Homework 5: Even More Recursion
Due: 10:00 PM, Oct 11, 2017

Contents

1 Odds and Evens (Practice) 2
2 Count Up (Practice) 2
3 Flip and Flipper 3
4 Set? 4
5 Set-Equal? 5
6 Analysis 5

Objectives

By the end of this homework you will be able to:

1. Revise standard data definitions
2. Analyze the work performed by recursive procedures

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, you should answer the Flip and Flipper, Set?, Set-Equal?, and Analysis 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., set.rkt corresponds to Set?):

- README.txt
- flip-flipper.rkt
- set.rkt
- set-analysis.txt
- set-equal.rkt
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/hw05` 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 hw05 directory into a file hw05.zip, and hand in with [this Google form](#).

**Option 2:** Use cs0170_handin hw05 to submit each file from the department machines.

See [this Piazza post](#) for more details.

---

### Practice

**Note:** These practice problems involve natural number recursion. If you have not already done so, please refer to Lab 04 to learn about this concept.

---

### 1 Odds and Evens (Practice)

**Task:** Write the recursive procedure, `odd-digits`, that takes as input a natural number and returns the number of odd digits in the number.

For example, 1, 3, and 5 are odd digits in 12345. Hence,

```
(odd-digits 12345)
=> 3
```

```
(odd-digits 2468)
=> 0
```

**Hint:** The built in Racket procedure `quotient` may come in handy, as may `remainder`.

**Note:** Doing this problem may help you write a `num-to-bignum` procedure that converts a Racket number into your particular bignum representation. Such a procedure makes it easy to write test cases, because you can write

```
(check-expect
 (bignum+
  (num-to-bignum 63)
  (num-to-bignum 22))
 (num-to-bignum 85))
```
for instance, which is pretty easy to read as “63 + 22 should be 85.” You’re under no obligation to write your bignum test-cases this way, however. And if you write num-to-bignum incorrectly, then all your test-cases for bignum+ might fail because of that, rather than because there’s a bug in bignum+, so there’s a tradeoff to consider. (This also helps you understand why testing of the procedures that you write is so important!)

2 Count Up (Practice)

Task: Write a procedure count-up that consumes two natural numbers, \(n\) and \(k\), and produces a list consisting of \(k\) numbers, in ascending order, counting up from \(n\) by ones.

Examples:

\[
\begin{align*}
(\text{count-up } 17 & \ 0) \\
& \Rightarrow \ \text{empty}
\end{align*}
\]

\[
\begin{align*}
(\text{count-up } 17 & \ 3) \\
& \Rightarrow (\text{list } 17 \ 18 \ 19)
\end{align*}
\]

Problems

3 Flip and Flipper

You were just to hired to work at an extremely disorganized pet shop. Your job is to make it easier for the staff to sort through and identify their many lists of pets and pet types. To do this, you will need to write some procedures that manipulate lists of paired data effectively.

Task: Write a procedure flip that takes as input a list of pairs (i.e., two-element lists) of strings and outputs a new list of pairs, where the pairs are in the same order as the corresponding pairs in the input list, but each pair is in reverse order.

The reverse of a pair contains the same elements as the original, but they appear in the opposite order. For example, the reverse of the pair (list "a" "b") is (list "b" "a").

Examples:

\[
\begin{align*}
(\text{flip } (\text{list } (\text{list } "\text{Clifford}" \ "\text{Big Red Dog}"") \ (\text{list } "\text{Marley}" \ "\text{Yellow Lab}")) & \Rightarrow (\text{list } (\text{list } "\text{Big Red Dog}" \ "\text{Clifford}"") \ (\text{list } "\text{Yellow Lab}" \ "\text{Marley}"))
\end{align*}
\]

\[
\begin{align*}
(\text{flip } (\text{list } (\text{list } "\text{Tom}" \ "\text{Alley cat}"") \ (\text{list } "\text{Garfield}" \ "\text{Tabby cat}")) & \Rightarrow (\text{list } (\text{list } "\text{Alley cat}" \ "\text{Tom}"") \ (\text{list } "\text{Tabby cat}" \ "\text{Garfield}")
\end{align*}
\]

Hint: Before getting started, you may want to rewrite the example above in longhand, meaning rewrite (list "Clifford" "Big Red Dog") as (cons "Clifford" (cons "Big Red Dog" empty)) and so on. If you have not fully mastered the internal structure of lists, you will find this problem very difficult.
Task: Write a procedure `flipper` that takes as input a list of strings and outputs a new list in which the first input element becomes the second output element and the second input element the first output element, the third input element becomes the fourth output element and the fourth input element the third output element, and so on. If the input list has no other element with which to flip the last element, then it should remain in its initial position.

Examples:

```
(flipper (list "Grumpy" "fish" "Happy" "turtle" "Sleepy" "hamster" "Dopey" "guinea pig"))
=> (list "fish" "Grumpy" "turtle" "Happy" "hamster" "Sleepy" "guinea pig" "Dopey")
```

```
(flipper (list "Bashful" "snake" "Sneezy" "fish" "Bob" "dog" "adopted in 2017"))
=> (list "snake" "Bashful" "fish" "Sneezy" "dog" "Bob" "adopted in 2017")
```

```
(flipper (list "Captain America" "Labradoodle" "adopted in 2015"))
=> (list "Labradoodle" "Captain America" "adopted in 2015")
```

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.

4 Set?

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.

```
(my-set? (list "dog" "dog" "cat"))
=> 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 `(member? datum alod)`, where `datum` is any atomic value, and `alod` is a list of atomic data of length $n$. Then it’s easy to
see that
\[
M(0) = D \quad \text{(1)}
\]
\[
M(n) \leq C + M(n - 1) \quad \text{for } n > 0 \quad \text{(2)}
\]
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 \texttt{my-set?} and make a conjecture on the upper bound of the run time using plug-and-chug.

## 5 Set-Equal?

**Task:** Define a procedure called \texttt{set-equal?} that determines whether two sets are equal (i.e., whether two lists, each with no duplicates, contain the same elements, regardless of order). Once again, the elements of the sets can be of any atomic type.

**Note:** Your procedure need only work on sets (i.e. lists with no duplicates). It need not function correctly on arbitrary lists.

**Hint:** Two sets \( A \) and \( B \) are equal exactly when

- every element of \( A \) is an element of \( B \), and
- every element of \( B \) is an element of \( A \).

**Hint:** It may simplify your program a little if you write a helper procedure.

```
(set-equal? (list "dog" "cat" "fish") (list "dog" "fish" "cat")) => true

(set-equal? (list "snake" "rat" "guinea pig") (list "snake" "fish" "cat")) => false

(set-equal? (list) (list)) => true
```

## 6 Analysis

Here are the implementations for a procedure \texttt{slow-reverse} and its helper \texttt{cons-last}.

```
(define (cons-last alod datum)
  (cond
   [(empty? alod) (cons datum empty)]
   [(cons? alod) (cons (first alod) (cons-last (rest alod) datum))]))

(define (slow-reverse alod)
  (cond
   [(empty? alod) empty]
   [(cons? alod) (cons-last (slow-reverse (rest alod)) (first alod))]))
```
Task: Write “Let $C(n)$ denote the number\footnote{Usually we write “the largest number of operations . . . . on any list ...” But in this case, the number of operations doesn’t depend on anything but the length of the list, so we can omit the word ‘largest’ and simply go for the exact operation count.} of operations involved in evaluating cons-last on any item and any list of length $n$...” and then write down a recurrence relation satisfied by $C$.

Hint: Since counting operations exactly can be a pain, you may use constants to denote operation counts, so that your answer might look something like this:

\begin{align}
C(0) &= A \\
C(n) &= B + Dn + C(n - 1) + C(n - 2), \text{ for } n > 0
\end{align}

where $A, B, \text{ and } D$ all represent constants. (Of course, this is not the actual answer!)

Task: Use plug-n-chug (or theorems, if you have them) to conjecture a closed-form expression for $C$ (i.e., one that does not involve recursion). Make sure to show the steps you take to get from the recurrence relation to your conjecture.

Task: Write a recurrence relation for $S$, an operation-counting procedure for slow-reverse. You may refer to $C$ (from the previous two tasks) in your solution.

Task: Write a conjecture for a closed form, or non-recursive, formula for $S$.

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)