Practice 1

Practice 1.3

```
(define (fun x y z)
    (cond
        ((and (> x z) (> y z)) (+ (* x x) (* y y)))
        ((and (> x y) (> (z y))) (+ (* x x) (* z z)))
        (else (+ (* y y) (* z z)))))
```

Practice 1.4

$a + |b|$

Practice 1.5

- 인자 먼저 계산법(applicative order evaluation):
  인자를 각각 먼저 eval하기 때문에 무한루프(함수 p)가 실행된다.
- 정의대로 계산법(normal-order evaluation): 인자의 식들을 eval하지 않고 일단 함수에 적용시키기 때문에 0이 먼저 비교가 되어 0이 리턴된다.

Practice 1.6

special form if의 경우 predicate이 참이라면 then-clause만 eval하고 거짓이라면 else-clause를 eval하기 때문에 루프가 종결될 수 있지만, new-if의 경우 new-if에 인자를 적용시킬 때 각 인자들을 모두 eval한 후 적용시키기 때문에 무한하게 프로시저가 호출된다.

Practice 1.7

```
(define (good-enough??? guess x)
    (= (improve guess x) guess))
```

Pratice 1.8

```
(define (sqrt3-iter guess x)
    (if (good-enough? guess x)
    guess
    (sqrt3-iter (improve guess x) x)))

(define (improve guess x)
    (/ (+ (* 2 guess) (/ x (sqr guess))) 3))

(define (good-enough? guess x)
    (= (improve guess x) guess))

(define (sqrt3 x)
    (sqrt3-iter 1.0 x))
```

Practice 1.9

- 첫번째 : 되도는(recursive) 프로세스
- 두번째 : 반복되는(iterative) 프로세스

Practice 1.10

```
(define (A x y)
    (cond ((= y 0) 0)
          ((= x 0) (* 2 y))
          ((= y 1) 2)
          (else (A (- x 1) (A x (- y 1))))))
```

- (A 1 10) = 1024 (2^{10})
- (A 2 4) = 65536 $(2^{2^{2^2}})$

- (A 3 3) = 65534
- $f(n) = 2n$
- $g(n)$ = if $n = 0$, 0 else 2^n
- $h(n) = 2^{h(n-1)}$, where $h(1) = 2$, $h(0) = 0$

Practice 1.11

- Recursive Process

```
(define (fr n)
    (if (< n 3)
        n
        (+ (fr (- n 1)) (* 2 (fr (- n 2))) (* 3 (fr (- n
3))))))
```

- Iterative Process

```
(define (fi n)
    (define (fi-iter cnt r1 r2 r3)
        (cond ((< n 3) n)
              ((= cnt n) (+ r1 (* 2 r2) (* 3 r3)))
              (else (fi-iter (+ 1 cnt)
                (+ r1 (* 2 r2) (* 3 r3)) r1 r2))))
    (fi-iter 3 2 1 0))
```

Pracitce 1.12

```
(define (pascal row col)
    (if (or (= col 1) (= col row))
        1
        (+ (pascal (- row 1) (- col 1)) (pascal (- row 1)
col))))
```

Practice 1.13

- 1
  - 2번에서, $Fib(n) = \frac{(\phi^n - \psi^n)}{\sqrt{5}}$이므로, $Fib(n) - f(n) = -g(n) = -\frac{\psi^n}{\sqrt{5}}$인데, $|\psi| < 1$이므로 $|Fib(n) - f(n)| < 0.5$이고 따라서 $Fib(n)$은 $\frac{\phi^n}{\sqrt{5}}$에 가장 가까운 정수이다.
- 2
  - $f(n) = \frac{(\phi)^n}{\sqrt{5}}$, $g(n) = \frac{(\psi)^n}{\sqrt{5}}$ 이라하고, $f(n) + g(n) = h(n)$이라했을 때, ϕ, ψ는 $x^2 - x - 1 = 0$의 두 근이므로 $n >= 2$에서 $h(n) = h(n-1) + h(n-2)$이다. $h(0) = 0$, $h(1) = 1$이므로 $Fib(n) = h(n)$이다.

Practice 1.14

- Time Complexity: $\Theta(n^5)$
- Space Complexity: $\Theta(n)$

Practice 1.15

a. 5 times
b. $\Theta(\log_3 a)$

Practice 1.16

```
(define (fast-expt-iter b n)
    (define (iter a k pow)
        (cond ((= pow 0) a)
              ((even? pow) (iter a (sqr k) (/ pow 2)))
              (else (iter (* a k) k (- pow 1)))))
    (iter 1 b n))
```

Practice 1.17

```
(define (fast-mul a b)
    (cond ((= b 0) 0)
          ((even? b) (double (fast-mul a (halve b))))
          (else (+ a (fast-mul a (- b 1))))))
```

Practice 1.18

```
(define (double x) (+ x x))
(define (halve x) (/ x 2))

(define (fast-mul-iter a b)
    (define (iter result i j)
        (cond ((= j 0) result)
              ((even? j) (iter result (double i) (halve
j)))
              (else (iter (+ result i) i (- j 1)))))
    (iter 0 a b))
```

Practice 1.19

a, b에 대해 $T_{pq}(a) = a' = bq + aq + ap, T_{pq}(b) = b' = bp + aq$라 하면,

$$T_{pq}{}^2(a) = b'q + a'q + a'p = b(q^2 + 2pq) + a(q^2 + 2pq) + a(p^2 + q^2)$$
$$T_{pq}{}^2(b) = b'p + a'q = b(p^2 + q^2) + a(q^2 + 2pq)$$

이며, 이식으로부터

$$p' = p^2 + q^2,\ q' = q^2 + 2pq$$

를 도출할 수 있다.

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

Practice 1.20

```scheme
(gcd 206 40) ;with normal-order application
(gcd 40 (remainder 206 40))
if (= (remainder 206 40) 0) ;v
(gcd (remainder 206 40) (remainder 40 (remainder (206
40))))
if (= (remainder 40 (remainder (206 40))) 0) ;vv
(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) ;vvvv
(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 (= (remainder (remainder 40 (remainder (206 40)))
(remainder ((remainder 206 40)) (remainder 40 (remainder
(206 40))))) 0) ;조건식 #t, vvvvvvv
(remainder ((remainder 206 40)) (remainder 40 (remainder
(206 40)))) ;결과, vvvv
```

주석으로 v된 부분이 remainder가 실행되는 표시이며 총 18번 실행된다.

Practice 1.21

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

(define (find-divisor n test-divisor)
    (cond ((> (sqr 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))
```

- 199 : 199
- 1999 : 1999
- 19999 : 7

Practice 1.22

% 주의사항: runtime 프로시저 대신 racket에서 current-inexact-milliseconds 사용. 자세한 내용은 racket docs에서.

- (search-for-primes), (iter) source code

```scheme
(define (search-for-primes start end cnt)
    (cond ((= cnt 0) (display "All Found")) ;cnt만큼 다 찾을 시
          ((= start end) (display "Not enough range")) ;range 안에 cnt만큼의 소수가 없을 시
          ((prime? start) (display start) (display " ")
           (search-for-primes (+ start 2) end (- cnt 1))) ;띄어쓰기 단위로 소수 출력
          (else (search-for-primes (+ start 2) end cnt))))

(define (iter a b c f) (f a) (f b) (f c)) ;f를 a, b, c에 적용
```

- 위 코드로 1000, 10000, 100000, 1000000 근처 3개의 소수들을 구해 연산 시간을 측정해 봤지만 숫자가 너무 작아 결과가 나오지 않으므로 그다음 4범위로 측정한 결과

> $10000019 *** 0.011962890625$
> $10000079 *** 0.010009765625$
> $10000103 *** 0.010009765625$
> $100000007 *** 0.05712890625$
> $100000037 *** 0.033203125$
> $100000039 *** 0.033935546875$
> $1000000007 *** 0.10498046875$
> $1000000009 *** 0.10595703125$
> $1000000021 *** 0.1259765625$
> $10000000019 *** 0.344970703125$
> $10000000033 *** 0.342041015625$
> $10000000061 *** 0.346923828125$

대체로 $\sqrt{n}$에 비례해서 연산시간이 증가함을 알 수 있다.

Practice 1.23

```
(define (next n) (if (= n 2) 3 (+ n 2)))
```

next 프로시저를 적용시켜 timed-prime-test2를 만들어 다음 코드를 실행시켰다.

```
(define (iter2 f g a b c) (f a) (g a) (f b) (g b) (f c) (g c))
(iter2 timed-prime-test timed-prime-test2 10000019 10000079 10000103)
(iter2 timed-prime-test timed-prime-test2 100000007 100000037 100000039)
(iter2 timed-prime-test timed-prime-test2 1000000007 1000000009 1000000021)
(iter2 timed-prime-test timed-prime-test2 10000000019 10000000033 10000000061)
```

그 결과,

> $10000019 * * * 0.011962890625$
> $10000019 * * * 0.006103515625$
> $10000079 * * * 0.010986328125$
> $10000079 * * * 0.005859375$
> $10000103 * * * 0.010986328125$
> $10000103 * * * 0.005859375$
> $100000007 * * * 0.072998046875$
> $100000007 * * * 0.01904296875$
> $100000037 * * * 0.0341796875$
> $100000037 * * * 0.01904296875$
> $100000039 * * * 0.0341796875$
> $100000039 * * * 0.01904296875$

$$1000000007 *** 0.10693359375$$
$$1000000007 *** 0.057861328125$$
$$1000000009 *** 0.120849609375$$
$$1000000009 *** 0.05908203125$$
$$1000000021 *** 0.10595703125$$
$$1000000021 *** 0.058837890625$$
$$10000000019 *** 0.33984375$$
$$10000000019 *** 0.198974609375$$
$$10000000033 *** 0.366943359375$$
$$10000000033 *** 0.18603515625$$
$$10000000061 *** 0.3427734375$$
$$10000000061 *** 0.197998046875$$

즉 대략 2배 정도이지만 2배에는 조금 못미치는 성능을 낸다는 것을 알 수 있다. 이는 비록 input의 갯수는 줄었지만 next 프로시저에서 if연산을 하는 시간이 더 늘었기 때문으로 추정된다.(그럼에도 if 연산의 속도가 빨라서인지 거의 두배, 또는 그 이상의 성능을 내준다...)

Practice 1.24

```
;To implement Fermat test
(define (expmod base exp m) ;밑 지수 법
  (cond ((= exp 0) 1)
        ((even? exp) (remainder (sqr (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)))))

;prime procedure with Fermat test
(define (fast-prime? n times)
```

```
    (cond ((= times 0) #t)
          ((fermat-test n) (fast-prime? n (- times 1)))
          (else #f)))
(define (prime? n) (fast-prime? n 100))
```

- Fermat test에서 expmod 프로시저는 a^n을 n으로 나눈 나머지는, $a^{\frac{n}{2}}$을 n으로 나눈 나머지의 제곱을 n으로 나눈 값과 같다는 사실을 이용한 알고리즘이다. 과거 fast-expt처럼 시행단계가 $\Theta(\log n))$으로 증가한다는 것을 보장할 수 있다. 이를 timed-prime-test에 적용시켜 다음과 같은 결과를 얻었다.

$$
\begin{aligned}
&1009 *** 0.02099609375\\
&1013 *** 0.01904296875\\
&1019 *** 0.01904296875\\
&10007 *** 0.02197265625\\
&10009 *** 0.02099609375\\
&10037 *** 0.0239257812 5\\
&100003 *** 0.02783203125\\
&100019 *** 0.02685546875\\
&100043 *** 0.02685546875\\
&10000019 *** 0.036865234375\\
&10000079 *** 0.038818359375\\
&10000103 *** 0.037841796875\\
&100000007 *** 0.0439453125\\
&100000037 *** 0.0439453125\\
&100000039 *** 0.072998046875\\
&1000000007 *** 0.050048828125\\
&1000000009 *** 0.047119140625\\
&1000000021 *** 0.047119140625
\end{aligned}
$$

- 이 결과로부터 대략적으로 n에 대해 $\log n$에 비례하여 증가한다는 것을 확인할 수 있다.

- 더 큰 숫자(예컨대 100자리가 넘어갈 정도의 숫자들)에 대해서는 로그보다는 좀더 증가폭이 크다고 한다. 이는 큰 숫자에 대한 primitive operations가 작은 숫자들과는 달리 constant하지 않기 때문이다.

Practice 1.25

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

이 프로시저를 사용하면 알고리즘이 작동은 하지만 연산 시간이 굉장히 길어진다. 똑같은 프로시저임에도 고작 5자리 수 100043을 소수 판별 해내는데 5849.830810546875ms나 걸리는 것을 확인할 수 있었다. 이는 지수 계산 자체는 빠를지라도 나머지를 구하는 연산이 숫자가 커지면 굉장히 느려지기 때문이다.

Practice 1.26

```
(* (expmod base (/ exp 2) m) (expmod base (/ exp 2) m))
```

코드에서만 봐도 알 수 있듯이 매번 expmod를 각각 호출하기 때문에 결국 지수 exp만큼 계산 단계를 거치게 된다.

Practice 1.27

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

```
(define (test start n)
    (cond ((= start n) (display "end") (newline))
          ((= (expmod start n n) start)
            (display start) (newline) (test (+ start 1)
n))
          (else (test (+ start 1) n))))
```

위 코드로 소수를 테스트하면 반드시 그 소수보다 작은 모든 정수가 나온다. 그리고 대부분의 합성수들은 대부분의 수들이 빠져서 나오게 된다. 하지만 561같은 카마이클 수를 테스트해보면 마치 소수처럼 그 수보다 작은 모든 정수가 나온다.

Practice 1.28

```
(define (expmod base exp m);밑 지수 법
    (define (check k)
        (define trivial? (or (= k 1) (= k (- m 1))))
        (if (and (= (remainder (sqr k) m) 1) (not
trivial?)) 0 k))
    (cond ((= exp 0) 1)
          ((even? exp) (remainder (sqr (check (expmod
base (/ exp 2) m))) m))
          (else (remainder (* base (expmod base (- exp 1)
m)) m))))

(define (fermat-test n a)
    (define result (expmod a (- n 1) n))
    (if (= result 0) #f (= result 1)))
```

- expmod 프로시저 안의 check 프로시저는 k(이것은 프로시저 본체에서 sqr의 대상이 될 값이다)가 1 또는 n - 1이 아니며 $k^2 \mod m$이 1인지 확인하여 맞다면 0을, 아니라면 k를 그대로 반환한다. 이 알고리즘은 만약 x가

소수라면 $x^2 \equiv 1 \pmod{n}$의 근은 반드시 $x \equiv 1 \pmod{m}$ 또는 $x \equiv -1 \pmod{m}$라는 것을 이용한다. 이 성질은 소수인 x에만 해당되기 때문에(이는 모듈러의 성질로 증명 가능) 만약 x가 $x^2 \equiv 1 \pmod{m}$의 비자명한 근($x \equiv 1 \pmod{m}$또는 $x \equiv -1 \pmod{m}$)이라면 이는 합성수라는 것이다. 그래서 매번 sqr의 대상인 녀석에 대해 check를 해주는 것이다.

- 밀러-라빈 판별법에 대해 위키백과에서 이론적 배경을 검색해보면 뭔가 SICP에서 제시하는 알고리즘과 상충하는 느낌을 받을 수 있다. 하지만 근본적인 아이디어는 비슷하다. n이 2가 아닌 소수라 할 때, $n-1=2^s d$(s는 정수, d는 홀수)라 표현한다. 즉, $n-1$을 홀수를 만날때까지 2로 나눈 것이다. 여기서 이제 $a < n$인 정수 a에 대해 다음 식 중 한가지가 성립한다. 이는 페르마 소정리에서 유도된다.

> - $a^d \equiv 1 \pmod{n}$
> - $a^{2^r d} \equiv -1 \pmod{n}$ for some $0 \leq r \leq s-1$

- 따라서 대우명제로 다음 두 식이 성립하면 n은 합성수이다.

> - $a^d \not\equiv 1 \pmod{n}$
> - $a^{2^r d} \not\equiv -1 \pmod{n}$ for all $0 \leq r \leq s-1$

- 위 식이 성립하지 않으면 n은 높은 확률로 소수이다고 말할 수 있다. 이 이론적 배경을 살펴보면 우리의 코드에서 check를 사실 재귀를 돌다가 첫 홀수에 도달했을때 그 홀수까지만 체크해보면 된다는 것을 알 수 있지만(물론 check 프로시저의 수정도 필요) SICP에서는 애초에 정확한 밀러-라빈 판별법과는 조금 다르 컨셉을 이용했기 때문에 애초에 조금 다른 알고리즘이라고 볼 수 있다.

Practice 1.29

```
(define (sigma x_n start end)
    (if (> start end)
        0
```

```
        (+ (x_n start) (sigma x_n (+ start 1) end))))

(define (integral f a b n)
    (define h (/ (- b a) n))
    (define (x_n n)
        (if (even? n)
            (* 2 (f (+ a (* n h))))
            (* 4 (f (+ a (* n h))))))
    (* (- (sigma x_n 0 n) (+ a b)) (/ h 3)))

(define (cube x) (* x x x))

(integral cube 0 1 100); 1/4
(integral cube 0 1 1000); 1/4
```

Practice 1.30

```
(define (sum term a next b)
    (define (iter a result)
        (if (> a b)
            result
            (iter (next a) (+ (term a) result))))
    (iter (a 0)))
```

Practice 1.31

a. Recursive Process

```
(define (product f a b)
    (if (> a b)
        1
        (* (f a) (product f (+ a 1) b))))
```

```
(define (func n)
    (if (even? n)
        (/ n (- n 1))
        (/ (- n 1) n)))

(exact->inexact (* 4 (product func 3
100000)));3.141608361592331 (10000부터는 시간 오지게 걸린다...)
```

b.Iterative Process

```
(define (product_iter f a b )
    (define (iter i result)
        (if (> i b)
            result
            (iter (+ i 1) (* (f i) result))))
    (iter a 1))

(define (func n)
    (if (even? n)
        (/ n (- n 1))
        (/ (- n 1) n)))

(exact->inexact (* 4 (product_iter func 3
100000)));3.141608361592331 (당연하지만 똑같이 나온다)
```

Practice 1.32

a. Recursive Process

```
(define (accumulate combiner null-value term a next b)
    (if (> a b)
```

```
        null-value
        (combiner (term a)
                  (accumulate combiner null-value term
(next a) next b))))

(define (sum term a next b)
    (accumulate + 0 term a next b))
(define (product term a next b)
    (accumulate * 1 term a next b))
```

b. Iterative Process

```
(define (accumulate_iter combiner null-value term a next
b)
    (define (iter a result)
        (if (> a b)
            result
            (iter (next a) (combiner result (term a)))))
    (iter a null-value))
```

Practice 1.33

- filtered-accumulate procedure

```
(define (filtered-accumulate filter combiner null-value
term a next b)
    (cond ((> a b) null-value)
          ((filter a) (combiner (term a) (filtered-
accumulate filter combiner null-value term (next a) next
b)))
          (else (combiner null-value (filtered-accumulate
```

```
filter combiner null-value term (next a) next b)))))
```

a.

```
(define (sum-of-squared-primes a b)
    (define (inc n) (+ 1 n))
    (filtered-accumulate prime? + 0 sqr a inc b))
```

b.

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

(define (product-of-relatively-prime-numbers n)
    (define (relative-prime? b) (= (gcd n b) 1))
    (define (inc x) (+ 1 x))
    (define (give x) x)
    (filtered-accumulate relative-prime? * 1 give 1 inc
n))
```

Practice 1.34

- (f f)는 결국 (f 2)를 호출하고 이는 다시 (2 2)를 호출하지만 2는 applicable한 인자가 아니기 때문에 오류가 발생한다.

Practice 1.35

- $x = 1 + \frac{1}{x}$의 근은 결국 $x^2 - x - 1 = 0$의 근과 같고 이는 결국 황금비이다.

```
(fixed-point (lambda (x) (+ 1 (/ 1 x)))
1.0);1.6180327868852458
```

Practice 1.36

- Without average damping

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
    (define (close-enough? v1 v2)
        (< (abs (- v1 v2)) tolerance))
    (define (try guess)
        (display guess)
        (newline)
        (let ((next (f guess)))
            (if (close-enough? guess next)
                next
                (try next))))
    (try first-guess))

(fixed-point (lambda (x) (/ (log 1000) (log x))) 4.0)
```

4.0
4.9828921423310435
4.301189432497896
4.734933901055578
4.442378437719526
4.632377941509958
4.505830646780212
4.588735606875766
4.533824356566501

```
4.56993352418142
4.546075272637246
4.561789745175654
4.55141783665413
4.558254212070625
4.553744140202578
4.556717747893265
4.554756404545319
4.5560497413912975
4.5551967522618035
4.555759257615811
4.555388284933278
4.555632929754932
4.555471588998784
4.555577989320218
4.555507819903776
4.555554095154945
4.555523577416557
4.555543703263474
4.555530430629037
4.555539183677709
```

- With average damping

```
(define (fixed-point f first-guess)
    (define (close-enough? v1 v2)
        (< (abs (- v1 v2)) tolerance))
    (define average (lambda (x y) (/ (+ x y) 2.0)))
    (define (try guess)
        (display guess)
        (newline)
        (let ((next (average (f guess) guess)))
            (if (close-enough? guess next)
```

```
                next
                (try next))))
    (try first-guess))
```

```
4.0
4.491446071165521
4.544974650975552
4.553746974742814
4.555231425802502
4.555483906560562
4.5555268862194875
4.5555342036887705
```

Practice 1.37

- Iterative Process

```
(define (cont-frac-iter n d k)
    (define (iter i result)
        (if (= i 0)
            result
            (iter (- i 1) (/ (n i) (+ (d i) result)))))
    (iter k 0))

(cont-frac-iter (lambda (i) 1.0)
           (lambda (i) 1.0)
           12) ;0.6180257510729613
(/ 1 1.6180327868852458) ;0.6180344478216819
```

- Recursive Process

```scheme
(define (cont-frac n d k)
    (define (recur s)
        (if (= s k)
            (/ (n s) (d s))
            (/ (n s) (+ (d s) (recur (+ s 1))))))
    (recur 1))
```

Practice 1.38

```scheme
(define euler-e
    (+ 2 (cont-frac (lambda (x) 1.0)
                (lambda (x)
                    (if (= 2 (remainder x 3))
                        (* 2 (+ 1 (quotient x 3)))
                        1))
                15))) ;2.718281828470584
```

Practice 1.39

```scheme
(define (tan-cf x k)
    (cont-frac
        (lambda (i) (if (= i 1) x (- (sqr x))))
        (lambda (i) (- (* 2 i) 1))
        k))
```

Practice 1.40

```scheme
(define (cubic a b c)
    (define cube (lambda (i) (* i i i)))
    (lambda (x) (+ (cube x) (* a (sqr x)) (* b x) c)))
```

Practice 1.41

```scheme
(define (double p)
    (lambda (x) (p (p x))))

(define (inc i) (+ i 1))
(((double (double double)) inc) 5) ;21
```

Practice 1.42

```scheme
(define (compose f g)
    (lambda (x) (f (g x))))
```

Practice 1.43

```scheme
(define (compose f g)
    (lambda (x) (f (g x))))

(define (repeated f n)
    (if (= n 1) f (compose f (repeated f (- n 1)))))
```

Practice 1.44

```scheme
(define (smooth f)
    (lambda (x) (/ (+ (f (- x dx)) (f x) (f (+ x dx)))
3.0)))
(define dx 0.00001)

(define (n-fold-smoothed-function f n)
    ((repeated smooth n) f))
```

Practice 1.45

```scheme
;fixed-point
(define (fixed-point f first-guess)
    (define (close-enough? v1 v2)
        (< (abs (- v1 v2)) tolerance))
    (define (try guess)
        (let ((next (f guess)))
            (if (close-enough? guess next)
                next
                (try next))))
    (try first-guess))
(define tolerance 0.00001)

;repeated
(define (compose f g)
    (lambda (x) (f (g x))))
(define (repeated f n)
    (if (= n 1) f (compose f (repeated f (- n 1)))))

;average-damp
(define (average-damp f)
    (lambda (x) (/ (+ x (f x)) 2)))
(define (n-folded-damped f n)
    (lambda (x) (((repeated average-damp n) f) x)))

(define (n-th-root x n)
    (fixed-point (n-folded-damped (lambda (y) (/ x (expt
y (- n 1)))) 2) 1.0))
```

실험적으로, $n < 2^k$를 만족시키는 최소의 k번만큼 damping을 해줘야 한다는 것을 알 수 있었고 나는 이것을 획기적인 방법으로 증명했지만 여백이 작아 이곳

에는 적지 못할 것 같다.

Practice 1.46

```
(define (iterative-improve improvingf good-enough?)
    (define (recur guess)
        (if (good-enough? guess)
            (improvingf guess)
            (recur (improvingf guess))))
    recur)

(define (fixed-point f first-guess)
    (define tolerance 0.00001)
    (define (improve x) (f x))
    (define (close-enough? v) (< (abs (- v (improve v)))
tolerance))
    ((iterative-improve improve close-enough?) first-
guess))

(define (sqrt- x)
    (define tolerance 0.00001)
    (define first-guess 1.0)
    (define (improve guess) (/ (+ guess (/ x guess)) 2))
    (define (close-enough? v) (< (abs (- (sqr v) x))
tolerance))
    ((iterative-improve improve close-enough?) first-
guess))
```