Lab 1: Higher-Order Functions

October 15, 2016

Contents

Objectives

By the end of this lab, you will:

- know how labs work in cs019
- understand that functions can be treated as first-class citizens (and what that means)
- learn the names of several common higher-order functions, implement them, and practice using them
- be able to write your own higher-order functions

1 Labs in cs019

There are a few important lab policies to keep in mind in cs019:

- All labs, unless otherwise specified, should be completed in pairs (if there are an odd number of students, a group of three may be formed).

- Partners should take turns using the keyboard, with each turn lasting around 15 minutes.
- You cannot work with the same person for more than one lab.
- You must get checked off by a TA at each checkpoint before moving on to the next section (no skipping ahead).
- In order to pass a checkpoint, relevant code must be fully functional and be augmented with docstrings and sufficient test cases. Some functions may have additional implementation-specific requirements.
- If you are unable to make your assigned lab section for a given week, you must email the head TAs by the preceding Saturday at 11:59pm (exceptions granted for emergency circumstances).
- If you do not finish a lab during your assigned lab section, you must finish it (either independently or with your partner(s)) and get it checked off during TA hours (of any TA) within a week of your section in order to receive full credit.
- During TA hours, there will be a "lab checkoff line" separate from the normal question line. You must use this if you're getting checked off for a lab, but can only use it for this purpose. If you have both a lab checkoff and a normal question, you must put yourself in both queues.

Labs are graded under the following criteria:

- If the lab is completed during section or checked off within a week it is given full credit.
- If the lab is started but not completed during section (and not checked off within a week) it is given half credit.
- If you're more than 20 minutes late for your lab section you are only eligible to receive half credit, and must complete the lab in order to receive said credit.
- If you do not show up for section you will receive no credit.

2 What Is a Higher-Order Function?

So far, you have seen all sorts of functions that input and output `Numbers`, `Strings`, and `Lists`. Did you know that *the functions themselves* can be passed as inputs and outputs? For instance, a function may take both a `Number` and another function as parameters, and use them both to generate some output:

```
fun fun-plus-one(num :: Number, func :: (Number -> Number)) -> Number:
  func(num) + 1
end
```

Because functions can be passed and produced in the same way as other data types (like `Number`s, `String`s, and `List`s), functions can be referred to as "first-class citizens."

A higher-order function is a function that either consumes a function as input, produces a function, or both. In the example above, `fun-plus-one` is a higher-order function because it accepts a function as a parameter. In this lab, we will focus exclusively on higher-order functions that consume other functions.

Note: Data types like `Number`, `String`, and `List` are "first-class citizens" too.

3 Map

Say that you have a list of the temperatures in Providence for every day in August. The temperatures were measured in degrees Fahrenheit, but you want to report them in degrees Celsius. With the same data, you want to report each day as "too hot" if the temperature is above 90 degrees Fahrenheit, "too cold" if the temperature is below 70 degrees Fahrenheit, and "just right" otherwise.

```
check:
  f-to-c([list: 131, 77, 68]) is [list: 55, 25, 20]
  goldilocks([list: 131, 77, 68]) is
  [list: "too hot", "just right", "too cold"]
end
```

Task: Implement these two functions, called `f-to-c` and `goldilocks`, respectively, without using `map`. Be sure to write function annotations and test cases for each. Each person in your pair should type for one of these. The formula for converting degrees Fahrenheit, F, to degrees Celsius, C, is $C = \frac{5}{9}(F - 32)$

What you might find with your implementations of these functions is that they have seemingly important commonalities. In particular, in each, you probably recurred through the list of temperatures, and transformed each following some procedure. Even though the procedure used for transforming each element of the list varied between the two functions, there was probably a lot of reused code in your implementations.

It is tedious to write a unique recursive implementation for each function like this that you use in your code (as you may have found working through Nile so far!). Even more important, code like this is more verbose and more difficult to read than it could be.

It'd be ideal to abstract out the common structure and draw attention to just the inner procedure. As you may have guessed (or known), this is where higher-order functions (HOFs) come in! In particular, the HOF `map` can be used to implement `f-to-c` and `goldilocks`.

Map is a higher-order function for lists. Given a list, `l`, of elements of some type `T`, and a function `f` (called the *functor*) that operates on an input of the same type `T`, `map` produces a list of the same length, where each element is the output of applying the functor `f` to the corresponding element in `l`.

Here's how `f-to-c` could have been written using `map`, shown without their doc strings and test cases for brevity:

```
fun f-to-c(f-list :: List<Number>) -> List<Number>:
  map(lam(f): 5/9 * (f - 32) end, f-list)
end
```

Task: Implement `goldilocks` with `map`. Hint: You may not want to use the lambda notation here.

Note: If you need a refresher on Pyret's lambda notation for anonymous functions, check out the documentation

Note: There is a shorthand syntax for lambdas in Pyret that you may prefer. Wherever you would write `lam(<parameter list>): <function body> end`, you can instead write `{(<parameter list>): <function body>}`. For example, we can write the sum function as `{(a, b): a + b}`.

Task: Now, to prove your understanding of the `map` function, write your own implementation of it. Name your implementation `my-map`. Use the map documentation as a guide. Feel free to use the two functions that you wrote earlier for two of your test cases.

⋆ **Before continuing, call over a TA to check your work so far.**

4 Filter

Map is not the only higher-order function that will be useful to you in cs019. We will learn some others, starting with Filter. `filter` works just like a filter in the real world: some things can pass through, but others cannot. For a list l of elements of some type `T`, you specify which elements can pass through by supplying a function (the *predicate*) that consumes elements of type `T` and produces a `Boolean`. If the predicate applied to an element is `false`, then that element is filtered out, and if its value is `true`, then that element is kept in the output list.

Below is an example of how you could use `filter` to get only the positive numbers in a list of numbers.

```
check:
    lon = [list: -1, 6, -2, 5, 7, 0, -4]
    filter(lam(num): num > 0 end, lon) is [list: 6, 5, 7]
end
```

Below are function headers for two other functions that can be implemented using `filter`. `tl-dr` eliminates all lists in `lol` whose lengths are greater than `length-thresh`. `eliminate-e` eliminates all Strings from `words` that contain the letter e.

```
fun tl-dr<T>(lol :: List<List<T>>,
             length-thresh :: Number) -> List<List<T>>:

fun eliminate-e(words :: List<String>) -> List<String>:
```

Task: Implement the two functions above using `filter`. As in Nile, you may not use any String built-ins except string-to-code-point(s). Hint: one way to implement `eliminate-e` involves using `filter` twice.

Task: Write your own implementation of the `filter` function. Name your implementation `my-filter`.

⋆ **Before continuing, call over a TA to check your work so far.**

5 Fold

Fold, sometimes called *reduce*, is a higher-order function that is used to combine the elements of a list into a single value. Let's look at the type signature to break down the different parts:

```
fun fold<T1, T2>(f :: (T2, T1 -> T2),
                 base :: T2, lst :: List<T1>) -> T2:
```

`fold` applies a procedure, `f` (referred to as the *folder*), to combine each element of the input list, `lst`, into one value of type `T2`. `f` takes two parameters: the result so far (the accumulator) and the current element of the list, beginning with `base` and the first element of the list.

Perhaps a simple example will make things more clear. Suppose we want to find the sum of the elements of a list of numbers. Our list `lon` will be our list of numbers; our folder `f` will be the addition function, which takes in two numbers and computes their sum; and our base value will be 0 (do you see why?).

Here is that example written in code, with `lon` as our list of numbers.

```
check:
    lon = [list: 7, 3, 2]
    fold(lam(acc, cur): acc + cur end, 0, lon2) is 12
end
```

Table 1: Step-by-step Execution

`acc`	`cur`	`f(acc, cur)`
0	7	7
7	3	10
10	2	12

This use of `fold` begins by invoking the folder function with the `base` (0) as the initial `acc` (accumulator) and the first element of the list (7) as the `cur` (current). Then the output of that first invocation ($7 + 0 = 7$) becomes the accumulator in the next step and the next element in the list becomes the current. . . and so on.

Now we're going to look at a similar example of how to use `fold`:

```
fun add-boolean(acc :: Number, cur :: Boolean) -> Number:
  if cur:
    acc + 1
  else:
    acc
  end
end

check:
  attempts = [list: true, false, false, true]
  fold(add-boolean, 5, attempts) is 7
end
```

Table 2: Step-by-step Execution

`acc`	`cur`	`add-boolean(acc, cur)`
5	true	6
6	false	6
6	false	6
6	true	7

Note that the data type of the output is different from the data type of the input list and how the folder `add-boolean` handles this. Also note that we started with a `base` of 5, which became the initial accumulator.

Below are function headers for two other functions that can be implemented using `fold`.

`list-product` computes the product of all of the elements of a list of numbers, and `list-max` finds the maximum number in a list of numbers.

```
fun list-product(lon :: List<Number>) -> Number:

fun list-max(lon :: List<Number>) -> Number:
```

Task: Implement the two functions above using `fold`.

Task: Write your own implementation of the `fold` function. Name your implementation `my-fold`.

⋆ **Before continuing, call over a TA to check your work so far.**

6 Map2

In situations where you have two lists of the same length and for all indices i, the ith element of one list corresponds to the ith element of another list of the same length (say, for example, they're two different columns from the same Table), you may find that you want to recur through them simultaneously, applying the same procedure to each pair of corresponding elements. As a simple example, suppose you have two lists: one is a list of nouns, `noun-list`, and the other is a list of adjectives that describe those nouns respectively, `adj-list`. You want to produce a list of the nouns with their descriptions. This can easily be done with Pyret's built-in map2 function with the code below.

```
check:
    noun-list = [list: "kitten", "puppy", "student"]
    adj-list = [list: "fluffy", "ugly", "smart"]
    map2(lam(n, a): a + " " + n end, noun-list, adj-list)
        is [list: "fluffy kitten", "ugly puppy", "smart student"]
end
```

Now you try. Suppose that you're teaching a class, and you want to write a program to determine whether or not each student has passed the class. You have two lists of data, `score` and `extra-credit`. `score` has, for each student, a Number corresponding to their final score in the class. `extra-credit` has, for each student, a Boolean indicating whether or not they completed an extra credit activity. The lists are parallel, meaning that the ith student in `score` is the same student as the ith student in `extra-credit`. A student passes the class if they have a score of over 75, or if they have a score of over 65 and completed the extra credit. You want to have a list that stores, for each student, a Boolean indicating whether or not they passed the class.

Task: Using `map2`, implement the function `who-passed`, which will take the lists `score` and `extra-credit` and produce the desired list.

Note: Another way to do this is to create your own `Student` structure that contains the relevant data for a single student and then `map` over a list of `Student`s. All the same, implement using `map2` here.

Task: Write your own implementation of the `map2` function. Name your implementation `my-map2`.

Note: The analogous `map3` and `map4` are also built in to Pyret.

⋆ **Before continuing, call over a TA to check your work so far.**

7 Creating Your Own: Maximizing Revenue

So far, we've seen a few of the higher-order functions built into Pyret's `List` library. But that's just the tip of the higher-order function iceberg; recall that *any* function that takes in another function as input is a higher-order function. There are cases in which you might want to write your own higher-order function to make your code more general. Here we explore one of those examples.

A firm (which, for simplicity, sells only one good) wants to maximize its revenue, the amount of money that it receives from customers. It faces a trade-off: if the firm increases the price of the good, fewer customers will buy it, but the firm will receive more money for each good sold. However, if the firm decreases the price of the good, more customers will buy it, but the firm will receive less money for each good sold.

This trade-off is formalized by a *demand function*, which maps the price of a good to (an approximation of) the total quantity of that good that people will buy.

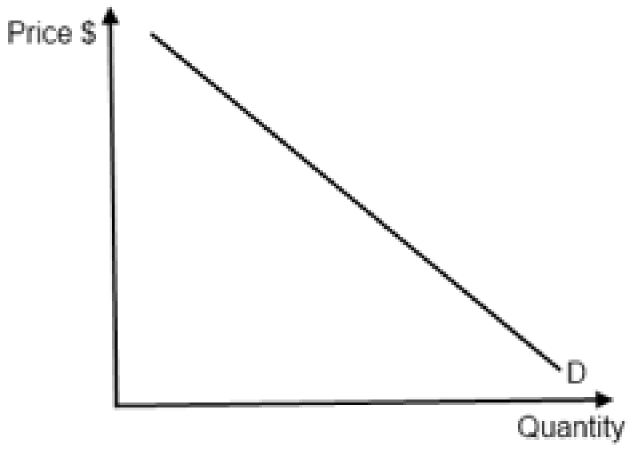

Figure 1: A simple demand function

In a typical demand function, the quantity decreases as the price increases, because it is usually the case that fewer people are interested in purchasing a more expensive item.

Task: Write a higher-order function, `best-price`, which takes in a demand function and a list of potential prices and computes the optimal price, the price that will yield the greatest revenue.

Note: `revenue = price * quantity demanded`

Imagine how you would write `best-price` if you didn't use a higher-order function. You wouldn't be able to take the demand function as an input, so you would probably have to hard-code it in and then change your code when the market changes or when you need it for a different good (with a different demand function). Since we wrote `best-price` as a higher-order function, though, we avoided this. This is what we mean when we say that higher-order functions allow you to write more general code.

8 An Important Final Note

As we've seen, higher-order functions are useful to abstract out repetition in code and to make your code more general. Often, they also make your code easier to read, but not always! When you write code, make sure to prioritize readability, and try to use higher-order functions only if they make your code easier to understand. Keep this in mind, and use your best judgment. Still, feel free to experiment in this class. We TAs will help you by pointing out if your code is difficult to read.

⋆ **Once a lab TA signs off on your work, you've finished the lab! Congratulations! Before you leave, make sure both partners have access to the code**

you've just written.