lex and tokenizing

- lex (lexical analyzer) allows us to describe the tokens of a language using regular expressions
- It also allows us to specify what information to record and other actions to take once a token is identified as a specific type
- The actions taken and information recorded is done using C (e.g. we can use C functions, structs, data types, etc)
- Our lex tokenizing code will usually be embedded into yacc to generate a full analysis tool for a specified language

lex/yacc sequence

- To use lex/yacc to produce a program to analyze source code in language L, the steps are as follows:
  - Create a file, e.g. L.lex, containing L’s tokenization rules
  - Create a file, e.g. L.yacc, containing L’s parsing rules
  - Run lex on L.lex to produce lex.yy.c
  - Run yacc on L.yacc to produce y.tab.c
  - Compile the two .c files to produce our tool
  - Gcc y.tab.c lex.yy.c -o analyzeL
- We can now use the tool on source code written in L, e.g.

```
./analyzeL < someLsourcecode
```

Lex file organization

- A lex file uses the extension .lex, and is divided into three core sections, separated from one another with %%
    - The first section is used to define useful regex character sets (e.g. Alpha, Digit, etc), for C library includes, prototypes, global vars, and typedefs
    - The second section is used for the actual token rules and code to be applied when a token is identified
    - the final section contains regular C code, typically the implementations for any functions prototyped in the first section
- Syntax gotcha: when using C-style comments in your lex file, don’t start them on the first character of the line, put space(s) first

just the .lex file layout...

```
 /* declarations  */
%{
#include <stdio.h>
#include “y.tab.h”
int yywrap();
int yyerror(char *s);
%}
 /* char sets go here */
%%
 /* token rules go here */
```

```
%%
 /* C function bodies */
int yywrap()
{
   return 1;
}
int yyerror(char *s)
{
   return 1;
}
```

Notes on the C files/functions

- y.tab.h will be auto-generated for us by lex
- yywrap is automatically called at the end of processing and is generally used to clean up any dynamically-allocated memory and call any summary routines we want to run
- yywrap must return 1 if processing completes normally
- yyerror is called in the event of tokenizing errors, and gets passed a string representing the error information – typically it displays the string and returns 1

The declarations section

- In the `%{ ... %}` we can include any additional C declarations (e.g. data types, constants, variables, function prototypes, etc)
- The items declared can be used in both our .lex and .yacc
- For each prototype, we must include an implementation of that function at the end of the file (with `yyerror` and `yywrap`)
- In the declarations section, typically before the `%{`, we can also define character sets of interest, e.g. alphabetic, digits

```
Alpha [a-zA-Z]
Digit [0-9]
```

The token rules section

- This is the section after the first %%, and contains the rules describing each token
- Each token rule has a regular expression defining what makes up a valid token of that type, plus a block of C code (in { }) dictating what to do with that token type
- The block of C code associated with the token must return a constant identifying that token, either a character (e.g. `‘x’`) or a token named in our .yacc file, e.g. `STRING`

(our .yacc and .lex files wind up depending on each other)

Token rule ordering

- Rules are processed in the order they appear in the .lex file, and the first matching rule is applied
- Thus if tokens overlap we want to be careful about our rule order
- e.g. suppose we have a keyword named ‘while’, and our identifiers are any non-empty string of alphabetic characters
- We want our .lex file to check for the keywords first, so it doesn’t accidentally treat `while` as an identifier

Token rule example

- Our regular expressions can make use of the character sets we described at the top of the file, and the block of C code can make use of the items we declared inside the `%{ ... %}`
- Example: suppose IDENTIFIER is listed as a valid token name in our .yacc file, and we want identifiers to be any sequence of one or more alphas
- For now, our rule just recognizes the pattern as an identifier:

```
({Alpha})+    { return(IDENTIFIER); }
```

Working with the token information

- For our identifier, if we were building up a symbol table of information we probably want to know more than just “we saw an identifier” ... perhaps what was the actual identifier (`x`, `foo`, ...?), perhaps what scope we were in when we saw it, etc
- Within the code section for the token, a variable named yytext is available, and holds the actual string for the token (e.g. `“x”`, `“foo”`, `“==”`, `“99.4”`, etc)
- Another variable, named `yylval`, is available to store things we want to remember about that specific token, to use later (the data type for `yylval` for each token type gets specified in the `.yacc` file)

Example: using yylval, yytext

- Suppose for our identifier token we want our code to store the actual text (e.g. “foo”)
- When we enter the code block the text is in yytext, but that will get overwritten when the next token gets read
- Each individual token has its own yylval however, so we’ll copy the content from yytext to yylval (assume the .yacc file set up the yylval for identifiers as a character array)

```
({Alpha})+ { strcpy(yylval, yytext);
             return IDENTIFIER; }
```

More on code blocks and yylval

- For each token type, the .yacc file can specify any C data type for its yylval: ints, chars, structs, etc
- In our .lex code block for the token type we need to make sure we treat the yylval as the matching type
- Our .lex code block can also make use of any of the global variables we declared in the top declarations section, updating them based on the current token type and content of yytext

Example: tracking row/col in file

- Our token rules could allow us to keep track of what row/column each token starts at in the file
- In the declarations section up top, create and initialize global variables for row and column, e.g.

```
  int row = 0;
  int col = 0;
```

- With each token type, add the length of the token to col:

```
({Alpha})+ { col += strlen(yytext); return IDENTIFIER; }
```

- Add a rule for newlines to reset col to 0 and increment row

```
  ([\n]) { row++; col = 0; }
```

Complexity of our token rules

- We can have as many token rules as we like, and the associated code can be as complex as desired
- Perhaps we make the yylval for each token type a struct, with integer fields for the row/column it was encountered on, a character array for the actual text for the token, an identifier for which scope level it was found in, an identifier for which row of a global symbol table it is recorded in, ... the possibilities are endless

The final section, C functions

- The section after the second set of %% must contain the implementations of all the C functions we prototyped
- These are written as normal C functions, with the one gotcha mentioned earlier about comments (indent them at least one space from the start of the line)