

## Laboratory: Further List Recursion

**Summary:** In a previous laboratory, you experimented with and developed procedures that recurse over lists of numbers. In this laboratory, you will continue your exploration of recursion over lists, focusing on lists of symbols or lists of lists.

### Contents:

- Exercises
  - Exercise 0: Preparation
  - Exercise 1: How Much Did You Skip?
  - Exercise 2: No Skipping Allowed
  - Exercise 3: Tallying Symbols
  - Exercise 4: Lengths of Lists
  - Exercise 5: Counting Odd Numbers
  - Exercise 6: Extracting Odds
  - Exercise 7: Finding the Gaps
  - Exercise 8: Are They All Valid?
  - Exercise 9: Is It There?
  - Exercise 10: Riffing Lists
- For Those With Extra Time
  - Extra 1: Unriffing Lists
  - Extra 2: Membership, Revisited
  - Extra 3: Riffing, Revisited
  - Extra 4: Length, Revisited

## Exercises

### Exercise 0: Preparation

- a. Reflect on the patterns of list recursion you learned in the first lab on recursion and the reading on recursion. By “patterns”, I mean the general forms the recursive procedures took.
- b. Start DrScheme.

### Exercise 1: How Much Did You Skip?

Define and test a Scheme procedure, (`tally-skips lst`), that takes one argument, a list, and determines how many times the symbol `skip` occurs in the list.

For example,

```
> (tally-skips (list 'hop 'skip 'jump 'skip 'and 'skip 'again))
3
```

## Exercise 2: No Skipping Allowed

Define and test a Scheme procedure, (`remove-skips lst`), that takes a list of symbols as its argument and returns a list that does not contain the symbol `skip`, but is otherwise identical to the given list. (Use the predicate `eq?` to test whether two symbols are alike.)

```
> (remove-skips (list 'hop 'skip 'jump 'skip 'and 'skip 'again))
(hop jump and again)
```

The example illustrates the intended effect of the procedure. By itself, however, it's not an adequate test of your procedure. It would be a good idea to test the case in which the given list is empty, a case in which it contains only `skips`, and one in which it contains only symbols other than `skip`. You might also test different positions of `skip`: at the front, at the end, and in the middle.

We recommend that you test the procedures you create very thoroughly. In most cases, testing does not reveal any errors in your procedures; but finding and correcting the errors that testing exposes is one of the most productive and rewarding uses of a programmer's time.

## Exercise 3: Tallying Symbols

Define and test a Scheme procedure, (`tally-occurrences sym symbols`), that takes two arguments, a symbol and a list of symbols, and determines how many times the given symbol occurs in the given list.

**Hint:** Use direct recursion. Here are the questions that you must resolve: What is the base case? What value should the procedure return in that case? How can you simplify the problem in order to recursively invoke the procedure being defined? What do you need to do with the value of the recursive procedure call in order to obtain the final result?

```
> (tally-occurrences 'apple (list 'pear 'apple 'cranberry 'banana 'apple))
2
> (tally-occurrences 'apple (list 'oak 'elm 'maple 'spruce 'pine))
0
```

When you are done implementing this procedure, **add `tally-occurrences`** to your library.

## Exercise 4: Lengths of Lists

Define and test a Scheme procedure, (`lengths lists`) that takes a list of lists as its argument and returns a list of their lengths:

```

> (lengths (list (list 'alpha 'beta 'gamma)
                 (list 'delta)
                 null
                 (list 'epsilon 'zeta 'eta 'theta 'iota 'kappa)))
(3 1 0 6)

```

## Exercise 5: Counting Odd Numbers

Write a Scheme procedure, (`tally-odds values`), that returns the number of odd numbers in a list. For example,

```

> (tally-odds (list 1 2 3 4 5))
3
> (tally-odds null)
0
> (tally-odds (list 2 4 6 8 10))
0
> (tally-odds (list 1 2 1 2 1 1))
4

```

If you'd like to be extra careful, make sure that your procedure works correctly even if some of the values in the list are not numbers. For example,

```

> (tally-odds (list 'one 2 3 4 'five))
1

```

## Exercise 6: Extracting Odds

Write a Scheme procedure, (`odds values`), that, given a list, produces another list that contains only the odd numbers in the first list. For example,

```

> (odds (list 1 2 3 4 5))
(1 3 5)
> (odds (list 'one 'two 'three 4 'five))
()
> (odds (list (list 1 2 3) (list 4 5 6)))
()

```

## Exercise 7: Finding the Gaps

Define and test a Scheme procedure, (`gaps values`), that takes a non-empty list of real numbers as its argument and returns a list of the disparities between numbers that are adjacent on the given list.

```

> (gaps (list 30 16 21 9 42))
(14 5 12 33)
> (gaps (list 1 2))
(1)
> (gaps (list 2 1))
(1)

```

*Hint:* What is the base case?

*Note:* The `gaps` procedure always returns a list one element shorter than the one it is given.

## Exercise 8: Are They All Valid?

Define and test a Scheme predicate, `(all-in-range? values)`, that takes a list as argument and determines whether all of its elements are in the range from 0 to 100, inclusive.

For example,

```
> (all-in-range? (list 40 50 100 10))
#t
> (all-in-range? (list 40 -50 100 10))
#f
> (all-in-range? (list))
#t
```

## Exercise 9: Is It There?

Define and test a Scheme predicate, `(member? sym symbols)`, that takes two arguments, a symbol and a list, and determines whether the given symbol appears within the given list.

For example,

```
> (member? 'a (list 'a 'b 'c))
#t
> (member? 'a (list 'b 'c 'b 'd))
#f
> (member? 'a (list 'd' 'c 'b 'a))
#t
```

*Note:* Some of you have already discovered the standard procedure, `member`, which does something very similar. Please define `member?` recursively, without using `member`.

When you are done implementing this procedure **add `member?`** to your library.

## Exercise 10: Riffling Lists

Develop a Scheme procedure, `(riffle lst1 lst2)`, that takes two lists as arguments and returns a list that results from riffling the given lists together, like two halves of a deck of cards: Element 0 of the result list should be element 0 of `lst1`, element 1 of the result list should be element 0 of `lst2`, element 2 of the result list should be element 1 of `lst1`, element 3 of the result list should be element 1 of `lst2`, and so on and so forth. Keep taking subsequent elements from alternating lists until the shorter list is exhausted. Once the shorter of the given lists is exhausted, all the rest of the elements of the result list should come from the other list.

```

> (riffle (list 'a 'b 'c 'd 'e) (list 'x 'y 'z))
(a x b y c z d e)
> (riffle null (list 'x 'y 'z))
(x y z)
> (riffle (list 'x 'y 'z) null)
(x y z)
> (riffle (list 'x 'y 'z) (list 'a))
(x a y z)
> (riffle (list 'a) (list 'x 'y 'z))
(a x y z)

```

*Note:* There are a few ways to approach this problem. The approach you take will be guided by your selection of a base case or cases. In particular, you might choose only one base case (typically, when *lst1* is null) or you might choose two base cases (when *lst1* is null; when *lst2* is null).

When you are done implementing this procedure **add riffle** to your library.

## For Those With Extra Time

### Extra 1: Unriffling Lists

Define and test a Scheme procedure, (*unriffle lst*), that takes a list as argument and returns a list of two lists, one comprising the elements in even-numbered positions in the given list, the other comprising the elements in odd-numbered-positions.

```

> (unriffle (list 'a 'b 'c 'd 'e 'f 'g 'h 'i))
((a c e g i) (b d f h))

```

*Hint:* One strategy is to define a separate helper procedure that takes the car of the given list and the result of the recursive call as its arguments and rearranges the pieces as necessary to obtain the final result.

### Extra 2: Membership, Revisited

It is possible to write the `member?` function mentioned above using `if/cond` expressions or by using a combination of `and`, `or`, and `not`.

- Rewrite `member?` using whichever technique you didn't use above.
- Which solution do you prefer? Why?

### Extra 3: Riffling, Revisited

The hint for the riffling question above suggests that there are two different solutions to the problem. Write a different solution than the one you first came up with. (If you had one base case, use two. If you had two base cases, use one.)

## Extra 4: Length, Revisited

Some of you have criticized the built-in `length` method for counting only the number of values in the top-level list. Write a procedure, `count-values` that counts the total number of non-list values in a list or its sublists.

For example,

```
> (count-values (list 'a 'b 'c))
3
> (count-values null)
0
> (count-values (list (list 1 2 3 (list 4 5)) (list 'a 'b) (list (list 2))))
8
> (count-values (list null null null null))
0
```

---

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.