SICP sections 1.2.4 - 1.2.5

July 04, 2007 at 04:45 Tags SICP

Section 1.2.4

Exercise 1.16

This is the fast recursive version, for reference:

```
(defun square (x)
  (* x x))

; Recursive version
(defun fast-expt (b n)
  (cond ((= n 0) 1)
        ((evenp n) (square (fast-expt b (/ n 2))))
        (t (* b (fast-expt b (- n 1))))))
```

And this is the iterative version, the requirement of the exercise. Note my usage of the `&optional` function parameter for `a`. Common Lisp supports optional and keyword parameters, which makes writing functions more convenient sometimes.

```
; Iterative version
; Note: using CL's &optional arguments instead
; of the auxiliary function
;
(defun fast-expt-iter (b n &optional (a 1))
  (cond ((= n 0) a)
        ((evenp n) (fast-expt-iter (square b) (/ n 2) a))
        (t (fast-expt-iter b (- n 1) (* b a)))))
```

Exercise 1.17

I will implement `double` and `halve` using normal CL operators. If our “machine” really doesn’t have multiplication instructions, the doubling and halving can be implemented by means of left and right shift, respectively.

```
(defun double (x)
  (* x 2))

(defun halve (x)
  (/ x 2))

(defun fast-mult (a b)
  (cond ((= b 0) 0)
        ((= b 1) a)
        ((evenp b) (double (fast-mult a (halve b))))
        (t (+ a (fast-mult a (- b 1))))))
```

Note that there are two special cases – for 0 and for 1. The case `(= b 0)` will not be reached during the recursion – it's only there to provide an answer when `(fast-mult a 0)` is requested by the user.

Exercise 1.18

After understanding the technique of solution in exercise 1.16, this one is quite simple. The auxuliary state variable is `acc` – it accumulates the sum, and the invariant `answer = a*b + acc` holds on each iteration.

```
(defun fast-mult-iter (a b &optional (acc 0))
  (cond ((= b 0) acc)
        ((evenp b) (fast-mult-iter (double a) (halve b) acc))
        (t (fast-mult-iter a (1- b) (+ a acc)))))
```

Exercise 1.19

```
Tpq(a,b) = {a <- bq + aq + ap
           {b <- bp + aq
```

To apply `Tpq` twice, we replace `a` and `b` on the right hand side in the equations above with their definitions by `Tpq` (that is, with the full expansion).

```
Tp'q'(a,b) = {a <- bpq + aqq + bqp + aqp + app + bqq + aqq + apq
             {b <- bpp + aqp + bqq + aqq + apq

           = {a <- (pq + qq + qp)*b + (pq + qq + qp)*a + (qq + pp)*a
           = {b <- (pp + qq)*b + (pq + qp + qq)*a
```

Restructured this way, we see that indeed the double transformation works, and we get `Tpq` back if we define:

```
p' = pp + qq
q' = pq + qp + qq
```

Now we are ready to write the code:

```
(defun fast-fib-iter (n)
  (fast-fib-iter-aux 1 0 0 1 n))

(defun fast-fib-iter-aux (a b p q count)
  (cond ((= count 0) b)
        ((evenp count)
          (fast-fib-iter-aux a
                             b
                             (+ (* p p) (* q q))     ; p'
                             (+ (* 2 p q) (* q q))   ; q'
                             (/ count 2)))
        (t (fast-fib-iter-aux (+ (* b q) (* a q) (* a p))
                              (+ (* b p) (* a q))
                              p
                              q
                              (1- count)))))
```

Section 1.2.5

The authors present *Euclid's Algorithm* for computing the GCD (Greatest Common Divisor) of two numbers. It's a very interesting and important algorithm, worth to understand. Here is a simple proof I lifted from TAOCP section 1.1[1].

```
GCD(a, b)

The remainder of the division a/b is r.
We can write this as:

qb + r = a
(for some integer q - the quotient of the division)

This means that any number that divides b and r must
also divide a.

The equation can also be rewritten as:

r = a - qb

This means that any number that divides a and b must
also divide r. So, the set of divisors of b and r
is the same as the set of divisors of a and b. In
particular, the greatest common divisor is the same.
Q.E.D.
```

Also, the Lamé theorem is presented – it provides a fascinating connection between Euclid's algorithm and Fibonacci numbers, and this is used to prove the the runtime of Euclid's algorithm is logarithmic.

Exercise 1.20

We get back to applicative vs. normal order evaluation. In applicative order, the process generated

is as follows (the number following a comment counts actual executions of `remainder`):

```
(gcd 206 40)
(gcd 40 (remainder 206 40)) ; 1
(gcd 40 6)
(gcd 6 (remaider 40 6)) ; 2
(gcd 6 4)
(gcd 4 (remainder 6 4)) ; 3
(gcd 4 2)
(gcd 2 (remainder 4 2)) ; 4
(gcd 2 0)
2
```

So, `remainder` is called 4 times, exactly in the sequence given in the example process. By `->` I mean that a call to `gcd` occurs.

```
(gcd 206 40)
(gcd 40 (remainder 206 40))
->
(if (= (remainder 206 40) 0) ; 1 time
  40
  (gcd (remainder 206 40)
    (remainder 40 (remainder 206 40))))
->
(if (= (remainder 40 (remainder 206 40)) 0) ; 2 times
  (remainder 206 40)
  (gcd
    (remainder
      40
      (remainder 206 40))
    (remainder
      (remainder 206 40)
      (remainder
        40
        (remainder 206 40)))))
->
if (= (remainder (remainder 206 40) (remainder 40 (remainder 206 40))) 0) ; 4 time
  (remainder 40 (remainder 206 40))
  (gcd
    (remainder
      (remainder 206 40)
      (remainder
        40
        (remainder 206 40)))
    (remainder
      (remainder 40 (remainder 206 40))
      (remainder
        (remainder 206 40)
        (remainder
          40
          (remainder 206 40)))))
->
if (= (...)) ; 7 times (b from above)

And this indeed equals 0, so the next is evaluated (a from above):
  (remainder
    (remainder 206 40)
    (remainder
      40
      (remainder 206 40)))

Another 4 calls.
In total 1 + 2 + 4 + 7 + 4 = 18 calls to remainder
```

Footnotes

[1] Donald Knuth's *The Art Of Computer Programming*.

For comments, please send me an email.

© 2003-2025 Eli Bendersky

Back to top