Assignment 4 Solutions

Nathan Hitchings

January 3, 2019

1 Exercise 1.16

Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does `fast-expt`. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an invariant quantity that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

1.1 Solution

```
(define (nh-fast-expt base exponent)
  (define (square x) (* x x))
  (define (iter a b n)
    (cond ((= n 0) a)
          ((even? n) (iter a (square b) (/ n 2)))
          (else (iter (* a b) b (- n 1)))))
  (iter 1 base exponent))
```

2 Exercise 1.17

The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the `expt` procedure:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

This algorithm takes a number of steps that is linear in `b`. Now suppose we include, together with addition, operations `double`, which doubles an integer, and `halve`, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to `fast-expt` that uses a logarithmic number of steps.

2.1 Solution

```
(define (fast-mult a b)
  (define (double x) (+ x x))
  (define (halve x) (if (even? x) (/ x 2) x))
  (cond ((= b 0) 0)
        ((even? b) (double (fast-mult a (halve b))))
        (else (+ a (fast-mult a (- b 1))))))
```

3 Exercise 1.18

Using the results of Exercise 1.16 and Exercise 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.

3.1 Solution

```
(define (nh-fast-mult a b)
  (define (double x) (+ x x))
  (define (halve x) (/ x 2))
  (define (iter product a b)
    (cond ((= b 0) product)
          ((even? b) (iter product (double a) (halve b)))
          (else (iter (+ product a) a (- b 1)))))
  (iter 0 a b))
```

4 Exercise 1.19

There is a clever algorithm for computing the Fibonacci numbers in a logarithmic number of steps. Recall the transformation of the state variables a and b in the `fib-iter` process of Section 1.2.2: $a \leftarrow a + b$ and $b \leftarrow a$. Call this transformation T, and observe that applying T over and over again n times, starting with 1 and 0, produces the pair $\text{Fib}(n+1)$ and $\text{Fib}(n)$. In other words, the Fibonacci numbers are produced by applying T^n, the n^{th} power of the transformation T, starting with the pair (1, 0). Now consider T to be the special case of $p = 0$ and $q = 1$ in a family of transformations T_{pq}, where T_{pq} transforms the pair (a, b) according to $a \leftarrow bq + aq + ap$ and $b \leftarrow bp + aq$. Show that if we apply such a transformation T_{pq} twice, the effect is the same as using a single transformation $T_{p'q'}$ of the same form, and compute p' and q' in terms of p and q. This gives us an explicit way to square these transformations, and thus we can compute T_n using successive squaring, as in the `fast-expt` procedure. Put this all together to complete the following procedure, which runs in a logarithmic number of steps:

```
(define (fib n) (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   -1 ; compute p'
                   -1 ; compute q'
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

4.1 Solution

```
(define (fib n) (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (define (square x) (* x x))
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
```

```
           b
           (+ (square p) (square q))
           (+ (square q) (* 2 p q))
           (/ count 2)))
(else (fib-iter (+ (* b q) (* a q) (* a p))
                (+ (* b p) (* a q))
                p
                q
                (- count 1)))))
```