# 15-122: Principles of Imperative Computation

## Lab 5: Misclaculations

Tom Cortina, Rob Simmons

**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 your neighbors as you work through this lab!

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

**Grading:** For two points, you must correctly answer (1.a), (2.a), and (2.b). Write down your answers and get a TA to check them. For three points, finish the rest of the lab.

## Postfix expressions

You are used to using infix expressions where the operator is in between its two operands (e.g. `3 + 4`). In postfix expressions, the operand follows ("post") its two operands (e.g `3 4 +`). Postfix expressions can be used as operands in other postfix expressions. 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 - * +
```

Note that order of operations determines what is converted from infix to postfix first. Also, postfix expressions never have parentheses.

To evaluate a postfix expression, we can separate it into a queue of *tokens* of operands and operators, 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, combined using that operator, and the result is pushed back on the stack. Once all tokens are processed from the queue, 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 the next programming assignment for class, you will implement a stack-based calculator named Clac that evaluates postfix expressions. 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?

Clac has additional features. The operator < pops an operand $y$ off the stack, then pops another operand $x$ off the stack, and pushes 1 on the stack if $x < y$ or 0 if $x \geq y$. The sequence of tokens

`if` $a$ `1 skip` $b$ (where $a$ and $b$ are replaced by integers)

pops the top operand off the stack, and pushes $a$ on the stack if the popped value was 1 or $b$ if the popped value was 0.

**(2.b)** Determine what this function computes in Clac, substituting different values for $x$ as you test it.

```
x x 0 < if -1 1 skip 1 *
```

We can create new functions in Clac by using the `:` token followed by the function name and the operations and a final `;` token. For example, here is a function that squares the number on top of the stack:

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

**(2.c)** Implement a function that performs the computation in (2.b) by assuming that one copy of $x$ is on the top of the stack before the function is executed.

## Clac reference

Most Clac tokens, when removed from the queue, only manipulate the stack:

| 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` | : | $S, x, y$ | $\longrightarrow$ | $S, y, x$ | |
| `rot` | : | $S, x, y, z$ | $\longrightarrow$ | $S, y, z, x$ | |
| `pick` | : | $S, x_n, \ldots, x_1, n$ | $\longrightarrow$ | $S, x_n, \ldots, x_1, x_n$ | error, if $n <= 0$ |
| `print` | : | $S, x$ | $\longrightarrow$ | $S$ | print $x$ followed by newline |
| `quit` | : | $S$ | $\longrightarrow$ | _ | exit Clac |

The `if` and `skip` operations, on the other hand, also manipulate the token queue:

| Before | | | | After | | | |
|---|---|---|---|---|---|---|---|
| **Stack** | | **Queue** | | **Stack** | | **Queue** | **Cond** |
| $S, n$ | $\|\|$ | `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`, $tok_1, \ldots, tok_n, Q$ | $\longrightarrow$ | $S$ | $\|\|$ | $Q$ | $n \geq 0$ |