CS201: Data Structures and Algorithms

Assignment 1

Version 1

Introduction

For your second programming assignment, you are to implement a infix expression calculator using stacks and queues. For all lines of output, there should be no initial whitespace and no trailing whitespace (except for a newline), unless otherwise indicated. A single space should separate tokens in a line of output, unless otherwise directed.

I/O

Your executable must be named *matilda*. The executable reads in a series of variable declarations (possibly empty) and an expression from a file. It will produce, on *stdout*, the result of the computation. 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:

option	*example*	*action*
`-v`	`matilda -v`	give author's name and exit
`-i`	`matilda -i FILENAME`	print the original input to evaluating the expression
`-p`	`matilda -p FILENAME`	print the postfix conversion of the infix expression before evaluating the expression
`-b`	`matilda -b FILENAME`	print the BST holding variable values before evaluating the expression

If multiple options are given, print the `-i` output before the `-p` output and print the `-p` output before the `-b` output. Those options can come in any order.

Here are some example invocations using options:

```
$ matilda -v
Alyssa P. Hacker
$ cat testfile
var a
   = 3 ;
var b = 4.0 ;
b * a
 ;
$ matilda -i -b testfile
var a = 3 ;
var b = 4.0 ;
b * a ;
[a=3.000000 [b=4.000000]]
12.000000
$ matilda -p testfile
b a *
12.000000
$
```

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 and real numbers) variables (tokens beginning with an alphabetic character), operators (`= + - * / % ^`), and parentheses. All tokens will be surrounded by whitespace. For example, the expression: `(x+1)*2;` is illegal and should be written as `( x + 1 ) * 2 ;` to be proper.

A token in the input stream can be identified as follows:

- semicolons and parentheses will appear in tokens by themselves
- the keyword `var` is itself
- the equals sign is itself
- a number will start with a `.`, a minus sign, or a digit
- 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

Note, the equals sign in a variable declaration is just syntactic sugar and can otherwise be ignored. 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:

```
echo "5 + 3 * ( 4 - 2 ) - 1 ;" > input
$ matilda -p input
5 3 4 2 - * 1 - +
10.000000
$
```

Declarations

Variables are declared with the `var` keyword:

```
$ echo "var x = 3.2 ;" > testvar
$ echo "x ;" >> testvar
$ matilda -b testvar
[x=3.200000]
3.200000
$
```

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

Variables and their values should be stored in a binary search tree for later retrieval during the processing of the postfix expression. A variable will only be declared once, so the binary search tree will contain unique keys. The specification for a binary search tree can be found here: `http://beastie.cs.ua.edu/cs201/assign-bst.html`.

Reading the input

You may find the *Art and Science of Programming - C Edition* scanner to be useful for this task:

```
wget troll.cs.ua.edu/ACP-C/scanner.c
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 *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:

```
$ echo "x + 3 ;" > badvar
$ matilda badvar
variable x was not declared
$
```

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 with a zero exit code. Other than this one exception concerning undeclared variables your program will only be tested with valid input. Normally, you would print error messages to *stderr* rather than *stdout* and exit with a non-zero exit code, but for testing purposes, you should print to stdout and use a zero exit code.

You must follow the C programming style guide for this project: `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:

```
make
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, other than your modules from assignment 0 and your *BST* module, may differ in number and name):

```
$ make clean
rm -f real.o string.o da.o cda.o queue.o stack.o scanner.o matilda.o process.o bst.o matilda
$ make
gcc -Wall -Wextra -g -std=c99 -c real.c
gcc -Wall -Wextra -g -std=c99 -c string.c
gcc -Wall -Wextra -g -std=c99 -c da.c
gcc -Wall -Wextra -g -std=c99 -c cda.c
gcc -Wall -Wextra -g -std=c99 -c queue.c
gcc -Wall -Wextra -g -std=c99 -c stack.c
gcc -Wall -Wextra -g -std=c99 -c scanner.c
gcc -Wall -Wextra -g -std=c99 -c matilda.c
gcc -Wall -Wextra -g -std=c99 -c process.c
gcc -Wall -Wextra -g -std=c99 -c bst.c
gcc -Wall -Wextra -g -std=c99 -o matilda real.o string.o da.o cda.o queue.o stack.o scanner.o matilda.o process.o bst.o -l
$ make
make: `matilda' is up to date.
$ make test
running: matilda -p -b -i mytestfile
var b = 5 ;
var a = 3 ;
var c = 2 ;
( a * b + c ) / ( 4 - 1 ) ;
a b * c + 4 1 - /
[[a=3.000000] b=5.000000 [c=2.000000]]
5.666667
$
```

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.