Introduction
In this post I will talk a little about lexical analysis. For more information you can check out Dr. Lusth’s notes on the subject here.

Lexical analysis is the process of constructing lexemes and tokens* from a sequence of characters. One way to think of this is that we take the contents of a file, split it into strings where each string represents some basic idea, such as identifier, operator, or left parenthesis. We call these generated strings lexemes.

* The raw text is the lexeme and tokens are tuples containing the lexeme type and lexeme value.

In addition to creating these strings we also pair them with their type resulting in a token. For instance you might generate a token with type identifier for the lexeme value printf

\[<\text{IDENT,printf}>\]

Some lexical analysis is so common that some languages, in some limited fashion, provide it natively. For instance, Java has the Scanner class and C (and automatically C++) has the strtok function. These two examples limit users to constructing lexemes, and not tokens, based on delimiters. For instance the default delimiter is whitespace and therefore you return all contiguous string of non-whitespace characters.

For the purposes of this class we need something more powerful. For instance, given the input below how should we tokenize it?

```plaintext
int main() {
    print("Hi");
}
```

Would you expect these lexemes (types omitted on purpose)

\[<\text{int}>\ <\text{main}>\ <\{>\ <\text{print}>\ <"Hi">\ <\}>\]

or these lexemes (types ommitted on purpose)

\[<\text{int}>\ <\text{main}>\ <\{>\ <\text{print}>\ <\{>\ <"Hi">\ <\}>\ <\};>\]

In the first example we see something that Scanner or strtok would return; while the second example would require a more complicated solution, which you will end up creating in this course.
A program which is dedicated to creating lexemes, and additionally tokens, from some input is called a lexer. To determine how a lexer constructs lexemes we need to look at regular expressions.

**Regular Expressions**
A regular expression is a string which can be used to match other strings. This might sound confusing so we provide an example below which matches all strings of upper and lowercase letters of length at least 1

```
[a-zA-Z]+  
```

The brackets and plus have special meanings which are described in the list of common operations below

- `[]` - character classes (`[a-z]` matches all lowercase letters)
- `()` - grouping
- `+` - 1 or more times
- `*` - 0 or more times
- `?` - 0 or 1 times
- `^` - Depending on context, beginning of line, or not. `[^s]` means any character that is non-whitespace
- `\s` - Whitespace characters, equivalent to `[ \t\n\r\v\f]`

For instance the regular expression below would match an identifier in C

```
[_a-zA-Z][_a-zA-Z0-9]*
```

which matches strings that begin with a letter or underscore and is followed by 0 or more letters, digits, or underscores.

If you want to match strings which contain the symbols in those table above you would need to escape the character by preceding it with a backslash. This also means that if you want to match a string with backslashes you would need to write `\` to match a single backslash.

**Lexemes and Tokens**
Now that we have talked a little about regular expressions we can talk about how they relate to lexical analysis. Usually, we want to associate a lexeme type with a regular expression. For this post we use the notation:

```
TYPE = RegEx
```

Using the above definition and regular expressions we can now examine the two examples we gave above. We can that for the first example the regular expression is any string of non-whitespace of length at least 1 and for lack of a better name, type LEXEME:
LEXEME = [\s]+

and the second is much more complicated. We match quoted strings, identifiers, periods, parenthesis, braces, and the semicolon

STRING = "[a-zA-Z]*"
IDENT = [a-zA-Z]+
PERIOD = \.
L_PAREN = \\(
R_PAREN = \\)
L_BRACE = {
R_BRACE = }
SEMICOLON = ;

From this we can see that the lexeme types with the tokens from the second example would have resulted in

<IDENT,int> <IDENT,main> <L_PAREN> <R_PAREN> <L_BRACE> <IDENT,print> <L_PAREN> <STRING,"Hi"> <R_PAREN> <SEMICOLON> <RIGHT_BRACE>

Notice that for some of the tokens we don't include the lexeme value since the lexeme type is enough to generate the value if needed.

Lexer (theory)

Now that we know how regular expression map to lexeme types and generate tokens let's take a look at how it can be used in a lexer. Conceptually what a lexer does is it reads the text of a file and tries to match the longest sequence of characters it can to a lexemes regular expression. For instance 1234.6 we could match the characters 123 to a lexeme of type INTEGER and 4.6 to a lexeme of type REAL. However, we want to match the longest sequence of character possible so we match 1234.6 to REAL.

More concretely it starts at a position in a file, either at the beginning or after just reading a lexeme, and performs the following

NEXT(DATA,START):
   REGS = ALL_REGS
   CURR = START
   S = ""
   WHILE CURR < LEN(DATA):
      S = CONCAT(S,DATA[CURR])
      FOR R in REGS:
         IF NO_CHANCE(R,S):# There is no chance that R can match S
            REGS = REGS - {R}
      IF REGS == []:
IF ACCEPTS(R,S[start:curr-1])# The regular express R matches S[start:curr-1]
    RETURN (R,S[start:curr-1])
ELSE
    ERROR("Unknown Lexeme type")
CURR += 1
IF SIZE(REGS) == 1# If a lexeme is accepted by two regular expressions you will have a bad
day
    IF ACCEPTS(REGS[0],S)
        RETURN (REGS[0],S)
    ERROR("Unknown Lexeme Type")
You should be able to follow most of what the psuedocode is doing except for NO_CHANCE and
ACCEPTS.

What NO_CHANCE, and similarly ACCEPTS, does is checks to see if given this string would it
ever be possible to match this regular expression. To help with this lets look at three regular
expressions and some input.

R1 = [a-z]+[0-9]?[A-Z]+  
R2 = [a-zA-Z]+           
R3 = [a-z]+               
S1 = helloEveryone

"h"
    S1 -> has a partial match [a-z]+[0-9]?[A-Z]+  
    S2 -> has a match [a-zA-Z]+               
    S3 -> has a match [a-z]+               

"he"
    S1 -> has a partial match [a-z]+[0-9]?[A-Z]+  
    S2 -> has a match [a-zA-Z]+               
    S3 -> has a match [a-z]+               
    ...  

"helloE"
    S1 -> has a match [a-z]+[0-9]?[A-Z]+ ([0-9] is optional remember!)  
    S2 -> has a match [a-zA-Z]+               
    S3 -> cannot match, remove S3 from REGS

"helloEv"
    S2 -> has a match [a-zA-Z]+               
    ...  

"helloEveryone"
    S2 -> has a match, on next iteration we break out and return (S2,"helloEveryone"

Lexer (practice)
Now that you see some idea of what is going on conceptually, how would you implement this in code? There are general purpose programs which do work with regular expressions to generate lexers, such as lex and Flex. Luckily you don't have to work with regular expressions if your lexeme types are simple. For instance if you recall identifiers in C: begin with either a letter or underscore followed by 0 or more letters, underscores, or digits.

How would you implement this in code? You would not have to create a regular expression but rather just check to make sure you meet the requirements listed

```c
// Assumption: we already validated the first characters
Token* readIdentifier(char* data, int* pos, int N) {
    int start = pos[0];
    ++pos[0];
    while(isLetter(pos[0] < N && (data[pos[0]]) || isDigit(data[pos[0]]) || isUnderscore(data[pos[0]])) )
    {
        ++pos[0];
    }
    int length = (pos[0] - 1) - start;
    return createToken(IDENTIFIER, data + start, length);
}

Token* readLexeme(char* data, int* pos, int N) {
    ... 
    else if(identifierWaiting(data,pos,N)) return readIdentifier(data,pos,N);
    ...
}
```

Some of the more astute readers might ask the question: what if I want to have a lexeme type for each keyword in my language? Luckily most keywords match identifiers, therefore when you read in an identifier just check to see if it is a keyword and change the type from IDENTIFIER to your keyword type, for instance FOR or IF.

From the above paragraph, and possible regular expression matching example, you might also ask the question: what do I do if it is possible for two lexeme types to start with the same value? You will need to read until there is a difference in the two. This can happen if you want a lexeme type INTEGER and REAL. Both can begin with a digit but as you reading you might realize you are reading a REAL instead of an INTEGER or vice versa. To handle this you can have a generic readNumber function which handles the cases inside.

Comments
In your language you are required to have comments. The easiest comments to handle are single character comments such as the # symbol. While you are reading whitespace if you see the # symbol read until the end of line. If you want to support more complicated symbols such as // or /* ... */ you will need to do some extra work. The best way to handle this is to have a check in the readLexeme function which check for the first character, then checks the second and handles the remaining characters as necessary. Make sure you can handle escaped characters if you do...
multiline comments.

Error Reporting
One issue that you will be required to deal with is error reporting (in both your lexer and parser). What happens if you read an unsupported character? You should be able to display the line and column that the error appears. Additionally, what happens in your parser when you find an operator, say division, and the left value is an integer and the right is an array? You should be able to print out the file, line, and column this error appears on.

The best way to handle this is to have your lexer keep track of the current line and column. When you read a lexeme save the file, line and starting column with your lexeme. This can then be used by your parser to display with error messages. In addition, can also store extra information such as the starting position of the line the lexeme is on if you wish to print the entire line.

Case Study
Let's look at a lexer which supports the following lexeme types:

```plaintext
IDENT       = [a-zA-Z]+  
L_PAREN     = \(  
R_PAREN     = \)  
```

The three lexeme types we have are left parenthesis, right parenthesis, and identifiers. For the parenthesis it should be obvious and for identifiers we accept 1 or more letters.

Let us look at some C code to see how we might structure our code to handle these requirements. There will be two methods that are called the most by the user, lexerHasToken and lexerNextToken. The lexerHasToken function returns true if there are, possibly, more Tokens to be read. The lexerNextToken function tries to read the next token and prints an error if one occurs, otherwise it returns a new Token.

The code for lexerHasToken is below and returns true if there is (possibly) another lexeme waiting to be read

```c
int lexerHasToken(Lexer* l) {  
   // We ignore whitespace  
   while(lexerHasMore(l) && isSpace(lexerPeek(l))) {  
      lexerNextCharacter(l);  
   }  
   return lexerHasMore(l);  
}
```

In this function we have four helper functions that you will have to create.

lexerHasMore takes in the lexer and returns true if there is another character in the file.isSpace
return true if the character supplied is whitespace

lexerPeek returns the current character without moving to the next

lexerNextCharacter moves to the next character in the file

In addition to handling whitespace this is where you would handle comments. This can be implemented by having a loop that has two jobs, one is to remove whitespace, the other is to remove comments. If you look at the attached code I have provided you can see that how I perform these operations. Unfortunately, you will have to change how this would be done in your lexer because you are only allowed to read one character at a time from the file.

The code below is for lexerNextLexeme and returns the next lexeme, unless there is an error

```
Token* lexerNextToken(Lexer* l) {
    if(lexerPeek(l) == '(') {
        return lexerReadLeftParen(l);
    } else if(lexerPeek(l) == ')') {
        return lexerReadRightParen(l);
    } else if(isLetter(lexerPeek(l))) {
        return lexerReadIdentifier(l);
    } else {
        lexerError(l);
        return NULL;
    }
}
```

We can see there are five new functions

lexerReadLeftParen reads in the left parenthesis and construct a new Token for it, this will move to the next character in the file

lexerReadRightParen reads in the right parenthesis and construct a new Token for it, this will move to the next character in the file

isLetter checks to see if the current character is a letter as defined by the regular expression \[a-zA-Z\]

lexerReadIdentifier will read in 1 or more characters and construct a new Token with type IDENT, this will move over 1 or more characters in the file

lexerError will print out any information we have about the error, including possibly the file, line, and column along with an error message

If you wanted to add a new lexeme type this is where you would have to add a new else if statement. For instance adding a new lexeme type for the + operator would add

```
} else if(lexerPeek(l) == '+') {
    return lexerReadOperator(l);
}
```

If you are interested on how we handle identifiers you can either check out the code attached or scroll up and examine the code for parsing C identifiers.

Extra Fun
Here is some things you can try to add to my lexer, in order of easier to difficult
   Handle operators   Handle integers   Handle characters and strings   Handle reals   Realize you can handle scheme

Attached Code
The attached code can be compiled using either the makefile, or with the command

gcc lexer.c -o lexer

You can then use the program with the command

./lexer filename

I have attached three sample files, due to attachment limitation, you can run on the lexer. An additional input which can be used is

((()aa{

File Attachments
1) makefile, downloaded 4 times
2) lexer.c, downloaded 4 times
3) correct1.txt, downloaded 5 times
4) wrong1.txt, downloaded 4 times
5) correct2.txt, downloaded 4 times