전산통계 및 실험 - Midterm

Jiho Kwak, 2020-17530

2025-10-16

1. Building Abstractions with Procedures

계산적 과정 (computational process)

- 컴퓨터 안에 사는 추상적 존재
- 전개되면서 *데이터*라 부르는 또다른 추상적 존재를 조작
- 프로그램: 계산적 과정의 전개를 지시하는 규칙의 패턴
- 프로그래밍 언어: 프로그램을 작성하는 기호 표현식 -> 계산적 과정이 수행하기 원하는 과업을 상술
- 일상적 생각:자연어 = 정량적 현상:수학 기호 = 계산적 과정:프로그래밍 언어
- 버그: 프로그램의 오류 -> 디버깅 능력 갖춰야 함
- 잘 설계된 계산적 시스템은 모듈식, 즉 개별 부품을 따로 작성, 교체, 디버깅할 수 있는 형태로 작성됨

1.1. The Elements of Programming

프로그래밍의 기본요소

1. 원초 표현식(primitive expressions): 언어가 다루는 가장 단순한 대상
2. 조합 수단(means of combination): 단순한 원소로부터 복합적인 원소를 만드는 방법
3. 추상화 수단(means of abstraction): 복합적인 원소에 이름을 붙이고 하나의 단위로 다루는 방법

프로그래밍의 원소

- 데이터: 조작하고자 하는 재료
- 함수: 데이터를 다루는 규칙의 서술
- 프로그래밍 언어는 원초 데이터와 원초 함수를 서술하는 기능 + 이들을 조합하고 추상화하는 수단을 제공

1.1.1. Expressions

- R 해석기(interpreter)에 표현식(expression)을 입력하면 해석기는 이를 평가(evaluate)하여 결과를 표시
- 원초 표현식의 예: 수(number)

```
486
```

```
## [1] 486
```

- 복합 표현식(compound expression): 표현식을 원초 함수(+, * 등)로 결합 -> 절차 적용을 표현하는 조합(combination)

```
137 + 349
```

```
## [1] 486
```

- 전위 표기법(prefix notation) cf. 중위 표기법(infix notation)

```
`+`(137, 349)
```

```
## [1] 486
```

- 중첩(nesting)

```
(3 * 5) + (10 - 6)
```

```
## [1] 19
```

```
`+`(`*`(3, 5), `-`(10, 6))
```

```
## [1] 19
```

- 연산의 우선순위: 중위 표현법만 해당

```
3 * 5 + 10 / 2
```

```
## [1] 20
```

- Read-eval-print loop(REPL): 해석기의 기본 사이클

1.1.2. Naming and the Environment

- 계산적 개체(computational objects)를 이름 짓고 그 이름으로 지칭: 프로그래밍 언어의 필수 요소
- 이름은 값이 어떤 객체(object)인 변수(variable)를 식별함
- 예) `size`가 2라는 수와 연관되면 2를 `size`라는 이름으로 지칭 가능

```
size <- 2
size; 5*size
```

```
## [1] 2
## [1] 10
```

- 추상화: 복합적인 연산을 간단한 이름으로 지칭하는 것

```
radius <- 10
circumference <- 2 * pi * radius
circumference
```

```
## [1] 62.83185
```

- 환경(environment): (이름, 개체) 쌍을 보관하는 메모리 공간

1.1.3. Evaluating Combinations

조합 평가 절차

1. 조합의 하위 표현식들을 평가
2. 가장 왼쪽 하위 표현식의 값인 함수를 다른 하위 표현식들의 값인 전달인수(arguments)에 적용한다.

- 평가는 재귀적(recursive)

```
# evaluation rule applied to four different combinations
(2 + 4 * 6) * (3 + 12)
```

```
## [1] 390
```

원초 표현식의 평가

- 단계 1을 반복하다 보면 원초 표현식을 평가해야 하는 지점에 도달
  - 수치, 원초 함수, 이름 등
- 평가 규칙

  - 수의 값은 해당 숫자가 나타내는 값 자체
  - 원초 함수의 값은 해당 연산을 수행하는 일련의 기계 명령
  - 이름의 값은 환경에서 그 이름과 연관된 객체 (이름의 의미는 환경이 결정)

```
x <- 2
`<-`(x, 2)  # prefix form
```

조합 평가 규칙의 예외

- 함수 <-를 전달인자 2개에 적용하지 않음 (기호 x의 값을 평가해야 함) -> "특수형"(special forms)

1.1.4. Compound Procedures

- 복합 함수: 강력한 프로그래밍 언어로서의 요소
  - 수치, 산술 연산: 원초 데이터와 원초 함수
  - 조합 중첩(Nesting of combinations): 연산을 결합하는 수단
  - 이름 붙이기: (제한적인) 추상화 수단

함수 정의

- 복합 연산에 이름을 붙이고 하나의 단위로 지칭
- 보다 강력한 추상화 기법

```
square <- function(x) {
  x * x
}
```

예제: 제곱 계산

- 이름이 square인 복합 함수(compound function)
  - x: 지역(local) 이름 -> 자연어의 대명사와 같은 역할
- <-를 이용하여 함수와 이름 square를 연관시킴

```
<함수 이름> <- function(<형식 매개변수 리스트>){<함수 본체>}
```

복합 함수 정의

- 이름: 환경 안에서 함수와 연관시킬 기호
- 형식 매개변수: 함수 본체 안에서 함수의 전달인자를 지칭하는 데 사용할 지역 이름
- 함수 본체: 형식 매개변수가 함수가 적용되는 실제 전달인자(actual arguments)로 대체될 때 함수 적용의 값을 산출하는 표현식

```
square(21)
```

함수 적용(함수 호출)

```
## [1] 441
```

```
sum_of_squares <- function(x, y) {
  square(x) + square(y)
}
sum_of_squares(3, 4)
```

함수로 함수 정의하기

```
## [1] 25
```

- 한 단계 더

```
f <- function(a) {
  sum_of_squares(a + 1, a * 2)
}
f(5)
```

```
## [1] 136
```

- 복합 함수는 원초 함수와 똑같은 방식으로 사용됨
- `sum_of_squares`의 정의만 보면 `square`가 내장되어 있는 함수인지 복합 함수로 정의된 것인지 구분할 수 없음

1.1.5. The Substitution Model for Procedure Application

- 복합 함수를 포함한 조합의 평가도 1.1.3절의 조합 평가 절차를 따름
- 복합 함수를 전달인자에 적용할 때는 형식 매개변수를 대응되는 전달인자로 치환하여 함수본체를 평가

```
f(5)
```

예제: `f(5)`의 평가

- `f`의 본체

```
sum_of_squares(a + 1, a * 2)
```

- 형식 매개변수 `a`를 전달인자의 값 5로 치환

```
sum_of_squares(5 + 1, 5 * 2)
```

- 3가지 subproblem

1. 가장 왼쪽 하위 표현식 `sum_of_squares`의 값을 구함: `function(x, y) { square(x) + square(y) }`
2. 나머지 하위 표현식 `5 + 1`과 `5 * 2`의 값을 구함: 6과 10
3. 위 함수를 6과 10에 적용

```
square(6) + square(10)
```

- 같은 방식으로 `square` 적용

```
(6 * 6) + (10 * 10)
```

- `*` 적용 (2회)
- `+` 적용
- caveat: 치환 모형은 함수 적용에 대한 머릿속 모형일 뿐, 해석기가 반드시 같은 방식으로 동작하는 것은 아님

적용 순서 평가와 정상 순서 평가

- 적용 순서 평가(applicative-order evaluation): 앞서의 평가 방식
- 정상 순서 평가: 완전 전개 후 축약

```
f(5)
sum_of_squares(5 + 1, 5 * 2)
square(5 + 1) + square (5 * 2)
(5 + 1) * (5 + 1) + (5 * 2) * (5 * 2)
```

- 차이: `5 + 1`과 `5 * 2`가 두 번 평가됨
- 함수 평가를 치환 모형으로 설명할 수 있고 적법한 값을 산출할 경우 두 평가 방식은 동등 (연습문제 1.5.)

지연 평가

- R 해석기는 정상 순서를 사용하되 전달인자가 두 번 평가되지 않도록 하는 지연 평가(lazy evaluation) 메커니즘을 사용한다.
- 전달인자의 값이 필요한 시점에서 평가

```
try_me <- function(a, b) {
  if (a==0) {
    1
  } else {
    b
  }
}
try_me(0, head(NULL))
```

```
## [1] 1
```

- 적용 순서 평가에서는 오류

1.1.6. Conditional Expressions and Predicates

$$|x| = \begin{cases} x, & \text{if } x \geq 0, \\ -x, & \text{else.} \end{cases}$$

```
my_abs <- function(x) {
  if (x >= 0) {
    x
  } else {
    -x
  }
}
```

```
if (<서술어>) <표현식1> else <표현식2>
```

- 서술어(predicates): 값이 `TRUE` 아니면 `FALSE`인 표현식 또는 함수
- 서술어가 `TRUE` 값을 가지면 이, 그렇지 않으면 가 평가됨
- 원초 서술어: `<`, `>`, `==`, `<=`, `>=`
- 논리 연산자: `&&`(논리곱), `||`(논리합), `!`(논리부정)
  - 복합 서술어 구축
  - `and`와 `or`은 특수형: 하위 표현식이 모두 평가되지 않을 수 있음

1.1.7. Example: Square Roots by Newton's Method

$$\sqrt{x} = y \quad \text{such that} \quad y \geq 0 \quad \text{and} \quad y^2 = x$$

- 수학 함수는 선언적 (declarative)
- 컴퓨터 함수는 명령적 (imperative)
  - $x = 2$, y의 최초 추측값이 1일 때 $\sqrt{x}$의 계산:

Guess	Quotient	Average
1	$2 / 1 = 2$	$(2 + 1) / 2 = 1.5$
1.5	$2 / 1.5 = 1.3333$	$(1.3333 + 1.5) / 2 = 1.4167$
1.4167	$2 / 1.4167 = 1.4118$	$(1.4167 + 1.4118) / 2 = 1.4142$
1.4142	...	...

```
improve <- function(guess, x) {
  average(guess, x / guess)
}
average <- function(x, y) {
  (x + y) / 2
}
is_good_enough <- function(guess, x) {
  my_abs(square(guess) - x) < 0.001
}
```

```
sqrt_iter <- function(guess, x) {
  if (is_good_enough(guess, x) ) {
    guess
  } else {
    sqrt_iter(improve(guess, x), x)
  }
}

my_sqrt <- function(x) {
  sqrt_iter(1, x)
}

my_sqrt(9)
```

```
## [1] 3.000092
```

- 함수 호출만으로 반복법(iterative method) 구현

1.1.8. Procedures as Black-Box Abstractions

```
is_good_enough <- function(guess, x) {
  my_abs(square(guess) - x) < 0.001
}
```

- `is_good_enough`에 있어 `square`의 구체적인 구현 방식은 중요하지 않음 / 제곱만 계산해 주면 됨 (블랙박스)

```
square <- function(x) {
  exp(doubled(log(x)))
}
doubled <- function(x) {
  x + x
}
```

```
square <- function(y) {
  y * y
}
```

- 구별할 수 없음
- 형식 매개변수 이름은 함수 본체 안에서만 유효
  - 아래의 x는 square의 매개변수 x와 구별되어야 함

```
is_good_enough <- function(guess, x) {
  my_abs(square(guess) - x) < 0.001
}
```

- 형식 매개변수는 함수 정의에 묶여 있음(bound) -> 이름을 찾아바꾸기 해도 됨
- 묶이지 않은 이름은 자유로움(free) -> 이름을 바꾸면 동작이 달라짐
  - e.g. my_abs, square
- 범위(scope): 주어진 이름의 묶임이 유효한 표현식의 집합
  - 형식 매개변수로 선언된 묶인 이름들의 범위 = 함수 본체

내부 정의와 블럭 구조

- my_sqrt에서 중요한 함수는 my_sqrt 뿐
- 이름 충돌을 방지하기 위해 다른 함수들을 my_sqrt안으로 숨기기
- 블럭 구조: 함수 내부에 지역적으로 함수를 정의한 것

```
my_sqrt <- function(x) {
  is_good_enough <- function(guess, x) {
    my_abs(square(guess) - x) < 0.001
  }
  improve <- function(guess, x) {
    average(guess, x / guess)
  }
  sqrt_iter <- function(guess, x) {
    if (is_good_enough(guess, x) ) {
      guess
    } else {
      sqrt_iter(improve(guess, x), x)
    }
    sqrt_iter(1, x)
  }
}
```

- 추가 단순화: my_sqrt에 묶인 x의 범위 내에 is_good_enough, improve, sqrt_iter도 있음
- 내부 함수들 안에서 x는 자유 이름
- x는 sqrt가 불러와질 때 값을 얻음 -> 자구적(字句的) 범위 적용 (lexical scoping)

```
my_sqrt <- function(x) {
  is_good_enough <- function(guess) {
    my_abs(square(guess) - x) < 0.001
  }
  improve <- function(guess) {
    average(guess, x / guess)
  }
  sqrt_iter <- function(guess) {
    if (is_good_enough(guess, x) ) {
      guess
    } else {
      sqrt_iter(improve(guess))
```

```
    }
    sqrt_iter(1, x)
  }
}
```

- 재귀 함수: sqrt_iter의 정의가 자기 자신 호출을 포함

```
sqrt_iter <- function(guess, x) {
  if (is_good_enough(guess, x) ) {
    guess
  } else {
    sqrt_iter(improve(guess, x), x)
  }
}
```

1.2. Procedures and the Processes They Generate

계산적 과정

- 함수는 계산적 과정의 지역적 전개 패턴: 과정의 각 단계가 이전 단계를 기반으로 어떻게 만들어지는지를 명시
- 목표: 간단한 함수가 생성하는 과정의 공통된 형태 알아보기

1.2.1. Linear Recursion and Iteration

계승 계산: 선형 재귀

$$n! = n(n-1)!, \quad 1! = 1$$

```
Factorial <- function(n) {
  if (n==1) {
    1
  } else {
    n * Factorial(n-1)
  }
}
Factorial(6)
```

```
## [1] 720
```

계승 계산: 선형 반복

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

- 상태변수 counter와 product를 두고 아래 과정을 counter가 1에서 n이 될 때까지 반복

```
product <- counter * product
counter <- counter + 1
```

```
Factorial <- function(n) {
  fact_iter(1, 1, n)
}

fact_iter <- function(product, counter, max_count) {
  if (counter > max_count) {
    product
  } else {
    fact_iter(counter * product, counter + 1, max_count)
```

```
  }
}

Factorial(6)
```

```
## [1] 720
```

비교

- 선형 재귀 과정: 확장 후 수축
  - 확장: 미뤄둔 연산(deferred operations)의 사슬(chain) 구축
  - 수축: 연산 수행
  - 재귀 과정(recursive process): 미뤄둔 연산의 사슬로 특징지어지는 계산적 과정
  - 나중에 수행해야 할 연산 기억이 필요
  - 계승 계산 예제: 사슬의 길이와 메모리 요구량 모두 n에 비례 -> **선형** 재귀 과정
- 선형 반복 과정: 상태 변수
  - 각 n에 대해 `product`, `counter`의 값만 기억
  - 확장/수축 없음
  - 반복 과정(iterative process): 상태 변수(state variables), 상태 갱신 규칙, 종료 조건
  - 필요 메모리는 상태 변수 개수에 비례 (n과 무관)
  - 계승 계산 예제: 필요한 단계 수가 n에 비례 -> **선형** 반복 과정

차이

- 반복 과정에서 상태 변수는 임의의 시점에서 과정의 상태를 완전히 서술 -> 멈춘 상태에서 다시 진행 가능
- 재귀 과정은 해석기가 관리하는 보조 정보가 필요 -> 멈춘 상태에서 다시 진행 불가능
- 재귀 함수가 반복 과정을 생성할 수 있음: `fact_iter`

꼬리 재귀 (tail recursion)

- 반복 과정을 생성하는 재귀 함수는 꼬리 재귀(tail recursion) 형태
  - 함수의 마지막 동작이 자기 자신을 호출, 추가 연산 없이 바로 반환
  - 예시: 앞의 `fact_iter`
- 대부분의 프로그래밍 언어는 꼬리 재귀 함수라 하더라도 소비하는 메모리가 호출 횟수에 비례하도록 설계되어 있음
- but 별도의 꼬리 재귀 최적화(tail call optimization) 과정을 거쳐 고정된 메모리 소비량으로 동작하게 할 수 있음 (`Tailcall`)

```
fact_iter <- function(product, counter, max_count) {
  if (counter > max_count) {
    product
  } else {
    Tailcall(fact_iter, counter * product, counter + 1, max_count)
  }
}
Factorial(6)
```

```
## [1] 720
```

1.2.2. Tree Recursion

피보나치 수열 계산: 트리 재귀

$$\mathrm{Fib}(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2), & n \geq 2 \end{cases}$$

```
fib <- function(n) {
  if (n==0) {
    0
  } else if (n==1) {
    1
  } else {
    fib(n - 1) + fib(n -2)
  }
}
```

- 중복 계산: 이파리 노드(`fib(0)`, `fib(1)`의 개수 = `fib(n+1)`
- 필요 계산 단계 수: 노드 수에 비례 (n에 대해 지수적으로 증가)
- 필요 메모리: 트리의 최대 깊이에 비례 (n에 대해 선형으로 증가)

피보나치 수열 계산: 선형 반복

$$a_n = \mathrm{Fib}(n+1), \quad b_n = \mathrm{Fib}(n), \quad a_0 = 1, \quad b_0 = 0$$

으로 두면

$$\begin{bmatrix} a_n \\ b_n \end{bmatrix} = \begin{bmatrix} a_{n-1} + b_{n-1} \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_{n-1} \\ b_{n-1} \end{bmatrix}$$

```
fib <- function(n) {
  fib_iter(1, 0, n)
}
fib_iter <- function(a, b, count) {
  if (count == 0) {
    b
  } else {
    fib_iter(a + b, a, count - 1)
  }
}
```

예제: 잔돈 세기

- 50, 25, 10, 5, 1센트 동전으로 1달러(=100센트) 만들기
- (일반화 재귀) n 종류 동전으로 금액 a를 만드는 경우의 수 =
  - 첫 번째 종류의 동전을 제외한 동전들로 금액 a를 만드는 경우의 수 +
  - n 종류의 동전을 사용해서 금액 $a - d$를 만드는 경우의 수 (d = 첫 번째 종류 동전의 가치)
- 경계 조건
  - $a = 0$이면 경우의 수는 1
  - $a \leq 0$이면 경우의 수는 0

```r
count_change <- function(amount) {
  cc(amount, 5)
}
cc <- function(amount, kinds_of_coins) {
  if (amount == 0) {
    1
  } else if (amount < 0 || kinds_of_coins == 0) {
    0
  } else {
    cc(amount, kinds_of_coins - 1) +
      cc(amount - first_denomination(kinds_of_coins), kinds_of_coins)
  }
}
first_denomination <- function(kinds_of_coins) {
  if (kinds_of_coins == 1) {
    1
  } else if (kinds_of_coins == 2) {
    5
  } else if (kinds_of_coins == 3) {
    10
  } else if (kinds_of_coins == 4) {
    25
  } else if (kinds_of_coins == 5) {
    50
  } else {
    0
  }
}

count_change(100)
```

```
## [1] 292
```

1.2.3. Orders of Growth

- n: 문제 크기
- $R(n)$: 필요한 자원의 양(계산량, 공간 요구량 등)
- 양수 k_1, k_2가 있어 다음을 만족할 때,

$$k_1 f(n) \leq R(n) \leq k_2 f(n)$$

  $R(n) = \Theta(f(n))$으로 표기

- 계승 계산
  - 선형 재귀: 계산량 $\Theta(n)$, 공간 요구량 $\Theta(n)$
  - 선형 반복: 계산량 $\Theta(n)$, 공간 요구량 $\Theta(1)$
- 피보나치 수열
  - 선형 재귀: 계산량 $\Theta(\phi^n)$, 공간 요구량 $\Theta(n)$, $\phi = (1+\sqrt{5})/2$
  - 선형 반복: 계산량 $\Theta(n)$, 공간 요구량 $\Theta(1)$

1.2.4. Exponentiation

$$b^n = b \cdot b^{n-1}, \quad b^0 = 1$$

- 선형 재귀: 계산량 $\Theta(n)$, 공간 요구량 $\Theta(n)$

```
expt <- function(b, n) {
  if (n == 0) {
    1
  } else {
    b * expt(b, n - 1)
  }
}
```

- 선형 반복: 계산량 $\Theta(n)$, 공간 요구량 $\Theta(1)$

```
expt <- function(b, n) {
  expt_iter(b, n, 1)
}
expt_iter <- function(b, counter, product) {
  if (counter == 0) {
    product
  } else {
    expt_iter(b, counter - 1, b * product)
  }
}
```

연속 제곱법

$$b^n = \begin{cases} \left(b^{n/2}\right)^2, & n \text{ even}, \\ b \cdot b^{n-1}, & n \text{ odd}. \end{cases}$$

```
fast_expt <- function(b, n) {
  if (n == 0) {
    1
  } else if (is_even(n)) {
    square(fast_expt(b, n / 2))
  } else {
    b * fast_expt(b, n - 1)
  }
}
is_even <- function(n) { n %% 2 == 0 }
```

- 계산량 $\Theta(\log n)$, 공간 요구량 $\Theta(\log n)$
- $R(n) = R(n/2) + 1$

1.2.5. Greatest Common Divisors

유클리드 호제법

- a를 b로 나눈 나머지가 r일 때,

$$GCD(a, b) = GCD(b, r)$$

```
gcd <- function(a, b) {
  if (b == 0) {
    a
  } else {
    gcd(b, a %% b)
  }
}
```

- 계산량: $\Theta(\log \min(a, b))$

1.2.6. Example: Testing for Primality

1이 아닌 최소 약수 찾기

- n이 소수가 아니면 $\sqrt{n}$ 이하의 약수가 반드시 존재

```
smallest_divisor <- function(n) {
  find_divisor(n, 2)
}
find_divisor <- function(n, test_divisor) {
  if (square(test_divisor) > n ) {
    n
  } else if (divides(test_divisor, n) ) {
    test_divisor
  } else {
    find_divisor(n, test_divisor + 1)
  }
}
divides <- function(a, b) { b %% a == 0 }
```

- 소수 판정: 계산량 $\Theta(\sqrt{n})$

```
is_prime <- function(n) { n == smallest_divisor(n) }

is_prime(43)
```

```
## [1] TRUE
```

페르마 판정법

- 페르마의 소정리: n이 소수이고 $a < n$이 양의 정수이면 $a^n \equiv a (\text{mod } n)$

```
expmod <- function(base, exp, m) {
  if (exp == 0) {
    1
  } else if (is_even(exp) ) {
    square(expmod(base, exp / 2, m)) %% m
  } else {
    (base * expmod(base, exp - 1, m)) %% m
  }
}
fermat_test <- function(n) {
  try_it <- function(a) {
    expmod(a, n, n) == a
  }
  try_it(1 + floor(runif(1) * (n - 1)));
}
fermat_test(97)
```

```
## [1] TRUE
```

- 계산량: $\Theta(\log n)$
- cf. 확률적 방법
  - `fermat_test(n) == FALSE`: n은 소수가 아님
  - `fermat_test(n) == TRUE`: n은 소수일 가능성이 있음

– 시행 횟수를 늘리면 판정을 통과할 경우 소수일 가능성이 커짐

```
fast_is_prime <- function(n, times) {
  if (times == 0) {
    TRUE
  } else if (fermat_test(n) ) {
    fast_is_prime(n, times - 1)
  } else {
    FALSE
  }
}

fast_is_prime(97, 3)
```

```
## [1] TRUE
```

1.3. Formulating Abstractions with Higher-Order Procedures

고차 함수

- 강력한 프로그래밍 언어의 필수 요소: 공통의 패턴에 이름을 부여해 추상체(abstraction)을 만들고 이를 활용해 프로그램을 작성할 수 있게 하는 능력 -> 함수가 제공하는 능력!
- 함수의 매개변수와 변환값이 반드시 수치일 필요는 없음
- 고차 함수(higher-order functions): 함수를 전달인자로 받거나 값으로 변환하는 함수

1.3.1. Procedures as Arguments

1. sum_integers: a부터 b까지의 정수의 합

```
sum_integers <- function(a, b) {
  if (a > b) {
    0
  } else {
    a + sum_integers(a + 1, b)
  }
}
```

2. sum_cubes: a부터 b까지의 정수 세제곱의 합

```
cube <- function(x) { x * x * x }

sum_cubes <- function(a, b) {
  if (a > b) {
    0
  } else {
    cube(a) + sum_cubes(a + 1, b)
  }
}
```

3. pi_sum: 급수 $\frac{1}{a(a+2)} + \frac{1}{(a+4)(a+6)} + \cdots + \frac{1}{b(b+2)} \simeq \sum_{k=0}^{\infty} \frac{1}{(4k+1)(4k+3)} = \frac{\pi}{8}$

```
pi_sum <- function(a, b) {
  if (a > b) {
    0
  } else {
    1 / (a * (a + 2)) + pi_sum(a + 4, b)
  }
```

```
}
```

- 공통 패턴:

```
<name> <- function(a, b) {
  if (a > b) {
    0
  } else {
    <term>(a) + <name>(<next>(a, b)
  }
}
```

- 필요한 함수를 형식 매개 변수로 하여 그대로 코드로 옮겨주면 된다.

```
Sum <- function(term, a, Next, b) {
  if (a > b) {
    0
  } else {
    term(a) + Sum(term, Next(a), Next, b)
  }
}
```

```
inc <- function(n) { n + 1 }
identity <- function(x) { x }
sum_integers <- function(a, b) {
  Sum(identity, a, inc, b)
}
sum_integers(1, 10)
```

```
## [1] 55
```

```
sum_cubes <- function(a, b) {
  Sum(cube, a, inc, b)
}

sum_cubes(1, 10)
```

```
## [1] 3025
```

```
pi_sum <- function(a, b) {
  pi_term <- function(x) {
    1 / (x * (x + 2))
  }
  pi_next <- function(x) {
    x + 4
  }
  Sum(pi_term, a, pi_next, b)
}

8 * pi_sum(1, 1000)  # approx to pi
```

```
## [1] 3.139593
```

함수의 함수

- 정적분

$$I(f) \triangleq \int_0^1 f(x)\,dx$$

  - $f(x) = 1$이면 $I(f) = 1$
  - $f(x) = x$이면 $I(f) = 1/2 \cdots$
  - $f(x) = x^2$이면 $I(f) = 1/3 \cdots$
- 수치적분

$$\int_a^b f \approx \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \cdots \right] dx$$

```
integral <- function(f, a, b, dx) {
  add_dx <- function(x) {
    x + dx
  }
  Sum(f, a + dx / 2, add_dx, b) * dx
}

integral(cube, 0, 1, 0.01);   # exact value is 0.25
```

```
## [1] 0.2499875
```

```
integral(cube, 0, 1, 0.001)
```

```
## [1] 0.2499999
```

1.3.2. Constructing Procedures Using `lambda`

무명함수

- `Sum`에 인자로 전달하기 위해 정의되는 `pi_term` 이나 `pi_next` -> 무명 함수로 사용

```
pi_sum <- function(a, b) {
  Sum(function(x) { 1 / (x * (x + 2)) }, a,
      function(x) { x + 4 }, b )
}
8 * pi_sum(1, 1000)
```

```
## [1] 3.139593
```

- `integral`의 `add_dx`

```
integral <- function(f, a, b, dx) {
  Sum(f, a + dx / 2,
      function(x) { x + dx }, b) * dx
}
integral(cube, 0, 1, 0.01)
```

```
## [1] 0.2499875
```

지역 이름으로 사용

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$$

$$a = 1 + xy, \quad b = 1 - y$$

$$\Longrightarrow f(x,y) = xa^2 + yb + ab$$

- 보조 함수: `f_helper`

```
f <- function(x, y) {
  f_helper <- function(a, b) {
    x * square(a) + y * b + a * b
  }
  f_helper(1 + x * y, 1 - y)
}
f(3, 4)
```

```
## [1] 456
```

- 무명 함수

```
f_2 <- function(x, y) {
  (function(a, b) { x * square(a) + y * b + a * b })(1 + x * y, 1 - y)
}
f_2(3, 4)
```

```
## [1] 456
```

- 지역 변수 사용

```
f_3 <- function(x, y) {
  a <- 1 + x * y
  b <- 1 - y
  x * square(a) + y * b + a * b
}

f_3(3, 4)
```

```
## [1] 456
```

- cf. 무명 함수 vs. 지역 변수

```
x <- 2
(function(x, y) { x * y })(3, x + 2); x
```

```
## [1] 12
```

```
## [1] 2
```

```
x <- 2
{
  x <- 3
  y <- x + 2
  x * y
}; x
```

```
## [1] 15
```

```
## [1] 3
```

1.3.3. Procedures as General Methods

이분법을 이용한 방정식의 근 구하기

- 비선형 방정식 $f(x) = 0$의 특정 구간에서의 근을 수치적으로 구하는 방법

- Remark. 중간값 정리: 실수 값을 갖는 함수 f가 구간 (a,b)에서 연속일 때, $f(a) < 0 < f(b)$이면 (a,b)에 근이 반드시 하나 있다.
- 이분법
  - $x_{mid} \triangleq (a+b)/2$
  - $f(x_{mid}) > 0 \Longrightarrow (a, x_{mid})$에 근 하나 존재
  - $f(x_{mid}) < 0 \Longrightarrow (x_{mid}, b)$에 근 하나 존재
  - $(a,b) \leftarrow (a, x_{mid})/(a,b) \leftarrow (x_{mid}, b)$로 두고 반복
  - 구간의 길이가 허용오차 τ 이하이면 종료
  - 주어진 구간의 길이가 L이면 계산량은 $\Theta(\log(L/\tau))$

```
bisection <- function(f, neg_point, pos_point) {
  midpoint <- (neg_point + pos_point) / 2
  if (close_enough(neg_point, pos_point)) {
    midpoint
  } else {
    test_value <- f(midpoint);
    if (test_value > 0 ) {
      bisection(f, neg_point, midpoint)
    } else if (test_value < 0 ) {
      bisection(f, midpoint, pos_point)
    } else {
      midpoint
    }
  }
}
close_enough <- function(x, y) {
  abs(x - y) < 0.001
}

bisection(function(x) { x * x - 1 }, 0, 2)
```

```
## [1] 1
```

```
half_interval_method <- function(f, a, b) {
  a_value <- f(a)
  b_value <- f(b);
  if (a_value < 0 && b_value > 0 ) {
    bisection(f, a, b)
  } else if (b_value < 0 && a_value > 0 ) {
    bisection(f, b, a)
  } else {
    stop("values are not of opposite sign")
  }
}

half_interval_method(sin, 2, 4)
```

```
## [1] 3.141113
```

함수의 고정점 구하기

- 고정점(fixed point): 함수 f에 대해 방정식 $f(x) = x$의 근
- 고정점 반복: 초기 추측값 x에서 출발하여 함수값이 별로 변하지 않을 때까지 $x \leftarrow f(x)$를 반복

```
tolerance <- 0.00001
fixed_point <- function(f, first_guess) {
```

```
  close_enough <- function(x, y) {
    abs(x - y) < tolerance;
  }
  try_with <- function(guess) {
    nxt <- f(guess)
    if (close_enough(guess, nxt) ) {
      nxt
    } else {
      try_with(nxt)
    }
  }
  try_with(first_guess)
}

fixed_point(cos, 1)
```

```
## [1] 0.7390823
```

제곱근 구하기

$$\sqrt{x} = y \quad \text{such that} \quad y \geq 0 \quad \text{and} \quad y^2 = x$$

- $y = x/y$의 고정점: 발산

```
Sqrt <- function(x) { fixed_point(function(y) { x / y }, 1) }
```

- $y = \frac{1}{2}(y + x/y)$의 고정점: average damping (1.1.7)

```
Sqrt <- function(x) {
  fixed_point(function(y) { (y + x / y) / 2 }, 1)
}
Sqrt(2)
```

```
## [1] 1.414214
```

1.3.4. Procedures as Returned Values

- 함수를 돌려주는 함수: 도함수, 부정적분
- 평균감쇄의 추상화

```
average_damp <- function(f) {
  function(x) {
    (x + f(x)) / 2
  }
}
```

- 제곱근 근사

```
Sqrt <- function(x) {
  fixed_point(average_damp(function(y) { x / y }), 1)
}
Sqrt(2)
```

```
## [1] 1.414214
```

- 세제곱근 근사

```
cube_root <- function(x) {
  fixed_point(average_damp(function(y) { x / square(y)}), 1)
```

```
}
cube_root(8)
```

```
## [1] 1.999998
```

뉴턴법

- 뉴턴법: 비선형 방정식 $g(x) = 9$의 근을 수치적으로 구하는 방법

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

  - $g(x) = 0$의 근: f의 고정점
  - 즉, 뉴턴법은 f에 대한 고정점 근사
- 보조함수: 도함수 근사

```
dx <- 0.00001
deriv <- function(g) {
  function(x) { (g(x + dx) - g(x)) / dx }
}

deriv(function(x) { x * x * x})(5)  # Df(5) where f(x) = x^3
```

```
## [1] 75.00015
```

- 뉴턴법 표현

```
newton_transform <- function(g) {
  function(x) { x - g(x) / deriv(g)(x) }
}
newtons_method <- function(g, guess) {
  fixed_point(newton_transform(g), guess)
}
```

뉴턴법으로 제곱근 계산

- 제곱근 $= g(y) \triangleq y^2 - x$일 때 $g(y) = 0$의 근

$$\begin{aligned} f(y) = y - \frac{g(y)}{Dg(y)} &= y - \frac{y^2 - x}{2y} \\ &= \frac{1}{2}\left(y + \frac{x}{y}\right) \end{aligned}$$

  - 평균감쇄법과 일치

```
Sqrt <- function(x) {
  newtons_method(function(y) { square(y) - x }, 1 )
}

Sqrt(2)
```

```
## [1] 1.414214
```

추상화와 일급 함수

- 고정점 반복법의 추상화 (`transform` 달라지게)

```
fixed_point_of_transform <- function(g, transform, guess) {
  fixed_point(transform(g), guess)
}
```

- 평균감쇄법

```
Sqrt <- function(x) {
  fixed_point_of_transform(
    function(y) { x / y }, average_damp, 1
  )
}

Sqrt(2)
```

```
## [1] 1.414214
```

- 뉴턴법

```
Sqrt <- function(x) {
  fixed_point_of_transform(
    function(y) { square(y) - x }, newton_transform, 1
  )
}

Sqrt(2)
```

```
## [1] 1.414214
```

- 일급 개체: 계산적 개체의 조작에 가해지는 제약이 가장 적은 개체
- 특성
  - 이름으로 지칭할 수 있음
  - 함수의 인자로 전달될 수 있음
  - 함수의 반환값이 될 수 있음
  - 자료 구조에 포함될 수 있음
- R에서는 함수가 일급 개체 -> 함수를 일급 개체로 다루면 표현력 크게 높아짐

2. Building Abstractions with Data

데이터를 이용한 추상화

- cf. 1장: 함수를 조합하여 복합 함수 만들기
- 2장: 데이터 개체를 조합하여 복합 데이터 만들기
- 장점
  - 프로그램 설계의 개념적 수준 제고
  - 설계 모듈성 확충
  - 언어 표현력 확장

유리수의 표현

$$\text{유리수} = \frac{\text{분자(정수)}}{\text{분모(정수)}}$$

- 원초 데이터만 사용 (정수) + 분모 계산 함수, 분자 계산 함수 필요 + 어느 분자가 어느 분모에 대응되는지 추적, 관리하는 장치 필요 - 복합 데이터 사용 ('유리수' 개체) + 유리수를 하나의 개념적 단위로 취급 + 모듈성 강화: 유리수를 그 자체로 다루는 부분 / 유리수가 정수 두 개의 순서쌍으로 표현되는 부분 분리 + 데이터 추상화

표현력: 선형 결합

- a, b, x, y가 원초 연산이 작용될 수 있는 원초 데이터임을 알아야 하는 경우

```
linear_combination <- function(a, b, x, y) {
    a * x + b * y
}
```

- a, b, x, y를 전달인자로 넘겨주기만 하면 됨
- 유리수, 복소수, 다항식 등의 선형 결합을 하나의 `linear_combination`으로 다룰 수 있음

```
linear_combination <- function(a, b, x, y) {
    add(mul(a, x), mul(b, y))
}
```

2.1. Introduction to Data Abstraction

데이터 추상화

- 분리 정복
  - (복합) 데이터 개체가 사용되는 방식
  - 이를 보다 기본적인 데이터 개체로 구현하는 방식
- 양자간 인터페이스
  - 선택자(selector)
  - 생성자(cosntructor)
- 프로그램은 데이터에 관해 최소한의 가정만 함

2.1.1. Example: Arithmetic Operations for Rational Numbers

- 생성자
  - `make_rat(n, d)`: 분자가 정수 `n`이고 분모가 정수 `d`인 유리수 반환
- 선택자
  - `numer(x)`: 유리수 `x`의 분자 반환
  - `denom(x)`: 유리수 `x`의 분모 반환

```
add_rat <- function(x, y) {
  make_rat(numer(x) * denom(y) + numer(y) * denom(x),
           denom(x) * denom(y))
}
sub_rat <- function(x, y) {
  make_rat(numer(x) * denom(y) - numer(y) * denom(x),
           denom(x) * denom(y))
}
mul_rat <- function(x, y) {
  make_rat(numer(x) * numer(y),
           denom(x) * denom(y))
}
div_rat <- function(x, y) {
  make_rat(numer(x) * denom(y),
           denom(x) * numer(y))
}
equal_rat <- function(x, y) {
  numer(x) * denom(y) == numer(y) * denom(x);
}
```

순서쌍 자료구조를 이용한 구현

- 복합 데이터

```
x <- pair(1, 2)
head(x); tail(x)
```

```
## [1] 1
```

```
## [1] 2
```

- 순서쌍의 순서쌍

```
x <- pair(1, 2)
y <- pair(3, 4)
z <- pair(x, y)
head(head(z)); head(tail(z))
```

```
## [1] 1
```

```
## [1] 3
```

- 유리수의 표현

```
make_rat <- function(n, d) { pair(n, d) }
numer    <- function(x) { head(x) }
denom    <- function(x) { tail(x) }

print_rat <- function(x) {
  cat(numer(x), "/", denom(x))
  cat("\n")
}
```

```
one_half <- make_rat(1, 2)
print_rat(one_half)
```

```
## 1 / 2
```

- 약분 기능 추가

```
make_rat <- function(n, d) {
    g <- gcd(n, d)
    pair(n / g, d / g)
}

print_rat(add_rat(one_half, one_half))
```

```
## 1 / 1
```

- 실제 유리수 연산을 구현하는 `add_rat` 등은 손대지 않고 생성자인 `make_rat`만 고쳐서 해결 가능

2.1.2. Abstraction Barriers

- 데이터 추상화의 이념: 데이터 개체의 형식마다 그 자료형의 데이터 개체를 조작하는 데 필요한 기본 연산들을 정의 -> 오직 그 연산들로만 데이터를 조작

```
make_rat <- function(n, d) {
    pair(n, d)
}
numer <- function(x) {
    g <- gcd(head(x), tail(x))
```

```
    head(x) / g
}
denom <- function(x) {
    g <- gcd(head(x), tail(x))
    tail(x) / g
}

print_rat(add_rat(one_half, one_half))
```

기약분수 표현의 또 다른 방식

```
## 1 / 1
```

- 유리수 생성은 드물고 분모/분자에의 접근이 빈번: 생성시 gcd 호출 (make_rat)
- 유리수 생성은 빈번하고 분모/분자에의 접근이 드묾: 선택자에서 gcd 호출 (numer, denom)
- but 어떤 표현을 사용하더라도 add_rat 등의 함수는 수정이 없어야 함

2.1.3. What Is Meant by Data?

- 유리수 데이터와 생성자/선택자 사이의 관계: x가 make_rat(n, d)의 반환값이면

$$\frac{\text{numer}(x)}{\text{denom}(x)} = \frac{n}{d}$$

  - 생성자/선택자는 유리수 표현을 위해 필요충분
  - 데이터 = 그 표현을 위해 특정 조건을 만족하는 선택자/생성자의 모임으로 정의할 수 있음

순서쌍이란 무엇인가?

- pair, head, tail의 필요충분 관계: z가 pair(x, y) $Longleftrightarrow$ head(z)는 x이고 tail(z)는 y
- 순서쌍의 함수적 표현

```
pair <- function(x, y) {
  dispatch <- function(message) {
    if (message == 0) {
      x
    } else if (message == 1) {
      y
    } else {
      stop("argument not 0 or 1 -- pair")
    }
  }
  dispatch
}
head <- function(pair) { pair(0) }
tail <- function(pair) { pair(1) }
```

```
z <- pair(1,2)
head(z); tail(z)
```

```
## [1] 1
```

```
## [1] 2
```

- pair, head, tail 만으로 순서쌍 개체에 접근하는 한 함수적 구현을 자료구조를 사용한 구현과 구분할 수 없음
- 함수를 전달인자로 받거나 돌려 주는 함수(first-class function)가 있으면 복합 데이터를 표현할 수 있음

2.2. Hierarchical Data and the Closure Property

순서쌍의 시각적 표현

- 1, 2, 3, 4를 조합하는 두 가지 방법
  - `(cons (cons 1 2) (cons 3 4))`
  - `(cons (cons 1 (cons 2 3)) 4)`
- 닫힘(closure): 순서쌍을 원소로 갖는 순서쌍을 생성하는 능력
  - 데이터 개체의 집합은 `pair` 연산에 대해 닫혀 있음
  - 집합이 어떤 연산에 대해 닫혀 있다면 해당 연산의 결과에 다시 그 연산을 적용 가능
  - 계층적(hierarchical) 또는 재귀적(recursive) 자료 구조 생성 가능

2.2.1. Representing Sequences

- 수열(sequence): 데이터 개체의 순서 집합
- 예시: 중첩된 순서쌍 -> 순서쌍 사슬, 리스트

```
pair(1,
  pair(2,
    pair(3,
      pair(4, NULL))))
```

- `NULL`: 사슬의 끝을 나타냄
- 다음과 동등: `List(1, 2, 3, 4)`

```
one_through_four <- List(1, 2, 3, 4)
head(one_through_four); tail(one_through_four)
```

```
## [1] 1
```

```
## [ 2, [ 3, [ 4, NULL ] ] ]
```

```
head(tail(one_through_four))
```

```
## [1] 2
```

```
pair(10, one_through_four)
```

```
## [ 10, [ 1, [ 2, [ 3, [ 4, NULL ] ] ] ] ]
```

리스트 연산

- 리스트 따라가기 (n번째 아이템 출력)

```
# returns the n-th item of the list
list_ref <- function(items, n) {
  if (n == 0) {
    head(items)
  } else {
    list_ref(tail(items), n - 1)
  }
}
squares <- List(1, 4, 9, 16, 25)
list_ref(squares, 3)
```

```
## [1] 16
```

- 리스트의 길이

```
# recursive style
len <- function(items) {
  if (is.null(items)) {
    0
  } else {
    1 + len(tail(items))
  }
}

odds <- List(1, 3, 5, 7)
len(odds)
```

```
## [1] 4
```

```
# iterative style
len <- function(items) {
  len_iter <- function(a, count) {
    if (is.null(a)) {
      count
    } else {
      len_iter(tail(a), count + 1)
    }
  }
  len_iter(items, 0)
}

odds <- List(1, 3, 5, 7)
len(odds)
```

```
## [1] 4
```

- 꼬리에 붙이기 (append)

```
appnd <- function(list1, list2) {
  if (is.null(list1)) {
    list2
  } else {
    pair(head(list1), appnd(tail(list1), list2))
  }
}

squares <- List(1, 4, 9, 16, 25)
odds <- List(1, 3, 5, 7)
appnd(squares, odds)
```

```
## [ 1, [ 4, [ 9, [ 16, [ 25, [ 1, [ 3, [ 5, [ 7, NULL ] ] ] ] ] ] ] ] ]
```

- 만약 `list1`이 빈 리스트 -> `list2`
- 그렇지 않으면 `list1`의 꼬리에 `list2`를 붙이고 `list1`의 머리를 결과의 머리로 함

리스트에 대한 사상(寫像, map)

- 리스트의 각 원소를 변환하여 결과값들의 리스트를 생성하는 함수

```
scale_list <- function(items, factor) {
  if (is.null(items)) {
    NULL
```

```
  } else {
    pair(head(items) * factor,
         scale_list(tail(items), factor))
  }
}

scale_list(List(1, 2, 3, 4, 5), 10)
```

```
## [ 10, [ 20, [ 30, [ 40, [ 50, NULL ] ] ] ] ]
```

- 이를 고차함수로 일반화

```
map <- function(fun, items) {
  if (is.null(items)) {
    NULL
  } else {
    pair(fun(head(items)),
         map(fun, tail(items)))
  }
}

map(abs, List(-10, 2.5, -11.6, 17));
```

```
## [ 10, [ 2.5, [ 11.6, [ 17, NULL ] ] ] ]
```

```
map(function(x) { x * x }, List(-10, 2.5, -11.6, 17))
```

```
## [ 100, [ 6.25, [ 134.56, [ 289, NULL ] ] ] ]
```

```
scale_list <- function(items, factor) {
  map(function(x) { x * factor }, items)
}

scale_list(List(1, 2, 3, 4, 5), 10)
```

```
## [ 10, [ 20, [ 30, [ 40, [ 50, NULL ] ] ] ] ]
```

- 원래의 `scale_list`: 리스트의 원소별 처리 방식이 그대로 드러남
- `map`으로 구현한 `scale_list`: 처리 방식의 세부 사항 숨겨짐 (보다 높은 수준의 추상화 - 원소의 리스트를 받이 리스트를 출력)
- `map`을 통해 구축한 추상화 장벽
  - 리스트를 변환하는 함수의 구현 vs. 리스트의 원소를 추출하고 조합하는 상세한 절차

2.2.2. Hierarchical Structures

```
x <- pair(List(1, 2), List(3, 4))
```

```
is_pair <- function(x) {
  inherits(x, "List")
}

count_leaves <- function(tree) {
  if (is.null(tree)) {
    0
```

```
  } else if (!is_pair(tree)) {
    # 일반 숫자(원자형)면 leaf 하나
    1
  } else {
    # head, tail 재귀
    count_leaves(head(tree)) + count_leaves(tail(tree))
  }
}
```

```
len(x);  # as a List
```

트리 재귀

```
## [1] 3
```

```
count_leaves(x) # as a tree
```

```
## [1] 4
```

```
List(x, x)
```

```
## [ [ [ 1, [ 2, NULL ] ], [ 3, [ 4, NULL ] ] ], [ [ [ 1, [ 2, NULL ] ], [ 3, [ 4, NULL ]
↪  ] ], NULL ] ]
```

```
len(List(x, x)); # as a List
```

```
## [1] 2
```

```
count_leaves((List(x, x))) # as a tree
```

```
## [1] 8
```

- len의 재귀
  - 리스트 x의 길이는 tail(x)의 길이에 1을 더한 것
  - 빈 리스트의 길이는 0
- count_leaves의 재귀
  - 트리 x의 이파리 수는 head(x)의 이파리 수와 tail(x)의 이파리 수의 합
  - 빈 트리의 이파리 수는 0
  - 잎 노드의 이파리 수는 1

```
count_leaves <- function(x) {
  if (is.null(x)) { # empty
    0
  } else if (!is_pair(x)) { # leaf
    1
  } else {
    count_leaves(head(x)) + count_leaves(tail(x))
  }
}
```

```
scale_tree <- function(tree, factor) {
  if (is.null(tree)) { # empty
    NULL
  } else if (!is_pair(tree)) { # leaf
    tree * factor
```

```
  } else {
    pair(scale_tree(head(tree), factor),
         scale_tree(tail(tree), factor))
  }
}

scale_tree(List(1, List(2, List(3, 4), 5), List(6, 7)), 10)
```

트리에 대한 사상

```
## [ 10, [ [ 20, [ [ 30, [ 40, NULL ] ], [ 50, NULL ] ] ], [ [ 60, [ 70, NULL ] ], NULL ]
↪  ] ]
```

- map 사용 (tree를 list로 간주)

```
scale_tree <- function(tree, factor) {
  map(function(sub_tree) {
        if (is_pair(sub_tree)) {
          scale_tree(sub_tree, factor)
        } else {
          sub_tree * factor
        }
    }, tree)
}

scale_tree(List(1, List(2, List(3, 4), 5), List(6, 7)), 10)
```

```
## [ 10, [ [ 20, [ [ 30, [ 40, NULL ] ], [ 50, NULL ] ] ], [ [ 60, [ 70, NULL ] ], NULL ]
↪  ] ]
```

2.2.3. Sequences as Conventional Interfaces

```
square <- function(x) { x * x }
is_odd <- function(n) { n %% 2 == 1 }

sum_odd_squares <- function(tree) {
  if (is.null(tree)) {
    0
  } else if(!is_pair(tree)) {
    if (is_odd(tree)) {
      square(tree)
    } else {
        0
    }
  } else {
    sum_odd_squares(head(tree)) +
              sum_odd_squares(tail(tree))
  }
}

sum_odd_squares(List(List(2, 3), List(4, 5)))
```

```
## [1] 34
```

```
is_even <- function(n) { n %% 2 == 0 }
fib <- function(n) {
```

```
  if (n == 0) { 0
  } else if (n == 1) { 1
  } else {
    fib(n - 1) + fib(n - 2);
  }
}

even_fibs <- function(n) {
  nxt <- function(k) {
    if (k > n) {
      NULL
    } else {
      f <- fib(k)
      if (is_even(f)) {
        pair(f, nxt(k + 1))
      } else {
        nxt(k + 1)
      }
    }
  }
  nxt(0)
}

even_fibs(9)
```

```
## [ 0, [ 2, [ 8, [ 34, NULL ] ] ] ]
```

두 함수의 추상적 유사성

- `sum_odd_squares`
  - 트리의 이파리를 **나열**
  - 홀수값을 갖는 잎만 선택하도록 **필터링**
  - 선택된 잎의 값의 제곱을 **계산**
  - 0에서 출발, +를 이용해서 **누적**합 계산
- `even_fibs`
  - 0부터 n까지의 정수를 **나열**
  - 각 정수에 대한 피보나치 수를 **계산**
  - 짝수값을 갖는 피보나치 수만 선택하도록 **필터링**
  - 빈 리스트에서 출발, `pair`를 이용해서 선택된 값을 **누적**

수열 연산

1. 사상 모듈

```
map(square, List(1, 2, 3, 4, 5))
```

```
## [ 1, [ 4, [ 9, [ 16, [ 25, NULL ] ] ] ] ]
```

2. 필터링 모듈

```
filtr <- function(predicate, sequence) {
  if (is.null(sequence)) {
    NULL
  } else if (predicate(head(sequence))) {
    pair(head(sequence),
         filtr(predicate, tail(sequence)))
```

```
  } else {
    filtr(predicate, tail(sequence))
  }
}

filtr(is_odd, List(1, 2, 3, 4, 5))
```

```
## [ 1, [ 3, [ 5, NULL ] ] ]
```

3. 누적 모듈

```
accumulate <- function(op, initial, sequence) {
  if (is.null(sequence)) {
    initial
  } else {
    op(head(sequence),
       accumulate(op, initial, tail(sequence)))
  }
}

accumulate(`+`, 0, List(1, 2, 3, 4, 5));
```

```
## [1] 15
```

```
accumulate(pair, NULL, List(1, 2, 3, 4, 5))
```

```
## [ 1, [ 2, [ 3, [ 4, [ 5, NULL ] ] ] ] ]
```

4. 열거 모듈

- even_fibs

```
enumerate_interval <- function(low, high) {
  if (low > high) {
    NULL
  } else {
    pair(low, enumerate_interval(low + 1, high))
  }
}

enumerate_interval(2, 7)
```

```
## [ 2, [ 3, [ 4, [ 5, [ 6, [ 7, NULL ] ] ] ] ] ]
```

- sum_odd_squares

```
enumerate_tree <- function(tree) {
  if (is.null(tree)) { # empty
    NULL
  } else if ( !is_pair(tree) ) { # leaf
    List(tree)
  } else {
    appnd(enumerate_tree(head(tree)),
          enumerate_tree(tail(tree)))
  }
}

enumerate_tree(List(1, List(2, List(3, 4)), 5))
```

```
## [ 1, [ 2, [ 3, [ 4, [ 5, NULL ] ] ] ] ]
```

- 신호 흐름식 구현: 통용 인터페이스(map, filter 등)를 활용한 모듈식 설계

```
sum_odd_squares <- function(tree) {
  accumulate(`+`,
             0,
             map(square,
                 filtr(is_odd,
                       enumerate_tree(tree))))
}

sum_odd_squares(List(List(2, 3), List(4, 5)))
```

```
## [1] 34
```

```
even_fibs <- function(n) {
  accumulate(pair,
             NULL,
             filtr(is_even,
                   map(fib,
                       enumerate_interval(0, n))))
}

even_fibs(9)
```

```
## [ 0, [ 2, [ 8, [ 34, NULL ] ] ] ]
```

- 모듈 재활용 예시들

```
list_fib_squares <- function(n) {
  accumulate(pair,
             NULL,
             map(square,
                 map(fib,
                     enumerate_interval(0, n))))
}

list_fib_squares(10)
```

```
## [ 0, [ 1, [ 1, [ 4, [ 9, [ 25, [ 64, [ 169, [ 441, [ 1156, [ 3025, NULL ] ] ] ] ] ] ] ]
↪  ] ] ] ]
```

```
product_of_squares_of_odd_elements <- function(sequence) {
  accumulate(`*`,
             1,
             map(square,
                 filtr(is_odd, sequence)))
}

product_of_squares_of_odd_elements(List(1, 2, 3, 4, 5))
```

```
## [1] 225
```

```
is_programmer <- function(record) {
  identical(head(tail(record)), "programmer")
}
```

```
salary <- function(record) {
  head(tail(tail(record)))
}

salary_of_highest_paid_programmer <- function(records) {
  accumulate(max,
             0,
             map(salary,
                 filtr(is_programmer, records)))
}

my_records <- List(List("Linus", "programmer", 30000),
                   List("Richard", "programmer", 25000),
                   List("Bill", "manager", 2500000))
salary_of_highest_paid_programmer(my_records)
```

```
## [1] 30000
```

- (리스트로 구현된) 수열은 처리 모듈을 결합시키는 통용 인터페이스로 작용
- 자료구조를 고르게 수열로 표현하면 프로그램의 자료 구조 의존성을 소수의 수열 연산(`map`, `filter` 등)으로 국한시킬 수 있음
  - 수열의 다른 representation으로 실험할 때 프로그램 설계는 그대로 두고 수열 연산만 변경

중첩된 사상

- 주어진 양의 정수 n에 대해 $1 \le j < i \le n$이고 $i+j$가 소수인 서로 다른 양의 정수 i, j의 순서쌍?
- 절차 1
  - 주어진 정수 $i < n$에 대해 i보다 작은 정수 j 나열
  - 순서쌍 (i, j) 생성

```
n <- 6

accumulate(appnd,
           NULL,
           map(function(i) {
                 map(function(j) { List(i, j) },
                     enumerate_interval(1, i - 1))
               },
               enumerate_interval(1, n)
            )
        )
```

```
## [ [ 2, [ 1, NULL ] ], [ [ 3, [ 1, NULL ] ], [ [ 3, [ 2, NULL ] ], [ [ 4, [ 1, NULL ]
↪  ], [ [ 4, [ 2, NULL ] ], [ [ 4, [ 3, NULL ] ], [ [ 5, [ 1, NULL ] ], [ [ 5, [ 2, NULL
↪  ] ], [ [ 5, [ 3, NULL ] ], [ [ 5, [ 4, NULL ] ], [ [ 6, [ 1, NULL ] ], [ [ 6, [ 2,
↪  NULL ] ], [ [ 6, [ 3, NULL ] ], [ [ 6, [ 4, NULL ] ], [ [ 6, [ 5, NULL ] ], NULL ] ]
↪  ] ] ] ] ] ] ] ] ] ] ] ] ]
```

```
# shortcut: map, then accumulate with appnd
flatmap <- function(f, seq) {
  accumulate(appnd, NULL, map(f, seq))
}
```

- 절차 2

  - 합이 소수인 것만 필터링

```
# predicate
is_prime_sum <- function(pair) {
  is_prime(head(pair) + head(tail(pair)))
}

is_prime_sum(List(8, 9))
```

```
## [1] TRUE
```

- 절차 3
  - map

```
# tripple
make_pair_sum <- function(pair) {
  List(head(pair), head(tail(pair)),
       head(pair) + head(tail(pair)))
}

make_pair_sum(List(8, 9))
```

```
## [ 8, [ 9, [ 17, NULL ] ] ]
```

```
prime_sum_pairs <- function(n) {
  map(make_pair_sum,
      filtr(is_prime_sum,
            flatmap(function(i) {
                      map(function(j) { List(i, j) },
                          enumerate_interval(1, i - 1))
                    },
                    enumerate_interval(1, n)
                )
        )
    )
}

prime_sum_pairs(6)
```

```
## [ [ 2, [ 1, [ 3, NULL ] ] ], [ [ 3, [ 2, [ 5, NULL ] ] ], [ [ 4, [ 1, [ 5, NULL ] ] ],
↪  [ [ 4, [ 3, [ 7, NULL ] ] ], [ [ 5, [ 2, [ 7, NULL ] ] ], [ [ 6, [ 1, [ 7, NULL ] ]
↪  ], [ [ 6, [ 5, [ 11, NULL ] ] ], NULL ] ] ] ] ] ]
```

집합 S의 순열: 재귀적 생성

- 모든 $x \in S$에 대해
  - $S \setminus \{x\}$의 순열을 생성하고
  - x를 그 앞에 붙임

```
permutations <- function(s) {
  if (is.null(s)) {      # empty set?
    List(NULL)           # sequence containing empty set
  } else {
    flatmap(function(x) {
              map(function(p) { pair(x, p) },
                  permutations(remov(x, s))
```

```
                )
            },
            s
        )
  }
}

remov <- function(item, sequence) {
  filtr(function(x) { !(x == item) },
        sequence
    )
}

len(remov(3, List(1, 2, 3, 4, 5)))
```

```
## [1] 4
```

```
permutations(List(1, 2, 3))
```

```
## [ [ 1, [ 2, [ 3, NULL ] ] ], [ [ 1, [ 3, [ 2, NULL ] ] ], [ [ 2, [ 1, [ 3, NULL ] ] ],
↪  [ [ 2, [ 3, [ 1, NULL ] ] ], [ [ 3, [ 1, [ 2, NULL ] ] ], [ [ 3, [ 2, [ 1, NULL ] ]
↪  ], NULL ] ] ] ] ] ]
```