

## Recursion on Numbers

- Exercises
  - Exercise 0: Preparation
  - Exercise 1: Power of two
  - Exercise 2: Counting down
  - Exercise 3: Filling lists
  - Exercise 4: Counting To
  - Exercise 5: Counting Between
  - Exercise 6: Counting To, Revisited
  - Exercise 7: How Many Digits?
- Notes

### Exercises

#### Exercise 0: Preparation

- Make sure you understand the reading on recursion and the second reading on recursion.
- Start DrScheme

#### Exercise 1: Power of two

Using recursion over natural numbers, define and test a recursive Scheme procedure, (`power-of-two power`) that takes a natural number as its argument and returns the result of raising 2 to the power of that number. For instance, the value of (`power-of-two 3`) should be  $2^3$ , or 8.

It is possible to define this procedure non-recursively, using Scheme's primitive `expt` procedure, but the point of the exercise is to use recursion.

#### Exercise 2: Counting down

Define and test a Scheme procedure, (`count-down val`), that takes a natural number as argument and returns a list of all the natural numbers less than or equal to that number, in descending order:

```
> (count-down 5)
(5 4 3 2 1 0)
> (count-down 0)
(0)
```

### Exercise 3: Filling lists

Define and test a Scheme procedure, `(fill-list value count)`, that takes two arguments, the second of which is a natural number, and returns a list consisting of the specified number of repetitions of the first argument:

```
> (fill-list 'sample 5)
(sample sample sample sample sample)
> (fill-list (list 'left 'right) 3)
((left right) (left right) (left right))
> (fill-list null 1)
(())
```

### Exercise 4: Counting To

Define and test a recursive Scheme procedure that takes a natural number as argument and returns a list of all the natural numbers that are strictly less than the argument, in ascending order. (The traditional name for this procedure is `iota` -- another Greek letter.)

### Exercise 5: Counting Between

You may recall the `count-from` procedure from the reading on recursion over natural numbers.

What is the value of the call `(count-from -10 10)`?

- Write down what you think that it should be.
- Copy the definition of `count-from` into DrScheme and use it to find out what the call actually returns.

### Exercise 6: Counting To, Revisited

Using `count-from`, define and test a Scheme procedure that takes a natural number as argument and returns a list of all the natural numbers that are strictly less than the argument, in ascending order. Note that your procedure **must** use `count-from` as a helper.

### Exercise 7: How Many Digits?

Here is the definition of a procedure that computes the number of digits in the decimal representation of number:

```
(define number-of-decimal-digits
  (lambda (number)
    (if (< number 10)
        1
        (+ (number-of-decimal-digits (quotient number 10)) 1))))
```

a. Test this procedure.

The definition of `number-of-decimal-digits` uses direct recursion.

b. Describe the base case of this recursion.

c. Identify and describe the way in which a simpler instance of the problem is created for the recursive call.

d. Explain how the procedure correctly determines that the decimal numeral for the number 2000 contains four digits.

e. What preconditions does `number-of-decimal-digits` impose on its argument?

## Notes

For those of you unable to find the reading on recursion over natural numbers and for completeness, here is the `count-from` procedure.

```
;;; Procedure:
;;; count-from
;;; Parameters:
;;; lower, a natural number
;;; upper, a natural number
;;; Purpose:
;;; Construct a list of the natural numbers from lower to upper,
;;; inclusive, in ascending order.
;;; Produces:
;;; ls, a list
;;; Preconditions:
;;; lower <= upper
;;; Both lower and upper are numbers, exact, integers, and non-negative.
;;; Postconditions:
;;; The length of ls is upper - lower + 1.
;;; Every natural number between lower and upper, inclusive, appears
;;; in the list.
;;; Every value in the list with a successor is smaller than its
;;; successor.
;;; For every natural number k less than or equal to the length of
;;; ls, the element in position k of ls is lower + k.
(define count-from
  (lambda (lower upper)
    (if (= lower upper)
        (list upper)
        (cons lower (count-from (+ lower 1) upper)))))
```