SICP section 2.2.3

August 22, 2007 at 05:45 Tags SICP

First I'm going to implement the `filter` and `accumulate` functions in CL (although `filter` is just another name for the CL library function `remove-if-not` and `accumulate` can be trivially modeled with `reduce`):

```
(defun filter (predicate sequence)
  (cond ((null sequence) nil)
        ((funcall predicate (car sequence))
          (cons (car sequence)
                (filter predicate (cdr sequence))))
        (t (filter predicate (cdr sequence)))))

(defun accumulate (op initial sequence)
  (if (null sequence)
    initial
    (funcall op
      (car sequence)
      (accumulate op initial (cdr sequence)))))
```

Exercise 2.33

```
(defun my-map (p sequence)
  (accumulate
    (lambda (x y) (cons (funcall p x) y))
    nil sequence))

(defun my-append (seq1 seq2)
  (accumulate #'cons seq2 seq1))

(defun my-length (sequence)
  (accumulate (lambda (x y) (1+ y)) 0 sequence))
```

Exercise 2.34

We can think of this computation as the accumulation of the current coefficient and `x` multiplied by *next*, where *next* is the same operation for a higher coefficient, until the end is reached (read the Horner formula shown in the book from right to left). This easily translates into the code:

```
(defun horner-eval (x coeffs)
  (accumulate
    (lambda (this-coeff higher-terms)
      (+ (* x higher-terms) this-coeff))
    0
    coeffs))
```

Exercise 2.35

This exercise is an interesting brain-stretcher. The authors show in their template that the last argument to `accumulate` is a call to `map`. My solution is different. First of all, it is important to understand that we'll need a recursive call to `count-leaves`. This is always the case for tree recursion – `accumulate` itself only goes over a linear sequence, and we need to delve recursively into the tree. This is a good tip that should get us started.

Now, consider how `accumulate` works. The `lambda` it accepts as `op` has the result of sub-sequence accumulation as its second argument. The first argument is the interesting one, because this is where the real work is done. If `x` is a leaf (`consp` returns false), we add 1. If it's a node with children, we delve recursively (think of the original implementation of `count-leaves`).

```
(defun count-leaves (tree)
  (accumulate
    (lambda (x y)
      (+  y
          (if (consp x)
            (count-leaves x)
            1)))
    0
    tree))
```

Exercise 2.36

```
(defun accumulate-n (op init seqs)
  (if (null (car seqs))
    nil
    (cons (accumulate op init (mapcar #'car seqs))
          (accumulate-n op init (mapcar #'cdr seqs)))))
```

Exercise 2.37

```
(defun dot-product (v w)
  (accumulate #'+ 0 (mapcar #'* v w)))

(defun matrix-*-vector (m v)
  (mapcar
    (lambda (row)
      (dot-product row v))
    m))

(defun transpose (m)
  (accumulate-n #'cons nil m))

(defun matrix-*-matrix (m n)
  (let ((n-t (transpose n)))
    (mapcar (lambda (row) (matrix-*-vector n-t row)) m)))
```

Exercise 2.38

First let's rewrite `accumulate` as `fold-right` and add the definition of `fold-left`:

```
(defun fold-right (op init seq)
  (if (null seq)
    init
    (funcall op
      (car seq)
      (fold-right op init (cdr seq)))))

(defun fold-left (op init seq)
  (labels (
    (iter (result rest)
      (if (null rest)
        result
        (iter (funcall op result (car rest))
              (cdr rest)))))
    (iter init seq)))
```

Note that for simple operations like addition, the two produce equivalent results:

```
(fold-right #'* 1 '(1 2 3 4 5))
=> 120
(fold-left #'* 1 '(1 2 3 4 5))
=> 120
```

But for the examples asked about in the exercise:

```
(fold-right #'/ 1 '(1 2 3))
=> 3/2
(fold-left #'/ 1 '(1 2 3))
=> 1/6

(fold-right #'list nil '(1 2 3))
=> (1 (2 (3 NIL)))
(fold-left #'list nil '(1 2 3))
=> (((NIL 1) 2) 3)
```

`fold-right` and `fold-left` will produce the same results if `op` is an associative operation.

Exercise 2.39

```
(defun reverse-r (seq)
  (fold-right (lambda (x y) (append y (list x))) nil seq))

(defun reverse-l (seq)
  (fold-left (lambda (x y) (cons y x)) nil seq))
```

Exercise 2.40

Here is the scaffolding for this and the following exercises:

```
(defun enumerate-interval (low high)
  (if (> low high)
    nil
    (cons low (enumerate-interval (1+ low) high))))

(defun flatmap (proc seq)
  (accumulate #'append nil (mapcar proc seq)))

(defun sum (lst)
  (accumulate #'+ 0 lst))

(defun prime-sum? (pair)
  (prime? (sum pair)))

(defun make-pair-sum (pair)
  (list (car pair) (cadr pair) (sum pair)))
```

The solution for the exercise is:

```
(defun unique-pairs (n)
  (flatmap
    (lambda (i)
      (mapcar (lambda (j) (list i j))
              (enumerate-interval 1 (1- i))))
    (enumerate-interval 1 n)))

(defun prime-sum-pairs (n)
  (mapcar
    #'make-pair-sum
    (filter #'prime-sum? (unique-pairs n))))
```

Exercise 2.41

```
(defun unique-triples (n)
  "Unique triples of numbers <= n"
  (flatmap
    (lambda (i)
      (flatmap
        (lambda (j)
          (mapcar
            (lambda (k) (list i j k))
            (enumerate-interval 1 (1- j))))
        (enumerate-interval 1 (1- i))))
    (enumerate-interval 1 n)))

(defun triples-sum-s (s n)
  "Triples of numbers <= n summing to s"
  (filter
    (lambda (triple)
      (= (sum triple) s))
    (unique-triples n)))
```

Exercise 2.42

A single position is defined by a row and a column:

```
(defun make-position (row col)
  (cons row col))

(defun position-row (pos)
  (car pos))

(defun position-col (pos)
  (cdr pos))

(defun positions-equal (a b)
  (equal a b))
```

A set of positions is just a list of position objects:

```
(defvar empty-board '())

(defun adjoin-position (row col positions)
  (append positions (list (make-position row col))))
```

Note how `adjoin-position` is implemented: in order to append the new position to the list's end, I use `append`. But `append` expects lists as its argument, so the new position is wrapped in a call to `list`. There must be a more elegant way to do this :-)

And this is the implementation of `safe?` and its helper function `attacks?`:

```
(defun attacks? (a b)
  "Both a and b are positions. This function
  checks if a queen in position a attacks the
  queen in position b."
  (let ((a-row (position-row a))
        (a-col (position-col a))
        (b-row (position-row b))
        (b-col (position-col b)))
    (cond
      ((= a-row b-row) t) ; row attack
      ((= a-col b-col) t) ; column attack
      ((= (abs (- a-col b-col)) ; diagonal attack
          (abs (- a-row b-row))) t)
      (t nil))))

(defun safe? (k positions)
  "Is the queen in the kth column safe with
  respect to the queens in columns 1..k-1?"
  (let ((kth-pos (nth (1- k) positions)))
    (if (null (find-if
                (lambda (pos)
                  (and  (not (positions-equal kth-pos pos))
                        (attacks? kth-pos pos)))
                positions))
        t
        nil)))
```

To complete the picture, this is `queens` translated to CL:

```
(defun queens (board-size)
    (queen-cols board-size board-size))

(defun queen-cols (k board-size)
  (if (= k 0)
    (list empty-board)
    (filter
      (lambda (positions) (safe? k positions))
      (flatmap
        (lambda (rest-of-queens)
          (mapcar
            (lambda (new-row)
              (adjoin-position new-row k rest-of-queens))
            (enumerate-interval 1 board-size)))
        (queen-cols (1- k) board-size)))))
```

I tested this implementation and it works correctly for boards with size < 8. However, for board size 8 and higher, CLISP reports a stack overflow. When I traced `queen-cols` to see what the problem is, it appeared that the lists returned by `queen-cols` are very large. Since they are probably returned by value on the stack, this is what causes the overflow. I tried to increase CLISP's memory consumption parameter, but it doesn't help. It is apparently possible to do some more serious tweaking with CLISP's stack size allocation, but it's far from trivial. Seems like an annoying limitation in CLISP (imagine that this code ran without problems in the environment the authors used 25 years ago!).

When I tried it on SBCL running on Linux (Ubuntu 7), it worked without problems and generated correct solutions for board size 8.

Exercise 2.43

Here is Louis Reasoner's version of `queens`:

```
(defun louis-queens (board-size)
  (louis-queen-cols board-size board-size))

(defun louis-queen-cols (k board-size)
  (if (= k 0)
    (list empty-board)
    (filter
      (lambda (positions) (safe? k positions))
      (flatmap
        (lambda (new-row)
          (mapcar
            (lambda (rest-of-queens)
              (adjoin-position new-row k rest-of-queens))
            (louis-queen-cols (1- k) board-size)))
        (enumerate-interval 1 board-size)))))
```

Since the recursive call to `queen-cols` is a costly operation in terms of time, Louis's implementation looks suspicious just for the fact that it places it inside another loop[1]. Let's analyze it a little further:

The “normal” implementation takes the result of `queen-cols` for one column less, and for each position in the list attaches all the possible placements of the new queen. These new sets of positions are then filtered by `safe?`.

Louis’s implementation, on the other hand, does it in a different order, which is crucial. For each new possible position, it re-generates the result of `queen-cols` for one column less, and attaches the new position to it. Note that the call to `louis-queen-cols` doesn’t need the “loop variable” `new-row`, and hence it’s quite clear that placing it in the inner loop is needless[2].

Runtime analysis: `queen-cols` calls itself only once per execution. Therefore, for some board size `N`, `queen-cols` is called `N+1` times (the call with `N=0` returns immediately, though).

`louis-queen-cols`, however, calls itself `board-size` times per execution. But this continues recursively, because each `louis-queen-cols` called calls itself `board-size` times too. So the call tree is N-nary, each node has N children. The tree is N levels deep, so the total amount of nodes (calls) in it is `O(N^N)`. To find the ratio between it and the original `queen-cols`, we divide: `O(N^N) / N+1 = O(N^N)`. Therefore, asymptotically `louis-queen-cols` is `O(N^N)` times slower.

[1] Think of `filter`, `flatmap` and `mapcar` as loops, since they are sequence functions.

[2] To be completely fair, neither does the call to `enumerate-interval` in the inner loop of `queen-cols` use the loop variable. We can actually precompute the list returned by `(enumerate-interval 1 board-size)` and use it in each iteration. However, since `enumerate-interval` is very fast, the speed gain won’t be too significant.

For comments, please send me an email.

Back to top