20180203_정리

노트북: 컴퓨터 프로그램의 구조와 해석
만든 날짜: 2017-12-15 오후 11:51 **수정한 날짜:** 2018-02-03 오후 11:45
작성자: haidn1994@gmail.com
태그: 문제풀이
URL: https://github.com/mathias/sicp/blob/master/exercise-1.11.rkt

---

- 1.1 연습문제

아래에 여러 식이 있다. 실행기가 찍어내는 값은 무엇인가? 아래에 적힌 식의 값을 차례대로 구한다고 하자.

```
10 -> 10

(+ 5 3 4) -> 12

(- 9 1) -> 8

(/ 6 2) -> 3

(+ (* 2 4) (- 4 6)) -> (+ 8 -2) -> -16

(define a 3) -> a = 3

(define b (+ a 1)) -> b = 4

(+ a b (* a b)) -> (+ 3 4 (* 3 4)) -> (+ 3 4 12) -> 19

(if (and (> b a) (< b (* a b)))
    b
    a)

-> (if (and #t #t) b a) -> 4

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))
-> 첫번째 조건: #f -> 두번째 조건 #t -> 세번째 조건은 else여서 자신보다 앞쪽에 #t인 술어가 없어야 실행된다. 따라서
답은 (+ 6 7 a) -> 16

(+ 2 (if (> b a) b a))
-> (+ 2 (if #t b a)) -> (+ 2 4) -> 6

(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))

(* 4 (+ a 1)) -> 16
```

-> 별로 어려운 문제는 아니고, 그냥 Lisp의 문법에 익숙해지라고 만든 문법이므로 그냥 풀면된다. 단, 처음에는 하지만 공부를 해보니 다음과 같은 점을 주의하면 좋겠다.

- Lisp는 함수형 프로그래밍 언어다. 따라서 명시적으로 상수를 지정(변수 지정 불가능)하고, (그리고 타입도 지정하고) state를 바꾸면서 하는 것은 꼬리재귀 형태로 만든 함수정도 밖에는 불가능하므로 주의한다.
  즉, 그 스택안에서는 상태가 불변이라는 것에 유의한다.
- 문법이 생소하므로 주의한다. 일단 1-1문제를 푸는 시점에서 익숙해져야 할 문법은 다음과 같다.
  (참고로, 아래에 나오는 모든 문법은 special form이다. special form으로 만들 수 밖에 없었던 이유가 있으므로 그 점을 하나씩 짚어 보자.)
  - (define <name> <object>) - object-like define:
    안쪽에 있지 않은 define은 제일 바깥쪽 환경(environment)에 해당한다. 그리고 이름에 객체를 묶고(binding) 그 이름은 해당하는 환경에서만 접근할 수 있다는 점에 주의하자.
  - (define (<name> <parameters...>) <body>) - procedure-like define:
    환경에 대한 설명은 위쪽을 참고하자. 그리고, <body>부분은 어지간하면 들여쓰기를 해주자.
    블록구조(네임스페이스를 제한 하여 바깥쪽에서 접근하지 않아도 되는 이름은 해당 프로시저 이름을 네

임스페이스로 삼아서 그 바깥쪽 네임스페이스와 안쪽 네임스페이스를 구분하는 기법! Algol60에서 따왔다.)를 적극적으로 활용하자.
- (cond (<pre1> <exp1>)
  (<pre2> <exp2>)
  ...
  (<pren> <expn>)):
  조건에 따라서 분기가 발생하는 구문인데, 앞서서 설명할 if와 분명한 차이점이 있다. 대표적으로 2가지 차이점이 있는데 이를 짚고 넘어가자.
  - 첫번째, Lisp는 기본적으로 applicative order를 사용하는데 비해서, if는 실행기(interpreter)에서 특수하게 취급하여 이와는 다르게 실행된다.
    cond의 경우 술어를 동시에 평가하고, 맞는 것을 취사 선택해서 프로세스를 이어가려고 한다. 만약 여러개의 술어가 전부 맞다면, 동시에 여러개의 실행문을 실행하려 들 것이다.
    하지만, if는 다르다. 일단 술어가 맞는 지 살펴보고, 맞으면 왼쪽에 있는것(또는 앞쪽에 있는것) 틀리면 오른쪽에 있는것(또는 뒤쪽에 있는것)을 실행할 든다는 차이점이 있다.

    applicative order와 normal order에 대해서 자세하게 알아보고(그리고 substitution model)에 대해서도 자세하게 알아보자. 어떤 차이가 있는지 자세하게 알아야 한다.
    여기서 알아야 할 교훈은 단순하게 생각하면 문법이 달라도 같아 보이는 동작이 실행기의 구현때문에 프로그램의 실행이 달라질수 있다는 점을 주의해야 한다는 점이다.
    즉, 문법에 따른 실행의 차이를 정확하게 인식하고 적용할 수 있어야 한다. 참고: https://courses.cs.washington.edu/courses/cse505/99au/functional/applicative-normal.pdf
  - 두번째, Lisp에서 if를 사용하면 여러개의 식을 나열해서 사용할수가 없다. 즉, 우리가 명령형 언어를 사용할 때 처럼 문(statement)의 나열을 사용하고 싶다면
    lambda를 사용하던지(아니면 let을 사용하던지), cond로 바꿔서 여러개의 실행문을 나열하는 수밖에는 없을 것 같다. 이것도 문법에 따른 차이라고 생각하면 좋겠다.
  - 그리고 맨 마지막에 (else <expression>)을 사용하면 모든 술어를 전부 평가하고 #t인 경우가 없으면, 그 때 (else <expression>)을 실행한다.
- (if <pre> <consequent-expression> <alternative>) - 평가해야 하는 술어가 1개밖에 없을때, 그리고 그에 따른 실행해야 하는 식이 각각 1개씩만 있다면,
  이 문법을 사용하는 것이 좋다. 그리고 이런 문법이 special form으로 만들수 밖에 없는 이유가 2가지나 있고, 위에 잘 설명해놓았으므로 주의한다.
  (즉, special form과 일반적인 procedure의 차이점이라고 보면 된다.)
- 잘 모르겠다면 다음 링크를 참고하자: https://stackoverflow.com/questions/1171252/whats-the-explanation-for-exercise-1-6-in-sicp
- 술어(predicate)가 무엇인지 알겠나? 술어를 평가하면 참(#t) 또는 거짓(#f)이 나온다.
  하지만 꼭 이런 식 말고도 다른 복잡한 술어들의 조합을 표현하고 싶을 때가 있을 것인데, 이럴 때 쓰라고 만들어 놓은 special form(논리 연산자(비트 논리 연산자 아님!)이 존재한다.
  - (and <pre1> <pre2> ... <pren>) -> 모든 술어가 참일때만 참을 반환하고 아니면 거짓을 반환한다.
  - (or <pre1> <pre2> ... <pren>) -> 모든 술어 중 하나라도 참이면 참을 반환하고, 모든 술어가 거짓이면 거짓을 반환한다.
  - (not <pre>) -> 단항 연산자로, 술어가 참이면 거짓, 거짓이면 참을 반환한다. 여집합 등을 생각하고 활용하기에 알맞다.
- 단항 연산자도 잊지 말아야 한다. -의 경우, 인자가 하나만 있으면 단항 연산자로 평가된다. 숫자 데이터가 들어오면 1의 보수를 반환한다는 사실을 잊지 말자.
- 그리고 보통 우리가 많이 접하는 프로그래밍 언어는(C, C++, C#, JAVA, JS등) 대체로 위에서 아래로 **순차적**으로 실행되지만, Lisp는 다르다.
  Lisp는 해당 프로시저가 실행되면, 안쪽에서 바깥쪽으로 풀려나오는 듯한 모양새를 가지기 때문에 다소 생소하다. 순서에 유의하자.
  현대적인 Lisp인 clojure의 경우에는 ->> 같은 매크로로 일반적은 프로그래밍 언어와 실행 방향을 같게 해줄 수 있지만 Shceme도 그런게 있는지는 잘 모르겠다.

- 연습문제 1.2

아래 식을 '앞가지 쓰기' 꼴로 고쳐보자.

```
(/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5)))))
   (* 3 (- 6 2) (- 2 7)))
```

-> 코드나 수식을 표현할 때, prefix, infix, postfix중 하나를 골라서 표현한다는 사실을 알고 있나?(효율성 때문에 내부적으로 infix를 사용해서 처리하는 경우는 없다.)
Lisp는 전위 표기법(prefix)를 사용해서 자신의 코드를 표현하는 언어인데, 여기에 함수형 프로그래밍 언어라는 특성이 합쳐져서, 굉장히 독특한 문법을 가지고 있다.

역시나 이 문제도 그런 표기법에 익숙해지라는 뜻에서 만든 문제인것 같은데, 여기서도 다른 언어에 적용할수 있는 배울만한 점이 하나 있다:

사람의 경우 수식을 표현할 때, 보통 중위 표기법(infix)을 사용하는데 이는 사람이 이해하기에는 편하지만 컴퓨터에서 사용하기에는 굉장히 복잡하다.
왜냐하면, 적절한 자료구조를 사용하기 어렵기 때문이다. 컴퓨터는 보통 tree를 사용해서 수식을 표현하고 처리하는데, 보통 스택과 배열을 사용해서 트리를 구현한다.
(중위 표기법은 tree로 처리하기 너무나 곤란하다! 즉, 중위 표기법은 사람의 편의를 위해서 존재한다는 것을 알아야 한다.)
따라서 결국은 중위 표기법으로 표현된 코드 또는 수식을 후위 표기법(postfix)으로 바꾸거나, 전위 표기법으로 변환하고, 이를 트리로 가공하는 과정을 거친다.
하지만 Lisp는 처음부터 전위 표기법을 사용하기 때문에 그럴 필요가 전혀 없다는 사실을 알 수 있다.

그리고 이렇게 트리로 표현된 수식을 재귀함수(되도는 프로세스)로 하나씩 탐색하면서 실행하기에도 적합하다는 사실을 기억하자.
이 두가지 사실은 나중에 컴파일러를 공부하거나 구현할 때 유용하게 사용할 수 있는 지식이므로 잘 기억하자.

- 1.3 연습문제

세 숫자를 인자로 받아 그 가운데 큰 숫자 두개를 제곱한 다음, 그 두 값을 덧셈하여 내놓는 프로시저를 정의하라.

-> 사실 이 문제를 현업에서 활용해야 하는 상황이 온다면, 이게 정답이 아닐까 싶다: 세 인자를 **리스트**로 받고, 이를 오름차순으로 정렬한다.
맨 앞에 있는 원소와, 그 다음에 있는 원소를 고른다. 그리고 그 숫자 2개를 제곱하고, 그 두값을 덧셈하여 내놓는 프로시저를 만들어서 호출한다. 하지만 그 방법은 이 연습문제에서 원하는 솔루션은 아닌것 같다.

아마 연습문제에서 원하는 솔루션은 술어를 활용하는 방식에 익숙해지라고, 그 방법을 연습하라고 만든 문제가 아닐까 싶은데, 강사님이 만든 코드가 이 연습문제의 해답에 가까울 것 같다.

```
#lang racket
(define (big-two-sum x y z)
     (sum (sqr (first x y z))
          (sqr (second x y z))))

(define (>= x y)
     (not (< x y)))

(define (<= x y)
     (not (> x y)))

(define (first x y z)
     (cond ((and (>= x y) (>= x z)) x)
           ((and (>= y x) (>= y z)) y)
           (else z)))

(define (second x y z)
     (cond ((or (and (<= x y) (>= x z))
                (and (>= x y) (<= x z))) x)
           ((or (and (<= y x) (>= y z))
                (and (>= y x) (<= y z))) y)
           (else z)))

(define (sqr x)
     (* x x))

(define (sum x y)
     (+ x y))
```

~~물론 이런 방법이 가장 적합하지만, 다른 방법도 있다. max?와 min?을 만들어서 상태변수로 계속 넘겨주는 형태로 만들수도 있다.~~
내가 잘못 생각한것 같다. 강사님이 알려주신 솔루션이랑 정렬을 사용하는 방법이 제일 나은것 같다.
하지만 문제의 의도는 1. 처음 풀 때는 이렇게 강사님이 알려준데로 풀고, 나중에 공부가 깊어지면 2. 함수의 합성을 통해 이를 해결하는 것도 바라지 않았을까? 하는 생각도 든다.

- 1.4 연습문제

엮은식(combination)의 연산자 자리에 복잡한 식(compound expression)이 다시 와도 앞에서 밝힌 규칙에 따라 식의 값을 구할 수 있다. 다음 프로시저에 인자를 주고 어떻게 계산되는지 밝혀 보라.

```
; TIP: 여기서 알 수 있는 사실은 비록 '+', '-', '*', '/'와 같은 기본적인 이항 연산자들은 비록 예약어 일지라도 Lisp에서 일반적인 프로시저와 취급이 같다는 사실이다.
; 이는 다른 언어에서 오버로딩된 '+'와 언어에서 기본적으로 제공하는 '+'등은 미묘하게 다르게 취급되는 것과는 확실히 구분된다.

(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

-> 여기서 처음으로 프로시저를 데이터로 취급하는 예제가 나온다. 그리고 트리를 그려서 이 문제를 해결해 보면 문제의 의미를 알기 쉬울 것 같다.

```
      결과값
/  |    |    \  \
if/¦\  + or -  a   b
 > b 0
```

이 트리가 맞는지는 잘 모르겠지만, 대충 이런 꼴이 나올 것이라 예상한다.

- 1.5 연습문제

```
; 인자 먼저 계산법(applicative order evaluation) <- 이렇게 되기는 하는데, 인자를 먼저 평가(evaluation)하게 되면, 인자를 구했을 때 계속 같은 값이 튀어나와서 무한 루프가 발생한다.
; 여기에 꼬리재귀 최적화까지 되어서 레지스터 하나에 계속 같은 값을 집어넣는 행위를 반복하고 되고, 스택 오버플로우도 생기지 않고 무한 루프가 발생하는 것이라고 생각한다.
(define (p) (p))

(define (test 0 (p)
    (if (= 0 0)
        0
       (p)))

// C로 쓰면 이런 상황이 발생하지 않았나 하는 생각이 든다.
int p = p;
while(p == p)
    p = p;

; 정의 먼저 계산법(normal order evaluation) <- 정의 먼저 계산법은 '끝까지 펼친 다음에 줄이는 계산 방법'을 쓰는데, 내 생각이 맞다면 2가지 가능성이 있다.
; 첫째로, 무한 루프가 발생하지 않는다는 것이다. (p)를 끝까지 펼쳤을 때 언어 실행기가 (p)가 된다는 사실을 안다면, 어느 순간에 (p)로 대치하는 행위를 멈출것이다. 따라서, 무한루프가 멈춘다.
; 둘째로, 여전히 무한 루프가 발생한다는 것이다. (p)를 끝까지 펼쳤을 때 언어 실행기가 (p)가 된다는 사실을 모르고, 계속 펼치는 시도를 한다면, 스택 오버플로우로 프로세스가 종료될 것이다.

; 인자 부분에 (p)가 끊임 없이 평가되면서(스택을 쌓지 않고 상태만 변한다.) 메모리가 터지지 않고 계속 돌아간다.(무한루프에 빠짐)
;
```

그리고 지금 여기서 쓰는 문법은 if special-form이라는 사실을 알아야 한다. cond를 사용해서 만든 프로시저와는 궤를 달리한다는 사실을 알아야 한다.

- 연습문제 1.6

Alyssa P. Hacker는 if가 왜 특별한 형태여야 하는지 받아들이기 어려웠다. "그냥 cond를 써서 if를 보통 프로시저처럼 정의하면 안될까?"라고 자신에게 되묻곤 했다. Alyssa의 친구 Eva Lu Ator는 그렇게 할 수 있을 거라고 하면서 if 대신 쓸 수 있는 new-if를 다음처럼 만들어 보았다.

```
; 스포일러: if와는 달리, predicate절과 else절이 동시에 평가되고, 참이면 실행된다.

(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-caluse)))
```

Eva는 Alyssa에게 이 프로그램이 if처럼 돌아간다는 걸 보여주려고 아래처럼 실험을 하였다.

```
(new-if (= 2 3) 0 5)
```
5

```
(new-if (= 1 1) 0 5)
```
0

Alyssa는 좋아하며 다음 제곱근 프로그램에서 new-if를 써보았다.

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

; 일단, new-if의 경우에는 프로시저다. 그리고 프로시저이기 때문에 applicative order(인자 먼저 계산법)를 따른다.
; 하지만 if는 Lisp Interpreter에서 제공하는 special-form이다. 따라서 일반적인 프로시저하고 동작하는 원리가 다르다.
; 잘 받아들여지지 않는다면 이렇게 생각하자: if에게 "인자"를 준다고 표현하지는 않는다. 하지만 new-if는 프로시저이기
때문에 인자를 준다고 표현할 수 있다.
; 이런 차이에서 두 표현이 계산되는 것에 차이가 있는 것이라고 생각할 수 있다.
```

Alyssa가 새로 만든 프로시저로 제곱근을 구하려 할 때 어떤 일이 일어나는가? 설명해 보자.

-> 일반적인 경우에는 같은 결과(정상적인 결과)를 내놓지만, consequent expression과 alternative중에 하나라도 되도는 프로시저의 형태로 짜게된다면, 무한루프에 빠지고 실행을 종료하게 된다.
왜 그런가? 여기서 만든 new-if는 실행기에 의해서 특별하게 정의된 special-form이나 미리 정의된 기계 프로시저가 아니라, 일반적인 그냥 '**프로시저**'이기 때문이다.
그냥 프로시저의 경우(cond를 사용해서 만든) consequent expression과 alternative를 동시에 값을 평가하고 그 값을 구하려든다. 따라서 불필요하게 계속 평가를 하게 되고, 결국은 무한 루프에 빠지게 되는 것이다.

그러면 왜 if는 될까? -> 여기서 if는 실행기에 의해서 applicative order를 따르지 않는다는 사실을 알 수 있다! if는 술어를 평가하고 참이면, 그 때 consequent expression를 평가하고, 술어가 거짓이라면, 그 때 alternative를 평가하여 실행하기 때문에 무한루프에 빠지지 않는다.

- 연습문제 1.7

앞서 만든 good-enough?로는 아주 작은 수의 제곱근을 구하지 못한다.
또 컴퓨터에서 수를 셈할 때에는 유효숫자가 딱 정해져 있다는 점 때문에 아주 큰 수의 제곱근을 구할 때 good-enough?가 올바로 답을 내지 못하는 보기를 들어 이런 문제를 설명해 보라.
good-enough?를 만드는 여러 방법 가운데 하나는, 참값에 더 가까운 값 guess를 구하기 위해 어림잡은 값을 조금씩 고쳐나가면서 헌 값에 견주어 고친 값이 그다지 나아지지 않을 때까지 계산을 이어가는 것이다.
이 방법에 따라 위에서 만든 제곱근 프로시저를 고쳐 보자. 그렇게 고치고 나니, 아주 작은 수나 큰 수의 제곱근을 구할 때 전보다 잘 돌아가는가?

-> 알고리즘 문제 해결 전략 2권(이하 종만북)에서 그 해답을 찾을 수 있다. 이 문제에서 필요한 만큼 본문의 내용을 보이겠다.
NOTE: 종만북에서는 이분법을 다루는 예제에서 이 문제를 다루고 있다.

...(중략)...

```
// 코드 13.1
double f(double x);

// 이분법의 예제 구현
double bisection(double lo, double hi) {
    // 반복문 불변식을 강제한다.
    if(f(lo) > 0)
        swap(lo, hi);
    // 반복문 불변식: f(lo) <= 0 < f(hi)
    while(fabs(hi - lo) > 2e-7) {
        // 사실 여기도 이런 식보다는 오류가 생기지 않는 좋은 표현을 사용하는 것이 좋다.
        double mid = (lo + hi) / 2;
        double fmid = f(mid);
        if(fmid <= 0)
            lo = mid;
        else
```

```
            hi = mid;
    }

    // 가운데 값을 반환한다.
    return (lo + hi)/2;
}
```

- 절대 오차를 이용한 종료 판정(NOTE: 앞서 만든 good-enough?는 절대 오차를 이용한 종료 판정을 사용한다.)
  이분법에서 가장 중요한 부분은 바로 while문의 종료 조건입니다. while문을 많이 수행할수록 오차가 줄어들 테지만,
  알고리즘을 영원히 수행할 수는 없는 노릇이니 우리는 정확도와 수행 속도 사이에서 적절히 타협하는 종료 조건을 선택해야 한다.
  그러나 모든 경우에 안정적으로 동작하도록 이분법의 종료 조건을 선택하기란 놀랍도록 어렵습니다.

  13.1 코드는 hi와 lo의 절대 오차를 이용한 반복문의 종료 판정을 구현합니다.
  hi와 lo가 충분히 가까워질 경우 [lo, hi]구간 내에서 임의의 답을 선택해서 반환하는 것이지요.
  이와 같은 방법을 사용할 수 있는 이유는 대부분의 프로그램 대회 문제에서는 정답과 아주 작은 오차가 있는 답들 또한 정답으로 인정하기 때문입니다.
  우리가 푸는 문제에서 정답과 10^-7이하의 차이가 나는 답들 또한 정답으로 인정한다고 합시다.
  |lo - hi| <= 10^-7인 상황에서 반복문을 종료하고, 두 값의 평균을 반환하면 최대 오차는 |lo - hi|/2 <= 10^-7이 됩니다.
  이것이 13.1 코드의 종료 조건이 lo와 hi의 차이를 2*10^-7과 비교하는 이유입니다.
  반복문 불변식을 강제하기 위해 lo > hi가 될 수도 있기 때문에 항상 hi-lo의 절대값을 취해야 한다는 데 유의하세요.

- 상대 오차를 이용한 종료 판정(NOTE: 문제 1.7은 good-enough?를 상대 오차를 이용한 종료 판정을 사용하도록 고치고, 이를 관찰하는 것을 요구하는 문제다.)
  이와 같이 절대 오차를 사용하는 방법은 다루는 값의 크기가 작을 때는 훌륭하게 동작하지만 값의 크기가 커지면 문제가 생길 수 있습니다.
  부동 소수점은 가수부(mantissa)라고 부르는 정수 부분과 이 변수에서 소수점의 위치를 나타내는 지수부(exponent)의 조합으로 표현되기 때문에,
  표현할 수 있는 수의 집합이 제한되어 있습니다. 따라서 숫자의 절대 값이 커지면 커질수록 표현할 수 있는 수들이 듬성듬성해지게 됩니다.

  13.2 코드는 이 현상에서 오는 문제를 보여줍니다. lo와 hi는 2*10^-5정도 차이 나는 값이기 때문에 이 함수는 금방 종료할 것 같습니다.
  그러나 lo와 hi사이에는 double 변수가 표현할 수 있는 값이 하나도 없습니다. (IEEE754: 하단의 링크를 참고할 것)
  따라서 (lo + hi)/2는 결국 hi와 같은 값이 됩니다. hi와 lo사이의 거리는 10^-7보다 훨씬 크기 때문에 이 함수는 절대로 종료되지 않지요.

  프로그래밍 대회에서는 이와 같은 문제 때문에 절대 오차 외에도 상대 오차를 허용하고 있습니다.
  반환 값의 오차가 10^-7보다 크더라도, 정답 A에 대해 [A*(1 - 10^7), A*(1 + 10^7)] 범위에 포함되는 답을 정답으로 인정하는 것이지요.
  현재의 근사 값 (lo+hi)/2이 범위 안에 어떤 정답에 대해서도 정답으로 인정되기 위해서는 다음 조건이 참이어야 합니다.

  (1 - 10^-7)*hi < (lo + hi)/2 < (1 + 10^-7)*lo

  단 이 조건은 lo와 hi가 전부 양수라고 가정합니다. 두 수가 음수인 경우, lo는 음수이고, hi는 양수인 경우 등을 모두 고려하려면 코드는 더더욱 복잡해지지요.
  그래서 경험 많은 프로그래밍 대회 참가자들은 이와 같은 방법을 쓰지 않습니다. 그럼 어떻게 하냐고요?

- 정해진 횟수만큼 반복하기
  단순해 보이지만 가장 유용한 방법은 while문을 적당한 for문으로 대체해서 반복문이 항상 정해진 횟수만큼 실행되도록 하는 것이다.
  실제로 12장에 나오는 연습 문제들의 구현은 항상 이와 같이 반복문을 100번 수행하도록 작성되어 있는 것을 확인할 수 있지요.
  반복분을 100번 수행하면 우리가 반환하는 답의 절대 오차는 최대 (|lo - hi|)/2^101이 됩니다.
  2^101은 대략 서른한 자리의 수로, |hi - lo|가 대략 10^20 미만의 수라면 이 오차는 항상 10^-7보다 작지요. 따라서 큰 숫자를 다루는 경우에도 충분히 답을 구할 수 있습니다.
  또한 이와 같은 방법은 절대로 무한 반복에 빠지지 않으며, 프로그램의 최대 수행 시간을 예상하기도 쉽다는 장점이 있습니다.
  반복문 내부를 100번 수행할 수 있는가만 확인하면 되니까요.

```
// 코드 13.2

// 종료되지 않는다.
void infiniteBisection() {
```

```
    double lo = 123456123456.1234588623046875;
    double hi = 123456123456.1234741210937500;
    while(fabs(hi - lo) > 2e-7)
        hi = (lo + hi)/2.0;
    printf("finished!\n");
}
```

...(후략)...

굳이 1.7문제를 풀려고 하지 않아도 Lisp Interpreter가 내부적으로 부동소수점 실수를 표현할 때 IEEE754를 사용한다면,
절대 오차를 사용하는 것보다 상대 오차를 사용하는 것이 딱히 더 낫다는 결론을 얻을 수 없다는 것을 알 수 있다.
여기서도 정해진 횟수만큼 반복하는 것이 간결하고 쉬운 코드를 만들고, 오차를 제어하는 데 도움이 된다는 사실을 알 수 있다.

- NOTE: 1장과 2장에서는 수치해석에 관한 문제가 종종 등장하는데, 여기서 오차를 다루는 문제는 매우 중요하게 여겨진다.
    - 수치해석은 관련 지식이 필요한 도메인이 아니라면 만날 일이 별로 없긴하지만,
      요즘같이 초대규모 데이터들에 대한 정량적 분석이 유행하는 시대에는 앞날이 어떻게 될지 모르므로 다음과 같은 단어가 나오면 관련 서적을 찾아보고 공부해야 한다는 것 정도는 알아두자.
        - 오차(실제값 x, 근삿값 x`일 때, 오차는 x - x`으로 정의된다. x - x` = 0이면 오차는 발생하지 않는다.)
            - 절대 오차(abs(x - x`))
            - 상대 오차(abs((x - x`)/x))
        - 이분법(half-interval method)
        - 뉴튼법(newton method)
        - 심슨의 규칙(Simphson's Rule)
    - 개인적으로는 보통 수치해석은 그런게 있다 정도만 알고, 필요한 상황이 오면 그 때 공부해도 늦지 않지만, 다음 주제는 그냥 공부하기를 추천한다.
        - 2진법, 8진법, 10진법, 16진법에 대한 이해와 다루는 법(즉 10진 표기법과 프로그래머가 주로 사용하는 수체계에 대한 이해)
        - 부동소수점 수에 대한 이해 (IEEE754)
        - 오차에 대한 이해

- IEEE754와 실수 비교에대한 URL을 첨부한다.
    - 영어 문서
        - https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/
        - https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
        - http://floating-point-gui.de/
    - 한글 문서
        - pdf주소를 달아놓자.

- 연습문제 1.8

뉴튼법 문제 3번

- 연습문제 1.9

다음 두 프로시저는 모두 0보다 큰 정수 두 개를 더하는 일을 하는데, 인자에 1을 더하는 inc프로시저와 인자에서 1을 빼는 dec프로시저를 가져다 쓴다.

```
(define (inc n)
    (+ n 1))

(define (dec n)
    (- n 1))

(define (+ a b)
    (if (= a 0)
        b
        (inc (+ (dec a) b))))

(define (+ a b)
```

```
    (if (= a 0)
        b
        (+ (dec a) (inc b))))
```

맞바꿈 계산법에 따라 두 프로시저가 (+ 4 5)를 계산하는 프로세스(과정)을 밝혀라. 이 프로세스는 반복하는가? 아니면 되도는가?

-> 맞바꿈 계산법을 적용해서 두 프로시저를 풀어보면, 첫번째 프로시저는 되도는 프로세스임을 알 수 있다.
inc가 필요한 만큼 쌓였다가 마지막 if에 이르러서야 하나씩 풀리며 답을 내놓는다.

하지만 두번째 프로시저는 두개의 상태변수를 가지면서, 한 단계를 마칠때마다 두 상태변수의 값을 바꾼다. 필요한 만큼 함수를 쌓거나 하는 일은 일어나지 않는다.
문제에서 의도하는 것은 아마도 두 프로시저의 코드를 잘 살펴서 되도는 프로세스와 반복하는 프로세스의 차이점을 알아보라는 것이 아닌가 싶은데,
그 차이점은 함수를 한번 호출할 때 그 바깥 쪽에 호출되지 않고 쌓이는 프로시저가 있다면, 그것이 바로 되도는 프로세스가 되는 차이점이 아닐까 싶다.

예시를 들어보겠다.

```
(define (facto n)
    (if (= n 1)
        1
        (* n (facto (- n 1)))))
;        ^ 이부분!

(define (+ a b)
    (if (= a 0)
        b
        (inc (+ (dec a) b))))
;        ^ 이부분!
```

이제 알겠는가?

- 연습문제 1.10

다음은 애커만 함수(Ackerman function)를 나타낸 프로시저다.

-> 죄송합니다.

- 연습문제 1.11

n < 3일 때 f(n) = n이고, n >= 3일 때 f(n) = f(n - 1) + 2f(n - 2) + 3f(n - 3)으로 정의한 함수 f가 있다.

```
; 되도는 프로세스: 그냥 수식을 그대로 옮기면 된다.
(define (f1 n)
    (if (< n 3)
        n
        (+ (f1 (- n 1)) (* 2 (f1 (- n 2))) (* 3 (f1 (- n 3))))))

; 반복하는 프로세스: 처음에는 잘 안됐는데 여기 이 링크를 보자.
; https://github.com/mathias/sicp/blob/master/exercise-1.11.rkt
(define (f2 n)
    (f-iter 2 1 0 n))

(define (f-iter a b c count)
    (if (= count 0)
        a
        (f-iter (+ a (* 2 b) (* 3 c))
                a
                b
                (dec count))))
```