Homework 4: More Recursion
Due: 10:59 PM, Oct 2, 2019

Contents

1 List Recursion (Practice) 2
2 Count Up (Practice) 3
3 Member (10 Points) 3
4 Append (12 Points) 4
5 Porky the Pig’s Mimicry School (12 Points) 5
6 Pair Count (6 Points) 6
7 Flip and Flipper (12 Points) 6
8 Writing Specifications (6 Points) 7
9 Oracle (Optional and Ungraded) 8
10 Simplifying Racket (Optional and Ungraded) 8

Objectives

By the end of this homework you will know:

1. Recursion is usually the answer
2. More about primitive recursion

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

1. Write more procedures that recur on lists
2. Use Racket’s built-in member? and append procedures

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 Member,
Append, Porky the Pig’s Mimicry School, Pair Count, Flip and Flipper, and Writing Specifications 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
- member.rkt
- member-analysis.txt
- append.rkt
- append-analysis.txt
- school.rkt
- pair-count.rkt
- flip-flipper.rkt
- specification.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 choose to also store these files on department machines, all your solution files should reside in your ~/course/cs0170/homeworks/hw04 directory.

Practice

1 List Recursion (Practice)

Task: Write a procedure, select-positives, that takes as input a list of integers and returns a list whose elements are the positive elements of the input list in the same order.

Examples:

```
(select-positives (list 1 2 3 4))
=> (list 1 2 3 4)
```
(select-positives (list 1 -2 3 -4))
=> (list 1 3)

(select-positives (list -1 -2 -3 -4))
=> empty

(select-positives empty)
=> empty

**Task:** Write a procedure, remove-until-zero, that takes as input a list of numbers and returns a list whose elements are the same as the elements of the input list appearing at and after the first zero in the input list. The output list should preserve the order of the input list.

**Examples:**

```
(remove-until-zero (list 15 16))
=> empty

(remove-until-zero (list 19 18 17 0))
=> (cons 0 empty)

(remove-until-zero (list 0 1 2 3))
=> (list 0 1 2 3)

(remove-until-zero (list 9 7 0 1 2))
=> (list 0 1 2)
```

## 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:**

```
(count-up 17 0)
=> empty

(count-up 17 3)
=> (list 17 18 19)
```

### Problems

## 3 Member (10 Points)

**Task:** Write a procedure, my-member?, that takes as input an integer and a list of integers, has call structure (my-member? item aloi), and outputs true if the input item is an element of the list, and false otherwise.
Examples:

(my-member? 0 empty)
=> false

(my-member? 17 (list 17))
=> true

(my-member? 15 (list 17 18 31 32))
=> false

Note: DrRacket includes a polymorphic procedure, member?, which takes as input a datum, datum, and a list of data, alod, and returns true if datum is an element of alod and false otherwise. For example:

(member? 17 (list 17))
=> true

(member? "seventeen" (list "seventeen"))
=> true

Needless to say, you are not allowed to use this procedure in your implementation or testing of my-member?. You are, however, welcome to use member? for all subsequent problems on this and future assignments. Task: Write a recurrence relation for the runtime of my-member?.

Here's some help getting started:

Let $M(n)$ be the largest number of operations used in evaluating my-member? applied to any list of length $n$.

4 Append (12 Points)

The cons procedure gives us a way of creating a list by attaching a single element to the beginning of another list. But what if we want to form a list from two preexisting lists?

Task: Write a procedure, my-append, that takes as input two lists of integers, has call structure (my-append alod1 alod2), and outputs another list of integers that contains all the elements of the first list in their input order, followed by all the elements of the second list in their input order.

Examples:

(my-append (list 1 2 3) (list 4 5 6 7))
=> (list 1 2 3 4 5 6 7)

(my-append empty (list 0 0 0))
=> (list 0 0 0)

(my-append empty empty)
=> empty
Note: DrRacket includes a polymorphic procedure, append, which takes as input two lists of data and returns another list that contains all the elements of the first, in order, followed by all the elements of the second, also in order. For example:

\[
\text{(append (list 1 2 3) (list 4 5 6 7))}
\]
\[
\Rightarrow \text{(list 1 2 3 4 5 6 7)}
\]

\[
\text{(append (list "tractor" "harvester") (list "wheelbarrow" "ox"))}
\]
\[
\Rightarrow \text{(list "tractor" "harvester" "wheelbarrow" "ox")}
\]

Needless to say, you are not allowed to use this procedure in your implementation or testing of my-append. You are, however, welcome to use append for all subsequent problems on this and future assignments.

Task: Write a recurrence relation for the runtime of my-append.

Here’s some help getting started:

Let \(A(n)\) be the largest number of operations used in evaluating my-append applied to any pair of lists where the first list has length \(n\).

5 Porky the Pig’s Mimicry School (12 Points)

Porky the Pig has decided to teach his fellow farm animals a class and has decided to take advantage of new computer technology to keep track of his students’ preferred names.

However, Porky doesn’t have time to create a ledger of students and their nicknames since he’s too busy rolling in mud, so he needs your help to keep track of his students. He wants you to create a database of the animals’ names and nicknames so that he can easily know what to call his students throughout the semester.

Here is an example of the kind of database Porky has in mind:

<table>
<thead>
<tr>
<th>Name</th>
<th>Nickname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robert Rooster</td>
<td>Bob</td>
</tr>
<tr>
<td>Gregory Goat</td>
<td>Greg</td>
</tr>
<tr>
<td>Harrison Horse</td>
<td>Harry</td>
</tr>
</tbody>
</table>

Task: Create two examples of student databases, represented as lists of string lists of length 2, with the first element being a student’s name and the second being that student’s nickname. One of your examples should encode the sample database depicted above.

Task: Write a procedure, update-nickname, that consumes a string, name, a string, nickname, and a database of students, in that order, and produces a new database of students. If the student were already in the database, the output will be nearly identical, with that student’s associated nickname updated to the newly provided nickname. If the student isn’t in the database already, the output will be the same database with the new student at the end.

For example, adding a new student to the database with name Lawrence Lamb and nickname Law yields:
Updating Robert Rooster’s nickname in the current database to Rob yields:

<table>
<thead>
<tr>
<th>Name</th>
<th>Nickname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robert Rooster</td>
<td>Rob</td>
</tr>
<tr>
<td>Gregory Goat</td>
<td>Greg</td>
</tr>
<tr>
<td>Harrison Horse</td>
<td>Harry</td>
</tr>
<tr>
<td>Lawrence Lamb</td>
<td>Law</td>
</tr>
</tbody>
</table>

Task: Write a procedure, `lookup`, with call structure `(lookup name database)`, that takes as input a string, name, and a database of students, and returns the associated nickname. In your procedure, you can assume that the database contains exactly one copy of the input name (after all, Porky knows who his students are!).

Note: Because your database is guaranteed to contain the input name, it will never be empty. Consider what this says about your base case.

6 Pair Count (6 Points)

Charlie the chicken plays a prank on Henry the Holstein Friesian cattle and goes through his precious sock collection! You are tasked with helping Henry find all of his matching socks.

Charlie has convinently created 2 lines of equal length of Henry’s socks across the beach. Henry is only able to compare 2 socks at a time—one from the left line with the corresponding sock from the right line. Also, the moment Henry finds a pair of non-matching socks, he will be so upset that he won’t be able to continue counting. Help Henry find out how many pairs of matching socks there are before he finds an unmatching pair!

Task: Write a procedure, `pair-count`, that takes as input two lists of integers of equal length, has call structure `(pair-count aloi1 aloi2)`, and returns the number of elements that pairwise match (that is, that have the same value in the same position in the list) up to the first pair that does not match or the end of the lists.

Examples:

```
(pair-count (list 5 4 2 1 4) (list 5 4 3 1 4))
=> 2

(pair-count (list 0 0 1) (list 0 0 1))
=> 3

(pair-count (list 5 1 0) (list 0 0 1))
=> 0
```
7  Flip and Flipper (12 Points)

You were just hired to work at an extremely disorganized cattle farm. Your job is to make it easier for the staff to sort through and identify their many lists of cattle and cattle 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:

(flip (list (list "Aaron" "Abondance") (list "Bobby" "Black Angus") (list "Henry" "Holstein-Friesian")))
=> (list (list "Abondance" "Aaron") (list "Black Angus" "Bobby") (list "Holstein-Friesian" "Henry"))

(flip (list (list "Sam" "Simmental") (list "Terrence" "Texas Longhorn")))
=> (list (list "Simmental" "Sam") (list "Texas Longhorn" "Terrence"))

Hint: Before getting started, you may want to rewrite the example above in longhand, meaning rewrite (list "Aaron" "Abondance") as (cons "Aaron" (cons "Abondance" 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 second input element the third output element. 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 "Farmer" "Spike" "has" "goats"))
=> (list "Spike" "Farmer" "goats" "has")

(flipper (list "Old" "MacDonald" "had" "a" "farm" "ee" "ai" "ee" "ai" "o"))
=> (list "MacDonald" "Old" "a" "had" "ee" "farm" "ee" "ai" "o" "ai")

(flipper (list "Bessie" "says" "moo"))
=> (list "says" "Bessie" "moo")

(flipper (list "Spike" "is" "a" "happy" "farmer"))
=> (list "is" "Spike" "happy" "a" "farmer")

8  Writing Specifications (6 Points)

In this problem, you will come to understand the importance of writing good specifications for your procedures. To demonstrate this, we will give you several poorly written specs and ask you for an
output that meets the spec, but does not fit into what the intended output should be.

**Example:** Suppose I write:

```plaintext
;; trim : num list -> num list
;;
;; input:
;;    aloi: a list of integers
;;
;; output:
;;    a list of the same numbers, but with any initial or final zeroes removed
```

Then if the input is:

```plaintext
(list 0 1 2 0)
```

the output could be:

```plaintext
(list 2 1)
```

Why? Because the output is a list with the same numbers...they just happen to be in a different order. And the specification didn’t say anything about order.

**Task:** Here are three more specs. For each of them, we hope the intent is fairly clear. You should write down an input/output pair that meets the spec, but violates the intent, i.e., you should do your best to deliberately misread the intent of the specification. Why? Because doing so helps you see how others might misinterpret your specifications, and writing a clear spec is one of the best things you can learn to do in CS17. Please include an explanation of how your input/output pair follows the specification but does not match the intent. The expectation with this explanation would be something like above:

"The output is a list with the same numbers, they just happen to be in a different order."

**Note:** This problem requires no programming. You’re not supposed to write the specified procedures; all you’re supposed to write are input/output pairs.

Spec 1. **Unique:** Remove all duplicates from a list of ints, leaving the remaining items in order.

Spec 2. **Sort:** Take a list of ints and produce a list containing the same ints, but in nondecreasing order.

Spec 3. **No cats:** from a list of strings, remove all occurrences of “cat”.

### 9 Oracle (Optional and Ungraded)

**Task:** Write a procedure, `equal-databases?`, that consumes two databases of students and outputs `true` if they contain the same students with the same nicknames.
10 Simplifying Racket (Optional and Ungraded)

In Racket, you can write `true` or `#t`, and both, when evaluated, produce the “true” value. There was really no reason to include both except history: some folks were used to typing `#t`. But if, in the course of the “read” part of the read-eval-print loop, Racket simply replaced every occurrence of `#t` with `true`, it would have no effect at all on how programs run. We refer to this kind of thing as *syntactic sugar*: having the ability to write booleans the way you’re used to kind of sweetens the language.

At a larger scale, think about an expression like:

```
(or expr1 expr2)
```

in your program, where both `expr1` and `expr2` evaluate to booleans (at least for CS17 racket!). Racket could convert it, during reading, to:

```
(if expr1 true expr2)
```

without changing the result of the program in any way.

This would violate the advice I gave in class about never putting `true` or `false` as one of the results in an if-expression, but this isn’t *you* putting that in the program, it’s *Racket* making this transformation every time it sees an or-expression.

If Racket did this, then the Rules of Evaluation wouldn’t have to have a special case for `or`, because Racket, during evaluation, would never encounter an `or`! Writing a Racket evaluator would be a little bit simpler.

**Task:** The `or` rule says something about which result-expressions get evaluated in various circumstances; the `if` rule has similar clauses. Check that if we perform the substitution above, nothing changes (i.e., if we evaluate the replacement rather than the original, exactly the same expressions get evaluated in both cases (aside from the one extra evaluation of `true`, which we’ll ignore).

**Task:** Figure out corresponding rewrite rules for `and` expressions and `cond` expressions, and show that the evaluations performed in each case are essentially the same. You may ignore any fixed number of primitive operations (e.g., evaluating `true` or `false`, or applying `not` to some boolean value).

We’ve just shown that we can reduce the number of rules in Racket by writing many expressions in terms of `if`. What about others? Could we get rid of `if`, `cond`, and `and` and write everything in terms of `or`?

Well, not in the form we currently know, because:

```
(if true 5 7)
```

evaluates to 5, but any `or` expression evaluates to either `true` or `false`.

The fact is (and this fact is for you to use only in this problem, and nowhere else in your CS17 code!) that `or` actually takes as arguments a pair of expressions. (It actually takes an arbitrary sequence of expressions, but for the purpose of this problem, let’s pretend it always takes exactly
two.) And the **or**-Rule for \(\texttt{or \ expr1 \ expr2}\) then reads “evaluate \(\texttt{expr1}\) to get a value \(v\); if the value \(v\) is **not** \texttt{false}, then \(v\) is the value of the or-expression. If \(v\) is false, evaluate \(\texttt{expr2}\) to get a value \(v'\). If \(v'\) is not false, then \(v'\) is the value of the or-expression; if it is false, then the value of the or-expression is \texttt{false}.” In short, any “non-false” value is treated as if it were the boolean \texttt{true} and gets returned! That means that:

\[
\texttt{(or 3 5)}
\]

evaluates to 3.

**Task:** With this interpretation of **or** (and a corresponding one for **and** — you can experiment with DrRacket set to \#lang Racket to see what it does) show that all \texttt{if}, \texttt{and}, \texttt{cond} expressions can be converted into \texttt{or} expressions, so that once again only a single rule (the **or** rule) is required in an evaluator.

Everything can also be written in terms of \texttt{and} or \texttt{cond}, by the way. The point isn’t that any one of these is better than the others, but rather that it’s possible to de-sugar the language and make expression-evaluation simpler.

You can ask “How much simpler can it get? Can we get rid of \texttt{true} and \texttt{false}? Can we get rid of procedure-application-expressions? Can we get rid of \texttt{let} expressions (which you’ll encounter in a week or two)? Just how simple can the “rules” be, if we allow this kind of rewriting during the “read phase”?

That’s a true *computer science* question — it asks about the theoretical limits for simplifying things. Not that you’d ever want to write programs in a language with almost no syntax — it’d be hell. But what if you wanted to *prove* something about all possible programs? The fewer possible constructs you have to consider, the easier your task is. And that’s a typical computer-science view of the world: sometimes you get power through simplicity.

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).