Assignment 1

Introduction

For your first programming assignment, you are to implement a calculator using a queue, a couple of stacks, and a binary search tree. The queue is used to store an infix expression, which will be read in from a file or from the keyboard. The first stack is used to convert the expression on the queue from infix to postfix, with the postfix expression stored on the second stack. The second stack is used to convert the postfix expression it holds into a single literal. The program generally displays the resulting literal. Since the expression can reference variables, the binary search tree is used to store the values of those variables.

I/O

Your executable must be named calculon. The executable reads in an infix expression, either from a file or from stdin, and will produce, on stdout, the result of the expression. Here is an example invocation:

```
$ echo var a = 3 \; > expr
$ echo var b = 4 \; >> expr
$ echo b * a \; >> expr
$ calculon expr
12
$
```

where $ is the system prompt. The file to be processed (expr in the example) is a free-format text file. That is, the tokens within are separated by arbitrary amounts of whitespace (i.e. spaces, tabs, and newlines) and every line ends with a newline. If a file name is not given, input is to be read from the keyboard:

```
$ calculon
var a = 3 ;
var b = 4 ;
b * a ;
12
$
```

The executable must handle the following options:

<table>
<thead>
<tr>
<th>option example</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>-v</td>
<td>calculon -v give author's name</td>
</tr>
<tr>
<td>-d FILENAME</td>
<td>calculon -d FILENAME print the postfix conversion of the infix expression found in FILENAME instead of printing the answer</td>
</tr>
<tr>
<td></td>
<td>calculon -d print the postfix conversion of the infix expression read in from stdin instead of printing the answer</td>
</tr>
</tbody>
</table>

Here are some example invocations using options:

```
$ calculon -v
Alyssa P. Hacker
$ calculon -d
var a = 3 ;
var b = 4 ;
b * a ;
$b a*$
$ echo var a = 3 \; > expr
$ echo var b = 4 \; >> expr
$ echo b * a \; >> expr
$ calculon -d expr
b a*
$
```

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 (+ - * / % ^). A token in the input stream can be identified as follows:

- 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 spaces and tabs within the string
- a variable will start with an ASCII letter (upper or lower case)
- a semicolon will appear in a token by itself
- everything else is an operator

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

```
$ calculon -d
5 + 3 * 4 - 2 - 1 ;
5 3 4 * 2 - 1 - +
$
```

An integer and a real combined together yields a real number:

```
$ calculon
3 * 123. ;
369.000000
$
```

Two strings added together results in a concatenation of the two operands:

```
$ calculon
"a b c" + "123" ;
"a b c123"
$
```

Here is a code snippet for safely concatenating two strings a and b, placing the result in c:

```c
char *c = malloc(sizeof(char) * (strlen(a) + strlen(b) + 1));
sprintf(c,"%s%s",a,b);
```

A string combined with a number causes the string to be converted to a number of the same type:

```
$ calculon
3.2 * "123" ;
393.600000
$ calculon
"123.4" * 3 ;
369
$
```

Use *atoi* or *atof* to convert a string to an integer or real number, respectively. You may find the CS100 C scanner to be useful for this task:

```
wget troll.cs.ua.edu/cs100/scanner.c
wget troll.cs.ua.edu/cs100/scanner.h
```

Declarations
Variables are declared with the *var* keyword:

```
$ calculon
var x = 3 ;
x ;
3
$
```

All declarations will appear before the infix expression. The initializer in a variable declaration may be an arbitrary complex expression:
 ```calculon
 var x = 3 ;
 var y =
 ( x - 1 ) * 4 - 2 ;
 x + y ;
 9
```

The initializer should be read into an input queue and processed when the semicolon that terminates the declaration is read. 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.

**Error checking**

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

 ```calculon
 x + 3 ;
 variable x was not declared
```

Other than this one exception, your program will only be tested with valid input.

**Implementation details**

Use a `value` structure to hold each element of the expression, where the `type` field specifies the type of value stored in the structure and the `ival`, `rval`, and `sval` hold an integer, a double, and a string (and variable name), respectively. Here is what the `value.h` file might look like:

```
typedef struct value
{
  int type;
  int ival;
  double rval;
  char *sval;
}
```

```
extern int INTEGER;
extern int REAL;
extern int STRING;
extern int VARIABLE;

 extern value *newValueI(int);
 extern value *newValueR(double);
 extern value *newValueS(char *);
 extern value *newValueV(char *);
```

Here is what the `value.c` file might look like:

```
#include "value.h"

extern void Fatal(char *, ...);

static value *newValue(void);

/***** Public Interface ****/

 int INTEGER = 0;
 int REAL = 1;
 int STRING = 2;
 int VARIABLE = 3;

 value *newValueI(int i)
 {
   value *v = newValue();
   v->type = INTEGER;
   ```
// similar implementations for newValueR, newValueS, and newValueV

/**** Private Interface ****/

static value *newValue(void) /* defaults to integer zero */
{
    value *v;
    if ((v = malloc(sizeof(value))) == 0)
        Fatal("out of memory\n");
    v->type = INTEGER;
    v->ival = 0;
    v->rval = 0;
    v->sval = 0;
    return v;
}

Now, the nodes for your stacks and queues can all references value “objects”. Variables are stored in the binary search tree with the variable name as key and a value object as value.

Your stack must support push and pop in constant time. Your queue must support enqueue and dequeue in constant time.

You must follow the C programming style guide for this project: [http://beastie.cs.ua.edu/cstyle.html](http://beastie.cs.ua.edu/cstyle.html).

### Compilation details

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

```
make
make test
make clean
```

The `make` command compiles your program, which should compile cleanly with no warnings or errors at the highest level of error checking (the `-Walt` option for gcc). 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 calculon.o calculon
$ make
  gcc  -Wall -std=c99 -c -g  scanner.c
gcc  -Wall -std=c99 -c -g   value.c
gcc  -Wall -std=c99 -c -g   node.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  calculon.c
gcc  -Wall -std=c99  scanner.o value.o node.o queue.o stack.o bst.o convert.o calculon.o -o calculon
$ make
  make: 'calculon' is up to date.
$ make test
  calculon -d mytestfile
  a b * c + 4 1 - /
  calculon mytestfile
  14
```

The compilation command must name the executable `calculon` (not `calculon.exe` for you Cygwin users). 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.

Note: depending on where you develop your code, uninitialized variables may have a tendency to start with a value of zero. Thus, a program with uninitialized variables may work on your system but fail when I run it. I won’t care, as you are mature
enough not to have uninitialized variables. You may have other errors as well that do not reveal themselves until run on my system. Again, that’s not my problem. If I am feeling generous and have the time, I may figure out where your error is and, perhaps, give you a few meager points back, but don’t depend on it.

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

Severe deductions will be made for poor performance or not following the specification (e.g. extra output or not allowing arbitrary whitespace). Here are some additional deductions which might apply:

- bad program style, lack of appropriate modularizations (20 point deduction max)
- placing executable code in .h files or including .c files. (20 point deduction max)
- include file names that do not match (in a case-sensitive sense) their include statements (10 point deduction)
- variables that are defined, but not set and used (5 point deduction)
- variables that are defined and set, but not used (5 point deduction)
- bad makefile style, unnecessary recompilation, missed recompilations (5 point deduction max)
- a makefile that does not use -Wall or -std=c99 for all compilations (5 point deduction)
- a makefile without a rule for each module (10 point deduction)
- a makefile without a clean rule or a clean rule that does not remove all object files and the executable (5 point deduction)

This list is not inclusive; other deductions may apply.

You may request a regrade of your program if you feel the testing suite did not do justice in uncovering the capabilities of your program. However, the maximum score a regraded program can achieve is 80 points, unless the problems your implementation has are deemed to be not your fault. A regrade request will not be considered if your program is not submitted for preliminary assessment.

Challenge

Rather than use a node-based linked list for implementing your stack and queue, try using a dynamic circular array instead.

Submission

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

- linux and cygwin instructions
- mac instructions

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

    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. I will deduct points if you submit object files or executables, so be careful. 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.