Tar Heel Basic Calculator - `thbc` - Part 1

PS04 - COMP590 - Spring 2019

Overview

The `bc` utility is a *basic calculator* with more familiar, infix-based syntax than `dc`'s reverse-polish notation. The original Unix `bc` was written in at Bell Labs in 1975 by Lorinda Cherry, Robert Morris, and others. Since `dc` was already tried and tested at the time, the original implementation of `bc` was as a front-end compiler whose source was `bc` syntax and whose target was `dc` syntax. This allowed `dc` to concern itself with the precise calculations, registers, and so on, while `bc` focused more on syntactical ergonomics. Your implementation of `thbc` will follow in this original spirit[1]. The `bc` utility is widely used today as it is a capable calculator, with support for variables, readily available in your shell.

Your `bc` impementation will span two problem sets:

1. Parsing Basic Expressions - In this part of the problem set you will parse input expressions such as `1 + 2 * 3` and output `dc` syntax such as `1 2 3 * + p`. From the command line, your `thbc` program can be *piped* to your `thdc` or the actual `dc` to perform the computations.
2. Growing the Language - In the next part of the problem set you will add exponentiation and variables to the language. This will allow you to do the following:

```
x = 2
n = 8
x^n
```

The purpose of this problem set is to gain experience with recursive descent parsing. Once your parser is complete, you will also visit its parse tree to emit the *target* `dc` syntax representation.

[1] Modern implementations of `bc` are self-contained and do not rely upon `dc` for the calculation machinery.

Example Usage of `thbc`

The skeleton implementation of `thdc` has two flags to help you during the development process. You can use neither, either, or both. The `-t` flag will output the tokens of your input string and the `-p` flag will output the parse tree. The demo below shows a few examples of the final product for this part of the problem set:

```
$ thbc
1 + 2 * 3
1 2 3 * + p
(1+2) * 3
1 2 + 3 * p
quit
```

```
$ thbc -tp
1 + 2 * 3
== Tokens ==
Number(1.0)
Operator('+')
LParen
Number(2.0)
Operator('*')
Number(3.0)
RParen

== Parse Tree ==
BinOp {lhs: Num(1.0), op: '+', rhs: BinOp {lhs: Num(2.0), op: '*', rhs: Num(3.0)}}

1 2 3 * + p
quit
```

```
$ thecho '1 + 2 * 3' | thbc | thdc
7
$ thecho '(1+2) * 3' | thbc | thdc
9
```

Note: For the final piping examples to work you'll need to add `thecho`, `thbc`, and `thdc` to your path. For a simpler pipeline, while in the `thbc` working directory, you could instead pipe with the built-in `echo` and `dc`. However, we encourage figuring out how to pipe between your own programs for the fully enlightened experience.

Differences Between `thbc` and `bc`

There are *many* differences between `thbc` and `bc`. The true `bc` is effectively a numerical programming language complete with C-like syntax for functions and loops. Please see the Wikipedia page for more information. Your `thbc` will be concerned with basic expressions in the first version and add support for variables in the second.

Program Architecture

Entrypoint

The file `src/main.rs` is already setup to parse the command-line options and correctly compose the `Tokenizer`, `Parser`, and `DCGen` components. You should read through this file first and understand its organization.

During the first phase of development, while you are working on the `Parser`, your interactive tests will want to make use of the `-t` and `-p` flags to display your Tokens and resulting Parse Tree.

Tokens and `Tokenizer`

All of the lexing infrastructure you'll need for `bc` is included in the starter code of `src/tokenizer.rs`. You should review it and it should look famaliar as it is not significantly different from `thdc`'s. It adds tokens for `LParen` and `RParen` (as well as for `Register` and `Assignment`, but you will not need those for this part of the problem set).

Parser and `Expr` Parse Tree Nodes

The skeleton of your recursive descent `Parser` can be found in `src/parser.rs`. The bulk of your work will be here. You'll see two variants of `Expr` nodes for now, `BinOp` for binary operations (`+`, `*`, and so on), and `Num` for number literals. You will add additional kinds of nodes to your parse tree in the next part of `bc`. There are two helper functions defined internal to the `parser` module to ease the construction of the enum variants, `binop`, and `num`, respectively.

The public API for the `Parser` is the static `parse` method. Notice it returns `Result<Expr, String>`, as do all of the helper methods you'll write. There *can* be errors in parsing and Rust's `Result` enum is the idiomatic way to return them. Refer to the book and your GRQs for more coverage on `Result`. The `parse` method is *mostly* setup for you. However, you'll want to add some additional logic at the `TODO` comment to ensure there are no remaining tokens to consume. If there are, you should return an `Err` whose `String` is `"Expected end of input, found {:?}"`, where the placeholder is substituted with the next `Token` found. A test case for this requirement is provided for you in the skeleton code.

The private API for the `Parser` is broken into two sections. The first is where you will need to implement the logic for each of the grammar rules. You should also test these rules. Basic test cases for levels 0, 1, and 2 are provided. You do not need to add additional cases for levels 0 or 1, but you should more thoroughly test level 2. You will need to write your own tests for level 3. The second section of the private API of `Parser` are helper methods that you should make use of to simplify your code. You should read the documentation for them in the file before beginning.

CodeGen in `src/dc_gen.rs`

Once your parser for this stage of the project is complete, you will need to implement the `to_dc` function and its helper `visit` function. These are given the resulting `Expr` parse tree and will produce a string that is a valid `dc` statement to compute the `Expr` and print its result.

Grammar

The grammar provided below is LL(1) and suitable for top-down recursived descent parsing.

```
Expr        -> MaybeAddSub
Atom        -> lparen Expr rparen | number
MaybeMulDiv -> Atom MulDivOp?
MulDivOp    -> ('*'|'/') Atom MulDivOp?
MaybeAddSub -> MaybeMulDiv AddSubOp?
AddSubOp    -> ('+'|'-') MaybeMulDiv AddSubOp?
```

It is organized such that the precedence of operators is explicitly addressed in the grammar itself. This grammar will be discussed in class, however, the intuition on the precedence is important to understand. Since your parser will recursively follow the rules, the closer a production happens to the "base case" (in our grammar the `Atom`) the higher precedence that rule will have as it greedily consumes next tokens.

The best way to implement a parser is to divide and conquer as few production rules as possible at a time. Ensure those rules are working and well tested, then take the next rule, and repeat. Since this is likely the first recursive descent parser you've implemented, we'll offer a suggested path toward growing it. We suggest you start with the *highest precedence* rules first, they naturally have either none or the fewest dependencies on other rules, and then work your way out to rules that depend on them, and so on.

Recommended Implementation Strategy

When fleshing out your `Parser`, you should run tests (many are provided for the initial levels). You can isolate your tests by name with `cargo test <name>`, for example, `cargo test lvl0`. Additionally, when you are using your program interactively you should run with `cargo run -- -tp` to show Tokens and your resulting parse tree. Finally, we encourage you to `comit` early and often as you're progressing through projects and reaching minor milestones along the way.

Level 0

The simplest form of the grammar are numbers or numbers optionally surrounded by parenthesis.

```
Expr        -> Atom
Atom        -> '(' Expr ')' | number
```

Level 1

Next, you'll want to add support for parsing expressions which may contain *one* multiplication or division operation. After this level is complete your grammar can handle expressions such as `2 * 3`, but not `2 * 3 * 4`. Should, in your parser's handling of `MaybeMulDiv`, you peek ahead to realize it is a multiplication or division expression, you'll want to *pass* the "left-hand side" atom you initially parsed. Notice, the production rule for `Expr` changes here.

```
Expr        -> MaybeMulDiv
Atom        -> lparen Expr rparen | number
MaybeMulDiv -> Atom MulDivOp?
MulDivOp    -> ('*'|'/') Atom
```

Level 2

This step extends the `MulDivOp` rule to be recursively defined. It enables your grammar to recognize strings like `2 * 3 * 4`. While the change in code is small, this is perhaps the most conceptually difficult and important step in growing your parser.

```
Expr        -> MaybeMulDiv
Atom        -> lparen Expr rparen | number
MaybeMulDiv -> Atom MulDivOp?
MulDivOp    -> ('*'|'/') Atom MulDivOp?
```

Since these operators are the same order of precedence and should be applied left-to-right, you'll want to ensure your resulting parse tree grows down the left-hand side, *not* the right-hand side. Convince yourself why this is useful by drawing a tree of non-commutative ops. For example, suppose you are parsing `1/2/3`, the resulting `Expr` should be:

```
BinOp {
    lhs: BinOp { lhs: Num(1.0), op: '/', rhs: Num(2.0) },
    op: '/',
    rhs: Num(3.0)
}
```

You'll want to add some additional tests for level 2, such as four chained factors and use of parenthesis to override the default left-to-right precedence.

Level 3

Finally, you'll need to add support for addition and subtraction which are lower precedence than multiplication and division. The same strategy you took on for multiplication and division should work here. Notice again here the production rule for `Expr` has changed to incorporate the rules for addition and subtraction.

```
Expr        -> MaybeAddSub
Atom        -> lparen Expr rparen | number
MaybeMulDiv -> Atom MulDivOp?
MulDivOp    -> ('*'|'/') Atom MulDivOp?
MaybeAddSub -> MaybeMulDiv AddSubOp?
AddSubOp    -> ('+'|'-') MaybeMulDiv AddSubOp?
```

`dc` Code Generation - Level 4

Once you believe your `Parser` code is complete, you're ready for `dc` code generation. Your implementation will be in `dc_gen.rs`. The purpose of the `to_dc` function is to take an `Expr` tree as input and return a valid `dc`/`thdc` `String` representing the computation that ends in a `p` to print its output. For example, the `Expr` tree for `1 + 2` should result in `1 2 + p`.

You should keep the `pub` function defined as is in terms of its signature. Beyond that, you are free to implement its logic and any helper methods you find necessary in this file with no prescription. You will want to add additional tests for this stage of the problem set, as well.

Getting Started

GitHub classroom starter URL: https://classroom.github.com/a/BbGpYmeu

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 ps04-thbc-<your-github-username>
```

You should go ahead and edit `Cargo.toml` to have your name in it and fill in the honor pledge on all files in `src`.

Grading Rubric Breakdown

Autograding Levels

Basic tests for all levels will be released as soon as they're ready. On the deadline, as per the `thdc` problem set, a complete suite of tests will be added.

1. 10 points - Parser Level 0 - Atom
2. 10 points - Parser Level 1 - Single Multiplication/Division Expr
3. 20 points - Parser Level 2 - Chained Multiplication/Division Expr
4. 20 points - Parser Level 3 - Chained Addition/Subtraction Expr
5. 20 points - CodeGen Level 4 - Correct Conversion of Expr to `dc` string

Hand-graded Points

1. 10 points - Style and Documentation
2. 10 points - Unit Tests