15-122: Principles of Imperative Computation, Spring 2020

Programming Homework 5: Clac and EXP

Due: Thursday 27th February, 2020 by 9pm

In this assignment, you will investigate stacks, queues, and dictionaries by implementing two programming languages. The first language, Clac, is a strange new programming language, but it is one that is easy to implement. The second language, EXP, looks more familiar.

The code handout for this assignment is on Autolab. 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 SEVEN (7) PENALTY-FREE HANDIN LIMIT. Every additional handin will incur a small (5%) penalty (even if using a late day). Your score for this assignment will be the score of your last Autolab submission.

Reminder about respecting interfaces: In this assignment, we will take interfaces seriously. When using the implementations of stacks and queues in the `lib` directory of the handout, you should *only* use the interfaces. We will enforce this, in the autograder, by compiling your code against different implementations of the interfaces. Similarly, when working on EXP in Section 3, you should respect the interface of the dictionaries you implemented in Section 2.

Style Grading: In this assignment, we will continue emphasizing *programming style*. We will be looking at your code and evaluating it based on the criteria outlined at `http://cs.cmu.edu/~15122/misc/styleguide.pdf`. We will make comments on your code via Autolab, and will assign an overall passing or failing style grade. A failing style grade will be temporarily represented as a score of -15 points. This -15 will be reset to 0 once you:

1. fix the style issues,
2. see a member of the course staff during office hours **within 5 days** after the grades are released, and
3. briefly discuss the style issues and how they were addressed.

We will evaluate your code for style in two ways. We will use `cc0` with the `-w` flag that gives style warnings — code that raises warnings with this flag is almost certain to fail style grading. Because the `-w` flag does not check for good variable names, appropriate comments, or appropriate use of functions, these issues will be checked by hand.

1 Introducing Clac

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.

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 *
Stack is now [12]
```

Clac responded by printing the stack, which now contains only `12`. We now enter `-9 2 /` and a newline:

```
clac>> -9 2 /
Stack is now [12,-4]
```

At this point the stack contains two integers: `12` (the result of the multiplication) and `-4` (the result of the integer division, which is at the top of the stack). We can add them simply by typing `+` and a newline.

```
clac>> +
Stack is now [8]
```

The stack contains only `8`, since $12 + (-4) = 8$. Note that the integers and operators we input into Clac (they are called *tokens* below) work like a queue: Clac processes them left to right and we enter new ones on the right.

1.1 Rules of Clac

Every Clac operation that you will implement removes a single string (a *token*) from an input queue, and then modifies the operand stack. To specify 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: *"When you dequeue the token `-` from the queue, 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.

The following table gives all the Clac operations that you will implement in this assignment.

Token		Before		After	Note
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 $y = 0$ or if $x = -2^{31}$ and $y = -1$
`**`	:	S, x, y	$\longrightarrow$	S, x^y	Error if $y < 0$

The addition, subtraction, multiplication, and exponent operations should give their answer in terms of 32-bit two's complement modular arithmetic, which makes it easy to implement Clac's math operations in terms of C0's math operations. Whenever the instructions say that Clac should raise an error, you should call the function **`error()`** with an appropriate error message. If a rule requires two integers to be removed from the stack and fewer than two integers are presently on the stack, **`error()`** should also be called.

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.

You can use the function `parse_int(tok, 10)` to check whether a token is an integer in base 10. If the function returns `NULL`, then the string stored as `tok` does not represent an integer. Otherwise, dereferencing the returned pointer will give you the integer that the string stored in `tok` represents.

1.2 Using the Reference Implementation

We can see how our introductory series of instructions affected the stack and the queue in detail by running the `clac-ref` binary that is available on the shared Unix servers, giving it the extra argument `-trace`. Using this argument, we can see the state of the stack and the queue after every step.

```
% clac-ref -trace
Clac top level
clac>> 3 4 * -9 2 / +
                        stack || queue
                              || 3 4 * -9 2 / +
                            3 || 4 * -9 2 / +
                          3 4 || * -9 2 / +
                           12 || -9 2 / +
                        12 -9 || 2 / +
                     12 -9 2 || / +
                        12 -4 || +
                            8 ||
8
```

We can see that `8` was printed out at the end because that was what was at the top of the stack after the queue had been completely consumed.

The full Clac implementation contains many other tokens for calculation, manipulating the stack and the queue, and even defining custom functions. However, the small set of operations above will be enough for this assignment.

1.3 Implementation

Task 1 (6 points) In the file `clac.c0`, you should implement the function `eval`:

```
void eval(istack_t S, queue_t Q)
//@requires S != NULL && Q != NULL;
//@ensures queue_empty(Q);
```

Note that the type is `istack_t`. In the code handout, you have both an interface of stacks of integers (`istack_t`, in `lib/stack_of_int.c0`) and stacks of strings (`stack_t`, in `lib/stack_of_string.c0`).

The `main` function in file `clac-main.c0` takes lines of input and converts 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.

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 leaving the stack in whatever state it was already in. If **`error()`** is ever called, it doesn't matter what remains on the stack or on the queue.

It is possible to write a lot of long and confusing code to safely and efficiently implementing this task, but it is also possible to use helper functions to write very clear and concise code. Try to think about how you would structure your code if you knew you were going to implement full Clac, which has a dozen more tokens. (No matter how good your code looks, your `eval()` function will probably still look like a loop containing a long series of **`if-else`** statements.)

You can reuse code from lecture and the lecture notes, but if you do that, cite your sources!

2 Dictionaries

The series of Clac tokens "`3 4 * -9 2 / +`" from Section 1.2 performs the computation that we would write down on paper as $3 \times 4 + -9 \div 2$. Correctly interpreting this mathematical computation relies on understanding *precedence* or *order of operations*. Specifically, we know that multiplication and division need to be done before addition, because these operations have higher precedence. The order of operations is clear when we look at our Clac program: the tokens `*` and `/` are earlier in the queue than the `+` token.

In this section, you will implement a *dictionary* that allows us to take a mathematical operator (represented as a string, like `"**"`), and find its precedence as an integer (say, `2`). When we initially create dictionaries with the function `dict_new()`, we populate them with an array of strings, where each string `A[i]` contains a whitespace-separated list of all the tokens that should be given precedence `i`. Here's an example:

```
string[] A = alloc_array(string, 3);
A[2] = "**";  // '**' has the highest precedence, 2
A[1] = "* /"; // '*' and '/' have the same precedence, 1
A[0] = "+ -"; // '+' and '-' have the same precedence, 0
dict_t D = dict_new(A, 3);
```

If you haven't already done so, you should look at the `num_tokens()` and `parse_tokens()` functions from the C0 library named `parse`. The `is_infix_array()` function provided to you in the file `lib/utils.c0` gives an example of using these functions.

Task 2 (6 points) In the file `dict.c0`, implement a precedence dictionary with the following interface:

```
//typedef ______* dict_t;

dict_t dict_new(string[] A, int n)
  /*@requires \length(A) == n; @*/
  /*@requires is_infix_array(A, n); @*/ ;

int dict_lookup(dict_t D, string oper)
  /*@requires D != NULL; @*/
  /*@ensures \result >= -1; @*/ ;
```

The function `dict_new(A, n)` must return `NULL` if the same token appears twice in the input. The function `dict_lookup(D, x)` should return `-1` if the string stored as `x` is not present in the dictionary `D`. Efficiency is not a primary concern for this task. See the file `dict-test.c0` for examples.

The implementation is completely up to you, but it should define and make appropriate use of a thorough specification function `is_dict`. Feel free to re-use ideas from previous assignments. If you use code from previous handouts or lecture, make sure to cite your sources.

Whatever you do, you should document your design clearly in comments, write and deploy reasonable data structure invariants, and use contracts to ensure safety and some degree of correctness for your code.

3 The EXP Language

Reverse Polish Systems has hired you to build a more user-friendly language, called EXP, leveraging their existing Clac technology. The EXP language has all the mathematical operations that the mini-Clac you implemented has: exponents, multiplication, division, addition, and subtraction. The difference is that EXP allows us to write expressions as infix. In EXP, we can write "`27 / 3**2 - 2**3`" and the result will be -5, because we evaluate the exponents first, getting $27/9 - 8$, and then evaluate the division, getting $3 - 8 = -5$.

3.1 Parsing EXP Programs

Both Clac and EXP use the same process for turning a string into a queue of tokens. The EXP program above will be transformed into a queue of tokens with the following contents:

```
"27", "/", "3", "**", "2", "-", "2", "**", "3"
```

We won't evaluate EXP programs directly. Directly evaluating infix expressions that are presented in this way is difficult at best. Instead, we will approach the problem indirectly: we will use an algorithm to transforms an EXP program into a Clac program. The example above will be transformed into the Clac program "`27 3 2 ** / 2 3 ** -`". We already have an interpreter that can evaluate this Clac program, so we can use that existing interpreter to run our EXP program!

You can see this happening using the `exp-ref` binary that is also available on the shared Unix servers:

```
% exp-ref
EXP top level
EXP>> 3*4 + -9/2
Corresponding Clac program: 3 4 * -9 2 / +
EXP>> -4*2/3 - 7*7
Corresponding Clac program: -4 2 * 3 / 7 7 * -
```

The algorithm for translating infix EXP programs into postfix Clac programs will need a dictionary in order to determine which tokens should be treated as infix mathematical operations. We'll use the word **operator** to describe a token that appears in the dictionary (and that therefore has a known precedence).

In addition to a dictionary, our translation algorithm will use a stack of operators and two queues. The input queue contains the EXP program we're reading, and the output queue contains the Clac program we're writing. Numbers always get moved immediately from the input queue to the output queue. Operators cannot be immediately moved to the output queue, though. They are placed on the operator stack until it is time to put them onto the output queue.

The algorithm, named the *shunting-yard algorithm* by Edsger Dijkstra,[1] has two phases: we start in the integer phase, and switch phases every time we remove something from the input queue.

[1]The algorithm is also described at `https://en.wikipedia.org/wiki/Shunting-yard_algorithm`. You're welcome to read the Wikipedia article, but don't refer to other sources.

Integer phase: There must be a token on the input queue, and it must be an integer. Put that token on the output queue and switch to the **operator** phase.

Operator phase: In this phase, you will sometimes pop operators off of the stack. *Whenever you pop an operator off the stack, you should add it immediately to the output queue.*

If there is a token *tok* on the input queue, it must be an operator (it must be in the precedence dictionary). Pop just enough operators off of the stack so that the operator on the top of the stack (if there is one) has precedence *strictly less* than *tok*'s precedence. (If the operator stack is initially empty, or if the operator on the top of the stack already has a lower precedence, then there's nothing to do.) Finally, push *tok* on the stack and switch to the **integer** phase.

If there are no tokens on the input queue, then you are done. Pop every operator off the stack (and put it on the output queue).

Example To give a few examples of how the operator phase works, imagine that the operator stack initially contains the following three operators, with `"**"` at the top of the stack:

`"+"` (precedence 0), `"/"` (precedence 1), `"**"` (precedence 2)

The behavior of the operator phase depends on what's on the queue. Here are three possible outcomes:

- If the input queue is empty, then enqueue `"**"`, then `"/"`, and then `"+"` onto the output queue in that order. Return the output queue.
- If the token at the front of the input queue is `"42"`, then the input was not a valid EXP program. Return `NULL`. The same happens when passed an invalid token like `"x"` or `"@"`.
- If the token at the front of the input queue is `"*"` and the dictionary gave this token precedence 1, first pop `"**"` off the stack and add it to the output queue. Then pop `"/"` off the stack and add it to the output queue. Now the precedence of the token on the top of the stack is strictly less than 1, so push `"*"` onto the stack. Switch to the integer phase and repeat.

Associativity This version of the shunting-yard algorithm results in all operators being treated as *left-associative*. We definitely want to interpret subtraction and division as left associative. As an example, consider the expression $1 - 2 - 3$. This is always evaluated the same way as $(1 - 2) - 3$, or "`1 2 - 3 -`" in Clac; it would be incorrect to read it as $1 - (2 - 3)$, or `1 2 3 - -` in Clac. However, the exponentiation operator is often treated as *right associative*. (Python does this, for instance.) EXP treats exponents as left associative to make your implementation simpler.

3.2 Invariants

Before you start implementing this algorithm, think about the invariants that are at work! Operationally, before you add a new operator to the input stack, you remove items from the stack until the new operator has a higher precedence than the operator on the top of the stack. We can write a specification function that captures this operational behavior as an invariant.

Task 3 (2 points) In a new file `parse.c0`, write a function which checks that a stack `S` contains valid operators with strictly increasing precedence. The highest-precedence operation must be on the top of the stack.

```
bool is_precstack(dict_t D, stack_t S)
//@requires D != NULL && S != NULL;
```

When the function returns, the contents of the stack should be the same as they were when the function was called.

Important: Even though `is_precstack(D, S)` will be an important loop invariant of our algorithm, we are not usually allowed to write it in a `//@loop_invariant` contract! This is because C0 has checks to make sure that programs will always run the same way with or without contracts enabled.

The function `is_precstack()` must actually modify the stack `S` in order to respect the stack interface. The C0 compiler assumes that calling a function that modifies the stack in any way could cause the program to behave differently when contracts are enabled. Even though your function returns the stack to its original condition, the C0 compiler cannot prove this and therefore refuses to allow `is_precstack()` in a contract.

If you want to use this function in a loop invariant to help you debug your code, you can call the compiler with the extra argument `--no-purity-check`. We'll compile your code with this option on Autolab — but that means you have to be extra careful that your `is_precstack()` function returns the stack to its original state!

3.3 Implementation

Task 4 (7 points) In the file `parse.c0`, implement the algorithm described in Section 3.1:

```
queue_t parse(dict_t D, queue_t input)
//@requires D != NULL && input != NULL;
//@ensures \result == NULL || queue_empty(input);
```

The function should return `NULL` if the input is not a well-formed EXP program according to the operators in the dictionary `D`. If the function returns `NULL` it doesn't matter what is left on the input queue.

The `exp-main.c0` file parses programs based on the following precedence table:

1. `||`
2. `&&`
3. `<`, `>`, `==`, `!=`
4. `<<`, `>>`
5. `+`, `-`
6. `*`, `/`, `%`
7. `**`

If you run your `exp` program with the `-x` option, it will attempt to run the parsed EXP program with your Clac implementation.

```
% ./exp -x
EXP top level
EXP>> 3 * 4 + -9 / 2
Corresponding Clac program: 3 4 * -9 2 / +
Result: 8
```

This will only work if you restrict yourself to EXP programs using `**`, `*`, `/`, `+`, or `-`. Otherwise, your Clac implementation will crash when it finds a token like `<` that it doesn't know how to process.

3.4 Extending EXP using Clac

For the last part of this assignment, you will write some code in Clac, instead of writing code in C0. You will need to use the full Clac implementation described in Figure 2, which is implemented by the `clac-ref` program.

The code you write will be in the form of Clac *definitions*, which you have already encountered in the lab activity on Clac. For example, the Clac definition "`: dup 1 pick ;`" defines a new token `dup` that takes whatever is at the top of the stack and duplicates it. Therefore, the user-defined `dup` token implements the rule $S, x \longrightarrow S, x, x$.

Task 5 (4 points) In a file `exp-defs.clac`, write Clac definitions that implement the Clac tokens described in Figure 1. The file already includes definitions of `>`, `==`, and `!=` to get you started.

Token		Before		After	Note
`>`	:	S, x, y	$\longrightarrow$	$S, 1$	If $x > y$
`>`	:	S, x, y	$\longrightarrow$	$S, 0$	If $x \leq y$
`==`	:	S, x, y	$\longrightarrow$	$S, 1$	If $x = y$
`==`	:	S, x, y	$\longrightarrow$	$S, 0$	If $x \neq y$
`!=`	:	S, x, y	$\longrightarrow$	$S, 1$	If $x \neq y$
`!=`	:	S, x, y	$\longrightarrow$	$S, 0$	If $x = y$
`||`	:	S, x, y	$\longrightarrow$	$S, 1$	If either $x \neq 0$ or $y \neq 0$
`||`	:	S, x, y	$\longrightarrow$	$S, 0$	If both x and y are 0
`&&`	:	S, x, y	$\longrightarrow$	$S, 1$	If both $x \neq 0$ and $y \neq 0$
`&&`	:	S, x, y	$\longrightarrow$	$S, 0$	If either x or y is 0
`<<`	:	S, x, y	$\longrightarrow$	$S, x \times 2^y$	If $0 \leq y < 32$
`>>`	:	S, x, y	$\longrightarrow$	$S, x/2^y$	If $x \geq 0$ and $0 \leq y < 32$

Figure 1: Clac functions to implement in Task 5

Task 6 (bonus) (Extra challenge 1) Make the shift operators always raise an error when $y < 0$ or $y \geq 32$.

Task 7 (bonus) (Extra challenge 2) Make the right-shift operator perform division that rounds towards negative infinity, so that it works correctly on negative x as well as nonnegative x.

With your definitions, you can parse any valid EXP program into Clac, and then run it with the Clac reference interpreter:

```
% ./exp
EXP top level
EXP>> 3 * 4 > -9 / 2 && 3 << 2 == 24 >> 2 - 1
Corresponding Clac program: 3 4 * -9 2 / > 3 2 << 24 2 1 - >> == &&
% clac-ref exp-defs.clac
clac>> 3 4 * -9 2 / > 3 2 << 24 2 1 - >> == &&
1
```

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