Homework 4: More Recursion
Due: 10:00 PM, Oct 3, 2017

Contents

1 List Recursion (Practice)  
2 Member  
3 Append  
4 Percy the Parrot’s Mimicry School  
5 Pair Count  
6 Oracle (Optional and Ungraded)  
7 Writing Specifications  
8 Simplifying Racket (Optional and Ungraded)

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 all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Member; Append; Percy the Parrot’s Mimicry School; Pair Count; and Writing Specifications questions.

To hand in your solutions to these problems, you must store them in appropriately-named files. In particular, each should be named for the corresponding problem, as follows (e.g., member.rkt corresponds to Member):
• README.txt
• member.rkt
• member-analysis.txt
• append.rkt
• append-analysis.txt
• school.rkt
• pair-count.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. If you are using a departmental linux system, all your solution files should reside in your `/course/cs0170/homeworks/hw04 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 hw04 directory into a file hw04.zip, and hand in with this Google form.

Option 2: Use cs0170_handin hw04 to submit each file from the department machines.
See this Piazza post for more details.

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 (cons 1 (cons 2 (cons 3 (cons 4 empty))))))
=> (cons 1 (cons 2 (cons 3 (cons 4 empty))))

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

(select-positives (cons -1 (cons -2 (cons -3 (cons -4 empty)))))
```
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 (cons 15 (cons 16 empty)))
empty

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

(remove-until-zero (cons 0 (cons 1 (cons 2 (cons 3 empty)))))
(cons 0 (cons 1 (cons 2 (cons 3 empty))))

Problems

2 Member

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

Examples:

(my-member? 0 empty)
false

(my-member? 17 (cons 17 empty))
true

(my-member? 15 (cons 17 (cons 18 (cons 31 (cons 32 empty)))))
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 (cons 17 empty))
true

(member? "seventeen" (cons "seventeen" empty))
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.

**Note:** This problem has to do with the idea of set membership, using lists to represent sets. Using lists is only one of many possible ways to implement sets.

**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 $n$.

### 3 Append

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:

```scheme
(my-append (cons 1 (cons 2 (cons 3 empty))) (cons 4 (cons 5 (cons 6 (cons 7 empty)))))
=> (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 (cons 7 empty)))))))
```

```scheme
(my-append empty (cons 0 (cons 0 (cons 0 empty))))
=> (cons 0 (cons 0 (cons 0 empty)))
```

```scheme
(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,

```scheme
(append (cons 1 (cons 2 (cons 3 empty))) (cons 4 (cons 5 (cons 6 empty)))))
=> (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 empty))))))
```

```scheme
(append (cons "shrimp" (cons "scallops" empty))
(cons "mussels" (cons "clams" empty)))
=> (cons "shrimp" (cons "scallops" (cons "mussels" (cons "clams" empty))))
```

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 
\texttt{my-append} applied to any pair of lists where the first list has length \( n \).

## 4 Percy the Parrot’s Mimicry School

Percy the Parrot has decided to give his fellow parrots a class on mimicking the human voice and has decided to take advantage of new computer technology to keep track of his students’ preferred names.

However, Percy doesn’t have time to create a ledger of students and their nicknames since he’s too busy \texttt{competitively riding his tricycle} so he needs your help to keep track of his parrot students. He wants you to create a database of the parrots’ 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 Percy has in mind:

<table>
<thead>
<tr>
<th>Name</th>
<th>Nickname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polly Wannacracker</td>
<td>Pol</td>
</tr>
<tr>
<td>Michael Macaw</td>
<td>Mike</td>
</tr>
<tr>
<td>James Beak</td>
<td>Jim</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 Tweeter Pan and nickname Tweety yields:

<table>
<thead>
<tr>
<th>Name</th>
<th>Nickname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polly Wannacracker</td>
<td>Pol</td>
</tr>
<tr>
<td>Michael Macaw</td>
<td>Mike</td>
</tr>
<tr>
<td>James Beak</td>
<td>Jimmy</td>
</tr>
<tr>
<td>Tweeter Pan</td>
<td>Tweety</td>
</tr>
</tbody>
</table>

Updating James Beak’s nickname in the current database to Jimmy yields:

<table>
<thead>
<tr>
<th>Name</th>
<th>Nickname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polly Wannacracker</td>
<td>Pol</td>
</tr>
<tr>
<td>Michael Macaw</td>
<td>Mike</td>
</tr>
<tr>
<td>James Beak</td>
<td>Jimmy</td>
</tr>
<tr>
<td>Tweeter Pan</td>
<td>Tweety</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, Percy knows who his students are!)[1]

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

5 Pair Count

Toto the border collie plays a prank on Pinky the flamingo and goes through his precious sock collection! You are tasked with helping Pinky find all of his matching socks.

Toto has convinently created 2 lines of equal length of Pinky’s socks across the beach. Pinky 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 Pinky finds a pair of non-matching socks, he will be so upset that he won’t be able to continue counting. Help Pinky 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.

Examples:

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

(pair-count (cons 0 (cons 0 (cons 1 empty))) (cons 0 (cons 0 (cons 1))))
=> 3

(pair-count (cons 1 (cons 0 (cons 1 empty))) (cons 0 (cons 0 (cons 1))))
=> 0
```

6 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.

7 Writing Specifications

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.

1If you’re having trouble coming up with bird names, try [this list](#)!
Here is an example:

Suppose I write

```racket
;; trim : (int list) -> (int list)
;;
;; input:
;;   alon: a list of integers
;;
;; output:
;;   a list of the same numbers, but with any initial or final zeroes removed
```

Then if the input is

```racket
(cons 0 (cons 1 (cons 2 (cons 0 empty)))
```

the output could be

```racket
(cons 2 (cons 1 empty)))
```

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”.

---

### 8 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 \textit{expr1} and \textit{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 \textit{true} or \textit{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 \textit{or}, because Racket, during evaluation, would never encounter an \textit{or}! Writing a Racket evaluator would be a little bit simpler.

**Task**: The \textit{or} Rule says something about which result-expressions get evaluated in various circumstances; the \textit{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 \textit{true}, which we’ll ignore).

**Task**: Figure out corresponding rewrite rules for \textit{and} expressions and \textit{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 \textit{true} or \textit{false}, or applying \textit{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 \textit{if}. What about others? Could we get rid of \textit{if}, \textit{cond}, \textit{and} and write everything in terms of \textit{or}?

Well, not in the form we currently know, because

\[
(if \ true \ 5 \ 7)
\]

evaluates to 5, but any \textit{or} expression evaluates to either \textit{true} or \textit{false}.

The fact is (and this fact is for you to use only in this problem, and nowhere else in your CS17 code!) that \textit{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 \textit{or}-Rule for \textit{(or expr1 expr2)} then reads “evaluate \textit{expr1} to get a value \textit{v}; if the value \textit{v} is \textit{not false}, then \textit{v} is the value of the or-expression. If \textit{v} is false, evaluate \textit{expr2} to get a value \textit{v’}. If \textit{v’} is not false, then \textit{v’} is the value of the or-expression; if it is false, then the value of the or-expression is \textit{false}.” In short, any “non-false” value is treated as if it were the boolean \textit{true} and gets returned! That means that

\[
(or \ 3 \ 5)
\]
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 if, and, cond expressions can be converted into 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 and or 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 true and false? Can we get rid of procedure-application-expressions? Can we get rid of 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 CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)