Fundamentals of Computer Science I: Media Computing (CS151.02 2007F)

Numeric Recursion

Summary: Our first explorations of recursion involved procedures that take lists as parameters. However, recursive procedures can also take other types as parameters, such as numbers. In this reading, we consider techniques to recurse over numbers, particularly integers.

Contents:

- Introduction
- The Structure of Recursive Procedures
- Numeric Base Cases
- An Example: Termial
  - Watching `termial` in Action
  - Preconditions for `termial`
  - A Note on the Implementation of `termial`
- Other Numeric Base Cases

Introduction

While the recursive procedures we've written so far have used lists as the basis of recursion, we can also write recursive procedures with other types as the basis of recursion. In particular, natural numbers provide a nice basis of recursion. Like lists, natural numbers have a recursive structure of which we can take advantage when we write direct-recursion procedures. A natural number, *n*, is either (a) zero, or (b) the successor of a smaller natural number, which we can obtain by subtracting 1 from *n*.

The Structure of Recursive Procedures

Recall that the common format of a recursive procedure is

```
(define recursive-proc
  (lambda (val)
    (if (base-case-test)
        (base-case val)
        (combine (partof val)
                 (recursive-proc (simplify val))))))
```

For lists, the test for a base case was typically "is the list empty" or "does the list have only one value", which we would express as `(empty? lst)` and `(empty? (cdr lst))`, respectively. We typically simplify a list by taking the cdr of the lst. Hence, the simplest form of a recursive procedure for lists is

```
(define recursive-proc
  (lambda (lst)
    (if (null? lst)
        (base-case)
        (combine (car lst)
                 (recursive-proc (cdr lst))))))
```

But we'll find different that with different data types, we'll use different base case tests and different mechanisms for simplifying values.

Numeric Base Cases

To write recursive procedures with numeric arguments, we first need a technique for identifying the base case. With natural numbers, 0 often provides an appropriate base case. Standard Scheme provides the predicate `zero?` to distinguish between the base and recursive cases, so we can again use an `if`-expression to ensure that only the expression for the appropriate case is evaluated. So we can write a procedure that applies to *any* natural number if we know (a) what value it should return when the argument is 0 and (b) how to convert the value that the procedure would return for the next smaller natural number into the appropriate return value for a given non-zero natural number.

Hence, a typical numeric recursive procedure will look something like the following:

```
(define recursive-proc
  (lambda (val)
    (if (zero? val)
        (base-case)
        (combine val (recursive-proc (- val 1))))))
```

In this sample code, we subtract 1 to simplify the number. However, we can also subtract more than 1, or divide the number by 2, or do anything else that gives a result that is closer to zero.

An Example: Termial

For instance, here is a procedure that, given a natural number, *number*, computes the result of adding together all of the natural numbers up to and including `number`. This result is traditional called the *termial* of the number.

```
;;; Procedure:
;;;   termial
;;; Parameters:
;;;   number, a natural number
;;; Purpose:
;;;   Compute the sum of natural numbers not greater than a given
;;;   natural number
;;; Produces:
;;;   sum, a natural number
;;; Preconditions:
;;;   number is a number, exact, an integer, and non-negative.
;;;   The sum is not larger than the largest integer the language
;;;     permits.
;;; Postconditions:
;;;   sum is the sum of natural numbers not greater than number.
;;;   That is, sum = 0 + 1 + 2 + ... + number
(define termial
  (lambda (number)
    (if (zero? number)
        0
        (+ number (termial (- number 1))))))
```

Whereas in a list recursion, we called the `cdr` procedure to reduce the length of the list in making the recursive call, the operation that we apply in recursion with natural numbers is reducing the number by 1.

Watching `termial` in Action

Here's a summary of what actually happens during the evaluation of a call to the `termial` procedure, say, `(termial 5)`:

```
   (termial 5)
=> (+ 5 (termial 4))
=> (+ 5 (+ 4 (termial 3)))
=> (+ 5 (+ 4 (+ 3 (termial 2))))
=> (+ 5 (+ 4 (+ 3 (+ 2 (termial 1)))))
=> (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (termial 0))))))
=> (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))
=> (+ 5 (+ 4 (+ 3 (+ 2 1))))
=> (+ 5 (+ 4 (+ 3 3)))
=> (+ 5 (+ 4 6))
=> (+ 5 10)
=> 15
```

Preconditions for `termial`

The restriction that `termial` takes only non-negative integers as arguments is an important one: If we gave it a negative number or a non-integer, we'd have a runaway recursion, because we cannot get to zero by subtracting 1 repeatedly from a negative number or from a non-integer, and so the base case would never be reached. For example,

```
   (termial -5)
=> (+ -5 (termial -6))
=> (+ -5 (+ -6 (termial -7)))
=> (+ -5 (+ -6 (+ -7 (termial -8))))
=> (+ -5 (+ -6 (+ -7 (+ -8 (termial -9)))))
=> ...
```

Similarly, if we gave the `termial` procedure an approximation rather than an exact number, we might or might not be able to reach zero, depending on how accurate the approximation is and how much of that accuracy is preserved by the subtraction procedure.

A Note on the Implementation of `termial`

Note that our "sum all the values" algorithm is not the only way to compute the termial of a natural number. Many of you may have learned a more efficient (or at least more elegant) algorithm. We'll return to this algorithm later.

Other Numeric Base Cases

The important part of getting recursion to work is making sure that the base case is inevitably reached by performing the simplification operation enough times. For instance, we can use direct recursion on exact positive integers with 1, rather than 0, as the base case.

```
;;; Procedure:
;;;   factorial
;;; Parameters:
;;;   number, a positive integer
;;; Purpose:
;;;   Compute number!, the product of positive integers not
;;;   greater than a given positive integer.
;;; Produces:
;;;   product, an integer
;;; Preconditions:
;;;   number is a number, exact, an integer, and positive.
;;;   The product is not larger than the largest integer the
;;;     language permits.
;;; Postconditions:
;;;   product is the product of the positive integers not
;;;   greater than number.  That is,
;;;     product = 1 * 2 * ... * number
(define factorial
  (lambda (number)
    (if (= number 1)
        1
        (* number (factorial (- number 1))))))
```

We require the invoker of this `factorial` procedure to provide an exact, strictly positive integer. (Zero won't work in this case, because we can't reach the base case, 1, by repeated subtractions if we start from 0.)

Similarly, we can use direct recursion to approach the base case from below by repeated additions of 1, if we know that our starting point is less than or equal to that base case. Here's an example:

```
;;; 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)))))
```

Why is this useful? Well, we can use the list of values to make a horizontal line of twenty red spots, as in

```
(define red-line
  (map (lambda (n) (spot.new n 0 color.red))
       (count-from 0 19)))
```

We could also use the list of values to make a grey diagonal line whose darkness changes gradually.

```
(define grey (lambda (n) (rgb.new n n n)))
(define grey-line
  (map (lambda (n) (spot.new n n (grey (+ 128 (* 8 n)))))
       (count-from 0 16)))
```

We'll return to some of these lines in the laboratory and in the next reading.
