「アルゴリズムとデータ構造入門」 第 4 回 資料 (*Oct. 26, 2004,* 奥乃)
http://winnie.kuis.kyoto-u.ac.jp/∼okuno/Lecture/04/IntroAlgDs/

1 Square Roots by Newton's Method

1.1 Original

```
(define (sqrt x)
  (sqrt-iter 1.0 x) )

(define (sqrt-iter guess x)
     (if (good-enough? guess x)
       guess
       (sqrt-iter (improve guess x) x) ))

(define (good-enough? guess x)
   (< (abs (- (square guess) x)) 0.001) )

(define (improve guess x)
  (average guess (/ x guess)) )

(define (average x y)
  (/ (+ x y) 2) )
```

1.2 Block Structure

```
(define (sqrt x)
  (define (good-enough? guess x)
     (< (abs (- (square guess) x)) 0.001) )
  (define (improve guess x)
    (average guess (/ x guess)) )
  (define (sqrt-iter guess x)
     (if (good-enough? guess x)
       guess
       (sqrt-iter (improve guess x)) )
  (sqrt-iter 1.0 x) )
```

1.3 Block Structure with Lexical Scoping of Internal Variables

```
(define (sqrt x)
  (define (good-enough? guess)
     (< (abs (- (square guess) x)) 0.001) )
  (define (improve guess)
    (average guess (/ x guess)) )
  (define (sqrt-iter guess)
     (if (good-enough? guess)
       guess
       (sqrt-iter (improve guess)) )
  (sqrt-iter 1.0) )
```

2 Factorial — Linear Recursion

2.1 Recursion

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1))) ))
```

2.2 Iteration

```
(define (factorial n)
  (fact-iter 1 1 n) )

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count) ))
```

2.3 Block Structure for Iteration

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
                   (+ counter 1) )))
  (iter 1 1) )
```

3 Fibonacci Function — Tree Recursion

3.1 Recursion

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2)) ))))
```

3.2 Iteration

```
(define (fib n)
  (fib-iter 1 0 n) )

(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1)) ))
```

4 Counting Change

```
(define (count-change amount)
  (cc amount 5) )

(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount (- kinds-of-coins 1))
                 (cc (- amount (first-denomination kinds-of-coins))
                     kinds-of-coins )))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50) ))
```

4.1 Exponentiation

4.2 Naïve Recursion

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1))) ))
```

4.3 Iteration

```
(define (expt b n)
  (expt-iter b n 1) )

(define (expt-iter b counter product)
  (if (= counter 0)
      product
      (expt-iter b
                 (- counter 1)
                 (* b product) )))
```

4.4 Divide and Conquor

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ b 2))))
        (else (* b (fast-expt b (- n 1)))) ))

(define (even? n)
  (= (remainder n 2) 0) )
```

5 Greatest Common Divisors

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b)) ))
```

6 Testing for Primality

6.1 Original

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

(define (find-devisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1))) ))

(define (divides? a b)
  (= (remainder b a) 0) )

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

6.2 HGO's Modification

```
(define (smallest-divisor n)
  (find-divisor n 3) )

(define (find-devisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 2))) ))

(define (divides? a b)
  (= (remainder b a) 0) )

(define (prime? n)
  (if (even? n)
      2
      (= n (smallest-divisor n)) ))
```

6.3 The Fermat Test based on Fermat's Little Theorem

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m)) m) )
        (else
         (remainder (* base (expmod base (- exp 1) m)) m) )))

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

(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false) ))
```