Homework 6: Higher-Order Procedures

Due: 11:59 PM, Oct 16, 2018

Contents

1 Fun with \textit{map} (Practice) \hfill 2

2 Unfold (Practice) \hfill 3

3 Map2 \hfill 3

4 Fold \hfill 4

5 All You Need is \ldots Fold \hfill 5

6 Testing/Debugging \hfill 7

\hspace{1em} 6.1 Is Contiguous Subsequence \hfill 7

\hspace{1em} 6.2 Increasing Subsequence \hfill 8

Introduction

We have of course seen lambda expressions but usually in the context of a \texttt{define}, in which a variable is bound to the value of such an expression.

However, it is useful to use lambda expressions without binding variables to them. The value of a lambda expression is of course a procedure; when we use such a procedure without binding a variable to it, we call the procedure an \textit{anonymous} procedure.

In this homework, you will learn about certain \textit{higher-procedures}, procedures that take procedures as arguments, that have become a traditional part of functional programming. When providing the procedure argument to a higher-order procedure, it is often convenient and elegant to use a lambda expression to provide the argument.

There is another, more technical reason that anonymous procedures are useful. You might get a hint of that in the coming lab.

We do require use of anonymous procedures as arguments to \texttt{foldr} in the second task of \texttt{Fold} and the entirety of \textbf{All You Need is \ldots Fold}.

As always, you should add a design recipe for each procedure you write, including a procedure specification (input/output), recursion diagram (only for recursive procedures), and test cases. Note, anonymous procedures do not need design recipes but named procedures that use anonymous procedures still do.
How to Hand In

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
- 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)

map is a built-in procedure that takes in two arguments, a procedure and a list of data. It outputs a list of data where the $i^{th}$ element of the output list is the result of applying the input procedure to the $i^{th}$ element of the input list.

By way of example:

```
(map zero? (quote (0 1 0 2 0 3)))
=> (#t #f #t #f #t #f)

(map (lambda (x) (* x x)) (quote (1 2 3)))
=> (1 4 9)
```

map is specifically a higher-order procedure. A procedure is known as a higher-order procedure if it takes in one or more procedures our outputs a procedure.
Task: Uncle Pennybags due to recent inflation wants to be called Mr. Moneybags. Using map, write a procedure `correct-name` that replaces every occurrence of the symbol `Pennybags` in a list of symbols with the symbol `Moneybags`.

Examples:

```lisp
(correct-name (quote (I met Pennybags at Reading Railroad)))
=> (I met Moneybags at Reading Railroad)

(correct-name (quote (I love Pennybags)))
=> (I love Moneybags)
```

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 symbols. The $i$th entry in the output list should be a symbol corresponding to the sign of the $i$th integer in the input list; positive if the integer is positive, negative if it is negative, or zero if it is zero.

Examples:

```lisp
(whats-your-sign (quote (1 2 3)))
=> (positive positive positive)

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

2 Unfold (Practice)

Write a procedure `unfold`, which consumes a one-argument procedure, $f$, and a natural number, $n$, and produces a list where the first element is $f(n)$, the second element is $f(n - 1)$, the third element is $f(n - 2)$, and so on, up until the last element which is $f(0)$.

Examples:

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

(unfold summorial 6)
=> (quote (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.

**Note:** The input lists do not necessarily have to be of the same type!

Example:

```
(map2 + (quote (1 2 3)) (quote (3 2 1)))
=> (4 4 4)
```

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

```
(map + (quote (1 2 3)) (quote (3 2 1)))
=> (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 ith element is the greater of the ith element of alon1 and the ith element of alon2.

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

Example:

```
(pairwise-max (quote (1 5 3)) (quote (4 2 6)))
=> (quote (4 5 6))
```

```
(pairwise-max (quote (1 5 3)) (quote (-4 10 -6)))
=> (quote (1 10 3))
```

### Fold

Consider a list of numbers:

```
(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 (quote (3 4 5)))
=> (+ 3 (+ 4 (+ 5 0)))
=> 12

(fold * 1 (quote (3 4 5)))
=> (* 3 (* 4 (* 5 1)))
=> 60

(fold cons empty (quote (c d e)))
=> (cons (quote c) (cons (quote d) (cons (quote 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. *You should use an anonymous procedure or a built-in as an argument to foldr for all subproblems in this task.*

**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 (quote (1 2 3)) (quote (4 5 6)))
=> (1 2 3 4 5 6)

(my-append (quote (#true #false)) (quote (#false)))
=> (#true #false #false)

(my-append empty (quote (0 0 0)))
=> (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 the same type. Feel free to use Racket’s built-in append procedure.

Examples:
5 All You Need is \ldots Fold

Task: Write each of the procedures below using a single call to Racket’s built-in fold procedure, \texttt{foldr}. You should use an anonymous procedure as an argument to \texttt{foldr} for all subproblems in this problem. You should do no explicit recursion yourself. You must follow these rules to earn full credit.

Note: After completing this problem, you may use Racket’s built-in \texttt{map} and \texttt{filter} procedures. To do so, switch to either "Advanced" or "Intermediate Student with lambda" in Dr. Racket.

1. \texttt{my-map}

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

   Examples:

   \[
   (\texttt{my-map even? (quote (1 2 3))})
   => (quote (#false #true #false))
   \]

   \[
   (\texttt{my-map sub1 empty})
   => \texttt{empty}
   \]

2. \texttt{my-filter}

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

   Examples:

   \[
   (\texttt{filter cons? (quote ((17) () (18 19))))}
   => (quote ((17) (18 19)))
   \]

   \[
   (\texttt{my-filter even? (quote (38 37 65 63 1 2 3 4 40))})
   => (quote (38 2 4 40))
   \]

3. \texttt{any?}

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

Examples:

\[
\begin{align*}
\text{(any? odd? (quote (1 3 5 7 9)))} & \Rightarrow \text{true} \\
\text{(any? even? empty)} & \Rightarrow \text{false}
\end{align*}
\]

4. all?

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

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

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

Examples:

\[
\begin{align*}
\text{(all? odd? (quote (91 3 5 7 9)))} & \Rightarrow \text{true} \\
\text{(all? (lambda (x) (equal? x (quote banana)) (quote (banana nana banana))))} & \Rightarrow \text{false} \\
\text{(all? empty? empty)} & \Rightarrow \text{true}
\end{align*}
\]

6 Testing/Debugging

6.1 Is Contiguous Subsequence

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

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

Here’s the code:

\[
\begin{align*}
\text{(define is-contiguous-subsequence}
\text{ (lambda (small big))}
\text{ (cond}
\text{ \quad [(and (empty? small) (empty? big)) true]}
\text{ \quad [(and (empty? small) (cons? big)) true]}
\text{ \quad [(and (cons? small) (empty? big)) false]}
\text{ (else false))}
\end{align*}
\]
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, (1 2) is a subsequence of (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 (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 (2 3 6 7). I say “a longest” instead of “the longest” because there may be two sequences of equal length. Consider (2 1 3); both (2 3) and (1 3) are “longest increasing subsequences” of (2 1 3).

To solve this problem, we 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
  (lambda (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

(define longest-increasing-subsequence-with-start
  (lambda (alon start)
    (cond
      [[(empty? alon) empty]
       [(cons? alon)
        (if (< (car alon) start)
            (longest-increasing-subsequence-with-start (cdr alon) start)
            (cons (car 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.