Computer Science 3MI3 – 2020 homework 5

"Fizzbuzz"-ing in Ruby

Mark Armstrong

October 12th, 2020

Contents

Introduction

The "fizzbuzz" problem is a very simple programming task, sometimes used in interviews to check for a basic understanding of iterating and branching constructs.

We will investigate various possible approaches to this problem in Ruby, as a way to become comfortable with the language. We begin with the familiar looping statements, and then move to using "higher-order" methods, as well as solving a generalisation of the problem.

Boilerplate

Submission procedures

Submission method

Homework should be submitted to your McMaster CAS Gitlab respository in the `cs3mi3-fall2020` project.

Ensure that you have **pushed** the commits to the remote repository in time for the deadline, and not just committed to your local copy.

Naming requirements

Place all files for the homework inside a folder titled `hn`, where `n` is the number of the homework. So, for homework 1, the use the folder `h1`, for homework 2 the folder `h2`, etc. Ensure you do not capitalise the `h`.

Unless otherwise instructed in the homework questions, place all of your code for the homework in a single file in the `hn` folder named `hn.ext`, where `ext` is the appropriate extension for the language used according to this list:

- For Scala, `ext` is `sc`.
- For Prolog, `ext` is `pl`.
- For Ruby, `ext` is `rb`.
- For Clojure, `ext` is `clg`.

If multiple languages are used in the homework, submit a `hn.ext` file for each language.

If the language supports multiple different file extensions, you must still follow the extension conventions above.

Incorrect naming of files may result in up to a 10% deduction in your grade.

Do not submit testing or diagnostic code

Unless you are instructed to do so in the homework questions, **you should not submit testing code with your homework submission**.

This includes

- any `main` function,
- any `print` statements which output information **that is not directly requested as console output in the homework questions**.

If you do not wish to remove diagnostic print statements manually, you will have to find a way to ensure that they disabled in your final submission.

For instance, by using a wrapper on the print function or macros.

Due date and allowance for technical difficulties

Homework is due on the second Sunday following its release, by the end of the day (midnight). Submissions past 00:00 may not be considered.

If you experience technical difficulties leading up to the submission time, please contact Mark **ASAP** with the details of the problem and, if possible, attach the current state of your homework to the communication. This information will help ensure we are able to accept your submission once the technical difficulties are resolved.

Proper conduct for coursework

Individual work

Unless explicitely stated in the homework questions, all homework in this course is intended to be *individually completed*.

You are welcome to discuss the content of the homework in the public forum of the class Microsoft Teams team homework channel, though obviously solutions or partial solutions should not be posted or described.

Private discussions about the homework cannot reasonably be forbidden, but such discussions should follow the same guidelines as public discussions.

Inappopriate collaboration via private discussions which is later discovered by course staff may be considered academic dishonesty.

When in doubt, make the discussion private, or report its contents to the course staff by making a note of it in your homework.

To clarify what is considered appropriate discussions of homework content, here are some examples:

1. Discussing the language features introduced or needed for the homework.
    - Such as relevant builtin datatypes and datatype definition methods and their general use.
    - Code snippets that are not partial solutions to the homework are welcome and encouraged.
2. Questions of the form "What is meant by `x`?", "Does `x` really mean `y`?" or "Is there a mistake with `x`?"
    - Of course, questions of those form which would be answered by partial solutions are not considered appropriate.
3. Questions or advice about errors that may be encountered.
    - Such as "If you see a `scala.MatchError` you should probably add a catch-all `_` case to your `match` expressions."

Language library resources

Unless explicitely stated in the questions, it is not expected that you will use any language library resources in the homeworks.

Possible exceptions to this rule include implementations of datatypes we discuss in this course, such as lists or options/maybes, if they are included in a standard library instead of being builtin.

Basic operations on such types would also be allowed.

- For instance, `head`, `tail`, `append`, etc. on lists would not require explicit permission to be used.
- More complex operations such as sorting procedures would require permission before you used them.

Additionally, the standard *higher-order* operations including `map`, `reduce`, `flatten`, and `filter` are permitted generally, unless the task is to implement such a higher-order operator.

Part 1: Fizzbuzzing by loops [5 points]

In Ruby, create a method `fizzbuzzLooper` which, given a list (presumably of integers, though it may contain any type) creates a new list whose elements are the elements of the original list converted into strings, unless they are

- an integer divisible by both `3` and `5`, in which case they are replaced by `"fizzbuzz"`,
- an integer divisible by `3` but not by `5`, in which case they are replaced by `"fizz"`, or
- an integer divisible by `5` but not by `3`, in which case they are replaced by `"buzz"`.

You may want to make use of the `to_s` method on integers; by convention, `to_s` on any type converts objects of that type into strings.

(Technically, your method should probably work given any type of collection, not just lists; but the result should be a list in any case.)

Your `fizzbuzzLooper` must make use of some manner of looping construct.

- Such as a `loop`, `while` loop or `for` loop.

Because this is a fairly trivial programming task, the marking of this question (and to some extent the marking of the remainder of the homework) *will take *elegance* more into account than usual. (meaning you are expected to follow good coding practices, especially *not repeating yourself.*)

Part 2: Fizzbuzzing by iterators (higher-order methods) [10 points]

Construct another method `fizzbuzzIterator`, whose behaviour is identical to `fizzbuzzLooper`, but which is defined using an "iterator" method rather than a looping construct.

See this online tutorial on collections and iterators. In particular, look into the iterators `each` and `collect`.

These iterators take a *block* as argument. A block is, essentially, Ruby's "lambda" construct. Blocks are delimited by braces, `{}`, and may have arguments, which are listed at the beginning and delimited by pipes, `||`. So the anonymous function `x → x + 1` would be written `{ |x| x + 1 }` in Ruby.

So for instance,

```
[1,2,3].each { |x| puts(x) }
```

outputs each element of the list `[1,2,3]`.

Part 3: Generalising fizzbuzzing [20 points]

We now consider a slight generalisation to the fizzbuzzing problem, which we will call "zuzzing".

To generalise the problem, we assume that we may have several rules which should be applied to the elements of this list, instead of just the two (if it's disible by 3, output "fizz", if it's divisible by 5, output "buzz".)

We want to create a method which accomodates any number of rules, and where the rules can be arbitrary predicates on the elements of the list (not just "`x → x is divisible by ~k`".)

To represent this multitude of rules, we use a list of lists, assuming each of the lists within the list contain two elements;

- the first element being a `lambda` for the rule, and

- the second element being a `lambda` for the string associated with that rule.
  - We use a `lambda` here as well so that the resulting string may depend upon the element.

(The keyword `lambda` applied to a block allows you to store that block using a variable, or in our case, in a list; we are still essentially using blocks in this question.)

For instance, to get the original behaviour of "fizzbuzz" using this "zuzz", we would use the rules

```
rules = [[lambda { |x| x % 3 == 0 }, lambda { |x| "fizz" }],
         [lambda { |x| x % 5 == 0 }, lambda { |x| "buzz" }]]
```

as in

```
zuzzer([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],rules)
```

The reason we use a list of lists of lambdas here to encode the rules, rather than a hash table or other construct, is that *the order of the rules matters*. If more than one rule applies to an element, all such rules should be applied *in order* to build the resulting string. For instance, with the "fizzbuzz" rules above, notice that the "fizz" rule comes before the "buzz" rule so that if an element is divisible by both 3 and 5, the result is `"fizzbuzz"`, not `"fizz"`, `"buzz"` or `"buzzfizz"`.

Create the method `zuzzer`.

Part 4: Generalised fizzbuzzing in Scala [10 bonus points]

Implement the generalised fizzbuzzing operation from part 3 in Scala.

Make what you feel are necessary adjustments to the types or implementation details, and describe your choices in comments. Your solution may be (sometimes subjectively) judged based on the choices you make. The purpose of the comments is then for you to convince us your choices are appropriate.

Part 5: Generalised fizzbuzzing in Prolog [10 bonus points]

Implement the generalised fizzbuzzing operation from part 3 in Scala.

Make what you feel are necessary adjustments to the types or implementation details, and describe your choices in comments. Your solution may

be (sometimes subjectively) judged based on the choices you make. The purpose of the comments is then for you to convince us your choices are appropriate.

Testing

:TODO: