SICP section 1.2.6

July 09, 2007 at 05:47 Tags SICP

The section begins with a discussion of the deterministic test of primality of `n` by testing all numbers between 2 and the square root of `n`. Here is the CL code:

```
(defun smallest-divisor (n)
  (find-divisor n 2))

(defun find-divisor (n test-divisor)
  (cond ((> (square test-divisor) n) n)
    ((divides? test-divisor n) test-divisor)
    (t (find-divisor n (+ test-divisor 1)))))

(defun divides? (a b)
  (= (rem b a) 0))

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

(defun prime? (n)
  (= n (smallest-divisor n)))
```

And this is the code for `expmod`:

```
(defun expmod (base exponent m)
  (cond ((= exponent 0) 1)
        ((even? exponent)
          (rem  (square (expmod base (/ exponent 2) m))
                m))
        (t
          (rem  (* base (expmod base (- exponent 1) m))
                m))))
```

As the authors said, you can easily see its similarity to `fast-expt` that was presented in an earlier section. Using a rule from number theory for multiplication with modulo, it is translated into this fast exponentation modulo function. Here is the rest of the code that implements fast probabilistic prime testing with the Fermat test:

```
(defun fermat-test (n)
  (defun try-it (a)
    (= (expmod a n n) a))
  (try-it (1+ (random (1- n)))))

(defun fast-prime? (n times)
  (cond ((= times 0) t)
        ((fermat-test n) (fast-prime? n (1- times)))
        (t nil)))
```

Exercise 1.21

Let's practice with CL's `dolist` and `format` forms:

```
(dolist (num '(199 1999 19999))
  (format t
    "The smallest divisor of ~d is ~d~%"
    num (smallest-divisor num)))
```

So, we see that while 199 and 1999 are primes, 19999's smallest divisor is 7.

Exercise 1.22

Common Lisp has a more convenient method for measuring the runtime of code. The macrc`time` can be called on any CL code and after running (evaluating) it, reports how long it took and how much memory it used. This is the `search-for-primes` function. Note that it uses several CL features like `when` and `do`:

```
(defun search-for-primes (start end)
  ; To go over the even numbers, we'll start from start+1
  ; if start is even and advance in steps of 2
  ;
  (let ((start (if (evenp start) (1+ start) start)))
    (do ((i start (+ i 2)))
        ((> i end))
      (when (prime? i)
        (format t "~d is prime~%" i)))))
```

And now I can run this function in any range I want, for example:

```
(search-for-primes 1000 1019)
=>
1009 is prime
1013 is prime
1019 is prime
```

Curiously, timing the runs for 1,000 10,000 100,000 and 1,000,000 doesn't work because my computer is too fast (SICP was written in the 80s!) – all the runs just take 0 seconds, which means that the runtime is below the resolution of the timer used in the timing. However, if I run it in a loop, I can get more reliable results:

```
(time (dotimes (i 1000 t) (search-for-primes 1000 1019)))
=>
1009 is prime
1013 is prime
1019 is prime
Real time: 0.6875 sec.
Run time: 0.6875 sec.
Space: 2472736 Bytes
GC: 4, GC time: 0.03125 sec.
```

So, the runtimes I'm getting are:

```
Range         Runtime for 1,000 iterations
-----         ----------------------------
1,000         0.68
10,000        2.5
100,000       6.15
1,000,000     17.8
```

These runtimes are definitely not `sqrt(10)` multiples of one another, but it's not too far from it. Of course, the `O(sqrt(10))` runtime is only an asymptotic bound. The real runtime depends on a lot of other things – the constants, the machine, the compiler, etc. The asymptotic bound is useful for an estimation, not exact prediction of the runtime.

Exercise 1.23

`find-divisor` will be rewritten as follows:

```
(defun next-divisor (n)
  (if (= n 2)
    3
    (+ n 2)))

(defun find-divisor (n test-divisor)
  (cond ((> (square test-divisor) n) n)
    ((divides? test-divisor n) test-divisor)
    (t (find-divisor n (next-divisor test-divisor)))))
```

When I run `(prime? 1000037)` 10000 times with the two methods I see that the speedup is indeed almost 2 (50 seconds vs. 26 seconds).

Exercise 1.24

While the speedup can't be matched exactly, the runtime grows very slowly which demonstrates logarithmic growth. For example, testing 1,037 (1k times) takes 0.1 seconds, 1,000,037 takes 0.67 seconds and 1,000,000,007 takes 1.07 seconds.

Exercise 1.25

There is a huge flaw in Alyssa's code. The call to `fast-expt` will result in huge numbers being crunched. Curiously, this problem takes time to show in Common Lisp which has built-in arbitrary precision arithmetic (huge integers). In many languages special libraries are required to handle integers larger than the machine word size (typically 2^32 – 1 for unsigned integers on 32-bit machines). Since prime tests are usually invoked on huge numbers (for example, the numbers used in RSA are typically hundreds of digits long), it just won't work. And if it will, it will take very long time because bignum arithmetic takes much much longer than ordinary arithmetic.

The `expmod` implementation provided by the authors, on the other hand, doesn't suffer from this problem. It uses a mathematical trick that keeps all numbers it works on low.

Exercise 1.26

This is a good time to present the Common Lisp built-in macro`trace`. When asked to `trace` a function prior to calling it, CL environments will display all calls to the function. This is very useful in order to analyze the call-graph of a function. For example:

```
; For CLISP, in order to indent recursive calls
(setf custom:*trace-indent* 1)

(trace expmod)
(expmod 15 10 10)
```

Shows:

```
;; Tracing function EXPMOD.
 1. Trace: (EXPMOD '15 '10 '10)
  2. Trace: (EXPMOD '15 '5 '10)
   3. Trace: (EXPMOD '15 '4 '10)
    4. Trace: (EXPMOD '15 '2 '10)
     5. Trace: (EXPMOD '15 '1 '10)
      6. Trace: (EXPMOD '15 '0 '10)
      6. Trace: EXPMOD ==> 1
     5. Trace: EXPMOD ==> 5
    4. Trace: EXPMOD ==> 5
   3. Trace: EXPMOD ==> 5
  2. Trace: EXPMOD ==> 5
 1. Trace: EXPMOD ==> 5
```

Implementing the alternative `expmod`:

```
(defun louis-expmod (base exponent m)
  (cond ((= exponent 0) 1)
        ((evenp exponent)
          (rem  (*  (louis-expmod base (/ exponent 2) m)
                    (louis-expmod base (/ exponent 2) m))
                m))
        (t
          (rem  (* base (louis-expmod base (- exponent 1) m))
                m))))
```

When traced with the same arguments we used to call`expmod` in the example above, this function generates:

```
;; Tracing function LOUIS-EXPMOD.
 1. Trace: (LOUIS-EXPMOD '15 '10 '10)
  2. Trace: (LOUIS-EXPMOD '15 '5 '10)
   3. Trace: (LOUIS-EXPMOD '15 '4 '10)
    4. Trace: (LOUIS-EXPMOD '15 '2 '10)
     5. Trace: (LOUIS-EXPMOD '15 '1 '10)
      6. Trace: (LOUIS-EXPMOD '15 '0 '10)
      6. Trace: LOUIS-EXPMOD ==> 1
     5. Trace: LOUIS-EXPMOD ==> 5
     5. Trace: (LOUIS-EXPMOD '15 '1 '10)
      6. Trace: (LOUIS-EXPMOD '15 '0 '10)
      6. Trace: LOUIS-EXPMOD ==> 1
     5. Trace: LOUIS-EXPMOD ==> 5
    4. Trace: LOUIS-EXPMOD ==> 5
    4. Trace: (LOUIS-EXPMOD '15 '2 '10)
     5. Trace: (LOUIS-EXPMOD '15 '1 '10)
      6. Trace: (LOUIS-EXPMOD '15 '0 '10)
      6. Trace: LOUIS-EXPMOD ==> 1
     5. Trace: LOUIS-EXPMOD ==> 5
     5. Trace: (LOUIS-EXPMOD '15 '1 '10)
      6. Trace: (LOUIS-EXPMOD '15 '0 '10)
      6. Trace: LOUIS-EXPMOD ==> 1
     5. Trace: LOUIS-EXPMOD ==> 5
    4. Trace: LOUIS-EXPMOD ==> 5
   3. Trace: LOUIS-EXPMOD ==> 5
  2. Trace: LOUIS-EXPMOD ==> 5
  2. Trace: (LOUIS-EXPMOD '15 '5 '10)
   3. Trace: (LOUIS-EXPMOD '15 '4 '10)
    4. Trace: (LOUIS-EXPMOD '15 '2 '10)
     5. Trace: (LOUIS-EXPMOD '15 '1 '10)
      6. Trace: (LOUIS-EXPMOD '15 '0 '10)
      6. Trace: LOUIS-EXPMOD ==> 1
     5. Trace: LOUIS-EXPMOD ==> 5
     5. Trace: (LOUIS-EXPMOD '15 '1 '10)
      6. Trace: (LOUIS-EXPMOD '15 '0 '10)
      6. Trace: LOUIS-EXPMOD ==> 1
     5. Trace: LOUIS-EXPMOD ==> 5
    4. Trace: LOUIS-EXPMOD ==> 5
    4. Trace: (LOUIS-EXPMOD '15 '2 '10)
     5. Trace: (LOUIS-EXPMOD '15 '1 '10)
      6. Trace: (LOUIS-EXPMOD '15 '0 '10)
      6. Trace: LOUIS-EXPMOD ==> 1
     5. Trace: LOUIS-EXPMOD ==> 5
     5. Trace: (LOUIS-EXPMOD '15 '1 '10)
      6. Trace: (LOUIS-EXPMOD '15 '0 '10)
      6. Trace: LOUIS-EXPMOD ==> 1
     5. Trace: LOUIS-EXPMOD ==> 5
    4. Trace: LOUIS-EXPMOD ==> 5
   3. Trace: LOUIS-EXPMOD ==> 5
  2. Trace: LOUIS-EXPMOD ==> 5
 1. Trace: LOUIS-EXPMOD ==> 5
```

What has happened here ? Well, we see that the double call to itself has transformed `expmod` from a linear recursion (like the one in the factorial computation) to a tree recursion (like the one in the Fibonacci computation), and thus the amount of calls grows exponentially. To understand it logically, note that we turned a single call to itself in the `evenp` branch to two calls. But this not merely multiplies the amount of calls by two – it multiplies it by two for each level, so if previously we had N calls, now we have `2^N` calls. In the case of `expmod`, instead of logarithmic runtime we get linear runtime `(2^logN = N)`.

Exercise 1.27

This is the solution of the exercise:

```
(defun full-fermat-test (n)
  (defun aux-test (a)
    (cond ((= a 1) t)
          ((/= (expmod a n n) a) nil)
          (t (aux-test (1- a)))))
  (aux-test (1- n)))
```

Using this function it is possible to test that 561, 1105, 1729, 2465, 2821, and 6601 are indeed Carmichael numbers, since they elude the test for all `a`. However, there is a slight problem worth mentioning here. On my Windows machine, with CLISP 2.41, the call `(full-fermat-test 6601)` produces a stack overflow. Let's understand why.

`full-fermat-test` is an iterative process, because `aux-test` is tail recursive. However, recall that I mentioned that, at least in CLISP, the tail recursion optimization is implemented only when functions are compiled. To make it work, I had to rewrite the test as:

```
(defun full-fermat-test (n)
  (defun aux-test (a)
    (cond ((= a 1) t)
          ((/= (expmod a n n) a) nil)
          (t (aux-test (1- a)))))
  (compile 'aux-test)
  (aux-test (1- n)))
```

Now the call `(full-fermat-test 6601)` doesn't overflow the stack. Just for fun, here is a version of the same test using a DO loop:

```
(defun full-fermat-test-loop (n)
  (do ((a 1 (+ a 1)))
      ((= a n) t)
      (when (/= (expmod a n n) a)
        (return nil))))
```

When timed against each other, these two tests (the compiled tail recursive one and the one using DO) run for about the same time, which makes sense because of their similarity.

Exercise 1.28

```
(defun expmod (base exponent m)
  (cond ((= exponent 0) 1)
        ((evenp exponent)
          (let* ( (candidate (expmod base (/ exponent 2) m))
                  (root (rem (square candidate) m)))
                (if (and (/= candidate 1) (/= candidate (1- m)) (= root 1))
                  0
                  root)))
        (t
          (rem  (* base (expmod base (- exponent 1) m))
                m))))

; Returns T from numbers that are probably prime
(defun miller-rabin-test (n)
  (let ((testnum (1+ (random (1- n)))))
    (= (expmod testnum (1- n) n) 1)))
```

Note the test in `miller-rabin-test`. It will fail when a nontrivial root of 1 modulo n is discovered, because in that case `expmod` propagates 0.