15-122: Principles of Imperative Computation, Spring 2013

Homework 3 Programming: Claclab

Due: Thursday, February 21, 2013 by 23:59

For the programming portion of this week's homework, you will implement two small interpreters and supporting code, one for a small postfix claculator™, the other for its extension to the Clac programming language. You will complete some starter code in three C0 files, and optionally submit some code in the Clac language for extra credit.

- `demo.c0` (described in Sections 1 and 2)
- `dict.c0` (described in Section 3),
- `clac.c0` (described in Section 4)
- `bonus.clac` (optional, described in Section 5)

You should submit these files electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

This assignment does not require a lot of code, but may be conceptually difficult.

- Please read all parts of the assignment carefully.
- Frequently and incrementally test your code. You can do small claculations interactively for the first two tasks, and run small Clac programs for Task 4. Take advantage of that to catch conceptual errors early.

The starter code in `demo.c0` is set up so that it compiles and runs as given, even though it cannot do anything interesting initially.

Modification to academic integrity policy: 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, freely. You may share Clac code with friends, with enemies, on public Piazza posts, on Facebook, whatever.

Assignment: Clac (25 points in total)

Starter code. Download the file `hw3-handout.tgz` from the course website. When you untar it, you will find three C0 files, `demo.c0`, `dict.c0`, and `clac.c0`. You will also see a `lib/` directory with some provided C0 libraries and `def/` with some sample Clac programs.

You should not modify or submit the library code in the `lib/` directory, nor should you rely on the internals of these implementations. When we are testing your code we will use our own implementations of these libraries with the same interface, but possibly different internals.

Compiling and running. You will compile and run your code using the standard C0 tools. You should compile and run the code with

```
% cc0 -d -o demo demo.c0 demo-main.c0
% ./demo
```

for Tasks 1 and 2 and

```
% cc0 -d -o clac dict.c0 clac.c0 clac-main.c0
% ./clac
```

for Tasks 3 and 4. Don't forget to include the `-d` switch to enable dynamic annotation checking while testing your program.

Submitting. Once you've completed some files, you can submit them to Autolab. There are two ways to do this:

From the terminal on Andrew Linux (via cluster or ssh) type:

```
% handin hw3 demo.c0 dict.c0 clac.c0 bonus.clac
```

(omitting `bonus.clac` if you are not providing that). Your score will then be available on the Autolab website.

Your files can also be submitted to the web interface of Autolab. To do so, please `tar` them, for example:

```
% tar -czvf sol.tgz demo.c0 dict.c0 clac.c0 bonus.clac
```

Then, go to `https://autolab.cs.cmu.edu/15122-s13` and submit them as your solution to homework 3 (claclab).

You may submit your assignment up to 25 times. When we grade your assignment, we will consider the most recent version submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. You should provide loop invariants and any assertions that you use to check your reasoning. If you write any auxiliary functions, include precise and appropriate pre- and postconditions.

You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Unit testing. You should write unit tests for your code. For Tasks 1 and 2 (file `demo.c0`) we have given you a head start and put some simple tests into `demo-test.c0` which you should extend. For Task 3 (file `dict.c0`) you should write a few small tests yourself. for Task 4 you can use the Clac files in the `def/` directory. For this assignment, unit tests will not be graded, but they will help you check the correctness of your code, pinpoint the location of bugs, and save you hours of frustration.

Reference implementation. You can run a reference implementation of Clac on Andrew if you'd like to see the results of examples. For your protection, 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).

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. If you find yourself writing the same code over and over, you should write a separate function to handle that computation and call it whenever you need it. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on Piazza if you're unsure of what constitutes good style.

Task 0 (0 points) No points on this assignment are explicitly allocated for style, but we will still review some or all submitted code, and reserve the right to deduct points for severe breaches of style or other issues.

1 Clac: The Demo Version

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.

There is a free demo version which one can use as a Claculator™, and a commercial version which, RPS claims, is a full-scale programming language. In the first task, you will implement the demo version.

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.

```
% ./demo
Clac top level
clac demo>> 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 demo>> -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 demo>> -
11
```

We obtain 11, since $7 - (-4) = 11$. We can quit our interactions by typing `quit`.

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

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

```
% ./demo
Clac top level
clac demo>> 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 supported by the demo version of Clac. 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.

Fundamentally, 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 `demo.c0`.

Task 1 (10 points) *Extend the implementation of the Clac demo version in file* `demo.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 `demo-main.c0` will take lines of input from the terminal (represented as strings) and convert them to a *queue of tokens*. Each token is just a string. This phase of the Clac demo has already been programmed for you, and you are welcome to examine it, but you may not change this code. In Clac, tokens are only separated by whitespace. 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.

2 Conditionals

As a first step in the extension of the Clac demo to the full Clac language we introduce *conditionals*. If we see *in the input queue*

`if`

then we branch based on top of the stack:

- If it is non-zero (which represents *true*), then we just continue with processing the remaining queue of tokens.
- If it is 0 (which represents *false*), we skip the next *two* tokens in effect ignoring them. It is an error if there are less than two tokens remaining in the queue.

Either way, the number we tested on top of the stack will be consumed by the operation.

This is used in conjunction with

`else`

which just skips the following token. If `else` is the last token in the queue, it is an error. `else` may seem like a strange name for this operation. It comes from the the idiom

`if` $token_1$ `else` $token_3$

If you carefully read the definitions of `if` and `else` above you will see that this idiom behaves as follows:

- If the top of the stack is non-zero (*true*), $token_1$ will be processed, and then `else` will cause $token_3$ to be skipped.
- If the top of the stack is zero (*false*), then $token_1$ and `else` will be skipped and $token_3$ will be processed.
- In either case, the top of the stack will be consumed.

The following example illustrates this:

```
clac demo>> 6 1 if 2 else -2 +
8
clac demo>> 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.

Task 2 (3 points) *Extend* `eval` *in file* `demo.c0` *to handle* `if` *and* `else` *according to the specification above.*

3 Dictionaries as Association Lists

We now would like to extend the demo version of Clac to the full version so it can be used as a programming language. Perhaps surprisingly, all we need are *definitions*.

This extension will require us to store defined names in a *dictionary* that associates each name with its definition. For this purpose we program dictionaries using so-called *association lists*. An association list is a linked list where each list node contains a key (here the name) and data item (here the definition). Definitions are simply queues of tokens, so an association list node is defined by

```
struct alist_node {
  string name;
  queue def;
  struct alist_node* next;
};
typedef struct alist_node alist;
```

A dictionary is just a header struct with a pointer to an assocation list.

```
struct dict_header {
  alist* assoclist;
};
typedef struct dict_header* dictionary;
```

The interface to dictionaries is given by:

```
dictionary dict_new();
queue dict_lookup(dictionary D, string name);
   /* lookup returns NULL if name is not defined */
void dict_insert(dictionary D, string name, queue def);
```

Task 3 (7 points) *We have already written the interface and struct definitions, as well as functions* `is_dict` *and* `dict_new`*, in the file* `dict.c0` *Complete the implementation of dictionaries by writing functions* `dict_lookup` *and* `dict_insert` *described next.*

- `dict_lookup(D, name)` returns the definition of *name* in D (which is a queue of tokens). If *name* is undefined (not in the dictionary), it returns `NULL`.
- `dict_insert(D, name, def)` updates the dictionary so that the lookup of *name* in D returns *def*. This must handle both cases: *name* may already be defined in D or not. In the first case we have to override or replace the old definition with the new.

4 Definitions

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

$$: \mathit{name}\ \mathit{token}_1 \ldots \mathit{token}_n\ ;$$

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. *name* can be any token, but if it is a built-in operator or number, then the definition can never be invoked since it is always superseded by the predefined meaning.

Let's consider a simple example.

```
: square dup * ;
```

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

```
% ./clac
Clac top level
clac>> : square dup * ;
(defined square)
(stack empty)
5 square
25
```

Note that `5 square` should be identical to `5 dup *` 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/stacks_queue_string.c0`). This is called the *return stack* or sometimes the *call stack*. 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.

Operand Stack	**Queue of Tokens**	**Return Stack**
(empty)	`3 square 4 square +`	(empty)
3	`square 4 square +`	(empty)
3	`dup *`	(`4 square +`)
3, 3	`*`	(`4 square +`)
9	(empty)	(`4 square +`)
9	`4 square +`	(empty)
9, 4	`square +`	(empty)
9, 4	`dup *`	(`+`)
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. In this example, the return stack has at most one queue on it at any time. When definitions are nested or recursive, it can become arbitrarily deep.

Task 4 (5 points) *After the demo version of your implementation (including conditionals) is working, copy the body of the demo implementation into the body of the* `eval` *function in the file* `clac.c0` *as a starting point for the full version. Note that* `eval` *takes different arguments here since we need to pass it the dictionary, so you cannot simply copy the whole function.*

Do not modify your (hopefully working!) file `demo.c0`*. Implement definitions (*`:...;`*) according to the specification above in the file* `clac.c0`*.*

[Hint: *When retrieving the definition of a name from the dictionary, you will need to copy the definition in order to make sure it is still available for future use. For this purpose we have supplied the function* `queue_copy_read_only` *in the file* `lib/queues_string.c0`*.]*

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

It has the following summary effect:

$$\texttt{fib} : S, n \longrightarrow S, \mathit{fib}(n)$$

(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.

5 Bonus: Clac Programs

Task 5 (bonus points) *Write and submit a file* `bonus.clac` *with one or several cool or surprising Clac programs. Before each example, use the idiom*

```
: comment ...some text... ;
```

to explain what the program does. You can even add unit tests to the file. See `def/fact.clac` *for an example of unit tests.*