Homework 6: Higher-Order Procedures
Due: 10:00 PM, Oct 17, 2017

Introduction and Objectives

For this assignment, the only problem that requires use of anonymous procedures (“lambdas”) is All You Need is ... Fold (Problem 5). All others can be done by using helper procedures, but many of them can be solved more cleanly using lambdas, which we’ll be talking about both on Wednesday and on Friday. We recommend that you start thinking about the problems (or even solving them) as soon as possible, but re-examine them in light of your newfound knowledge about lambdas after Friday.

Why use lambdas if you don’t have to? Because it’s a great skill, and will serve you well in your next two projects. Katie says “and you also don’t have to write out everything in the design recipe for lambda-based procedures, but you do for helper procedures!” (Note from Spike: not having to write it out doesn’t mean you shouldn’t think about it!)

By the end of this assignment, you will be able to:

1. Use map, filter, and fold, perhaps with anonymous procedures (i.e., lambdas)
2. Write familiar procedures using fold
3. Practice writing good test cases given a specification
How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Map2, Fold, All You Need is . . . Fold, and Testing/Debugging Practice questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files. In particular, each should be named for the corresponding problem, as follows (e.g., fold.rkt corresponds to Fold):

- README.txt
- map2.rkt
- fold.rkt
- allfold.rkt
- debugging.txt

For this assignment, all files you turn in that contain code must be Racket files, so they must end with extension .rkt. If you are using a departmental linux system, all your solution files should reside in your ~/course/cs0170/homeworks/hw06 directory.

For this and every assignment, you should also have a README.txt file whose first line contains only your CS-department email address, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

README.txt:
jhughes@cs.brown.edu
There’s nothing to say except that I’m turning in three code files plus this README the way the instructions say that I should.

To hand in your solutions to these problems, you have two options:

Option 1: Zip your hw06 directory into a file hw06.zip, and hand in with [this Google form].

Option 2: Use cs0170_handin hw06 to submit each file from the department machines.

See [this Piazza post] for more details.

Practice

1 Fun with map (Practice)

Task: Grace the greyhound absolutely hates when people misspell her breed as “grayhound”! Using map, write a procedure correct-name that replaces every occurrence of "grayhound" in a list of strings with "greyhound".

Examples:
(correct-name (list "he" "loves" "his" "grayhound"))
=> (list "he" "loves" "his" "greyhound")

(correct-name (list "she" "wants" "a" "grayhound" "or" "a" "pug"))
=> (list "she" "wants" "a" "greyhound" "or" "a" "pug")

(correct-name (list "grace" "the" "grayhound"))
=> (list "grace" "the" "greyhound")

Task: Sometimes you don’t care about the value of an integer, just whether it is positive, negative, or zero. Using map, write a procedure `whats-your-sign` that consumes a list of integers and produces a list of strings. The ith entry in the output list should be a string corresponding to the sign of the ith integer in the input list; "positive" if the integer is positive, "negative" if it is negative, or "zero" if it is zero.

Examples:

`(whats-your-sign (list 1 2 3))
=> (list "positive" "positive" "positive")

`(whats-your-sign (list -1 0 1))
=> (list "negative" "zero" "positive")

Task: Using map, write a procedure `make-singletons` that consumes a list of data and produces a list whose ith element is a list of data whose only element is the ith element of the input list.

Example:

`(make-singletons (list 1 2))
=> (list (list 1) (list 2))

`(make-singletons (list "A" "C" "G" "T"))
=> (list (list "A") (list "C") (list "G") (list "T"))

Note: You can code each of these procedures either by defining a helper or by using anonymous procedures with lambda.

2 Unfold (Practice)

Write a procedure `unfold`, which consumes a one-argument procedure, f, and a natural number, n, and produces the following list:

(list (f n) (f (- n 1)) (f (- n 2)) ... (f 0))

Examples:

`(unfold factorial 5)
=> (list 120 24 6 2 1 1)

`(unfold summorial 6)
=> (list 21 15 10 6 3 1 0)
Problems

3 Map2

Task: Write a procedure `map2` which consumes a two-argument procedure, `proc`, and two lists, `alod1` and `alod2`, of equal length, and which produces a single list whose first element is the result of applying `proc` to the first elements of `alod1` and `alod2`, whose second element is the result of applying `proc` to the second elements of `alod1` and `alod2`, and so on.

Examples:

```racket
(map2 + (list 1 2 3) (list 3 2 1))
=> (list 4 4 4)
(map2 string-append (list "CS" "01") (list "CI" "70"))
=> (list "CSCI" "0170")
```

Note: Racket’s built-in `map` procedure is very versatile. It can be used like this, for example:

```racket
(map + (list 1 2 3) (list 3 2 1))
=> (list 4 4 4)
```

As you might have guessed, you cannot use Racket’s built-in `map` procedure to complete this task. Instead, your solution should use recursion explicitly, thereby demonstrating your understanding of the functionality of `map` (and `map2`).

Task: Using `map2`, write a one-line procedure `pairwise-max` which consumes two lists of integers, `alon1` and `alon2`, of equal length, and which produces a list whose `i`th element is the greater of the `i`th element of `alon1` and the `i`th element of `alon2`.

Hint: Feel free to use Racket’s built-in `max` procedure.

Example:

```racket
(pairwise-max (list 1 5 3) (list 4 2 6))
=> (list 4 5 6)
(pairwise-max (list 1 5 3) (list -4 10 -6))
=> (list 1 10 3)
```

4 Fold

Consider a list of numbers:

```racket
(cons 5 (cons 4 (cons 3 (cons 2 (cons 1 empty)))))
```

What if we replaced every occurrence of `cons` with a different procedure, say `+`, and `empty` with a different value, say `0`:
(+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0))))))

From the point of view of higher-order procedures, these two expressions are equivalent. In each, a (lower-order) two-argument procedure is applied sequentially to a list, as follows: the procedure is first applied to the last element of the list and a base value in that order; the procedure is next applied to the second to last element of the list and the result of the first procedure application in that order; the procedure is then applied to the third to last element of the list and the result of the second procedure application in that order; and so on. This sequence of procedure applications is called folding.

**Task:** Write the procedure `fold`. It consumes three arguments: a two-argument procedure, a base value, and a list of data; and it applies the fold transformation to the input list using the given procedure and base value.

**Note:** Your procedure should take as input the aforementioned three arguments in the *exact* order given above.

**Examples:**

```
(fold + 0 (list 3 4 5))
=> (+ 3 (+ 4 (+ 5 0)))
=> 12
```

```
(fold * 1 (list 3 4 5))
=> (* 3 (* 4 (* 5 1)))
=> 60
```

```
(fold cons empty (list "c" "d" "e"))
=> (cons "c" (cons "d" (cons "e" empty)))
```

**Note:** The `fold` procedure is also built into Racket but, for reasons that will become apparent later on in the semester, it is called `foldr`.

**Task:** Write the following procedures using a single call to Racket’s built-in fold procedure, `foldr`. In other words, do not use recursion explicitly.

**Note:** Once you’ve written the two procedures below, feel free to use Racket’s built-in versions of `append` and `flatten` in your code.

1. `my-append`, which consumes two lists and produces another list that contains all the elements of the first, in their input order, followed by all the elements of the second, in their input order. You may assume the input lists are both of some type ’a

**Examples:**

```
(my-append (list 1 2 3) (list 4 5 6))
=> (list 1 2 3 4 5 6)
```

```
(my-append (list "I" "love") (list "CS17"))
=> (list "I" "love" "CS17")
```

```
(my-append empty (list 0 0 0))
=> (list 0 0 0)
```
2. my-flatten, which consumes a non-empty list of lists, and produces another list that contains all the elements of the first sublist, in order, followed by all the elements of the second sublist, also in order, and so on. You may assume the input lists all contain data of some type 'a.

Examples:

\[
\begin{align*}
\text{my-flatten (list (list 1) (list 2 3) (list 4 5 6))} & \Rightarrow (1 2 3 4 5 6) \\
\text{my-flatten (list (list "a" "b") (list "c" "d" "e"))} & \Rightarrow ("a" "b" "c" "d" "e") \\
\text{my-flatten (list empty empty empty)} & \Rightarrow \text{empty}
\end{align*}
\]

5 All You Need is . . . Fold

**Task:** Write each of the procedures below using a single call to Racket’s built-in fold procedure, foldr, and either an anonymous procedure or a helper procedure as an argument to foldr (though we suggest you use an anonymous procedure over a helper). You should do no explicit recursion yourself. You must follow these rules to earn full credit.

1. my-map

- Input: a one-argument procedure, \( \text{proc} \), and a list of data, \( \text{alod} \) such that \( \text{proc} \) operates on data of the type in \( \text{alod} \)
- Output: a list of data whose elements are the result of applying \( \text{proc} \) to each element of \( \text{alod} \)

Examples:

\[
\begin{align*}
\text{my-map even? (list 1 2 3)} & \Rightarrow (\text{false true false}) \\
\text{my-map sub1 empty} & \Rightarrow \text{empty}
\end{align*}
\]

2. my-filter

- Input: a one-argument predicate, \( \text{pred} \), and a list of data, \( \text{alod} \)
- Output: a list consisting of all the elements of \( \text{alod} \) on which \( \text{pred} \) evaluates to true

Examples:

\[
\begin{align*}
\text{my-filter cons? (list (list 17) empty (list 18 19))} & \Rightarrow (\text{list 17} (\text{list 18 19})) \\
\text{my-filter even? (list 38 37 65 63 1 2 3 4 40)} & \Rightarrow (\text{list 38 2 4 40})
\end{align*}
\]
3. any?

- Input: a one-argument predicate, pred, and a list of data, alod such that pred operates on data of the type in alod
- Output: a boolean indicating if pred evaluates to true for any datum in the list, and false otherwise

Examples:

```scheme
(any? odd? (list 1 3 5 7 9))
=> true

(any? (lambda (x) (string=? x "banana"))
  (list "banana" "nana" "banana"))
=> true

(any? even? empty)
=> false
```

4. all?

- Input: a one-argument predicate, pred, and a list of data, alod such that pred operates on data of the type in alod
- Output: a boolean indicating if pred evaluates to true for all data in the list, and false otherwise. all? produces true on the empty list, regardless of the predicate passed in.

**Hint:** Your implementation of all? should be awfully similar to your implementation of any?.

Examples:

```scheme
(all? odd? (list 1 3 5 7 9))
=> true

(all? (lambda (x) (string=? x "banana"))
  (list "banana" "nana" "banana"))
=> false

(all? empty? empty)
=> true
```

6 Testing/Debugging

6.1 Is Contiguous Subsequence

The following code tests two sequences of integers to see whether small is a contiguous subsequence of big. That means that each item in small appears in big, in the same order as in small, all elements contiguous.

For instance (list 1 2) is a subsequence of (list 1 2 5), but not a subsequence of (list 1 5 2).

Here's the code:
This code works by saying that if the first item in the two lists match, then we just need to check that the rest do, too, but if they DON’T match, then it’s still possible for the small list to appear later in the big list. (For instance, (list 1 2) is a subsequence of (list 5 1 2).)

This program is incorrect.

Task: Show that the program exhibits a kind of failure called a false positive, by showing a case where is-contiguous-subsequence evaluates to true, even though small is not a contiguous subsequence of big in the sense described. To do this you should write down a check-expect that fails.

Suggestion: you’ll get maximal value from this exercise by working it out by hand, rather than by pasting the code into DrRacket and trying to find something that breaks it by trial and error.

The point of this exercise is that thinking through such cases before you write a procedure helps you write better check-expects and saves you debugging time.

6.2 Increasing Subsequence

We have a sequence of natural numbers, like (list 2 5 3 6 7); we’d like to find a longest increasing subsequence, where this time the elements of the subsequence need not be contiguous. So for our example list, one possible answer is (list 2 3 6 7). I say “a longest” instead of “the longest” because there may be two sequences of equal length. Consider (list 2 1 3); both (list 2 3) and (list 1 3) are “longest increasing subsequences” of (list 2 1 3).

To solve this problem, we apply the idea from class, and generalize: we write

(longest-increasing-subsequence-with-start alon start)

which tells us the longest increasing subsequence of alon where the first item in the subsequence is at least as big as start. Assuming we have this, we can write

(define (longest-increasing-subsequence alon)
  (longest-increasing-subsequence-with-start alon 0))

We write the subprocedure as follows: first, if the list’s empty, the answer’s empty. Otherwise, if the first element of the list is less than start, we have to ignore it, so we recur on (rest alon). But if it’s at least as large as start, we use it as the start of our output. Here’s the code
```
(if (< (first alon) start)
    (longest-increasing-subsequence-with-start (rest alon) start)
    (cons (first alon)
        (longest-increasing-subsequence-with-start
            (rest alon)
            (+ 1 (first alon))))))
```

Notice how when we include the first item of the list, we recursively call our procedure, but we say that the remaining list has to start with a number that’s at least (+ 1 (first alon)), so that if the input sequence contains something like 2 3 3 6, we’ll only include the first 3.

*Once again, this code does not work.*

**Task:** Find an example of a sequence of natural numbers where the longest increasing subsequence is longer than the one found by this procedure. To do this you should write down a check-expect that fails. Also provide what the procedure will (incorrectly) evaluate to in Racket.

Suggestion: you’ll get maximal value from this exercise by working it out by hand, rather than by pasting the code into DrRacket and trying to find something that breaks it by trial and error.

---

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)