            - in C code
            - all C code can be used!!
            - if the action is more than a single command it needs to be in braces.

Rules' pattern and action setup

```
/* rules */
%%
stop                        printf("Stop command received\n");
start                       printf("Start command received\n");
[0123456789]+               printf("NUMBER\n");
[a-zA-Z]+                   printf("WORD\n");
```

- code
    - must call yylex
        - which is the lexical analyzer and ***parser that is built from the rules!!***
    - a default main is used which only calls yylex.
        - I personally use for simplicity

Compiling and running

- It is a two step process as usual
    - Edit and compile
    - Run and Test

Editing and Compiling

```
[slupoli@linux2 Lex]$ emacs example1.l      (last letter is an L)
[slupoli@linux2 Lex]$ lex example1.l
[slupoli@linux2 Lex]$ cc lex.yy.c -o example1 -ll
```

Running a Lex Script

Using the keyboard as input

```
[slupoli@linux2 Lex]$ ./example1
```

The program will continue to RUN until you hit <CTRL> <D>
The program is running, it's waiting for input from the keyboard

Using a file (with data already in it) as input

```
[slupoli@linux2 Lex]$ ./example1 < test.txt
```

Using a file (with data already in it) as an argument

```
// inside the main

…
int main(int argc, char *argv[])
{
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

- behind the scenes, a file named lex.yy.c is created
    - it's a C version of the .l file we created
    - will be used next week in Parsing (YACC)

Comments in a Lex file

- same as C/Java /* */
- BUT NOT //
    - UNLESS within a section!!!

String Matching examples

- learn the best from examples and their explanations

Lex Example 1

```
/* Whenever the 'stop' key is encountered in the input,
the rest of the line (a printf() call) is executed.
Besides 'stop', we've also defined 'start', which otherwise
does mostly the same. */

/* declarations */
%{
#include <stdio.h> // needed for “printf” below
%}
/* rules */
%%
stop    printf("Stop command received\n");
start   printf("Start command received\n");
%%
/* code (main) */
int main()
{
     yylex();
     return 0;
}
```

Lex Example 2

```
/* This Lex file describes two kinds of matches (tokens): VARIABLESs and
NUMBERs. It also tracks the number of total tokens counted.*/

/* declaration */
%{
#include <stdio.h>
int tokenCount=0;
%}
/* rules */
%%
[0123456789]+      { printf("%d NUMBER \"%s\"\n", ++tokenCount, yytext); }
[a-zA-Z]+          { printf("%d WORD \"%s\"\n", ++tokenCount, yytext); }
[a-zA-Z][a-zA-Z0-9]*   { printf("%d OTHER \"%s\"\n", ++tokenCount,yytext);
}
%%
/* code (main) */
int main()
{
        yylex();
        return 0;
}
```

A word about the Rules Section

- the order does matter if the Reg Ex. is close!!
- got the warning
  - warning, rule cannot be matched

Reg. Ex. Order does matter in Lex!!

Worked	Didn't Work!!

`[a-zA-Z]+ printf("WORD\n");`
`[0123456789]+ printf("NUMBER\n");`
`[a-zA-Z][a-zA-Z0-9]* printf("VARIABLE\n");`
	`[a-zA-Z][a-zA-Z0-9]* printf("VARIABLE\n");`
	`[0123456789]+ printf("NUMBER\n");`
	`[a-zA-Z]+ printf("WORD\n");`

// Why didn't this work??

String Matches to Reg Ex.

- need to review some of the symbols
- most of your exercises will require matching

More Common String Matches		
NUMBER	[0123456789]+ OR [0-9]+	a sequence of one or more characters from the group 0123456789
Variable(WORD)	[a-zA-Z][a-zA-Z0-9]*	The first part matches 1 and only 1 character that is between 'a' and 'z', or between 'A' and 'Z'. In other words, a letter. This initial letter then needs to be followed by zero or more characters which are either a letter or a digit.
Word	[a-zA-Z]+	
Individual Chars	\” \{ ; \n	You can specify individual characters to invoke a response
whitespace	[\t]+	
Choice	on|off	using an exact text match
single letter	[a-zA-Z]	
single digit	[0-9]	
punctuation	[,.:;!?]	
decimal	[0-9]*"."[0-9]+	[0-9]*"."[0-9]+
signed int	[-+]?[1-9][0-9]*	a (possibly) signed integer:

Regular Expressions	
c=character, x,y=regular expressions, m,n=integers, i=identifier.	
.	matches any single character except newline
*	matches 0 or more instances of the preceding regular expression
+	matches 1 or more instances of the preceding regular expression
?	matches 0 or 1 of the preceding regular expression
|	matches the preceding or following regular expression
[ ]	defines a character class
( )	groups enclosed regular expression into a new regular expression
“…”	matches everything within the " " literally
x|y	x or y
{i}	definition of i
x/y	x, only if followed by y (y not removed from input)
x{m,n}	m to n occurrences of x
ˆx	x, but only at beginning of line
x$	x, but only at end of line
"s”	exactly what is in the quotes (except for "\" and following character)

Using Variables and Macro Examples

- variables are declared in C in the definitions (declaration) section, but both rules and main have access
- here the MACRO (regular definitions) is used to shorten the code

Variables and Macro Examples 3

```
/* reads and interprets Roman numerals. */
/* Declarations */
%{
        int total=0;
%}

/* Lex Macros */
WS      [ \t]+

/* rules */
%%
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;
%%

/* code and main() */
int main ()
{
   int first, second;
   first = yylex ();
   second = yylex ();

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

Programmer Functions Example

- The third part can be used to provide auxiliary user functions that the pattern matching rules use; these will be simply copied along to the generated code

Functions and Macro Example 4

```
/* This example uses a declared <SYMBOL> to invoke a response. This is nicer and
reduced code */

/* declaration */
%{

void founddigit(void) { printf("Found a digit"); }

%}

/* Lex Macros */
Digit [0-9]

/* rules */
%%
{Digit} { founddigit();} // so once a digit is found it calls the function
%%
/* code (main and more) */

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

Useful LEX member functions and variables

- can be referred anytime in the .l file

LEX Built in Member Functions and Variables

returns	name	function
char *	yytext	the string being matched (\n or null terminated)
int	yyleng	The length of the matched string
FILE *	yyin	the input file (defaults to stdin (input file))
FILE *	yyout	the output file (defaults to stdout (output file))
	yylval	an additional variable for communication between functions

LEX Built in Member Function Example 5

```
%{
#include<stdio.h>
%}
%%
[0-9]+ {printf("Number %s\n", yytext);}
%%
int main(int argc, char* argv[])
{
        yyin = fopen("input", "r");
        yylex();
        return 0;
}
```

LEX Built in Member Function Example 6

```
%{
#include<stdio.h>
%}
%%
[0-9]+ {fprintf(yyout, "Number %s\n", yytext);}
%%
int main(int argc, char* argv[])
{
        yyin = fopen("input", "r");
        yyout = fopen("out", "w");
        yylex();
        return 0;
}
```

LEX Built in Member variable Example 7

```
%{
#include<stdlib.h>
#include<stdio.h>
%}
number [0-9]+
%%
{number} {printf("Found : %d of length %d",atoi(yytext),yyleng);}
%%
int main()
{
yylex();
return 0;
}
```

C's helpful functions

- ato??
    - #include <stdlib.h>
    - converts text to a literal number
    - will need to do this for any number manipulation
    - atof = to float
    - atoi = integer
    - atol = long
    - http://www.cplusplus.com/reference/clibrary/cstdlib/atoi/
- toupper
    - #include <ctype.h>
    - returns an uppercase character
    - toupper(c)
    - http://www.cplusplus.com/reference/clibrary/cctype/toupper/

Lex – Real world Problem

- remember this is only A PART of the whole process
- YACC will be use to parse the syntax later given
- this is to do simple calculations
- uses MACROS, atof

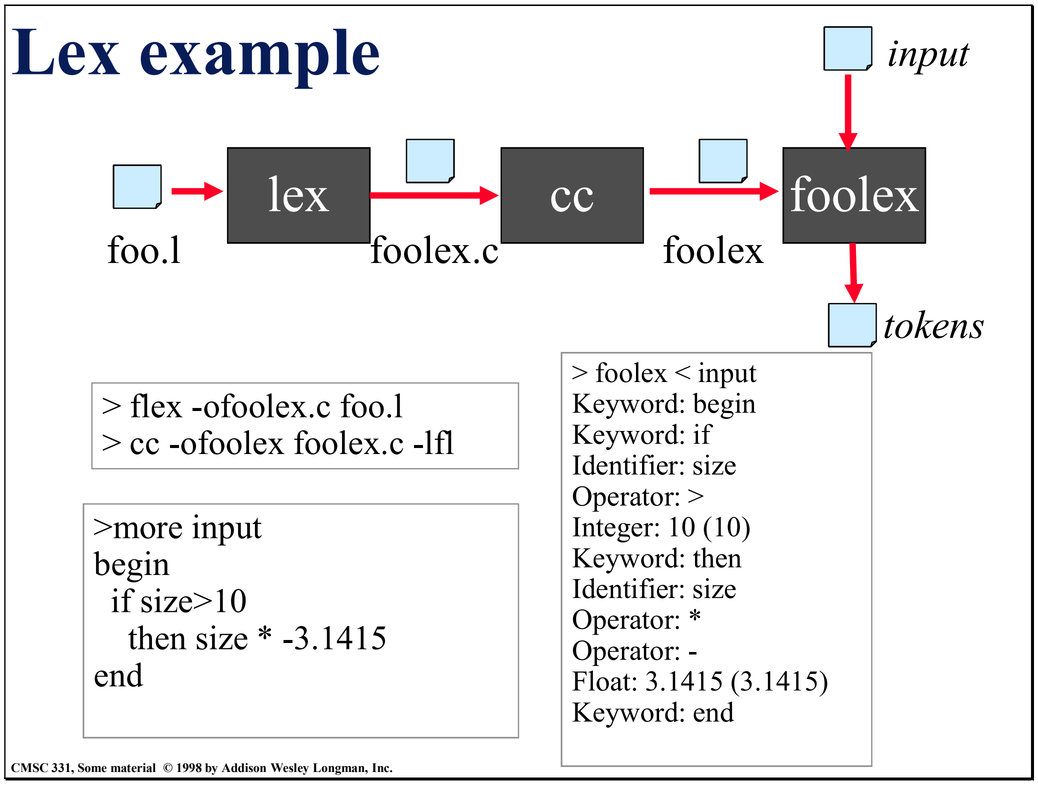
LEX Calculator – Setting up the Scanner

```
%{
#include "global.h"
#include "y.tab.h"
#include <stdlib.h>
%}
white [ \t]+
digit [0-9]
integer {digit}+
real {integer}("."{integer})?
%%
{white} { }
{real} {yyval=atof(yytext);
return (NUMBER);}
"+" return (PLUS);
"-" return (MINUS);
"*" return (TIMES);
"\n" return (END);
```

Construct using Lex a program for translating all letter appearances in a text file into uppercase. (Work as a group)

So after all this, what??

- Lex is a State-Machine Generator for a
- So far, we've discussed lex as if it interprets the regular expressions in our lex-specification at run-time
- Lex is in reality a C-code generator, more like a compiler than an interpreter. Our Lex Specification is "compiled" into C-code.
- The C-code Lex generates is in the form of a state-machine, which processed input character-by-character.


Lex example
input
lex
cc
foolex
foo.l
foolex.c
foolex
tokens
CMSC 331, Some material © 1998 by Addison Wesley Longman, Inc.

```
> flex -ofoolex.c foo.l
> cc -ofoolex foolex.c -lfl
```

```
>more input
begin
  if size>10
    then size * -3.1415
end
```

```
> foolex < input
Keyword: begin
Keyword: if
Identifier: size
Operator: >
Integer: 10 (10)
Keyword: then
Identifier: size
Operator: *
Operator: -
Float: 3.1415 (3.1415)
Keyword: end
```

Lex Tutorial
http://dinosaur.compilertools.net/lex/index.html
http://www.cs.rug.nl/~jjan/vb/lextut.pdf
http://ds9a.nl/lex-yacc/cvs/lex-yacc-howto.html
http://www.tldp.org/HOWTO/Lex-YACC-HOWTO-3.html

Lex Unix Manual Page
http://plan9.bell-labs.com/magic/man2html/1/lex

Other ressources
http://perso.ens-lyon.fr/bogdan.pasca/teaching/projetCompilation/year2009-2010/td/tp01.pdf
http://www.cs.unb.ca/~bgn/courses/cs4905/Examples/LexExample/index.html
http://www.iis.ee.ic.ac.uk/yiannis/lp/LPLecture6bw.pdf
http://luv.asn.au/overheads/lex_yacc/lex.html

LEX Program Examples:
http://www.inf.unibz.it/~fillottrani/Introduction%20to%20Lex.pdf
http://www.linuxquestions.org/questions/linux-software-2/lex-and-yacc-programming-in-linux-132377/

C Programming Stuff
http://www.cplusplus.com/reference/clibrary/cctype/toupper/
http://www.cplusplus.com/reference/clibrary/cstdlib/atoi/