Fundamentals of Computer Science I (CS151.02 2007S)

Numeric Recursion

Summary: Although most of our prior experiments with recursion have emphasized recursion over lists, it is also possible to use other values as the basis of recursion. In this laboratory, you will explore the use of natural numbers (non-negative integers) as the basis of recursion.

Contents:

- 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?
  - Exercise 8: Nesting Lists
- For Those Who Finish Early
  - Extra 1: Powers of Two, Revisited
  - Extra 2: `digit-of?`
  - Extra 3: Summing Digits
- Notes
  - `count-from`

Exercises

Exercise 0: Preparation

a. Make sure you understand the initial reading on recursion and the followup reading on numeric recursion.

b. 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 (an integer greater than or equal to 0) as its argument and returns the result of raising 2 to the power of that number. For example,

```
> (power-of-two 3)
8
> (power-of-two 10)
1024
> (power-of-two 1)
2
```

It is possible to implement 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)
```

Note that you should use `cons` to build up the list.

Note also that you are better off writing this with direct recursion, rather than using a helper procedure.

When you are finished writing this procedure, **add `count-down` to your Scheme library.**

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)
(())
> (fill-list null 2)
(() ())
> (fill-list 'hello 0)
()
```

Again, even if you know a built-in procedure to do this task, please implement it recursively.

When you are finished writing this procedure, **add `fill-list` to your Scheme library.**

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`, a Greek letter.)

For example,

```
> (iota 3)
(0 1 2)
> (iota 5)
(0 1 2 3 4)
> (iota 1)
(0)
```

Note that you will probably need to use a helper of some sort to write `iota`. You might use the traditional form of helper, which adds an extra parameter. You might also use a helper that simply computes iota in the reverse order. (Most students write a backwards iota in the first attempt; instead of throwing it away, rename it and call it from iota.)

When you are done, **add `iota` to your library**.

Exercise 5: Counting Between

You may recall the `count-from` procedure from the reading on recursion over natural numbers. That procedure is also reproduced at the end of this lab.

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

a. Write down what you think that it should be.

b. Copy the definition of `count-from` into DrScheme and use it to find out what the call actually returns.

When you are finished writing this procedure, **add `count-from` to your Scheme library.**

Exercise 6: Counting To, Revisited

Using `count-from` as a helper, define and test a Scheme procedure, `(new-iota` *`n`*`)`, 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.

For example,

```
> (new-iota 3)
(0 1 2)
> (new-iota 5)
(0 1 2 3 4)
> (new-iota 1)
(0)
```

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. That is, explain what problem is solved recursively and why you know that that problem is simpler.

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?

When you are finished writing this procedure, **add `number-of-decimal-digits` to your Scheme library.**

Exercise 8: Nesting Lists

Develop a Scheme procedure, (`nest` *`val depth`*) that takes any value, *val* as its first argument and any natural number, *depth*, as its second argument, and "nests" *val* in *depth* lists. To "nest" a value in a list, simply place it in a singleton list. To nest it in two lists, place that first list in a singleton list. To nest in three lists, place that depth-two list in a singleton list. And so on and so forth.

For example,

```
> (nest 'contents 5)
(((((contents)))))
> (nest #t 1)
(#t)
> (nest (list 'alpha 'beta) 2)
(((alpha beta)))
> (nest 'alpha 2)
((alpha))
> (nest 'notnested 0)
notnested
```

For Those Who Finish Early

Extra 1: Powers of Two, Revisited

Rewrite `power-of-two` to permit negative exponents.

Extra 2: `digit-of?`

Define and test a procedure, (`digit-of?` *`digit num`*), returns `#t` if *digit* is a digit of the natural number num, and `#f` otherwise. For example,

```
> (digit-of? 5 7523)
#t
> (digit-of? 5 999338)
#f
```

Note that you may want to use *number-of-decimal-digits* as a pattern for writing this procedure.

Extra 3: Summing Digits

Write a procedure, (`sum-of-digits` *`num`*), that sums the digits in natural number *num*.

Note that you may want to use *number-of-decimal-digits* as a pattern for writing this procedure.

Notes

`count-from`

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)))))
```

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.