Tar Heel Desk Calculator - `thdc` - Tokenizer

PS02 - COMP590 - Spring 2019

Overview

In the next series of problem sets you will implement the essence of `dc` (1969), the original "Desk Calculator" program. The little language of `dc` is Unix's *oldest surviving language.* It was originally implemented in B, the predecessor to the C programming language.

Using `dc` will feel a bit foreign because it uses reverse-polish notation (RPN). This means operators *follow* their operands as oposed to *infix notation* where operators are placed *between* their operands. For example, `1 + 2` is expressed in RPN as `1 2 +`. An interesting property of RPN languages is that there is no need for parenthesis to handle special cases for orders of operations. For example, using infix notation the expression `(1 + 2) * 3` requires parenthesis for the addition to occur before the multiplication. With RPN the previous expression is written as `1 2 + 3 *` . Notice that is distinct from `1 2 3 * +` which is equivalent to the infix expression `1 + 2 * 3`.

There are two key benefits to starting our journey into little languages with a simple, RPN language like `dc`'s:

1. It is a *regular language*, meaning its production rules can be expressed with a *regular definition.* You will come to learn such a language is less expressive than a *context-free language* (required for infix-based languages) and needs simpler machinery to process. You will implement a *context-free language* processor in the `thbc` assignment.
2. Since the *operands* of each *operator* are expressed *before* the operator, once an operator is encountered `thdc` can evaluate it immediately. The *structure* of the language is inherent in the ordering and no additional work is needed to *parse* it. Notice this simplification on the implementation side comes with the higher cost of users needing to do the legwork to *translate* their calculations into RPN.

This series of problem sets will break the implementation down into three steps:

1. Tokenization - Using a stream of characters as inputs you will generate a stream of typed *Tokens* (such as Operator, Number, Print) before concerning yourself with their interpretation.
2. Stack Machine - In the next part you will implement a software machine to process tokens one-by-one and carry out their instructions.
3. Registers - Once the fundamentals of `thdc` are working you will extend its language and machinery to support named registers (think variables).

Example Usage of `dc`

Here are some example use cases of `dc` to motivate the direction we are moving in. You are encouraged to follow along and tinker with your own examples. Note your behavior for *this part of the problem set* will not perform the calculations yet. The lines with comments following the pound symbol are the ones you can enter into `dc`. Those without comments are what it prints out in response.

```
$ dc
# In this program we'll add 1 + 2
p               # print the top value of stack
dc: stack empty
1               # push 1 onto the stack
p               # print the top value of stack
1
2               # push 2 onto the stack
p               # print the top value of the stack
2
+               # add the top two values to the stack and push result
p               # print the top value of the stack
3
q               # quit

$ dc
1 2 + 3 * p     # (1 + 2) * 3 and print value remaining on stack
9
1 2 3 * + p     # 1 + 2 * 3 and print value remaining on stack
7
f               # print the full stack
7
9
+               # add the top two values of the stack and push result
p               # print top of stack
16
q               # quit
```

In addition to the usual suspects of arithmetic (+, -, *, /), there are two other operators we're concerned with initially: `p` prints the top value of the stack and `f` prints the full stack from top-to-bottom. The character `q` quits the program.

Differences Between `thdc` and `dc`

The most fundamental difference between `thdc` and `dc` is that yours will not implement arbitrary precision arithmetic. In `dc` you can calculate things like *pi* precisely to however many digits you'd like without any roundoff loss. In `thdc` we'll make do with the limitations of 64-bit floating point arithmetic. There are many other advanced features of `dc` we will not implement in this series of problem sets, as well, such as strings, macros, conditionals, stack-based registers, and so on.

Tokenization

The first step of processing a language is called *lexical analysis*. Its purpose is to raise the level of abstraction from thinking in terms of *characters* to thinking in terms of *tokens*. For example, consider the following input string `10 20 + p`. At the character level, with space characters represented by `<sp>`, the nine input characters are:

```
'1' '0' <sp> '2' '0' <sp> '+' <sp> 'p'
```

The purpose of a *tokenizer*, also commonly called a *lexer*, is to take an input string such as the one above and yield *tokens* which have meaning in the context of their language.

```
Num(10) Num(20) Operator('+') Print
```

Notice the space characters were filtered out and are not expressed as *tokens*. Your assignment in this problem set is to implement the tokenizer for `thdc` given the following token definitions. In this regular definition, terminals are surrounded in single quotes:

```
digit       -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
digits      -> digit digit*
number      -> digits | digits '.' | digits '.' digits
print       -> 'p'
full_stack  -> 'f'
operator    -> '+' | '-' | '/' | '*'
unknown     -> _any terminal character not recognized by rules above_
token       -> print | full_stack | operator | number | unknown
```

A few notes on the regular definition above:

1. Whitespace is not addressed by this definition. It is expected you will ignore spaces, tabs, and new line characters.
2. The definition of the *digits* non-terminal can be read as "*digits* are a *digit* followed by zero or more digits". The `*` is the Kleene Star operator that means zero or more instances.
3. Greediness is expected in tokenization. For example, the input string `123` should be consumed greedily into a single `Token` whose value is `123` versus non-greedily into three `Token`s whose values are `1`, `2`, and `3`.
4. As in `dc`, you cannot enter negative numbers directly. *How would you get a negative value on the stack?*

Getting Started

GitHub classroom starter URL: https://classroom.github.com/a/5yXgU6UR

Please follow the link above to setup your repository for this problem set. Once you've done so, you'll want to find its Clone link and be sure you choose the SSH option that begins with `git@github.com:comp590-...`. Copy that link to your clipboard.

While logged in to your VM in a terminal, issue the following commands and replace the words in <tags> with the values specific to you:

```
$ cd $HOME
$ git clone <paste>
$ cd ps02-thdc-<your-github-username>
```

You should go ahead and edit `Cargo.toml` to have your name in it and fill in the honor pledge on both `src/main.rs` as well as `src/tokenizer.rs`.

Skeleton Code

To focus your work on the concept of tokenization, we setup a project structure for you. Here is an overview of the files provided:

`Cargo.toml` - The project's Cargo configuration file. You should update the author to be you. Also notice the `structopt` dependency is listed here. This is what informs Cargo your project depends on the `structopt` external library. It automatically downloads and compiles it as needed during the build process.

`src/main.rs` - The program's `main` entry point is in this file.

It uses an external library called `structopt` to handle command-line argument parsing. Try running with the `-h` flag as follows: `cargo run -- -h`. You'll notice a debug mode is established with the `-d` short flag or `--debug` long flag. The `structopt` library makes handling command-line arguments *much* easier than trying to interpret them yourself as you experienced in `thecho`. For this problem set you will want to run `cargo run -- -d`, the short flag for *debug mode*, for interacting with your tokenizer.

The `src/main.rs` file reads in lines of input from `stdin` and, in debug mode, iterates through the tokens generated by your `Tokenizer` printing them one by one. You should read through this file and its comments to get a feel for how this comes together. In future problem sets we will not provide as much skeleton code.

`src/tokenizer.rs` - The `Tokenizer` skeleton code establishes it as an implementation of the `Iterator` trait so it can iterate through an input string's `Token`s just the same as you can iterate through anything else in Rust. You should read through the comments and code in this file a few times over to become very familiar with it. You should also look in the `eval` function of `src/main.rs` to understand its usage. It is already setup to handle `Print` and `Unknown` tokens. Your job is to extend its implementation to produce `FullStack`, `Operator`, and `Number` tokens based on the grammar, while ignoring all spaces, tabs, and newlines.

You will notice the `Tokenizer` has as its only member a `Peekable Chars` iterator. This is just like the `Chars` iterator you used in `thecho` with one additional method available: `peek`.

The `peek` method returns an `Option<char>` but does not move the `Chars` iterator forward. This allows you to "peek" one character ahead without advancing the iterator.

The `Tokenizer`'s `Iterator impl` defines the `next` method which produces an `Option<Token>`. You will notice it peeks one character ahead to decide which kind of `Token` it is going to take next and then dispatches the work of producing the `Token` off to a helper function.

Both the `next` method and the `helper` functions have unit tests defined beneath them. In the course of adding functionality to your `Tokenizer`, you should add tests along the way. A large part of the hand graded score for this problem set will assess your tests. A few sample tests are provided and you can run them with `cargo test`.

Example Usage and Output

Before you begin work you should try `cargo run -- -d` and entering `p` and an unknown character to see the starting point the starter code leaves you in.

A few sample use case scenarios your final version should be able to produce are demonstrated below.

```
$ cargo run -- -d

f +   -   */
== TOKENS ==
FullStack
Operator('+')
Operator('-')
Operator('*')
Operator('/')

2.5
== TOKENS ==
Number(2.5)

100
== TOKENS ==
Number(100.0)

1 2.0+pf
== TOKENS ==
Number(1.0)
Number(2.0)
Operator('+')
Print
FullStack

q
```

Grading Rubric Breakdown

Autograding Levels

1. 10 points - FullStack Token
2. 10 points - Operator Token
3. 20 points - Number Token
4. 20 points - Ignoring Whitespace

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. 30 points - Unit Tests

- Did you write unit tests for helper methods?
- Did you write unit tests for the `next` method?
  - Do they cover variations of whitespace?
  - Do they cover each kind of Token `next` can generate?
- Are your unit tests logically organized and well named?