Stack Machine Interpreter for `thdc`

PS03 - COMP590 - Spring 2019

Overview

In the second part of Tar Heel Desk Calculator, `thdc`, you will implement a stack machine to interpret and compute the calculations a user enters into the program. It will rely upon your `Tokenizer` from part 1 to produce a stream of `Token` values.

Once this part of the problem set is completed, you will be able to enter reverse-polish notation expressions like `8 4 / 2 +`, the equivalent of `8 / 4 + 2` in infix notation, and see the result of `4` printed out. As each number is encountered, you will push it onto your machine's stack. As each numerical operator is encountered, you will pop the last two values off the stack and push the result of the operation back onto it. The `p` command will print the top-most value of the stack and the `f` command will print the full stack from top-to-bottom. There is an opportunity for 5 extra credit points by adding support for registers.

You should leave your "debug mode" which began with the "-d" flag intact. In part 2 you are implementing the normal, flagless mode of `thdc`.

For this problem set's autograding, only *basic tests* will be posted online before the deadline. The basic tests are to confirm your program successfully compiles and your output is aligned with the autograder's expectations. After the deadline, the *complete suite of tests* will run for your final autograding score. As such, you should follow the *test-driven development (TDD)* workflow as discussed in lecture 7. Test early and test often. After the *complete suite of tests* runs, there will be a 3-day grace period where you can earn back up to half of the points missed in the complete suite.

We are providing a skeleton outline for your `Machine`. You are free to ignore this starting point and design your own solution. All solutions will be autograded based on the same suite of *integration tests* and hand-graded on the same coding standards for your personal *unit tests*, style, and documentation. If you craft your own solution, its top-level comments should ellucidate what design parameters and trade-offs led to your design.

Example Usage of `thdc` Part 2

Here are some example use cases of `thdc` that should work after you've completed Part 2. The text in black is expected input, text in blue is expected output.

```
$ thdc
p
thdc:  stack empty
1 2 f
1
2
3
f
3
2
1
+ + p
6
q
```

```
$ thdc
1 +
thdc:  stack empty
p
1
2 + p
3
9 / p
0.3333333333333333
q
```

Differences Between `thdc` and `dc`

Your program should mimic the original `dc`'s arithmetic behavior with the exception of handling decimal points. You should play around with `dc` on your VM to test expected behaviors. One disclaimer: it is a precise numerical calculator with arbitrary precision while yours is only concerned with floating point arithmetic. By default it computes 0 digits of precision in division. To enter a mode in `dc` that will display 3 digits of precision, for example, begin `dc` enter `3k`.

Getting Started

You will continue working out of your current `thdc` repository.

Skeleton Code

A recommended starting point for your stack machine is provided at the following URL:

<raw.githubusercontent.com/comp590-19s/starter-ps02-thdc/master/src/machine.rs>

To get started with the skeleton code, you will need to add it to a `machine.rs` file in the `src` folder. From `src/main.rs`, you will need to import the `Machine` struct the same as how your `Tokenizer` was imported. You should read through the public methods of `Machine` before continuing on.

Your first task is to "wire" a `Machine` value into your program's flow of execution. You will want a `Machine` struct whose lifetime is longer than the REPL `loop` in `main`, but is not `static`. You will need to update the `eval` function in `main.rs` to accept a mutable reference to your `Machine` so that its state persists after each line of user input is evaluated. The `Machine` struct *also* has an `eval` method whose job is to take a string of input and carry out the instructions. You will notice in the skeleton code it returns a `Vec` of `String`s (to make it easier to test). The skeleton design encourages you to print those values from `main.rs`. It is your job to establish a Tokenizer in the `eval` function of `Machine` and to iterate through the input tokens to carry out their semantics.

There are a few specifications to note for each variant of `Token`.

1. `p` - (`Print`) - When the stack is empty, results in the message `thdc: stack empty`. When the stack is nonempty, results in the topmost value on the stack without popping it.
2. `f` - (`FullStack`) - When the stack is empty, results in nothing. When the stack is nonempty, results in each value on the stack being displayed on its own line starting from the top of the stack (most recently pushed) all the way to the bottom.
3. `Number` - All number token values should be pushed directly onto the stack.
4. `Operator` - All arithmetic operators are binary meaning they require two values from the stack. If fewer than two values are available on the stack, leave any value on the stack in its place and respond with `thdc: stack empty`. When two or more values are on the stack, pop them both, apply the operator such that the left hand operand was the earlier value pushed onto the stack and the right hand operand was the later value. Push the result back onto the stack.
5. `Unknown` - For Unknown tokens, respond with `thdc: <char> unimplemented` and substitute `<char>` with the unknown character.

Grading Rubric Breakdown

Autograding Levels

1. 50 points - Basic Tests
2. 30 points - Complete Suite of Tests

Hand-graded Points

1. 10 points - Style and Documentation

- Did you run `:RustFmt` in `vim` to ensure proper indentation and formatting?
- Did you name variables meaningfully?
- Did you add comments to segments of code that are not self-documenting?
- Did you organize your code well?

2. 10 points - Unit Tests

- Did you write unit tests for your `Machine`'s `eval` method?
- Did you write unit tests for your `Machine`'s helper methods (token type)?
- Are your unit tests logically organized and well named?

5pts Extra Credit - Registers

When you have completed your implementation for `thdc` you are encouraged to attempt adding support for registers. Each register can be thought of as a variable which stores a single number value. There are 26 registers whose names are the lowercase letters `a` through `z`. There are two kinds of register commands: **s**tore and **l**oad. Each register command is made of two characters, first, the command, and second, the register letter. There can be no whitespace between the command letter and the register. If either `s` or `l` is followed by a character not in the range `a` through `z`, the `s` or `l` should be considered `Unknown`.

1. `s - Store` - When the command `s<char>` is issued, the current value on the top of the stack should be popped off and stored in the register `<char>`.

2. `l - Load` - When the command `l<char>` is issued, the current value stored in the register `char` should be pushed onto the stack. All register values are initially `0`. Loading a value from a register *does not* clear the register.