15-122: Principles of Imperative Computation — Spring 2023

Lab 06: Misclaculations — Tuesday February 21st

Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. We encourage discussing problems with other students in this lab!

During the programming assignment on Clac (not just during this lab), we furthermore encourage you to share any interesting Clac programs you write with other students on Piazza.

Setup: *There is no handout for this lab.*

Postfix expressions

You are used to infix arithmetic expressions where the operator is in between its two operands (e.g., `3 + 4`). In *postfix* expressions, the operator follows ("post") its two operands (e.g., `3 4 +`). Postfix expressions can be used as operands in other postfix expressions without the need for parentheses. Here are some examples:

```
INFIX               POSTFIX
1 + 2 * 3 - 4       1 2 3 * + 4 -
(1 + 2) * 3 - 4     1 2 + 3 * 4 -
1 + 2 * (3 - 4)     1 2 3 4 - * +
```

In an infix expression, the order of operation is determined by precedence conventions and the use of parentheses. In a postfix expression, it is determined by the position of the operators.

To evaluate a postfix expression, we can treat it as a queue of *tokens* of operands and operators (the front of the queue is on the left and the back on the right), and then use a stack to evaluate it. For each token in the postfix expression, if it is an operand (e.g., `1`), it is pushed on the stack. If it is an operator, the top two operands are popped from the stack, evaluated using that operator, and the result is pushed back on the stack. Once all tokens are processed from the queue (from left to right), the final result of the computation should be at the top of the stack.

(1.a) Convert the infix expression

```
125 - 15 * (3 + 2) / (6 * 4 + 1)
```

to postfix by hand, and then trace the algorithm described above to compute the value of the postfix expression. The result should be the same as if you calculated the infix expression directly.

Clac

For your next 15-122 programming assignment, you will implement a stack-based calculator named *Clac* that evaluates postfix expressions.

We describe what each token does by means of *rules* of the form $S \mid\mid Q \longrightarrow S' \mid\mid Q'$, where S and Q are the stack and the queue before evaluating this token, and S' and Q' are the stack and the queue after evaluating it. Note that this token is at the front of Q (on its left).

Most Clac tokens, when removed from the queue, only manipulate the stack. Here is a description of a few tokens that stand for operations. For example, the rule for `+` is read as follows: *when the front of the queue is* `+` *and values* x *and* y *are a the top, they are replaced by the value* $x + y$. Here, S and Q denote the rest of the stack and the rest of the queue, respectively.

Before			After		
Stack	**Queue**		**Stack**	**Queue**	**Condition or Effect**
S	n, Q	$\longrightarrow$	S, n	Q	for $-2^{31} \leq n < 2^{31}$ in decimal
S, x, y	`+`$, Q$	$\longrightarrow$	$S, x+y$	Q	
S, x, y	`-`$, Q$	$\longrightarrow$	$S, x-y$	Q	
S, x, y	`*`$, Q$	$\longrightarrow$	$S, x*y$	Q	
S, x, y	`/`$, Q$	$\longrightarrow$	$S, x\,/\,y$	Q	error, if div by 0 or overflow
S, x, y	`%`$, Q$	$\longrightarrow$	$S, x\,\%\,y$	Q	error, if mod by 0 or overflow
S, x, y	`**`$, Q$	$\longrightarrow$	S, x^y	Q	error, if y < 0
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	
$S, x_n, \ldots, x_1, n$	`pick`$, Q$	$\longrightarrow$	$S, x_n, \ldots, x_1, x_n$	Q	error, if $n <= 0$
S, x	`print`$, Q$	$\longrightarrow$	S	Q	print x followed by newline
S	`quit`$, Q$	$\longrightarrow$	_	_	exit Clac

Some operations remove other tokens from the queue besides the one being evaluated. The operations `if` and `skip` are good examples of this:

Before			After		
Stack	**Queue**		**Stack**	**Queue**	**Condition or Effect**
S, n	`if`$, Q$	$\longrightarrow$	S	Q	$n \neq 0$
S, n	`if`$, tok_1, tok_2, tok_3, Q$	$\longrightarrow$	S	Q	$n = 0$
S, n	`skip`$, tok_1, \ldots, tok_n, Q$	$\longrightarrow$	S	Q	$n \geq 0$

Note that the queue must contain at least three tokens after **`if`** when the top of the stack (n) is 0, but it may contain any number of tokens after **`if`** (possibly even zero) if $n \neq 0$.

A reference (i.e., completed) implementation `clac-ref` is available on AFS. Use the `-trace` option to see how the stack and queue change as an expression is evaluated:

```
% clac-ref -trace
clac>> 2 3 * 4 +
                                    stack || queue
                                          || 2 3 * 4 +
                                        2 || 3 * 4 +
                                      2 3 || * 4 +
                                        6 || 4 +
                                      6 4 || +
                                       10 ||
```

Note that the stack is written left (bottom) to right (top). Enter `quit` to exit Clac.

(2.a) Use `clac-ref` to compute the value of your postfix expression from Exercise 1. What is the maximum size of the stack as this expression is evaluated?

1.5pt

We can define new tokens in Clac by using the "`:`" token followed by the name of the new token, then the sequence of tokens that it stands for, and a final "`;`" token. For example, say we are interested in a new operation that **dup**licates the topmost element on the stack, i.e., we would like to have a token `dup` described by the following rule:

$$\texttt{dup} \;:\; S, x \;||\; Q \;\longrightarrow\; S, x, x \;||\; Q$$

`dup` is not predefined in Clac, but we can use `:` and `;` to define it as a new token `dup` as follows:

```
: dup 1 pick ;
```

Make sure you understand why `1 pick` duplicates the element at the top of the stack.

(2.b) Now using `dup`, create a new token that **square**s the number on the top of the stack, as described by the following rule:

$$\texttt{square} \;:\; S, x \;||\; Q \;\longrightarrow\; S, x^2 \;||\; Q$$

3pt

Clac is a powerful language with some tricky tokens. Let's play with some of them!

When writing code, we often make use of control flow through writing if/else statements. A very common structure is as follows:

```
if (condition) {
  A
}
else {
  B
}
```

where `A` is a sequence of operations to be run if `cond` is true, and `B` is run otherwise.

(2.c) Determine the equivalent Clac expression to the above C0 code. You may assume that the stack is currently of the form $S, cond$. We consider the condition to be **false** when $cond = 0$ and **true** otherwise.

You can assume that the Clac translations of A and B are just single tokens.

(Hint: the `if` and `skip` tokens may be useful here.)

(2.d) Implement an operation that computes the absolute value of the number at the top of the stack, i.e., one that implements the following definition:

$$\texttt{abs} \ : \ S, x \ \longrightarrow \ S, |x|$$

You do not need to worry about overflow for this task.

It may be useful to first write out the absolute value function in C0 pseudocode with an if/else statement to help determine what A and B should be, then use your solution from (2.c)

```
: A _____ ;
: B _____ ;
: abs _____ ;
```

4pt

Once you have the definitions, check your answer by running it in `clac-ref` with the trace flag.

Challenge Problems (Just for Fun)

(3.a) In (2.c), you wrote a translation with the assumption that the Clac translations of A and B are single tokens.

Now, write the equivalent Clac expression given that the Clac translation of A is m tokens long, and the translation of B is n tokens long.

Although Clac lacks loop constructs such as **`for`** and **`while`**, it's still possible to create complex programs using recursion.

(3.b) Using one or more recursive helper definitions, implement the following Clac operation *without using the built-in `**` operator*:

$$\texttt{powtwo} \ : \ S, x \ \longrightarrow \ S, 2^x$$

(3.c) Again, *without using the built-in `**` operator*, implement the following Clac operation:

$$\texttt{pow} \ : \ S, x, y \ \longrightarrow \ S, x^y$$