Tail Recursion

- Exercises
  - Exercise 0: Preparation
  - Exercise 1: Identifying Tail-Recursive Procedures
  - Exercise 2: Does Tail Recursion Really Make a Difference?
  - Exercise 3: Finding the Longest String on a List
  - Exercise 4: Finding the Index
  - Exercise 5: Iota, Once Again
  - Exercise 6: Reverse
  - Exercise 7: Append

Exercises

Exercise 0: Preparation

a. Go over the short reading on tail recursion.

b. Start DrScheme.

Exercise 1: Identifying Tail-Recursive Procedures

Identify three (or more) tail-recursive procedures you’ve already written.

Exercise 2: Does Tail Recursion Really Make a Difference?

In DrScheme, you can find out how long it takes to evaluate an expression by using `(time expression)`. The `time` procedure prints out the time it takes to evaluate and then returns the value computed. If you care only about the time it takes, you can write

> `(let ((junk (time expression))))`

a. Try the two versions of `factorial` on some large numbers. Does one seem to be faster than the other?

b. Try the three versions of `add-to-all` on some lists of varying sizes (you’ll probably need at least a hundred values in each list, and possibly more) until you can determine a difference in running times.

Exercise 3: Finding the Longest String on a List

Write a tail-recursive `longest-string-on-list` procedure.
Exercise 4: Finding the Index

Define a tail-recursive procedure `index` that has two arguments, an item `a` and a list of items `ls`, and returns the index of `a` in `ls`, that is, the zero-based location of `a` in `ls`. If the item is not in the list, the procedure returns -1. Test your procedure on:

(index 3 (list 1 2 3 4 5 6)) --> 2
(index 'so (list 'do 're 'mi 'fa 'so 'la 'ti 'do)) --> 4
(index "a" (list "b" "c" "d")) --> -1
(index 'cat null) --> -1

Exercise 5: Iota, Once Again

Define and test a tail-recursive version of Iota.

Exercise 6: Reverse

Write your own tail-recursive version of reverse.

Exercise 7: Append

This is an optional extra-credit exercise.

Here’s a non-tail-recursive version of append.

(define append
  (lambda (list-one list-two)
    (if (null? list-one) list-two
      (cons (car list-one)
        (append (cdr list-one) list-two)))))

a. Write your own tail-recursive version.

b. Determine experimentally which of the three versions (built-in, given above, tail-recursive) is fastest.