15-122: Principles of Imperative Computation, Spring 2013

Homework 4 Programming: Clac

:@;

Due: Monday, September 30, 2013 by 22:00

For the programming portion of this week's homework, you will implement a small for a postfix claculator$^{\text{TM}}$, an interpreter for the Clac programming language.

The code handout for this assignment is at

`http://www.cs.cmu.edu/~rjsimmon/15122-f13/hw/15122-prog4.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a FIVE (5) HANDIN LIMIT - you may only submit to Autolab five times for this assignment without penalty; every additional handin will incur a half-point penalty.

Testing Because we are setting a very low handin limit, we are going to be very clear how we intend to test your programs. We will test your Clac implementation by running tests of the following form:

```
test_prog(clac_program, initial_stack, final_stack, result);
```

We will check four things:

1. Your code must compile without violating the library interfaces.
2. When given valid input, your interpreter must run without errors and wind up with the correct stack.
3. When given invalid input, your interpreter must halt with a call to `error`, signaling that the user has written an invalid program.
4. All operations have good asymptotic running time when we compile without `-d`. (This means that all operations should take constant time except for `skip` and `pick`. The time it takes to skip n elements or to copy the n^{th} element on the stack ought to be in $O(n)$.)

The file `clac-test.c0` includes examples of how to write and run tests of this form, and the `README.txt` explains how to compile and run these tests.

Sharing tests The academic integrity policy for this course does not allow you to view other people's C0 code or share your C0 code with others. However, you may share Clac code, including *Clac-only* test cases, via Piazza posts. If you share tests, do so non-anonymously and tag your post with `#clactest`.

1 Introducing the Claculator

Clac is a new stack-based programming language developed by a Pittsburgh-area startup called Reverse Polish Systems (RPS). Any similarities of Clac with Forth or PostScript are purely coincidental. In the first part of this assignment, we will be implementing the core features of the Claculator, and in Task 2 we will be adding a few more interesting features.

Clac works like an interactive calculator. When it runs, it maintains an *operand stack*. Entering numbers will simply push them onto the operand 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 number on top of the stack will be printed. For example, after we start the we type `3 4 +` and then a newline.

```
% ./claculator
Clac top level
clac>> 3 4 +
7
```

Clac responded by printing `7`, which is now on top of the stack (which is otherwise empty). We now enter `-9 2 /` and a newline, after which Clac responds with `-4`.

```
clac>> -9 2 /
-4
```

At this point the stack has `7` (the result of the addition) and `-4` (the result of the integer division) and we can subtract them simply by typing `-` and a newline.

```
clac>> -
11
```

We obtain 11, since $7 - (-4) = 11$. 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,

```
% ./claculator
Clac top level
clac>> 11 10 2 9 - + *
33
```

Please make sure you understand why the above yields `33` on the stack.

In addition to the arithmetic operations, there are a few special operations you will have to implement. The table below is the complete set of operations that you will be implementing in Task 1. To specify the 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 subtraction 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, subtract y from 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.

Token		**Before**		**After**	**Condition or Effect**
n	:	S	$\longrightarrow$	S, n	for $-2^{31} \leq n < 2^{31}$ in decimal
`+`	:	S, x, y	$\longrightarrow$	$S, x+y$	
`-`	:	S, x, y	$\longrightarrow$	$S, x-y$	
`*`	:	S, x, y	$\longrightarrow$	$S, x*y$	
`/`	:	S, x, y	$\longrightarrow$	$S, x\,/\,y$	error, if div by 0 or overflow
`%`	:	S, x, y	$\longrightarrow$	$S, x\,\%\,y$	error, if mod by 0 or overflow
`<`	:	S, x, y	$\longrightarrow$	$S, 1$	if $x < y$
`<`	:	S, x, y	$\longrightarrow$	$S, 0$	if $x \geq y$
`drop`	:	S, x	$\longrightarrow$	S	
`swap`	:	S, x, y	$\longrightarrow$	S, y, x	
`dup`	:	S, x	$\longrightarrow$	S, x, x	
`rot`	:	S, x, y, z	$\longrightarrow$	S, y, z, x	
`print`	:	S, x	$\longrightarrow$	S	print x followed by newline
`quit`	:	S	$\longrightarrow$	$\_$	exit Clac

Your implementation should explicitly detect and signal an error by calling the function `error` with an appropriate error message in the following situations:

- The token is illegal, that is, not one of the ones listed above. Tokens are treated as case-sensitive. For example, `DUP` and `Dup` are undefined, but `dup` duplicates the top element of the stack.

- There are an insufficient number of elements on the stack to carry out an operation. For example, it is an error to call `rot` when there are less than three integers on the stack.

- Division or modulus by 0, or division of `int_min()` by -1, would generate an overflow according to the definition of C0 (see page 3 of the C0 Reference). This is a 32 bit, two's complement language, so addition, subtraction, and multiplication behave just as in C0 without raising any overflow errors.

User errors (errors in Clac code) should always cause `error` to be called; they should never cause assertions to fail. An example is in the starter code in the file `clac.c0`.

Task 1 (10 points) *Extend the implementation of the Clac implementation in file* `clac.c0` *to behave according to the specification above. Do not change any of the* `#use` *directives in this file, and make sure your function will satisfy the following declaration:*

```
bool eval(queue Q, stack S)
//@ensures \result == false || queue_empty(Q);
```

You may freely change anything else in this file. The arguments and return value of the `eval` *are explained next.*

The `main` function in file `clac-main.c0` and the `test_prog` function in `clac-test.c0` both take lines of input and convert them to a *queue of tokens*. Each token is just a string. This part of the Clac implementation has already been programmed for you, and you are welcome to examine it, but you need not change this code. In Clac, tokens are only separated by white space. For example, `3 4+` will be read as two tokens (`"3"` followed by `"4+"`) and will therefore lead to an error since the token `"4+"` is not defined.

The *stack of integers S* will be initially empty. But since the input is processed line-by-line, the `eval` function may also be called with nonempty stacks, representing the values from prior computations.

The `eval` function should dequeue tokens from the queue Q and process them according to the Clac definition. When the queue is empty, `eval` should return `true`, leaving the stack in whatever state it is. Upon encountering the token "`quit`", `eval` should return `false`, indicating to the `main` function that it should exit.

You can find the interface to the implementations of queues and stacks in the files `lib/queues_string.c0` and `lib/stacks_int.c0`, which are just like the code from Lectures 9 and 10.

Reference implementation. You can run a reference implementation of Clac on Andrew if you'd like to see the results of examples. To try and avoid confusion, we have called the executable `clac-ref`.

```
% clac-ref
Clac top level
clac>> 4 7 - 2 *
-6
clac>> quit
-6
Bye!
```

You can also run `clac-ref -trace`. This will make the reference implementation print out all the intermediate steps of the computation. For example,

```
% clac-ref -trace
Clac top level
clac>> 4 7 - 2 *
     stack || queue
           || 4 7 - 2 *
         4 || 7 - 2 *
       4 7 || - 2 *
        -3 || 2 *
      -3 2 || *
        -6 ||
-6
clac>>
```

Here the left column displays the current stack and the right column displays the current queue followed by the return stack (which is not displayed in the example above because it is empty).

2 Manipulating the stack and the queue

In this part of the assignment, we will implement a few more advanced features of Clac that manipulate the stack and the queue. The first, `skip`, takes the number n on top of the stack (which must be non-negative) and removes n tokens from the queue – it is an error if n is negative or if there aren't enough tokens immediately available on the queue.

You'll also implement two special tokens that behave much like `skip`: `if` skips either 0 or 2 tokens depending on whether the top of the stack is 0 or not, and `else` behaves exactly like "`1 skip`". These may seem like strange names, but these two tokens together let us implement the idiom

$$\texttt{if}\ token_1\ \texttt{else}\ token_3$$

where 0 acts like `false` and any non-zero value acts like `true`. The following example illustrates this:

```
clac>> 6 1 if 2 else -2 +
8
clac>> 6 0 if 2 else -2 +
4
```

The first line computes 8: when `if` is encountered, 6 and 1 are on the stack, so we push 2, consuming 1 in the process. Then we see `else` and skip over the next token `-2`. Finally we add 6 and 2, resulting in 8 to be pushed on the stack which is printed back.

In the next line, when `if` is encountered, 0 is on top of the stack so we skip the next two tokens, namely `2` and `else`. Then `-2` is pushed on the stack and the addition computes $6 + (-2)$ and pushes it on the stack.

To be clear: *the two tokens* `if` *and* `else` *do not have to appear together.* They should be implemented as independent Clac features.

The final addition we will make to Clac is the token `pick`. If the positive number n appears at the top of the stack, `pick` should duplicate the n^{th} element on the stack at the top of the stack. It is an error if n isn't positive or if there aren't n things on the stack to look through.

This following table summarizes the features we are implementing in this part of the assignment:

Before				**After**			
Stack		**Queue**		**Stack**		**Queue**	**Cond**
S, n	$\|\|$	$\texttt{skip}, tok_1, \ldots, tok_n, Q$	$\longrightarrow$	S	$\|\|$	Q	$n \geq 0$
S, n	$\|\|$	$\texttt{if}, Q$	$\longrightarrow$	S	$\|\|$	Q	$n \neq 0$
S, n	$\|\|$	$\texttt{if}, tok_1, tok_2, Q$	$\longrightarrow$	S	$\|\|$	Q	$n = 0$
S	$\|\|$	$\texttt{else}, tok_1, Q$	$\longrightarrow$	S	$\|\|$	Q	
$S, x_n, \ldots, x_1, n$	$\|\|$	$\texttt{pick}, Q$	$\longrightarrow$	$S, x_n, \ldots, x_1, x_n$	$\|\|$	Q	$n > 0$

Task 2 (5 points) *Extend* `eval` *in file* `clac.c0` *to handle* `skip`, `if`, `else`, *and* `pick` *according to the specification above.*

3 Bonus: Clacworks

We've described the behavior of 10 tokens: `drop`, `swap`, `dup`, `rot`, `print`, `quit`, `skip`, `if`, `else`, and `pick`. As long as you leave the meaning of programs with numbers and these 10 tokens alone, you can extend Clac with new features and write programs that use these new tokens.

Task 3 (bonus points) *Add some new features to your Claculator by defining the meaning of new tokens. Write a cool or surprising program in* `bonus.clac`, *and describe the behavior of your new features and your Clac program in* `bonus.txt`.