COMP 211: Principles of Imperative Computation, Fall 2014

Programming 5: Clac

:@;

Due: Wednesday, 29 Oct, 2015 by 6pm

For the programming portion of this week's homework, you will implement a small postfix claculator.

1 Getting the code

The code handout for this assignment is available from the course web page. The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in.

The files you should hand in are:

- `clac.c0`
- `poly1.clac` [Task 2]
- `poly2.clac` [Bonus Task 3]

To download and unzip the code, do the following:

1.1 Mac OS

1. On the assignments page of the course web site, right-click the Programming 5 `Code` link, and then click "Save Link as...", and choose to save it in `Documents/comp211/`.
2. Start a new terminal window and do the following:

   ```
   cd comp211
   tar -xzvkf hw5-handout.tgz
   ```

   This will make a directory called `Documents/comp211/hw5-handout` and extract all of the code for the assignment.
3. The file `hw5-handout/README.txt` contains a description of the various files, and instructions on running them.

1.2 Windows

1. On the assignments page of the course web site, right-click the Programming 5 `Code` link, and then click "Save Link as...", and choose to save it in your `comp211/`. To find this directory in your web browser, look in

   ```
   C:\cygwin\home\YOURNAME\comp211\
   ```

2. Start a new terminal window and do the following:

   ```
   cd comp211
   tar -xzkvf hw5-handout.tgz
   ```

   This will make a directory called `comp211/hw5-handout` and extract all of the code for the assignment.

3. The file `hw5-handout/README.txt` contains a description of the various files, and instructions on running them.

2 Introducing the Claculator

Clac is a new stack-based programming language developed by a new startup called Reverse Polish Systems (RPS)[1]. Clac is similar to an old language called Forth, and also to the language PostScript, which is still used by printers. In this assignment, we will be implementing the core features of the Claculator.

Clac works like an interactive calculator. When it runs, it maintains an *argument stack*. Entering numbers will simply push them onto the argument stack. When an operation such as addition `+` or multiplication `*` is encountered, it will be applied to the top elements of the stack (consuming them in the process) and the result is pushed back onto the stack. When a newline is read, the stack will be printed. For example, suppose we start Clac and type `3 4 +` and then a newline.

```
Clac top level
clac>> 3 4 +
Stack: 7
```

(The `clac>>` prompt is signalling that this command should be typed into Clac, rather than `coin` or the Unix shell.)

Clac responded by printing `7`, which is now on top of the stack (which is otherwise empty). We now enter `3 2 *` and a newline, after which Clac responds with `7,6`.

```
clac>> 3 2 *
Stack: 7,6
```

[1] `http://en.wikipedia.org/wiki/Reverse_Polish_notation`

We write stacks with the top element on the right, so this stack has the *top* element of the stack as 6, and the *next-to-top* element as 7.

At this point the stack has `7` and `6` on it and we multiply them simply by typing `*` and a newline.

```
clac>> *
Stack: 42
```

We can quit our interactions by typing `quit`.

```
clac>> quit
11
Bye!
```

We can type multiple inputs (numbers and operations) on the same line. For example,

```
% ./clac
clac>> 11 10 3 9 + + *
Stack: 242
```

We can see what this does by writing out the state of the stack after each number/operation has been processed:

Operation	Stack After
11	11
10	11,10
3	11,10,3
9	11,10,3,9
+	11,10,12
+	11,22
*	240

In addition to the arithmetic operations, there are a few special operations you will have to implement, which manipulate the stack. `dup` duplicates the top element. `swap` swaps the first two elements. `third` moves the third element to the front, keeping the first and second in the same order. For example,

```
clac>> 11 10 swap drop 13 14 third
Stack: 13,14,10
```

Operation	Stack After
11	11
10	11,10
swap	10,11
drop	10
13	10,13
14	10,13,14
third	13,14,10

To precisely specify these operations, we use the notation

$$S \longrightarrow S'$$

to mean that the stack S transitions to become stack S'. Stacks are written with the *top element at the right end*! For example, the action of addition is stated as

$$\texttt{+} : S, x, y \longrightarrow S, x{+}y$$

which means: "*Pop the top element (y) and the next element (x) from the stack, add y to x, and push the result x+y back onto the stack.*" The fact that we write S in the rule above means that there can be many other integers on the stack that will not be affected by the operation.

Operation		**Before**		**After**	**Condition or Effect**
n	:	S	$\longrightarrow$	S, n	for $-2^{31} \le n < 2^{31}$ in decimal
`+`	:	S, x, y	$\longrightarrow$	$S, x + y$	
`*`	:	S, x, y	$\longrightarrow$	$S, x * y$	
`drop`	:	S, x	$\longrightarrow$	S	
`swap`	:	S, x, y	$\longrightarrow$	S, y, x	
`dup`	:	S, x	$\longrightarrow$	S, x, x	
`third`	:	S, x, y, z	$\longrightarrow$	S, y, z, x	
`quit`	:	S	$\longrightarrow$	_	exit Clac

Task 1 (7 points) *In* `clac.c0`*, fill in the function*

```
bool eval(string op, stack S)
```

with code that processes a single operation `op`*, which must be one of* `+ * drop swap dup third quit`*, or a decimal number* `n`*. The function should modify the given stack* `S` *according to the operation. You may assume that there will be enough arguments on the stack to do the operation (e.g., for* `+`*, the stack will always have at least two numbers on it).*

You can find the interface for stacks in the file `lib/stacks_int.c0`, which is just like the code from lecture, except it defines stacks of `int`s instead of strings. Your code must respect the library interfaces described in the `lib` directory: we will compile your code against *different* implementations of stacks, so you must only use the functions and types given as part of the interface.

When you compile and run Clac:

```
$ cc0 -d -o clac lib/*.c0 clac.c0 clac-main.c0
$ ./clac
```

your `eval` function will be called on each command the user types in, in order from left to right. The stack will be initially empty, but as the input is processed, the `eval` function will also be called with nonempty stacks, representing the values from prior computations.

To test, in the Unix terminal, run

```
% ./clac
```

Then you will see a Clac prompt, like

```
clac>>
```

and you can type Clac programs and see the results:

```
clac>> 10 20 +
30
```

You can also run `clac` on a file, where the file's lines are treated just as if you had typed them into clac. For example, in the Unix terminal, do

```
% ./clac def/add.clac
```

To test your implementation, you should either type things directly into `clac`, or you should create additional `<test>.clac` files in the `defs` directory.

Task 2 (1 point) *Write a Clac program, that, when run on a stack with x at the top of the stack, replaces x with the value of $x^2 + 2x + 1$. Hand in this program in a file* `poly1.clac`*.*
For example:

```
clac>> 3 <your program>
Stack: 16
```

Task 3 (Bonus) *Write a Clac program that, when run on a stack with x and y at the top, replaces them with the value of $3x^2 + 2xy + 5y^2$. Hand in this program in a file* `poly2.clac`*.*

2.1 Bonus Task: Type Checking

Above, we said that you could assume that there will always be enough arguments on the stack for each operation that you encounter. E.g. you will never encounter a `+` when there is only one thing on the stack. However, Clac programmers can type in programs that violate this, such as

```
clac>> 2 3 4 + + 2 * +
```

Bonus Task 2: Write a function

```
bool always_enough(string[] ops, int ops_size, int start_size)
```

which returns true iff there are always enough arguments on the stack while running the operations in `ops` from left to right, starting with a stack of size `start_size`. The idea is that, when the Clac programmer types in a program, you will *first* run `always_enough` on it, to determine it will run successfully, and *then* run the program itself. `./clac` does this, so you can test your `always_enough` just by typing incorrect programs into `clac`.

If you try to run an incorrect program before doing the bonus task, you will see a contract violation, notifying you that the program crashed:

```
clac>> 1 +
lib/stacks_int.c0:15.4-15.29: @requires annotation failed
```

After doing the bonus task, you should see

```
clac>> 1 +
Not enough arguments on stack for those operations; ignoring them.
```

and then `Clac` will wait for more input.

Checking that a Clac program will not crash before you run it is much like *type checking* a C0 program before you run it, and the error `Not enough arguments on the stack` is like C0's error

```
clac.c0:62.10-62.28:error:too few arguments in function call
  return always_enough(S,1);
```

Thus, the conceptial goal of this bonus task is to give you a bit of a sense for how programs are type checked.