Assignment 1
Rough Draft II (subject to changes)

Introduction
For your second programming assignment, you are to implement a infix expression calculator using a stack and a queue.

I/O
Your executable must be named *matilda*. The executable reads in a series of expressions from a file and will produce, on *stdout*, the result of the computations. Here is an example invocation:

```
$ echo 999 \* 8888 \; > items
$ matilda items
8879112.000000
$
```

where $ is the system prompt. The file to be processed (*items* in the example) is a free-format text file. That is, the numbers, variables, operators, and semicolons found within are separated by arbitrary amounts of whitespace (i.e. spaces, tabs, and newlines) and every line ends with a newline.

The executable must handle the following options:

<table>
<thead>
<tr>
<th>option</th>
<th>example</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>-v</td>
<td>matilda -v</td>
<td>give author's name and exit</td>
</tr>
<tr>
<td>-i</td>
<td>matilda -i FILENAME</td>
<td>print the original input to evaluating the expression</td>
</tr>
<tr>
<td>-d</td>
<td>matilda -p FILENAME</td>
<td>print the postfix conversion of the last infix expression</td>
</tr>
<tr>
<td>-b</td>
<td>matilda -b FILENAME</td>
<td>print the BST holding variable values</td>
</tr>
</tbody>
</table>

If multiple options are given, print the -i output before the -p output and print the -p output before the -b output.

Here are some example invocations using options:

```
$ matilda -v
Alyssa P. Hacker
$ matilda -i -p -b testfile
var a = 3 ;
var b = 4 ;
b * a ;
b a * ;
[a = 3 [b = 4]]
12.000000
$
```

The last expression shown, $b \ a * ;$, is the postfix conversion of the expression entered, $b \ast a ;$. Here is another example:

```
$ echo var a = 3 \; > expr
$ echo var b = 4 \; >> expr
$ echo b \* a \; >> expr
$ matilda -p expr
b a * ;
12.0000000
$
```

Here is a program that features some handy-dandy option-handling code that you may use verbatim without credit: *options.c*
Expressions
Expressions will be composed of literals (integers, real numbers, and quoted strings), variables (tokens beginning with an alphabetic character), and operators (\(= \div - * / \% ^\)), plus parentheses. All tokens will be surrounded by whitespace. For example, the expression: \((x+1) \times 2\); is illegal and should be written as \((x + 1) \times 2\); to be proper.

A token in the input stream can be identified as follows:
- semicolons and parentheses will appear in tokens by themselves
- a real number will have a . in it
- a integer will start with a digit or a minus sign and not have a . in it
- a string will start with a double quote and end with a double quote; there may be whitespace within the string
- a variable will start with an ASCII letter (upper or lower case) and will be followed by ASCII letters or digits
- everything else is an operator

The operators have precedence in increasing order from assignment to plus to minus to times to divides to modulus to exponentiation. Parentheses override precedence. All operators are left associative:

\[
\text{echo } "5 + 3 * ( 4 - 2 ) - 1 ;" \text{ > input} \\
\text{matilda -p input} \\
5 \; 3 \; 4 \; 2 \; - \; * \; 1 \; - \; + \; ; \\
10.000000 \\
\text{
}\]

Declarations
Variables are declared with the \texttt{var} keyword:

\[
\text{echo } "\texttt{var x = 3.2 ;}" \text{ > input} \\
\text{echo } "\texttt{x ;}" \text{ >> input} \\
\text{matilda -b input} \\
\{ x = 3.2 \} \\
3.200000 \\
\text{
}\]

All declarations will appear first. The initializer in a variable declaration will be a number.

Reading the input
You may find the \textit{Art and Science of Programming - C Edition} scanner to be useful for this task:

\[
\text{wget troll.cs.ua.edu/ACP-C/scanner.c} \\
\text{wget troll.cs.ua.edu/ACP-C/scanner.h} \\
\]

You can read a token with the scanner and then examine the token to see what kind of value it is. You may not use \texttt{scanf} to read into a fixed-length character array.

Error checking
The only error checking you must perform is detecting the use of a variable that was not declared:

\[
\text{echo } "\texttt{x + 3 ;}" \text{ > input} \\
\text{matilda input} \\
\texttt{variable x was not declared} \\
\text{
}\]

Display the error message as soon as you detect the undeclared variable. After printing the error message on standard out, you should exit the program. Other than this one exception, your program will only be tested with valid input. Normally, you would print error messages to \texttt{stderr} rather than \texttt{stdout}, but an exception is being made in this case.

You must follow the C programming style guide for this project: \url{http://beastie.cs.ua.edu/cs201/cstyle.html}. 
Compilation details

You must implement your calculator algorithm in portable C99. You must provide a `makefile` which responds properly to the commands:

```
makes
make test
make clean
```

The `make` command compile the `matilda` executable, which should compile cleanly with no warnings or errors at the highest level of error checking (the `-Wall` and `-Wextra` options.). The `make test` command should test your program and the `make clean` command should remove object files and the executable. Here are examples (your files may differ in number and name):

```
$ make clean
  rm -f scanner.o value.o node.o queue.o stack.o bst.o convert.o matilda.o matilda

$ make
  gcc -Wall -std=c99 -c -g scanner.c
  gcc -Wall -std=c99 -c -g real.c
  gcc -Wall -std=c99 -c -g da.c
  gcc -Wall -std=c99 -c -g cda.c
  gcc -Wall -std=c99 -c -g queue.c
  gcc -Wall -std=c99 -c -g stack.c
  gcc -Wall -std=c99 -c -g bst.c
  gcc -Wall -std=c99 -c -g convert.c
  gcc -Wall -std=c99 -c -g matilda.c
  gcc -Wall -std=c99 scanner.o real.o da.o cda.o queue.o stack.o bst.o convert.o matilda.o -o matilda

$ make
  make: `matilda' is up to date.

$ make test
  running: matilda -i -p -b mytestfile
  var b = 5;
  var a = 3;
  var c = 2;
  a b * c + 4 1 - /
  [[a = 3] b = 5 [c = 2]]
  14.000000

$ 
```

The compilation command must name the executable `matilda`. The You may develop on any system you wish but your program will be compiled and tested on a Linux system. Only the most foolish students would not thoroughly test their implementations on a Linux system before submission.

Documentation

All code you hand in must be attributed to its authors. Comment sparingly but well. Do explain the purpose of your program. Do not explain obvious code. If code is not obvious, consider rewriting the code rather than explaining what is going on through comments.

Grading

Grading will proceed as with assignment 0. In addition to testing your program, your binary search tree module and your matilda module will be tested individually.

Submission

To submit assignments, you need to install the `submit` system.

- `linux and windows 10 bash`
- `mac instructions`

You will hand in (electronically) your code for the preliminary assessment and for final testing with the commands:

```
submit cs201 lusth test1
submit cs201 lusth assign1
```
Make sure you are in the same directory as your makefile when submitting. The `submit` program will bundle up all the files in your current directory and ship them to me. Thus it is very important that only the source code and any testing files be in your directory. This includes subdirectories as well since all the files in any subdirectories will also be shipped to me. You may submit as many times as you want before the deadline; new submissions replace old submissions. Old submissions are stored and can be used for backup.