15-122: Principles of Imperative Computation, Summer 2017

Programming Homework 5: Clac

:@;

Due: Tuesday 18th July, 2017 by 11pm

In this assignment, you will implement a *claculator*™ for the Clac programming language.

The code handout for this assignment is on Autolab and at

`http://cs.cmu.edu/~15122/hw/clac-handout.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. Every additional handin will incur a small (5%) penalty.

Testing We will test your Clac implementation by running tests of the following form:

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

We will check four things:

1. Your code must respect the library interfaces described in the `lib` directory. We will compile your code against *different* implementations of stacks and queues, so you must only use the functions and types given as part of the interface.
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 must have good asymptotic running time when we compile without `-d`. (This means that all operations should take constant time except for `**`, `pick` and `skip`. The time it takes to copy the n^{th} stack element or delete n tokens from the queue should 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, tag your post with `#clactest`.

Every year one or two ambitious students write prime-number tests in Clac; one student in Fall 2014 wrote an implementation of *another* programming language, Bitwise Cyclic Tag, in the Clac language! What will you come up with?

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 later 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. This is an example where we start Clac and type `3 4 +`, followed by a newline:

```
% ./clac
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,

```
% ./clac
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 at the top of Figure 1 is the complete set of operations that you will be implementing in Task 1. To specify the operations, we use the notation

$$S \mid\mid Q \longrightarrow S' \mid\mid Q'$$

to mean that the stack S and the queue Q transitions to become stack S' and the queue Q'. We use a queue to hold the numbers and operations Clac evaluates (and for other purposes as well — see below). Stacks are written with the *top element at the right end*! For example, the action of multiplication is stated as

$$S, x, y \mid\mid *, Q \qquad \longrightarrow \qquad S, x \times y \mid\mid Q$$

which means: *"When you dequeue the token $*$ from the queue, pop the top element (y) and the next element (x) from the stack, multiply y by x, and push the result $x \times y$ 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.

Every operation in Clac is determined by the token that has just been dequeued from the queue. The `<` and `if` tokens cause *different* things to happen depending on the specific values on the stack.

$$\begin{array}{llll} S, x, y \mid\mid \texttt{<}, Q & \longrightarrow & S, 1 \mid\mid Q & \text{if } x < y \\ S, x, y \mid\mid \texttt{<}, Q & \longrightarrow & S, 0 \mid\mid Q & \text{if } x \geq y \end{array}$$

In Clac, we use the integer 0 to mean `false` and we treat non-zero values like 1 as `true`. The `if` token runs some code (the next three tokens) only if the integer on the top of the stack is `true` (that is, nonzero). If the `if` token is reached while 0 is at the top of the stack, the following three tokens are skipped.

As you implement the Clac operations in Figure 1, if the instructions indicate that Clac should raise an error, you should call the function **`error()`** with the appropriate error message. The **`error()`** function takes a string as its argument and is built in to C0, like **`assert()`**. User errors (errors in Clac code) should always cause **`error`** to be called; assertions should only be used for programmer errors.

2 Implementing Clac

In `clac.c0`, you should not change any of the `#use` directives, and you should not change the type of `eval`, its preconditions, or its postconditions:

```
bool eval(queue Q, stack S, state_t ST)
//@requires Q != NULL && S != NULL && ST != NULL;
//@ensures \result == false || queue_empty(Q);
```

You do not need to worry about the `state_header` struct, the `state_t` type, or the `init_state()` function until Task 3.

Before				**After**			
Stack		**Queue**	$\longrightarrow$	**Stack**		**Queue**	**Cond/Effect**
S	$\|\|$	n, Q	$\longrightarrow$	S, n	$\|\|$	Q	
S, n	$\|\|$	`print`, Q	$\longrightarrow$	S	$\|\|$	Q	See note #1
S	$\|\|$	`quit`, Q	$\longrightarrow$	S	$\|\|$	Q	See note #2
S, x, y	$\|\|$	`+`, Q	$\longrightarrow$	$S, x + y$	$\|\|$	Q	See note #3
S, x, y	$\|\|$	`-`, Q	$\longrightarrow$	$S, x - y$	$\|\|$	Q	See note #3
S, x, y	$\|\|$	`*`, Q	$\longrightarrow$	$S, x \times y$	$\|\|$	Q	See note #3
S, x, y	$\|\|$	`/`, Q	$\longrightarrow$	$S, x / y$	$\|\|$	Q	See note #4
S, x, y	$\|\|$	`%`, Q	$\longrightarrow$	$S, x \% y$	$\|\|$	Q	See note #4
S, x, y	$\|\|$	`**`, Q	$\longrightarrow$	S, x^y	$\|\|$	Q	See #3, #4
S, x, y	$\|\|$	`<`, Q	$\longrightarrow$	$S, 1$	$\|\|$	Q	if $x < y$
S, x, y	$\|\|$	`<`, Q	$\longrightarrow$	$S, 0$	$\|\|$	Q	if $x \geq y$
S, x	$\|\|$	`drop`, Q	$\longrightarrow$	S	$\|\|$	Q	
S, x, y	$\|\|$	`swap`, Q	$\longrightarrow$	S, y, x	$\|\|$	Q	
S, x, y, z	$\|\|$	`rot`, Q	$\longrightarrow$	S, y, z, x	$\|\|$	Q	
S, x	$\|\|$	`if`, Q	$\longrightarrow$	S	$\|\|$	Q	if $x \neq 0$
S, x	$\|\|$	`if`, tok_1, tok_2, tok_3, Q	$\longrightarrow$	S	$\|\|$	Q	if $x = 0$
$S, x_n, \ldots, x_1, n$	$\|\|$	`pick`, Q	$\longrightarrow$	$S, x_n, \ldots, x_1, x_n$	$\|\|$	Q	See note #5
S, n	$\|\|$	`skip`, $tok_1, \ldots, tok_n, Q$	$\longrightarrow$	S	$\|\|$	Q	See note #5

Clac should raise an error whenever there are not enough tokens on the stack or the queue for an operation to be performed, or whenever the token on the top of the stack is not one of the ones listed above. Tokens are case sensitive, so `Print` and `PRINT` are not defined, though `print` is.

Notes:

1. *The `print` token causes n to be printed, followed by a newline.*
2. *The `quit` token causes the interpreter to stop. The `eval` function should then return `false` to indicate that we should just stop, rather than asking for more input.*
3. *This is a 32 bit, two's complement language, so addition, subtraction, multiplication, and exponentiation should behave just as in C0 without raising any overflow errors.*
4. *Division or modulus by 0, or division/modulus of `int_min()` by -1, which would generate an overflow exception according to the definition of C0 (see page 3 of the C0 Reference), should raise an error in Clac. Negative exponents are undefined and should also raise an error.*
5. *The `pick` token should raise an error if n, the value on the top of the stack, is not strictly positive. The `skip` token should raise an error if n is negative; 0 is acceptable.*

Figure 1: Clac reference

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 should 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.

When `eval` is first called, the *stack of integers* S will be 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 was already in. Upon encountering the token "`quit`", `eval` should return `false`, indicating to the `main` function that it should exit.

Task 1 (6 points) In `clac.c0`, make sure the given implementations of `print` and `quit` are safe and correct, fixing them if they are not. Add implementations of `+`, `-`, `*`, `**`, `/`, `%`, and `<` according to the specification in Figure 1.

Task 2 (5 points) In `clac.c0`, add definitions of `drop`, `swap`, `rot`, `if`, `skip`, and `pick` according to the specification in Figure 1.

It is possible to write a lot of long and confusing code to safely and efficiently implementing these two tasks, but it is also possible to use helper functions to write very clear and concise code. As you design your helper functions, remember the conditions we said we would be checking on the first page!

The interfaces to stacks and queues, which are similar to the ones from lecture, are given in the `lib` directory. You may not change these implementations, and you must respect their interfaces.

3 Dictionaries

Before we can introduce our final Clac feature, *definitions*, we need to introduce a *dictionary* data type that will associate names with their definitions. The interface to dictionaries has three functions:

```
dict_t dict_new()
  /*@ensures \result != NULL; @*/;

queue_t dict_lookup(dict_t D, string name)
  /*@requires D != NULL; @*/;

void dict_insert(dict_t D, string name, queue_t def)
  /*@requires D != NULL; @*/
  /*@requires def != NULL; @*/;
```

Lookup returns the most recently inserted queue for a given name, or `NULL` if no such queue exists. Insertion with `dict_insert(D, name, def)` updates the dictionary so that future lookups on `name` will return `def`. This must handle the case where `name` is not already defined in the dictionary, in which case we have to add it, as well as the case where `name` is already defined, in which case we have to override or replace the old definition with the new.

You should take a look at the implementation of dictionaries we give you, which is probably the simplest but least efficient implementation imaginable, based on *association lists*, which are linked lists where each node contains a key (the string) and a value (the queue). The dictionaries don't ever use the queue interface, they just store and return references to queues.

4 Definitions

Finally, we add definitions to Clac. A *definition* has the form

$$\texttt{:}\ name\ token_1 \ldots token_n\ \texttt{;}$$

When we encounter the token `:` (colon) in the input queue, we interpret the following token as a *name*. Then we create a new (separate) queue, intended to hold the definition of *name*. Then we continue to scan the input queue, copying each token to the new queue until we encounter a token `;` (semicolon) which signals the end of the definition. Then we add *name*, with the new queue as its definition, to the dictionary.

If the queue ends after the colon (`:`), or if there is no semicolon (`;`) in the remainder of the queue after *name*, an error should be signaled. In a definition, *name* can be any token, but if it is a built-in operator or a number, then the definition can never be invoked since it is always superseded by the predefined meaning.

Let's consider a simple example.

```
: dup 1 pick ;
: square dup * ;
```

This defines `dup` and `square`. Whenever we see `square` in the input subsequently, it has the effect of replacing n on the top of the stack with n^2. For example,

```
% clac-ref
Clac top level
clac>> : dup 1 pick ; : square dup * ;
(defined dup)
(defined square)
(stack empty)
clac>> 5 square
25
```

Note that `5 square` should be identical to `5 1 pick *` which duplicates 5 on the stack and then performs a multiplication.

How do we process a defined name when we encounter it in the queue of tokens? This is not entirely straightforward, as the following example illustrates:

```
clac>> 3 square 4 square +
25
```

When we see the first occurrence of `square` we cannot simply replace the rest of the queue with the definition of `square`, since after squaring 3 we have to continue to process the rest of the queue, namely `4 square +`.

In order to implement this, we maintain a *stack of queues of tokens* (which has already been implemented for you, see the file `lib/stack_of_queue_of_string.c0`). This is called the *return stack* or sometimes the *call stack*. When we encounter a defined token, we push the remainder of the queue onto the return stack, and begin using the definition as our queue of tokens. When we finish executing a definition we pop the prior queue of tokens from the return stack and continue with processing it. When there are no longer any queues on the return stack we return from the `eval` function. Here's a trace of the above execution:

Operand Stack	**Queue of Tokens**	**Return Stack**
(empty)	`3 square 4 square +`	(empty)
3	`square 4 square +`	(empty)
3	`dup *`	(`4 square +`)
3	`1 pick`	(`4 square +`), (`*`)
$3, 1$	`pick`	(`4 square +`), (`*`)
$3, 3$	(empty)	(`4 square +`), (`*`)
$3, 3$	`*`	(`4 square +`)
9	(empty)	(`4 square +`)
9	`4 square +`	(empty)
$9, 4$	`square +`	(empty)
$9, 4$	`dup *`	(`+`)
$9, 4$	`1 pick`	(`+`), (`*`)
$9, 4, 1$	`pick`	(`+`), (`*`)
$9, 4, 4$	(empty)	(`+`), (`*`)
$9, 4, 4$	`*`	(`+`)
$9, 16$	(empty)	(`+`)
$9, 16$	`+`	(empty)
25	(empty)	(empty)

After the last step, both the token queue and the return stack are empty, so we return from the `eval` function with an operand stack consisting of the single number 25.

If you run this example in the reference Clac interpreter with the `-trace` option, you'll see that the reference interpreter orders the return stack in the other order, so that you always see the most recent part of the return stack first.

5 Implementing Definitions

Before you work on this task, make sure you have tested your previous implementation yourself and with Autolab. An Autolab submission will also make sure you have a working backup to go back to. While this task is worth fewer points than the previous tasks, it's also the most challenging part of the assignment.

You'll modify your existing interpreter to include dictionaries by adding a field to the `state_header` struct defined in `clac.c0`. This struct is initialized by the `init_state()` function one time when the Clac interpreter is first run; the pointer that `init_state()` returns is then passed back to `eval` every time it is run. You'll therefore modify `init_state` to initialize your new struct field(s).

You may remove the same queue from a dictionary more than one time, like the queue one associated with `square` in the example above. Therefore, it's important that each time you look up a queue in the dictionary, you make a copy of it before you dequeue from that queue. The interface to queues in `lib/queue_of_string.c0` has a function `queue_read_only_copy(Q)` that makes a read-only copy of a queue in $O(1)$ time.

You'll need to allocate a return stack as well. The return stack can be a field of the `state_header` struct or you can allocate a new return stack whenever `eval()` is called.

Task 3 (4 points) Modify the `state_header` struct and use it to implement definitions in your Clac interpreter.

Some Clac examples are given in the files in the `def/` directory with the handout. As an example, here is the definition of the Fibonacci function with several auxiliary names like `noop` (which does nothing).

```
: noop ;
: dup 1 pick ;
: fib dup if fib1 1 skip noop ;
: fib1 dup 1 - if fib_body 1 skip noop ;
: fib_body dup 1 - fib swap 2 - fib + ;
```

It has the following summary effect:

$$S, n \mid\mid \texttt{fib}, Q \quad\longrightarrow\quad S, \mathit{fib}(n) \mid\mid Q$$

(provided $n \geq 0$) where *fib* is the standard mathematical Fibonacci function. It's a useful exercise to work through by hand how, for example, `2 fib` computes, starting with the empty operand and return stacks.