Homework 5: Higher-Order Procedures
Due: 10:59 PM, Oct 9, 2019

Contents

1 Fun with map (Practice) .................................................. 2

2 Unfold (Practice) .......................................................... 3

3 Map2 (13 points) ............................................................ 3

4 Fold (13 points) ............................................................. 4

5 All You Need is ... Fold (12 points) ................................. 6

6 Set? (11 points) ............................................................. 8

7 Testing/Debugging (8 points) ......................................... 8

7.1 Is Contiguous Subsequence ........................................ 8

7.2 Increasing Subsequence ............................................. 9

Introduction and Objectives

For this assignment, the only problem that requires the 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 talked about in class.

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.

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 every other) homework assignment, 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.
To hand in your solutions to these problems, you must store them in appropriately-named files. Each should be named for the corresponding problem, as follows:

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

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

README.txt:
B01234567
There’s nothing to say except that I’m turning in some files plus this README the way the instructions say that I should.

To hand in your solutions to these problems, you must upload them to Gradescope. Do not zip or compress them. If you re-submit your homework, you must re-submit all files. If you choose to also store these files on department machines, all your solution files should reside in your ~/course/cs0170/homeworks/hw06 directory.

Practice

1 Fun with map (Practice)

Task: Donald the dairy farmer wants to write an article to show how amazing his cheese is. However, he is not a good writer and misspells the word “dairy” to “diary” all over his article. Wanting to help Donald, you decide to help him write the program to correct his article. Using map, write a procedure correct-name that replaces every occurrence of "diary" in a list of strings with "dairy".

Examples:

(correct-name (list "I" "daily" "make" "diary" "product"))
=> (list "I" "daily" "make" "dairy" "product")

(correct-name (list "diary" "farmer" "is" "the" "best" "job" "ever"))
=> (list "dairy" "farmer" "is" "the" "best" "job" "ever")

(correct-name (list "donald" "starts" "the" "diary"))
=> (list "donald" "starts" "the" "dairy")
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 i\textsuperscript{th} entry in the output list should be a string corresponding to the sign of the i\textsuperscript{th} 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 i\textsuperscript{th} element is a list of data whose only element is the i\textsuperscript{th} element of the input list.

Example:

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

(make-singletons (list "hay" "wheat" "soy" "barley"))
=> (list (list "hay") (list "wheat") (list "soy") (list "barley"))
```

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 (13 points)

Task: Write the type signature for a procedure `map2` which consumes a two-argument procedure, `proc`, and two lists, `alod1` and `alod2`, of equal length, and of possibly different types, 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.

**Note:** This first task asks you to write the type signature for map2 which is already required by the design recipe. However, we're asking you to explicitly do it as its own task to help with the implementation of map2 in the next task.

**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")

(map2 cons (list 1 2 3) (list empty empty empty))
=> (list (list 1) (list 2) (list 3))
```

**Task:** Write map2.

**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 (13 points)

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 type signature for a 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:** This first task asks you to write the type signature for `fold`, which is already required by the design recipe. However, we’re asking you to explicitly do it as its own task to help with the implementation of `fold` in the next task.

**Task:** Write `fold`.

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

**Examples:**

```racket
(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 "grass" "mulch" "wood chips"))
=> (cons "grass" (cons "mulch" (cons "wood chips" 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.

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:**

   ```racket
   (my-append (list 1 2 3) (list 4 5 6))
   => (list 1 2 3 4 5 6)
   (my-append (list "I" "love") (list "barns"))
   => (list "I" "love" "barns")
   ```
Note: Now that you’ve written my-append, feel free to use Racket’s built-in versions of append in your code, including in following problems on this homework.

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:

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

(my-flatten (list (list "apple" "cider") (list "donuts" "are" "yummy")))
=> (list "apple" "cider" "donuts" "are" "yummy")

(my-flatten (list empty empty empty))
=> empty
```

Note: Now that you’ve written flatten, feel free to use Racket’s built-in versions of `flatten` in your code.

## 5 All You Need is . . . Fold (12 points)

**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, `proc`, and a list of data, `alod` such that `proc` operates on data of the type in `alod`
   - **Output:** a list of data whose elements are the result of applying `proc` to each element of `alod`

   Examples:
   ```racket
   (my-map even? (list 1 2 3))
   => (list false true false)
   (my-map sub1 empty)
   => empty
   ```

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 \text{true}

Examples:

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

3. any?

• Input: a one-argument predicate, \( \text{pred} \), and a list of data, \( \text{alod} \) such that \( \text{pred} \) operates on data of the type in \( \text{alod} \)

• Output: a boolean indicating if \( \text{pred} \) evaluates to \text{true} for any datum in the list, and \text{false} otherwise

Examples:

\[
\begin{align*}
\text{(any? odd? (list 1 3 5 7 9))} \\
\Rightarrow \text{true} \\
\text{(any? (\textbf{lambda} (x) (string=? x "duck"))} \\
\text{(list "duck" "yuck" "duck"))} \\
\Rightarrow \text{true} \\
\text{(any? even? \textbf{empty})} \\
\Rightarrow \text{false}
\end{align*}
\]

4. all?

• Input: a one-argument predicate, \( \text{pred} \), and a list of data, \( \text{alod} \) such that \( \text{pred} \) operates on data of the type in \( \text{alod} \)

• Output: a boolean indicating if \( \text{pred} \) evaluates to \text{true} for all data in the list, and \text{false} otherwise. \text{all?} produces \text{true} on the empty list, regardless of the predicate passed in.

Hint: Your implementation of \text{all?} should be awfully similar to your implementation of \text{any?}.

Examples:

\[
\begin{align*}
\text{(all? odd? (list 1 3 5 7 9))} \\
\Rightarrow \text{true} \\
\text{(all? (\textbf{lambda} (x) (string=? x "duck"))} \\
\text{(list "duck" "yuck" "duck"))} \\
\Rightarrow \text{false} \\
\text{(all? empty? \textbf{empty})} \\
\Rightarrow \text{true}
\end{align*}
\]
Sets: An Introduction

A set is a collection of distinct elements, such that there are no repeated elements. In the next two problems, we will be representing sets via lists with no duplicates. We will only consider sets of atomic data (that is, lists of atomic data with no duplicates), as this is data for which member? is guaranteed to work.

6 Set? (11 points)

Task: Define a set predicate, called my-set?, that determines whether a list is a faithful representation of a set (i.e., whether all of its elements are distinct). Your procedure should be able to operate on lists of any single atomic data type.

Hint: Use Racket’s built-in member? procedure.

Examples:

```racket
(my-set? (list "shovel" "shovel" "watering can"))
=> false

(my-set? (list 1 2 3))
=> true

(my-set? empty)
=> true ;; vacuously -- empty has no elements, so they are all distinct
```

The built-in member? procedure is identical to the one we saw in class. And its analysis is essentially the same as that for the contains17? procedure:

Let \( M(n) \) be the number of elementary operations involved in evaluating \( \text{member? \ datum alod} \), where \( \text{datum} \) is any atomic value, and \( \text{alod} \) is a list of atomic data of length \( n \). Then it’s easy to see that:

\[
M(0) = D \\
M(n) \leq C + M(n - 1) \quad \text{for } n > 0
\]

and from the analysis we did for all recurrences like this, we see that \( M(n) \leq Cn + D \) for every natural number \( n \). (At least if you believe that we can prove the conjecture we made in class.)

Task: Write a recurrence relation for my-set? and make a conjecture on the upper bound of the run time using plug-and-chug. You can include this in the comments at the very bottom of your set.rkt file.

7 Testing/Debugging (8 points)

7.1 Is Contiguous Subsequence

The following code tests two sequences of integers to see whether small is a contiguous subsequence of big. That means that if each item in small appears in big, in the same order as in small, all elements contiguous.
For instance \((\text{list } 1 \ 2)\) is a subsequence of \((\text{list } 1 \ 2 \ 5)\), but not a subsequence of \((\text{list } 1 \ 5 \ 2)\).

Here’s the code:

\[
\text{(define (is-contiguous-subsequence small big)}
\begin{cases}
  \text{true} & \text{if both are empty lists} \\
  \text{true} & \text{if small is empty but big is not} \\
  \text{false} & \text{if big is empty but small is not} \\
  \text{(if (= (first small) (first big))}
  \begin{cases}
    \text{(is-contiguous-subsequence (rest small) (rest big))} & \text{if the first elements match} \\
    \text{(is-contiguous-subsequence small (rest big))} & \text{if the first elements don't match}
  \end{cases}
\end{cases}
\)
\]

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, \((\text{list } 1 \ 2)\) is a subsequence of \((\text{list } 5 \ 1 \ 2)\).)

This program is incorrect.

Task: Show that the program exhibits a kind of failure called a false positive, by showing a case where \(\text{is-contiguous-subsequence}\) evaluates to true, even though \(\text{small}\) is not a contiguous subsequence of \(\text{big}\) 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.

### 7.2 Increasing Subsequence

We have a sequence of natural numbers, like \((\text{list } 2 \ 5 \ 3 \ 6 \ 7)\); 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 \((\text{list } 2 \ 3 \ 6 \ 7)\). I say “a longest” instead of “the longest” because there may be two sequences of equal length. Consider \((\text{list } 2 \ 1 \ 3)\); both \((\text{list } 2 \ 3)\) and \((\text{list } 1 \ 3)\) are “longest increasing subsequences” of \((\text{list } 2 \ 1 \ 3)\).

To solve this problem, we apply the idea from class, and generalize: we write \((\text{longest-increasing-subsequence-with-start alon start})\) which tells us the longest increasing subsequence of \(\text{alon}\) where the first item in the subsequence is at least as big as \(\text{start}\). Assuming we have this, we can write:

\[
(\text{define (longest-increasing-subsequence alon)})
(\text{longest-increasing-subsequence-with-start alon 0}))
\]

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 \(\text{start}\), we have to ignore it, so we recur on \((\text{rest alon})\). But if it’s at least as large as \(\text{start}\), we use it as the start of our output. Here’s the code:

\[
(\text{define (longest-increasing-subsequence-with-start alon start})
\]

---

9
(cond
  [(empty? alon) empty]
  [(cons? alon)
    (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 CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback).