전산통계 및 실험 - Final

Jiho Kwak, 2020-17530

2025-12-14

2.3. Symbolic Data

2.3.1. Quotation

- 인용부호를 도입할 필요성: 값과 기호의 구분

```
a <- 1
b <- 2
List(a, b)
```

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

```
List("a", "b")
```

```
## [ a, [ b, NULL ] ]
```

- 문자열의 비교
  - 두 기호가 같은 순서 + 같은 문자로 구성되어 있으면 동일

```
member <- function(item, x) {
  if (is.null(x)) {
    NULL
  } else if (identical(item, head(x))) {
    x
  } else {
    member(item, tail(x))
  }
}

member("apple", List("x", "y", "apple", "pear"))
```

```
## [ apple, [ pear, NULL ] ]
```

2.3.2. EX: Symbolic Differentiation

- 대수 표현식을 기호적으로 미분하는 함수
- 데이터 추상화 전략 활용 (cf. 유리수 표현 및 산술)

추상적 데이터의 미분 프로그램

- 두 피연산자의 곱과 합으로만 구성된 대수식의 미분

$$\frac{dc}{dx} = 0 \text{ for } c \text{ a constant}$$
$$\frac{dx}{dx} = 1$$
$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$
$$\frac{d(uv)}{dx} = u\left(\frac{dv}{dx}\right) + v\left(\frac{du}{dx}\right)$$

- 마지막 두 줄은 재귀적

함수	의미
`is_variable(e)`	Is `e` a variable?
`is_same_variable(v1, v2)`	Are `v1` and `v2` the same variable?
`is_sum(e)`	Is `e` a sum?
`addend(e)`	Addend of the sum `e`.
`augend(e)`	Augend of the sum `e`.
`make_sum(a1, a2)`	Construct the sum of `a1` and `a2`.
`is_product(e)`	Is `e` a product?
`multiplier(e)`	Multiplier of the product `e`.
`multiplicand(e)`	Multiplicand of the product `e`.
`make_product(m1, m2)`	Construct the product of `m1` and `m2`.

```
deriv <- function(exp, variable) {
  if (is.numeric(exp)) {
    0
  } else if (is_variable(exp)) {
    ifelse(is_same_variable(exp, variable), 1, 0)
  } else if (is_sum(exp)) {
    make_sum(deriv(addend(exp), variable),
             deriv(augend(exp), variable))
  } else if (is_product(exp)) {
    make_sum(make_product(multiplier(exp),
                          deriv(multiplicand(exp), variable)),
             make_product(deriv(multiplier(exp), variable),
                          multiplicand(exp))
        )
  } else {
     stop("unknown expression type -- deriv")
  }
}
```

대수식의 표현

- 전위 표기법(prefix notation)
  - $ax + b$ -> `List("+", List("*", "a", "x"), "b")`

```
# 변수는 그냥 문자열
is_variable <- function(x) { is.character(x) }

# 두 변수는 해당 문자열이 같으면 같다
```

```
is_same_variable <- function(v1, v2) {
  is_variable(v1) && is_variable(v2) && identical(v1, v2)
}

# 함과 곱은 리스트
make_sum <- function(a1, a2) { List("+", a1, a2) }
make_product <- function(m1, m2) { List("*", m1, m2) }

# 리스트의 첫 원소가 "+"이면 합
is_sum <- function(x) {
  is_pair(x) && identical(head(x), "+")
}

# 더하는 수는 합 리스트의 두번째 원소
addend <- function(s) { head(tail(s)) }

# 더해지는 수는 합 리스트의 세번째 원소
augend <- function(s) { head(tail(tail(s))) }

# 리스트의 첫 원소가 "*"이면 합
is_product <- function(x) {
  is_pair(x) && identical(head(x), "*")
}

# 곱하는 수는 곱 리스트의 두번째 원소
multiplier <- function(s) { head(tail(s)) }

# 곱해지는 수는 곱 리스트의 세번째 원소
multiplicand <- function(s) { head(tail(tail(s))) }
```

```
deriv(List("+", "x", 3), "x")
```

```
## [ +, [ 1, [ 0, NULL ] ] ]
```

```
number_equal <- function(exp, num) {
  is.numeric(exp) && identical(exp, num)
}


make_sum <- function(a1, a2) {
  if (number_equal(a1, 0)) { # a1이 0
    a2
  } else if (number_equal(a2, 0)) { # a2가 0
    a1
  } else if (is.numeric(a1) && is.numeric(a2)) { # a1, a2 숫자 -> 간단히
    a1 + a2
  } else {
    List("+", a1, a2)
  }
}

make_product <- function(m1, m2) {
  if (number_equal(m1, 0) || number_equal(m2, 0)) {
```

```
    0
  } else if (number_equal(m1, 1)) {
    m2
  } else if (number_equal(m2, 1)) {
    m1
  } else if (is.numeric(m1) && is.numeric(m2)) {
    m1 * m2
  } else {
    List("*", m1, m2);
  }
}
```

식 간단히 하기

2.3.3. EX: Representing Sets

순서 없는 리스트로 집합 표현

- 집합의 데이터 추상화: `union_set`, `intersection_set`, `is_element_of_set`, `adjoin_set`

```
is_element_of_set <- function(x, set) {
  if (is.null(set)) {
    FALSE
  } else if (identical(x, head(set))) {
    TRUE
  } else {
    is_element_of_set(x, tail(set))
  }
}

adjoin_set <- function(x, set){
  if (is_element_of_set(x, set)) {
    set
  } else {
    pair(x, set)
  }
}

intersection_set <- function(set1, set2) {
  if (is.null(set1) || is.null(set2)) {
    NULL
  } else if (is_element_of_set(head(set1), set2)) {
    pair(head(set1), intersection_set(tail(set1), set2))
  } else {
    intersection_set(tail(set1), set2)
  }
}
```

```
intersection_set(
   adjoin_set(10, adjoin_set(20, adjoin_set(30, NULL))),
   adjoin_set(10, adjoin_set(15, adjoin_set(20, NULL))))
```

```
## [ 10, [ 20, NULL ] ]
```

순서 있는 리스트로 표현

- 원소 간 대소 비교가 가능해야 함
- 속도 향상: $\Theta(n^2)$ -> $\Theta(n)$
  - 최악의 경우: 찾으려는 원소가 리스트의 맨 끝 (= 순서 없는 리스트)
  - 평균적으로 리스트의 절반을 검색

```
is_element_of_set <- function(x, set) {
  if (is.null(set)) {
    FALSE
  } else if (x == head(set)) {
    TRUE
  } else if (x < head(set)) {
    FALSE
  } else {
    is_element_of_set(x, tail(set)) # x > head(set)
  }
}

intersection_set <- function(set1, set2) {
  if (is.null(set1) || is.null(set2)) {
    NULL
  } else {
    x1 <- head(set1)
    x2 <- head(set2)
    if (x1 == x2) {
      pair(x1, intersection_set(tail(set1), tail(set2)))
    } else if (x1 < x2) {
      intersection_set(tail(set1), set2) ## 방향 주의
    } else {
      intersection_set(set1, tail(set2))
    }
  }
}
```

```
intersection_set(List(10, 20, 30), List(10, 15, 20))
```

```
## [ 10, [ 20, NULL ] ]
```

이진 트리로 표현

- 트리의 각 노드 = 집합의 원소
- 노드의 왼쪽 부분 트리 항목들은 자신보다 작음
- 노드의 오른쪽 부분 트리 항목들은 자신보다 큼

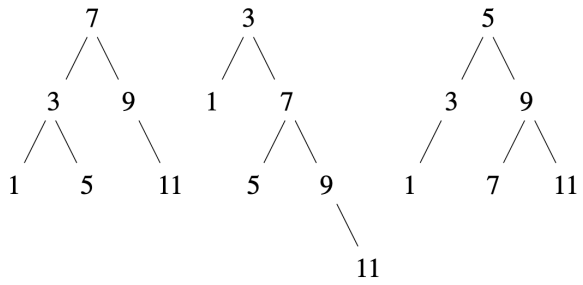

Figure 1: 집합 $\{1, 3, 5, 7, 9, 11\}$을 표현하는 이진 트리들

1. `is_element_of_set`의 속도 향상:

- `x` < 최상위 노드 -> 왼쪽 부분 트리만 검색
- `x` > 최상위 노드 -> 오른쪽 부분 트리만 검색
- 트리가 균형잡혀 있다면 크기가 $\frac{n}{2}$인 트리를 검색하는 문제로 줄어듦 -> $\Theta(\log n)$
- 트리의 노드는 원소가 3개인 리스트로 표현
  - `List(<entry>, <left_subtree>, <right_subtree>)`

```
entry <- function(tree) {head(tree)}
left_branch <- function(tree) {head(tail(tree))}
right_branch <- function(tree) {head(tail(tail(tree)))}
make_tree <- function(entry, left, right) {List(entry, left, right)}
```

```
is_element_of_set <- function(x, set) {
  if (is.null(set)) {
    FALSE
  } else if (x == entry(set)) {
    TRUE
  } else if (x < entry(set)) {
    is_element_of_set(x, left_branch(set))
  } else {
    is_element_of_set(x, right_branch(set))
  }
}
```

2. `adjoin_set`의 속도 향상:

- `x`를 항목에 추가할 때 양 부분 트리 중 하나로 내려보냄
- 균형잡힌 트리일 경우 $\Theta(\log n)$

```
adjoin_set <- function(x, set) {
  if (is.null(set)) {
    make_tree(x, NULL, NULL)
  } else if (x == entry(set)) {
    set
  } else if (x < entry(set)) {
    make_tree(entry(set),
              adjoin_set(x, left_branch(set)),
              right_branch(set))
  } else {
    make_tree(entry(set),
              left_branch(set),
              adjoin_set(x, right_branch(set)))
  }
}
```

- 균형이 깨진 트리 -> 해결책:
  - `adjoin_set` 연산 시 트리를 균형잡힌 트리로 변환
  - 별도의 자료 구조 사용

집합과 정보 검색(information retrieval)

- 데이터베이스 관리 시스템 -> 효율적 접근 방법 필요: key

- 순서 없는 리스트를 사용한 정보 검색 구현

```
make_record <- function(key, data) {pair(key, data)}
key <- function(record) {head(record)}
data <- function(record) {tail(record)}

lookup <- function(given_key, set_of_records) {
  if (is.null(set_of_records)) {
    FALSE
  } else if (identical(given_key, key(head(set_of_records)))) {
    head(set_of_records)
  } else {
    lookup(given_key, tail(set_of_records))
  }
}

lookup(3, List(make_record(2, "Venus"),
               make_record(5, "Jupiter"),
               make_record(4, "Mars"),
               make_record(3, "Earth"),
               make_record(6, "Saturn")))
```

```
## [ 3, Earth ]
```

2.3.4. EX: Huffman Encoding Trees

- 기호의 이진 부호화:
  - 고정 길이 부호(fixed-length codes)
    - A=000, B=001, C=010, D=011
    - E=100, F=101, G=110, H=111
  - 가변 길이 부호(variable-length codes)
    - 자주 쓰이는 기호에 더 짧은 부호 배정
    - A=0, B=100, C=1010, D=1011
    - E=1100, F=1101, G=1110, H=1111
    - 앞자리 부호(prefix code): 어느 기호의 부호도 다른 기호의 부호의 앞자리가 되지 않는 부호화
- 허프먼 부호(Huffman code): 빈출 기호에 가장 짧은 부호를 배정하는 가변 길이 부호

허프먼 부호 트리 만들기

- 핵심: 빈도가 낮은 기호를 뿌리 노드에서 먼 노드로 배치
  1. 기호와 상대도수를 담은 이파리 노드 생성
  2. 가중치(상대도수)가 가장 낮은 두 이파리 노드 병합, 내부 노드 생성 (가중치 = 상대도수의 합)
  3. 병합된 노드를 이파리 노드 취급하여 반복
- 이파리 노드의 표현

```
make_leaf <- function(symbol, weight) {
  List("leaf", symbol, weight)
}

is_leaf <- function(object) {
  identical(head(object), "leaf")
```

```
}

symbol_leaf <- function(x) {head(tail(x))}
weight_leaf <- function(x) {head(tail(tail(x)))}
```

- 트리의 표현

```
make_code_tree <- function(left, right) {
  List("code_tree", left, right,
       appnd(symbols(left), symbols(right)),
       weight(left) + weight(right))
}
```

- 선택자

```
left_branch <- function(tree) {head(tail(tree))}
right_branch <- function(tree) {head(tail(tail(tree)))}

symbols <- function(tree) {
  if (is_leaf(tree)) {
    List(symbol_leaf(tree))
  } else {
    head(tail(tail(tail(tree))))
  }
}

weight <- function(tree) {
  if (is_leaf(tree)) {
    weight_leaf(tree)
  } else {
    head(tail(tail(tail(tail(tree)))))
  }
}
```

- 예제

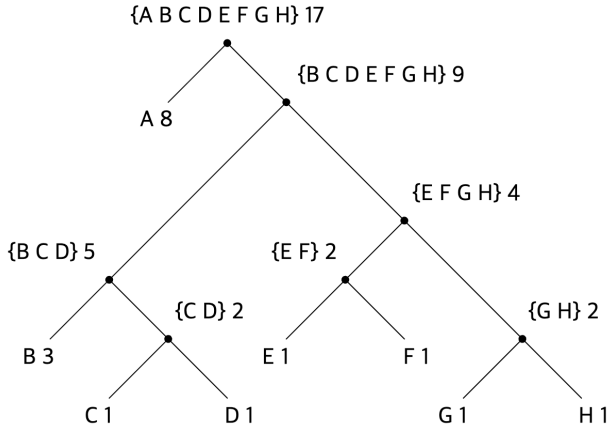

Figure 3: 허프먼 부호 트리

```
leaf_A <- make_leaf("A", 8)
leaf_B <- make_leaf("B", 3)
```

```
leaf_C <- make_leaf("C", 1)
leaf_D <- make_leaf("D", 1)
leaf_E <- make_leaf("E", 1)
leaf_F <- make_leaf("F", 1)
leaf_G <- make_leaf("G", 1)
leaf_H <- make_leaf("H", 1)

subtree_CD      <- make_code_tree(leaf_C, leaf_D)
subtree_EF      <- make_code_tree(leaf_E, leaf_F)
subtree_GH      <- make_code_tree(leaf_G, leaf_H)
subtree_EFGH    <- make_code_tree(subtree_EF, subtree_GH)
subtree_BCD     <- make_code_tree(leaf_B, subtree_CD)
subtree_BCDEFGH <- make_code_tree(subtree_BCD, subtree_EFGH)
tree_ABCDEFGH   <- make_code_tree(leaf_A, subtree_BCDEFGH)
```

```
subtree_CD
```

```
## [ code_tree, [ [ leaf, [ C, [ 1, NULL ] ] ], [ [ leaf, [ D, [ 1, NULL ] ] ], [ [ C, [
↪  D, NULL ] ], [ 2, NULL ] ] ] ] ]
```

- 복호화 함수
  - 0이면 왼쪽, 1이면 오른쪽으로 이동

```
choose_branch <- function(bit, branch) {
  if (bit == 0) {
    left_branch(branch)
  } else if (bit == 1) {
    right_branch(branch)
  } else {
    stop("bad bit -- choose branch")
  }
}

decode <- function(bits, tree) {
  decode_1 <- function(bits, current_branch) {
    if (is.null(bits)) {
      NULL
    } else {
      next_branch <- choose_branch(head(bits),
                                   current_branch)
      if (is_leaf(next_branch)) {
        pair(symbol_leaf(next_branch),
             decode_1(tail(bits), tree))
      } else {
        decode_1(tail(bits), next_branch)
      }
    }
  }
  decode_1(bits, tree)
}
```

```
decode(List(1, 0, 0, 0, 1, 0, 1, 0), tree_ABCDEFGH)
```

```
## [ B, [ A, [ C, NULL ] ] ]
```

가중치 정렬

- 중간 노드의 기호 집합에서 가장 작은 항목을 찾는 연산 여러 번 수행됨
  - 순서 있는 리스트로 표현 (ex. 2.61)

```
adjoin_set <- function(x, set) {
  if (is.null(set)) {
    List(x)
  } else if (weight(x) < weight(head(set))) {
    pair(x, set)
  } else {
    pair(head(set), adjoin_set(x, tail(set)))
  }
}

make_leaf_set <- function(pairs) {
  if (is.null(pairs)) {
    NULL
  } else {
    first_pair <- head(pairs)
    adjoin_set(make_leaf(head(first_pair), head(tail(first_pair))),
               make_leaf_set(tail(pairs)))
  }
}
```

```
make_leaf_set(List(List("A", 4),
                   List("B", 2),
                   List("C", 1),
                   List("D", 1) )
              )
```

```
## [ [ leaf, [ D, [ 1, NULL ] ] ], [ [ leaf, [ C, [ 1, NULL ] ] ], [ [ leaf, [ B, [ 2,
↪  NULL ] ] ], [ [ leaf, [ A, [ 4, NULL ] ] ], NULL ] ] ] ]
```

2.4. Multiple Representations for Abstract Data

2.4.1. Representations for Complex Numbers

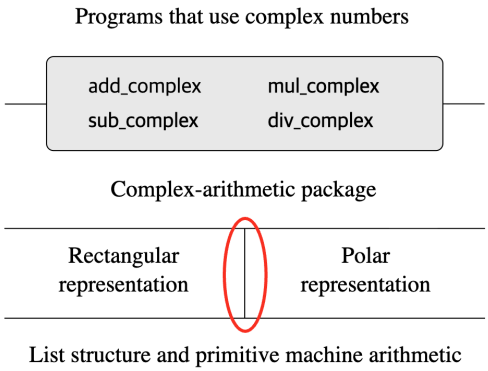

Figure 1: 복소수 체의 데이터 추상화 장벽 (유리수 집합과 비교)

- 복소수 표현하기
  - 직교좌표: $z = x + iy$
  - 극좌표: $z = r\exp(iA)$

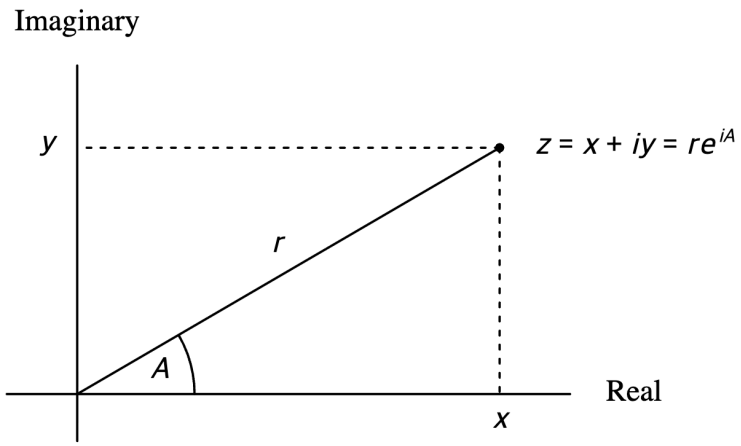

Figure 2: 복소평면

- 선택자: `real_part`, `imag_part`, `magnitude`, `angle`
- 생성자: `make_from_real_imag`, `make_from_mag_ang`

```
# 필요충분 조건
make_from_real_imag(real_part(z), imag_part(z)) == z
make_from_mag_ang(magnitude(z), angle(z) == z)
```

- 복소수 연산

$$\Re(z_1 + z_2) = \Re(z_1) + \Re(z_2), \quad \Im(z_1 + z_2) = \Im(z_1) + \Im(z_2)$$
$$|z_1 \cdot z_2| = |z_1| \cdot |z_2|, \quad \angle z_1 \cdot z_2 = \angle z_1 + \angle z_2$$

$$r = |z| = \sqrt{x^2 + y^2}, \quad a = \angle z = \operatorname{atan2}(y, x)$$
$$x = r\cos a, \quad y = r\sin b$$

```
add_complex <- function(z1, z2) {
  make_from_real_imag(real_part(z1) + real_part(z2),
                      imag_part(z1) + imag_part(z2))
}
sub_complex <- function(z1, z2) {
  make_from_real_imag(real_part(z1) - real_part(z2),
                      imag_part(z1) - imag_part(z2))
}
mul_complex <- function(z1, z2) {
  make_from_mag_ang(magnitude(z1) * magnitude(z2),
                    angle(z1) + angle(z2))
}
div_complex <- function(z1, z2) {
  make_from_mag_ang(magnitude(z1) / magnitude(z2),
                    angle(z1) - angle(z2))
}
```

- 직교좌표 표현: `pair(<실수부>, <허수부>)`

```
real_part <- function(z) {head(z)}
imag_part <- function(z) {tail(z)}

magnitude <- function(z) {
  sqrt(square(real_part(z)) + square(imag_part(z)))
}
angle <- function(z) {
  atan2(imag_part(z), real_part(z))
}

make_from_real_imag <- function(x, y) {
  pair(x, y)
}
make_from_mag_ang <- function(r, a) {
  pair(r * cos(a), r * sin(a))
}
```

```
my_co_num_1 <- make_from_real_imag(2.5, -0.5)
my_co_num_2 <- make_from_real_imag(2.5, -0.5)
result <- add_complex(my_co_num_1,
                      mul_complex(my_co_num_2,
                                  my_co_num_2))
imag_part(result)
```

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

- 극좌표 표현: `pair(<크기>, <편각>)`

```
real_part <- function(z) {
  magnitude(z) * cos(angle(z))
}
imag_part <- function(z) {
  magnitude(z) * sin(angle(z))
}

magnitude <- function(z) { head(z) }
angle <- function(z) { tail(z) }

make_from_real_imag <- function(x, y) {
  pair(sqrt(square(x) + square(y)),
       atan2(y, x))
}
make_from_mag_ang <- function(r, a) {
  pair(r, a)
}
```

```
my_co_num_1 <- make_from_real_imag(2.5, -0.5)
my_co_num_2 <- make_from_real_imag(2.5, -0.5)
result <- add_complex(my_co_num_1,
                      mul_complex(my_co_num_2,
                                  my_co_num_2))
imag_part(result)
```

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

- 추상화 장벽 형성을 통해 데이터 개체에 대한 구체적 표현의 결정을 마지막 순간까지 연기 -> 시스템 설계에서 최대한의 유연성을 유지

2.4.2. Tagged Data

- 직교좌표와 극좌표 표현 골라서 사용 -> 두 표현을 구분할 꼬리표(tag) 혹은 자료형(data type) 필요

```
attach_tag <- function(type_tag, contents){
  pair(type_tag, contents)
}
type_tag <- function(datum){
  if (is_pair(datum)){
    head(datum)
  } else {
    stop("bad tagged datum -- type_tag")
  }
}
contents <- function(datum){
  if (is_pair(datum)){
    tail(datum)
  } else {
    stop("bad tagged datum -- contents")
  }
}
```

- 술어

```
is_rectangular <- function(z){
  identical(type_tag(z), "rectangular")
}
is_polar <- function(z){
  identical(type_tag(z), "polar")
}
```

- 직교좌표, 극좌표 표현 개편

```
real_part_rectangular <- function(z) { head(z) }
imag_part_rectangular <- function(z) { tail(z) }
magnitude_rectangular <- function(z) {
  sqrt(square(real_part(z)) + square(imag_part(z)))
}
angle_rectangular <- function(z) {
  atan2(imag_part_rectangular(z),
        real_part_rectangular(z))
}
make_from_real_imag_rectangular <- function(x, y) {
  attach_tag("rectangular", pair(x, y))
}
make_from_mag_ang_rectangular <- function(r, a) {
  attach_tag("rectangular",
             pair(r * cos(a), r * sin(a)))
}
```

```
real_part_polar <- function(z) {
  magnitude_polar(z) * cos(angle_polar(z))
```

```
}
imag_part_polar <- function(z) {
  magnitude_polar(z) * sin(angle_polar(z))
}
magnitude_polar <- function(z) { head(z) }
angle_polar <- function(z) { tail(z) }
make_from_real_imag_polar <- function(x, y) {
  attach_tag("polar",
             pair(sqrt(square(x) + square(y)),
                  atan2(y, x)))
}
make_from_mag_ang_polar <- function(r, a) {
  attach_tag("polar", pair(r, a))
}
```

```
make_from_real_imag <- function(x, y) {
  make_from_real_imag_rectangular(x, y)
}
make_from_mag_ang <- function(r, a) {
  make_from_mag_ang_polar(r, a)
}
```

- 일반 선택자의 구현: 꼬리표를 보고 자료형에 맞는 함수 호출

```
real_part <- function(z) {
  if (is_rectangular(z) ) {
    real_part_rectangular(contents(z))
  } else if (is_polar(z) ) {
    real_part_polar(contents(z))
  } else {
    stop("unknown type -- real_part")
  }
}
imag_part <- function(z) {
  if (is_rectangular(z) ) {
    imag_part_rectangular(contents(z))
  } else if (is_polar(z) ) {
    imag_part_polar(contents(z))
  } else {
    stop("unknown type -- imag_part")
  }
}

magnitude <- function(z) {
  if (is_rectangular(z) ) {
    magnitude_rectangular(contents(z))
  } else if (is_polar(z) ) {
    magnitude_polar(contents(z))
  } else {
    stop("unknown type -- magnitude")
  }
}
angle <- function(z) {
  if (is_rectangular(z) ) {
```

```
    angle_rectangular(contents(z))
  } else if (is_polar(z) ) {
    angle_polar(contents(z))
  } else {
    stop("unknown type -- angle")
  }
}
```

```
z <- make_from_mag_ang(3.0, 0.7)
imag_part(z)
```

```
## [1] 1.932653
```

- 두 표현 모두 지원하는 경우의 장점:
  - 데이터 표현과 연산을 분리 -> 새로운 표현이나 연산을 기존 코드 수정 없이 독립적으로 추가할 수 있음(additivity)
- 자료형에 따라 다른 동작을 하도록 일반함수로 구현 가능

2.4.3. Data-Directed Programming and Additivity

- 자료형 기반 파송(dispatching on type): 자료 개체의 형을 따져 적절한 함수를 호출
- 일반 선택자 구현 -> 일반함수가 모든 가능한 표현을 알아야 함
  - 새로운 표현형을 추가하면 이들 함수를 모두 고쳐야 함
  - 개별 표현이 독립적이라도 함수 이름의 충돌이 있을 수 있음

  -> 비가산적

자료 지시형 프로그래밍

- 일반 함수에 대해 2차원 표를 관리
  - 행: 가능한 연산
  - 열: 가능한 자료형

	Types	
Operations	Polar	Rectangular
real_part	real_part_polar	real_part_rectangular
imag_part	imag_part_polar	imag_part_rectangular
magnitude	magnitude_polar	magnitude_rectangular
angle	angle_polar	angle_rectangular

Figure 4: 앞 절 복소수 산술 시스템의 선택자

- 필요 함수 (실제 구현 in 3장)
  - `Put(<op>, <type>, <item>)`
    - `<item>`을 `<op>`(행), `<type>`(열)에 따라 표에 등록
  - `Get(<op>, <type>)`
    - `<op>`(행), `<type>`(열)으로 표에 등록된 아이템 반환 (없으면 `FALSE`)

```
install_rectangular_package <- function() {
  # internal functions
```

```
  real_part <- function(z) {head(z)}
  imag_part <- function(z) {tail(z)}
  magnitude <- function(z) {
    sqrt(square(real_part(z)) + square(imag_part(z)))
  }
  angle <- function(z) {
    atan2(imag_part(z), real_part(z))
  }
  make_from_real_imag <- function(x, y) { pair(x, y) }
  make_from_mag_ang <- function(r, a) {
    pair(r * cos(a), r * sin(a))
  }

  # interface to the rest of the system
  tag <- function(x) { attach_tag("rectangular", x)  }
  Put("real_part", List("rectangular"), real_part)
  Put("imag_part", List("rectangular"), imag_part)
  Put("magnitude", List("rectangular"), magnitude)
  Put("angle", List("rectangular"), angle)
  Put("make_from_real_imag", "rectangular",
      function(x, y) { tag(make_from_real_imag(x, y)) }
     )
  Put("make_from_mag_ang", "rectangular",
      function(r, a) { tag(make_from_mag_ang(r, a)) }
     )
  "done"
}

install_rectangular_package()
```

```
## [1] "done"
```

- 극좌표 표현 패키지

```
install_polar_package <- function() {
  # internal functions
  real_part <- function(z) {
    magnitude(z) * cos(angle(z))
  }
  imag_part <- function(z) {
    magnitude(z) * sin(angle(z))
  }
  magnitude <- function(z) { head(z) }
  angle <- function(z) { tail(z) }
  make_from_real_imag <- function(x, y) {
    pair(sqrt(square(x) + square(y)),
         atan2(y, x))
  }
  make_from_mag_ang <- function(r, a) { pair(r, a) }

  # interface to the rest of the system
  tag <- function(x) { attach_tag("polar", x) }
  Put("real_part", List("polar"), real_part)
  Put("imag_part", List("polar"), imag_part)
  Put("magnitude", List("polar"), magnitude)
```

```
  Put("angle", List("polar"), angle)
  Put("make_from_real_imag", "polar",
      function(x, y) { tag(make_from_real_imag(x, y)) }
     )
  Put("make_from_mag_ang", "polar",
      function(r, a) { tag(make_from_mag_ang(r, a)) }
     )
  "done"
}

install_polar_package()
```

```
## [1] "done"
```

- 일반 선택자

```
apply_generic <- function(op, args) {
  type_tags <- map(type_tag, args)
  fun <- Get(op, type_tags)
  if(!identical(fun, FALSE)) {
    do_call(fun, map(contents, args))
  } else{
    stop("no method for these types -- apply_generic")
  }
}

do_call <- function(what, args) {
  if (!is_pair(args)) {
    stop("second argument must be a List")
  } else {
    args <- flatten_list(args)
    do.call(what, args) # R primitive
  }
}
```

```
real_part <- function(z) { apply_generic("real_part", List(z)) }
imag_part <- function(z) { apply_generic("imag_part", List(z)) }
magnitude <- function(z) { apply_generic("magnitude", List(z)) }
angle     <- function(z) { apply_generic("angle", List(z)) }
```

- 생성자

```
make_from_real_imag <- function(x, y) {
  Get("make_from_real_imag", "rectangular")(x, y)
}
make_from_mag_ang <- function(r, a) {
  Get("make_from_mag_ang", "polar")(r, a)
}
```

```
z  <- make_from_real_imag(1.0, 4.5)
z2 <- add_complex(z, z)
imag_part(z2); real_part(z2)
```

```
## [1] 9
```

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

메시지 전달

- 자료 지시형 프로그래밍의 또다른 기법
- 자료형 대신 연산 이름을 기반으로 파송을 처리하는 함수 개체 생성

```
make_from_real_imag <- function(x, y) {
  dispatch <- function(op) {
    if (identical(op, "real_part") ) {
      x
    } else if (identical(op, "imag_part") ) {
      y
    } else if (identical(op, "magnitude") ) {
      sqrt(square(x) + square(y))
    } else if (identical(op, "angle") ) {
      atan2(y, x)
    } else {
      stop("unknown op -- make_from_real_imag")
    }
  }
  dispatch
}
```

- 순서쌍의 메시지 전달식 정의

```
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) }
```

- 일반선택자 적용 함수 변경

```
apply_generic <- function(op, arg) { head(arg)(op) }
```

- 선택자는 이전과 동일

```
real_part <- function(z) { apply_generic("real_part", List(z)) }
imag_part <- function(z) { apply_generic("imag_part", List(z)) }
magnitude <- function(z) { apply_generic("magnitude", List(z)) }
angle     <- function(z) { apply_generic("angle", List(z)) }
```

```
z <- make_from_real_imag(3, 4)
real_part(z); imag_part(z); magnitude(z); angle(z)
```

```
## [1] 3
## [1] 4
## [1] 5
## [1] 0.9272952
```

2.5. Systems with Generic Operations

- 일반 함수를 통해 데이터 연산을 수행하는 코드를 여러 표현 방식에 연결
- 다른 종류의 **전달인자**에 대한 일반 연산을 같은 방식으로 정의할 수 있음

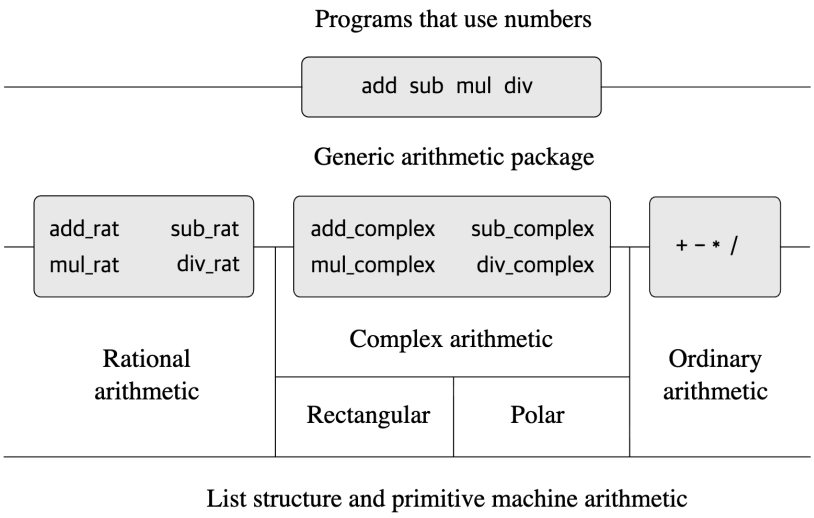

Figure 1: 일반 산술 시스템

2.5.1. Generic Arithmetic Operations

- 보통의 수에는 `+`, 유리수에는 `add_rat`, 복소수에는 `add_complex` 적용

```
add <- function(x, y) { apply_generic("add", List(x, y)) }
sub <- function(x, y) { apply_generic("sub", List(x, y)) }
mul <- function(x, y) { apply_generic("mul", List(x, y)) }
div <- function(x, y) { apply_generic("div", List(x, y)) }
```

- 보통 수 산술 연산 패키지

```
install_ordinary_number_package <- function() {
  tag <- function(x) {
    attach_tag("ordinary_number", x)
  }
  # generic arithmetic functions
  Put("add", List("ordinary_number", "ordinary_number"),
      function(x, y) { tag(x + y) } )
  Put("sub", List("ordinary_number", "ordinary_number"),
      function(x, y) { tag(x - y) } )
  Put("mul", List("ordinary_number", "ordinary_number"),
      function(x, y) { tag(x * y) } )
  Put("div", List("ordinary_number", "ordinary_number"),
      function(x, y) { tag(x / y) } )
```

```
  # constructor
  Put("make", "ordinary_number",
      function(x) { tag(x) } )
  "done"
}

install_ordinary_number_package()
```

```
## [1] "done"
```

- 생성자

```
make_ordinary_number <- function(n) {
  Get("make", "ordinary_number")(n)
}
```

- 유리수 산술 (2.1.절 재활용)

```
install_rational_package <- function() {
  # internal functions
  numer    <- function(x) { head(x) }
  denom    <- function(x) { tail(x) }
  make_rat <- function(n, d) {
    g <- gcd(n, d)
    pair(n / g, d / g)
  }
  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))
  }
  # interface to rest of the system
  tag <- function(x) {
    attach_tag("rational", x)
  }
  Put("add", List("rational", "rational"),
      function(x, y) { tag(add_rat(x, y)) } )
  Put("sub", List("rational", "rational"),
      function(x, y) { tag(sub_rat(x, y)) } )
  Put("mul", List("rational", "rational"),
      function(x, y) { tag(mul_rat(x, y)) } )
  Put("div", List("rational", "rational"),
      function(x, y) { tag(div_rat(x, y)) } )
  Put("make", "rational",
```

```
      function(n, d) { tag(make_rat(n, d)) } )
  "done"
}

install_rational_package()
```

```
## [1] "done"
```

- 생성자

```
make_rational <- function(n, d) {
  Get("make", "rational")(n, d)
}
```

- 복소수 산술 (2.4.절 재활용)

```
install_complex_package <- function() {
  # imported functions from rectangular and polar packages
  make_from_real_imag <- function(x, y) {
    Get("make_from_real_imag", "rectangular")(x, y)
  }
  make_from_mag_ang <- function(r, a) {
    Get("make_from_mag_ang", "polar")(r, a)
  }
  # internal functions
  add_complex <- function(z1, z2) {
    make_from_real_imag(real_part(z1) + real_part(z2),
                        imag_part(z1) + imag_part(z2))
  }
  sub_complex <- function(z1, z2) {
    make_from_real_imag(real_part(z1) - real_part(z2),
                        imag_part(z1) - imag_part(z2))
  }
  mul_complex <- function(z1, z2) {
    make_from_mag_ang(magnitude(z1) * magnitude(z2),
                      angle(z1) + angle(z2))
  }
  div_complex <- function(z1, z2) {
    make_from_mag_ang(magnitude(z1) / magnitude(z2),
                      angle(z1) - angle(z2))
  }
  # interface to rest of the system
  tag <- function(z) { attach_tag("complex", z) }
  Put("add", List("complex", "complex"),
      function(z1, z2) { tag(add_complex(z1, z2)) } )
  Put("sub", List("complex", "complex"),
      function(z1, z2) { tag(sub_complex(z1, z2)) } )
  Put("mul", List("complex", "complex"),
      function(z1, z2) { tag(mul_complex(z1, z2)) } )
  Put("div", List("complex", "complex"),
      function(z1, z2) { tag(div_complex(z1, z2)) } )
  Put("make_from_real_imag", "complex",
      function(x, y) { tag(make_from_real_imag(x, y)) } )
  Put("make_from_mag_ang", "complex",
      function(r, a) { tag(make_from_mag_ang(r, a)) } )
```

```
  "done"
}

install_rectangular_package();
```

```
## [1] "done"
```

```
install_polar_package();
```

```
## [1] "done"
```

```
install_complex_package()
```

```
## [1] "done"
```

- 생성자

```
make_complex_from_real_imag <- function(x, y) {
  Get("make_from_real_imag", "complex")(x, y)
}
make_complex_from_mag_ang <- function(r, a) {
  Get("make_from_mag_ang", "complex")(r, a)
}
```

- 2.4.3.과 비교: 2수준 태그 시스템

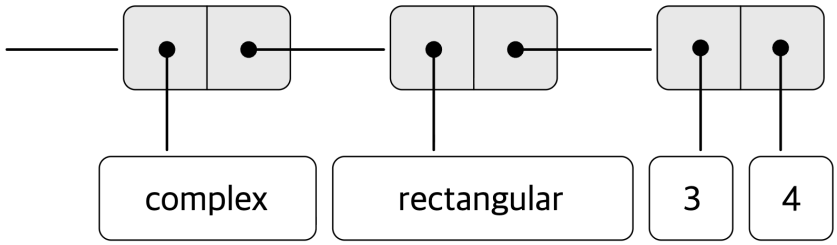

Figure 2: 2수준 태그 시스템

```
r <- make_complex_from_real_imag(4, 3)
p <- make_complex_from_mag_ang(5, 0.5)

add(r, p);    # results in a complex number in rectangular coord
```

```
## [ complex, [ rectangular, [ 8.387913, 5.397128 ] ] ]
```

```
mul(r, p)     # results in a complex number in polar coordinates
```

```
## [ complex, [ polar, [ 25, 1.143501 ] ] ]
```

2.5.2. Combining Data of Different Types

```
Put("add", List("ordinary_number", "ordinary_number"),
    function(x, y) { tag(x + y) } ) # 보통 수 + 보통 수
Put("add", List("rational", "rational"),
    function(x, y) { tag(add_rat(x, y)) } ) # 유리수 + 유리수
Put("add", List("complex", "complex"),
    function(z1, z2) { tag(add_complex(z1, z2)) } ) # 복소수 + 복소수
```

강제 형변환

- 한 자료형의 개체를 다른 자료형의 개체로 취급
  - e.g. 보통 수 = 허수부가 0인 복소수

```
ordinary_number_to_complex <- function(n) {
  make_complex_from_real_imag(contents(n), 0)
}

put_coercion("ordinary_number", "complex",
             ordinary_number_to_complex)
```

```
## [1] "done"
```

- apply_generic 수정

```
apply_generic <- function(op, args) {
  type_tags <- map(type_tag, args)
  fun <- Get(op, type_tags)
  if (!identical(fun, FALSE) ) {
    do_call(fun, map(contents, args))
  } else if (len(args) == 2 ) {
    type1 <- head(type_tags)
    type2 <- head(tail(type_tags))
    a1 <- head(args)
    a2 <- head(tail(args))
    t1_to_t2 <- get_coercion(type1, type2)
    t2_to_t1 <- get_coercion(type2, type1)
    if (!identical(t1_to_t2, FALSE) ) {
      apply_generic(op, List(t1_to_t2(a1), a2))
    } else if (!identical(t2_to_t1, FALSE) ) {
      apply_generic(op, List(a1, t2_to_t1(a2)))
    } else {
      stop(paste(capture.output(print(List(op, type_tags))), "no method for these types A
      ↪  -- apply_generic"))
    }
  } else {
    stop(paste(capture.output(print(List(op, type_tags))), "no method for these types B
    ↪  -- apply_generic"))
  }
}
```

```
z <- make_complex_from_real_imag(4, 3)
n <- make_ordinary_number(7)
add(z, n)
```

```
## [ complex, [ rectangular, [ 11, 3 ] ] ]
```

자료형의 계층 구조

- 하위형(subtype), 상위형(supertype)
  - 승격(raise): 정수 -> 유리수, 유리수 -> 실수
  - 상속(inheritance): `real_part(<정수>)`
  - 강등(lower): $(2 + 3i) + (4 - 3i) = 6$

- 계층 구조가 부적절한 경우: 복수 개의 하위/상위 자료형

2.5.3. EX: Symbolic Algebra

- 피연산자에 적용되는 연산자의 트리 구조
  - 원초 개체(상수, 변수)로부터 복합 개체 구성하기
- 자료형: 선형결합, 다항식, 유리함수, 삼각함수 등
- e.g. $x^2 \sin(y^2+1) + x\cos 2y + \cos(y^3 - 2y^2)$
  - 최상위 자료형: x의 다항식
  - x의 다항식의 계수: 삼각함수
  - 삼각함수의 인수: y의 다항식
  - y의 다항식의 계수: 정수

다항식 산술 연산

- 변수가 하나인 다항식만 고려
  - 단, 계수는 어떤 다른 변수의 다항식일 수 있음
- 덧셈, 곱셈만 고려
- 두 다항식을 결합: 같은 변수의 다항식끼리

- 선택자: `variable`, `term_list` - 변수와 항들의 리스트
- 생성자: `make_poly`

```
add_poly <- function(p1, p2) {
  if (is_same_variable(variable(p1), variable(p2))) {
    make_poly(variable(p1),
              add_terms(term_list(p1), term_list(p2)))
  } else {
    stop(paste(capture.output(print(List(p1, p2))),
               "polys not in same var -- add_poly"))
  }
}

mul_poly <- function(p1, p2) {
  if (is_same_variable(variable(p1), variable(p2)) ) {
    make_poly(variable(p1),
              mul_terms(term_list(p1), term_list(p2)))
  } else {
    stop(paste(capture.output(print(List(p1, p2))),
               "polys not in same var -- mul_poly"))
  }
}
```

```
is_same_variable <- function(v1, v2) {  # Sect 2.3.2
  is_variable(v1) && is_variable(v2) && identical(v1, v2)
}

is_variable <- function(x) { is.character(x) }
```

- 일반 산술 시스템에 통합

```
# template
install_polynomial_package <- function() {
  # internal functions

  ## representation of poly
  make_poly <- function(variable, term_list) {
    pair(variable, term_list)
  }
  variable <- function(p) { head(p) }
  term_list <- function(p) { tail(p) }
  ### <is_same_variable, is_variable>

  ## representation of terms and term lists
  adjoin_term <- function(term, term_list) { ... }
  ### <more functions...>
  coeff <- function(term) { ... }

  add_poly <- function(p1, p2) { ... }
  ### <functions used by add-poly>

  mul_poly <- function(p1, p2) { ... }
  ### <functions used by mul-poly>

  # interface to rest of the system
  tag <- function(p) { attach_tag("polynomial", p) }
  Put("add", List("polynomial", "polynomial"),
      function(p1, p2) { tag(add_poly(p1, p2)) } )
  Put("mul", List("polynomial", "polynomial"),
      function(p1, p2) { tag(mul_poly(p1, p2)) } )
  Put("make", "polynomial",
      function(variable, terms) {
            tag(make_poly(variable, terms))
      }
     )
  "done"
}
```

다항식의 덧셈

- 차수가 같은 두 항에 대해 차수는 같고 계수는 두 항의 계수의 합인 새 항 생성
- 항들의 리스트
  - `the_empty_termlist`: 생성자
  - `adjoin_term`: 항 리스트에 새 항 추가
  - `is_empty_termlist`: 술어
  - `first_term`, `rest_terms`: 선택자
- 항에 대한 함수
  - `make_term`: 생성자
  - `order`(차수), `coeff`(계수): 선택자

```
add_terms <- function(L1, L2) {
    if (is_empty_termlist(L1) ) {
```

```
      L2
    } else if (is_empty_termlist(L2) ) {
      L1
    } else {
      t1 <- first_term(L1)
      t2 <- first_term(L2)
      if (order(t1) > order(t2) ) {
        adjoin_term(t1, add_terms(rest_terms(L1), L2))
      } else if (order(t1) < order(t2) ) {
        adjoin_term(t2, add_terms(L1, rest_terms(L2)))
      } else {
        adjoin_term(make_term(order(t1),
                              add(coeff(t1), coeff(t2))), # t1의 차수, 계수의 합
                    add_terms(rest_terms(L1),
                              rest_terms(L2))) # 나머지 연산
      }
    }
}
```

- 동차항의 계수 더할 때 일반함수 add의 사용

다항식의 곱셈

- 앞 리스트의 각 항을 뒤 리스트의 모든 항에 곱함 (mul_term_by_all_terms)
  - 두 항의 곱: 계수 = (두 항의 계수의 곱), 차수 = (두 항의 차수의 합)인 항
  - 결과로 얻은 항 리스트들을 모두 더해 하나의 리스트로 만듦

```
mul_terms <- function(L1, L2) {
  if (is_empty_termlist(L1)) {
    the_empty_termlist
  } else {
    add_terms(mul_term_by_all_terms(first_term(L1), L2),
              mul_terms(rest_terms(L1), L2))
  }
}

mul_term_by_all_terms <- function(t1, L) {
  if (is_empty_termlist(L)) {
    the_empty_termlist
  } else {
    t2 <- first_term(L)
    adjoin_term(make_term(order(t1) + order(t2),
                          mul(coeff(t1), coeff(t2))),
                mul_term_by_all_terms(t1, rest_terms(L)))
  }
}
```

- 주안점: 일반함수 add, mul -> 일반 산술 패키지가 지원하는 모든 수를 계수로 사용 가능 (+ 강제 형변환 도입)

```
Put("add", List("polynomial", "polynomial"),
    function(p1, p2) { tag(add_poly(p1, p2)) } )
```

```
## [1] "ok"
```

```
Put("mul", List("polynomial", "polynomial"),
    function(p1, p2) { tag(mul_poly(p1, p2)) } )
```

```
## [1] "ok"
```

- polynomial 자료형의 일반 연산도 가능

항 리스트의 표현

- 항 리스트: 차수를 key로 하는 계수 집합
- 2.3.3.의 집합 표현 함수 활용 (순서 리스트)
- 조밀 다항식 $x^5 + 2x^4 + 3x^2 - 2x - 5$

```
List(1, 2, 0, -2, -5)
```

- 희소 다항식 $x^100 + 2x^2 + 1$

```
List(List(100, 1), List(2, 2), List(0, 1))
```

```
adjoin_term <- function(term, term_list){
  if (is_equal_to_zero(coeff(term))) {
    term_list
  } else {
    pair(term, term_list)
  }
}

the_empty_termlist <- NULL
first_term <- function(term_list) {head(term_list)}
rest_terms <- function(term_list) {tail(term_list)}
is_empty_termlist <- function(term_list) { is.null(term_list) }
```

```
make_term <- function(order, coeff){List(order, coeff)}
order <- function(term) {head(term)}
coeff <- function(term) {head(tail(term))}
```

```
install_ordinary_number_is_equal_to_zero <- function() {
  Put("is_equal_to_zero", List("ordinary_number"),
        function(x) {x == 0} )
  "done"
}
install_ordinary_number_is_equal_to_zero()
```

```
## [1] "done"
```

```
is_equal_to_zero <- function(x) {
  apply_generic("is_equal_to_zero", List(x))
}
```

- 생성자

```
make_polynomial <- function(variable, terms){
  Get("make", "polynomial")(variable, terms)
}
```

- 전부 합치기

```
install_polynomial_package <- function() {
  # internal functions

  ## representation of poly
  make_poly <- function(variable, term_list) {
    pair(variable, term_list)
  }
  variable <- function(p) { head(p) }
  term_list <- function(p) { tail(p) }

  ## representation of terms and term lists
  adjoin_term <- function(term, term_list) {
    if (is_equal_to_zero(coeff(term)) ) {
      term_list
    } else {
      pair(term, term_list)
    }
  }
  the_empty_termlist <- NULL
  first_term <- function(term_list) { head(term_list) }
  rest_terms <- function(term_list) { tail(term_list) }
  is_empty_termlist <- function(term_list) { is.null(term_list) }
  make_term <- function(order, coeff) { List(order, coeff) }
  order <- function(term) { head(term) }
  coeff <- function(term) { head(tail(term)) }

  add_poly <- function(p1, p2) {
    if (is_same_variable(variable(p1), variable(p2)) ) {
      make_poly(variable(p1),
                add_terms(term_list(p1), term_list(p2)))
    } else {
      stop(paste(capture.output(print(List(p1, p2))),
                 "polys not in same var -- add_poly"))
    }
  }
  add_terms <- function(L1, L2) {
    if (is_empty_termlist(L1) ) {
      L2
    } else if (is_empty_termlist(L2) ) {
      L1
    } else {
      t1 <- first_term(L1)
      t2 <- first_term(L2)
      if (order(t1) > order(t2) ) {
        adjoin_term(t1, add_terms(rest_terms(L1), L2))
      } else if (order(t1) < order(t2) ) {
        adjoin_term(t2, add_terms(L1, rest_terms(L2)))
      } else {
        adjoin_term(make_term(order(t1),
                              add(coeff(t1), coeff(t2))),
                    add_terms(rest_terms(L1),
                              rest_terms(L2)))
      }
```

```
    }
  }
  mul_poly <- function(p1, p2) {
    if (is_same_variable(variable(p1), variable(p2)) ) {
      make_poly(variable(p1),
                mul_terms(term_list(p1), term_list(p2)))
    } else {
      stop(paste(capture.output(print(List(p1, p2))),
                 "polys not in same var -- mul_poly"))
    }
  }
  mul_terms <- function(L1, L2) {
    if (is_empty_termlist(L1) ) {
      the_empty_termlist
    } else {
      add_terms(mul_term_by_all_terms(first_term(L1), L2),
                mul_terms(rest_terms(L1), L2))
    }
  }
  mul_term_by_all_terms <- function(t1, L) {
    if (is_empty_termlist(L)) {
      the_empty_termlist
    } else {
      t2 <- first_term(L)
      adjoin_term(make_term(order(t1) + order(t2),
                            mul(coeff(t1), coeff(t2))),
                  mul_term_by_all_terms(t1, rest_terms(L)))
    }
  }

  # interface to rest of the system
  tag <- function(p) { attach_tag("polynomial", p) }
  Put("add", List("polynomial", "polynomial"),
      function(p1, p2) { tag(add_poly(p1, p2)) } )
  Put("mul", List("polynomial", "polynomial"),
      function(p1, p2) { tag(mul_poly(p1, p2)) } )
  Put("make", "polynomial",
      function(variable, terms) {
            tag(make_poly(variable, terms))
      }
     )
  "done"
}

install_polynomial_package()
```

```
## [1] "done"
```

- 예시

$$(4x^2 + 3x + 7)(5x^2 + 2x + 10) = 20x^4 + 23x^3 + 81x^2 + 44x + 70$$

```
p1 <- make_polynomial("x",
                List(make_term(2, make_ordinary_number(4)),
```

```
                    make_term(1, make_ordinary_number(3)),
                    make_term(0, make_ordinary_number(7))))
p2 <- make_polynomial("x",
                List(make_term(2, make_ordinary_number(5)),
                    make_term(1, make_ordinary_number(2)),
                    make_term(0, make_ordinary_number(10))))
mul(p1, p2)
```

```
## [ polynomial, [ x, [ [ 4, [ [ ordinary_number, 20 ], NULL ] ], [ [ 3, [ [
↪  ordinary_number, 23 ], NULL ] ], [ [ 2, [ [ ordinary_number, 81 ], NULL ] ], [ [ 1, [
↪  [ ordinary_number, 44 ], NULL ] ], [ [ 0, [ [ ordinary_number, 70 ], NULL ] ], NULL ]
↪  ] ] ] ] ] ]
```

기호 대수에서의 자료형 계층 구조

- 현재는 탑 구조가 아님: (계수가 y의 다항식인 x의 다항식) $\subsetneq$ (계수가 x의 다항식인 y의 다항식)
- 변수에 순서를 매기고 우선순위가 높은 변수부터 정리하여 표현하면 탑 구조 가능
  - but 새로운 자료형을 동적으로 만들어 내는 데는 적합하지 않음 (e.g. 삼각다항식)
- 대규모 대수 조작 시스템의 복잡성은 주로 다양한 자료형간의 관계에서 나옴 (형 변환의 어려움)

정리

- 어떤 자료형의 개체(다항식 개체)가 실제로는 서로 다른 자료형의 여러 개체로 구성된 복잡한 개체일 수 있음
- 일반 산술 정의에 큰 어려움 없음: by 적절한 일반 연산 설치
- 재귀적 데이터 추상화: 다항식의 구성 요소 자체가 다항식일 수 있음
- 일반 연산과 데이터 지시형 프로그래밍을 이용하여 해결

3. Modularity, Objects, and State

3.1. Assignment and Local State

3.1.1. Local State Variables

- `withdraw`: 은행 계좌 인출 구현
  - 같은 입력, 다른 결과값
  - 상태(state)를 기억하도록 해야 구현 가능

```
balance <- 100

withdraw <- function(amount) {
  if (balance >= amount) {
    assign("balance", balance - amount, inherits = TRUE)
    balance
  } else {
    "Insufficient funds"
  }
}
```

- 대입(assignment) `assign(<name>, <value>, inherits=TRUE )`

  - <name>이 나타내는 변수의 값을 <value>로 변경

```
withdraw(25);
```

```
## [1] 75
```

```
withdraw(25)
```

```
## [1] 50
```

대입의 의의

- 이전까지 프로그램에서 사용한 이름들은 불변(immutable)
- 은행 계좌와 같은 시변(time-varying) 개체의 행동을 구현하려면 개체의 상태 변수(balance)의 값을 변경할 수 있어야 함
  - "변경"은 결국 같은 이름의 변수에 새로운 값을 대입하는 것
- balance가 withdraw 내부에서만 접근 가능하도록 하는 것이 모듈화에 유리한 **지역** 상태 변수(local state variable)의 개념을 더 정확히 구현

```
make_withdraw_balance_100 <- function() {
  balance <- 100
  function(amount) {
    if (balance >= amount) {
      assign("balance", balance - amount, inherits = TRUE)
      balance
    } else {
      "Insufficient funds"
    }
  }
}

new_withdraw <- make_withdraw_balance_100()
```

```
new_withdraw(25)
```

```
## [1] 75
```

```

make_withdraw <- function(balance) {
  function(amount) {
    if (balance >= amount) {
      assign("balance", balance - amount, inherits=TRUE )
      balance
    } else {
      "Insufficient funds"
    }
  }
}

W1 <- make_withdraw(100)
W2 <- make_withdraw(100)
```

인출 처리 장치

- W1과 W2는 독립적인 개체 -> 각자의 지역 상태 변수 balance를 가짐

```
W1(50); W2(70)
```

```
## [1] 50
```

```
## [1] 30
```

```
make_account <- function(balance) {
  withdraw <- function(amount) {
    if (balance >= amount) {
      assign("balance", balance - amount, inherits = TRUE)
      balance
    }
  }
  deposit <- function(amount) {
    assign("balance", balance + amount, inherits = TRUE)
    balance
  }
  dispatch <- function(m) {
    if (m == "withdraw") {
      withdraw
    } else if (m == "deposit") {
      deposit
    } else {
      stop(past("unknown request -- make_account", m))
    }
  }
  dispatch
}
```

입금 가능한 은행 계좌 개체

- `make_account` 호출마다 지역 상태 변수 `balance`를 갖는 환경(environment)이 만들어짐
- 반환값(`dispatch`)은 메시지 전달 방식(2.4.3)으로 은행 계좌 개체를 나타내는 함수
  - 환경 내의 지역 변수를 변경 가능

```
acc <- make_account(100)
acc("withdraw")(50);
```

```
## [1] 50
```

```
acc("deposit")(40)
```

```
## [1] 90
```

```
acc2 <- make_account(100)
acc2("withdraw")(30);
```

```
## [1] 70
```

```
acc2("deposit")(50)
```

```
## [1] 120
```

3.1.2. The Benefits of Introducing Assignment

- 몬테카를로 실험: 무작위로 행한 실험의 표본 성공 비율로부터 모집단의 성공 확률과 관련된 양을 추정
- 무작위로 추출된 두 자연수가 서로소일 확률이 $\frac{6}{\pi^2}$라는 사실로부터 π를 몬테카를로 실험을 통해 추정 가능

```
# General Monte Carlo Function
monte_carlo <- function(trials, experiment) {
  iter <- function(trials_remaining, trials_passed) {
    if (trials_remaining == 0) {
      trials_passed / trials
    } else if (experiment()) {
      iter(trials_remaining - 1, trials_passed + 1)
    } else {
      iter(trials_remaining - 1, trials_passed)
    }
  }
  iter(trials, 0)
}
```

- rand: 균일 분포에서 무작위 추출

```
rand_update <- function(x) {
  m <- 2^31 - 1
  a <- 7^5
  b <- 0L
  (a * x + b) %% m
}

x_1 <- 3 # given number
x_2 <- rand_update(x_1)
x_3 <- rand_update(x_2)
```

- 대입을 이용한 rand 구현

```
random_init <- 123456789L

make_rand <- function() {
  x <- random_init
  function() {
    assign("x", rand_update(x), inherits=TRUE )
    x
  }
}
rand <- make_rand()
```

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

dirichlet_test <- function() {
  gcd(rand(), rand()) == 1   # TRUE or FALSE
}

estimate_pi <- function(trials) {
```

```
  sqrt(6 / monte_carlo(trials, dirichlet_test))
}
```

```
estimate_pi(500)
```

π의 몬테카를로 추정

```
## [1] 3.285022
```

```
random_gcd_test <- function(trials, initial_x) {
  iter <- function(trials_remaining, trials_passed, x) {
    x1 <- rand_update(x)
    x2 <- rand_update(x1)
    if (trials_remaining == 0) {
      trials_passed / trials
    } else if (gcd(x1, x2) == 1) {
      iter(trials_remaining - 1, trials_passed + 1, x2)
    } else {
      iter(trials_remaining - 1, trials_passed, x2)
    }
  }
  iter(trials, 0, initial_x)
}

estimate_pi2 <- function(trials) {
  sqrt(6 / random_gcd_test(trials, random_init))
}

estimate_pi2(500)
```

대입을 사용하지 않은 구현

```
## [1] 3.285022
```

- 모듈성의 훼손
  - `rand`를 사용한 첫번째 버전: 임의의 `experiment` 함수를 전달인자로 받는 `monte_carlo` 함수로 π를 추정하는 몬테카를로 법을 표현
  - `rand`를 사용하지 않는 (즉, 지역 상태를 보관하지 않는) 두번째 버전: 난수 `x1`과 `x2`를 명시적으로 조작하고 `x2`를 반복문에서 재사용하여 `rand_update`의 입력으로 사용
    - 다른 몬테카를로 실험에서는 사용하는 난수의 개수가 달라질 수 있음
    - 최상위 함수 `estimate_pi2`에서도 난수 발생기의 초기값 `initial_x`를 제공해야 함
- 즉, 난수 발생기의 내부가 프로그램의 다른 부분으로 새어 나옴 -> 몬테카를로 코드를 다른 작업에 재사용하기 어려움
- 첫번째 버전에서는 난수 발생기의 상태를 `rand` 안에 가둬 두어 난수 발생의 세부가 프로그램의 다른 부분과 무관

3.1.3. The Costs of Introducing Assignment

- `assign` 연산을 사용하여 지역 상태를 갖는 개체 구현
- 대가
  - 프로그래밍 언어에서의 함수 적용을 더 이상 치환 모형(1.1.5.)으로 설명할 수 없음

  - 수리적으로 단순한 모형으로 개체와 대입을 다루기 어려움
- 대입이 없는 기존 방법: 같은 입력 -> 같은 출력
  - 함수 적용을 수학 함수의 계산으로 간주 가능: 함수형 프로그래밍(functional programming)

```
make_simplified_withdraw <- function(balance) {
  function(amount) {
    assign("balance", balance - amount, inherits = TRUE)
    balance
  }
}

W <- make_simplified_withdraw(25)
W(20); W(15)
```

대입의 복잡성

```
## [1] 5
```

```
## [1] -10
```

```
make_decrementer <- function(balance) {
  # do not use `assign`
  function(amount) {
    balance - amount
  }
}

D <- make_decrementer(25)
D(20); D(15)
```

```
## [1] 5
```

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

- 치환 모형으로는 표현 불가능

치환 모형의 기본 가정

- "프로그래밍 언어에서 기호(symbol)는 본질적으로 값을 지칭하는 이름이다."
  - 대입을 도입하기 전에는 참인 명제 (상수)
  - 대입의 도입으로 변수의 값은 갱신 가능하게 됨
    * 이제 변수는 값을 저장할 수 있는 어떤 장소를 의미
    * 저장소에 담긴 값은 프로그램 실행 도중 변할 수 있음 -> 환경 모형 (3.2.)

```
D1 <- make_decrementer(25)
D2 <- make_decrementer(25)
```

```
W1 <- make_simplified_withdraw(25)
W2 <- make_simplified_withdraw(25)
```

"같다"의 의미

- 어떤 표현식을 평가할 때 `W1`을 `W2`로 대체해서 같은 값이 나온다는 보장이 없음

```
W1(20);
```

```
## [1] 5
```

```
W1(20);
```

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

```
W2(20)
```

```
## [1] 5
```

- 프로그래밍 언어의 참조 투명성: 같은 것은 같은 것으로 치환 가능
  - 대입을 도입하면 참조 투명성이 깨짐 -> 분석의 어려움

- "Peter와 Paul은 잔고가 $100인 은행 계좌가 하나씩 있다"

```
peter_acc <- make_account(100)
paul_acc  <- make_account(100)
```

```
peter_acc <- make_account(100)
paul_acc  <- peter_acc
```

- 첫 번째 모형에서는 두 계좌가 별개
- 두 번째 모형에서는 `paul_acc`와 `peter_acc`가 같음
  - 두 사람의 공동 계좌
  - side effect: 프로그램에서 `paul_acc`가 변경될 수 있는 곳을 찾아 수정하려 한다면 `peter_acc`가 변경되는 곳도 찾아야 함

- 데이터 개체에 대한 변경이 허용되지 않는다면 합성 데이터 개체는 구성요소들의 총합 (e.g. 유리수)
- 변경이 허용된다면 합성 개체는 구성요소들과는 또다른 정체성을 가짐
  - 은행 계좌는 인출해서 잔고가 변해도 여전히 "같은" 은행 계좌
  - 계좌를 하나의 개체로 인식하는 것의 문제

명령형 프로그래밍의 함정

- 명령형 프로그래밍(imperative programming): 대입을 활발히 활용하는 프로그래밍 방식
- 계승 계산 (함수형)

```
factorial <- function(n) {
  iter <- function(product, counter) {
    if (counter > n) {
      product
    } else {
      iter(counter * product, counter + 1)
    }
  }
  iter(1, 1)
}
```

```r
factorial(5)
```

```
## [1] 120
```

- 계승 계산 (명령형)

```r
factorial <- function(n) {
  product <- 1
  counter <- 1
  iter <- function() {
    if (counter > n) {
      product
    } else {
      assign("product", counter * product, inherits = TRUE)
      assign("counter", counter + 1, inherits = TRUE)
      iter()
    }
  }
  iter()
}

factorial(5)
```

```
## [1] 120
```

- 명령형: 대입의 **순서** 바뀌면 안됨

3.2. The Environment Model of Evaluation

- 치환 모형(1.1.5.): 합성함수를 전달인자들에 적용할 때는 각 형식 매개변수를 대응되는 전달인자로 치환하여 평가
- 환경(environments)
  - 대입을 도입한 경우, 변수는 값이 저장되는 어떤 "장소"
  - 새로운 평가 모형: 환경 구조를 통해 이러한 장소를 관리
  - 환경: 프레임의 계층 구조
    - 프레임: 이름-값 묶음(binding) + 상위 환경으로의 포인터

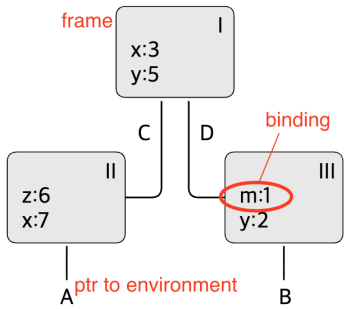

Figure 1: 간단한 환경 구조

- 환경 + (상위 환경으로의) 포인터 구조
  - 환경 C==D에서 x: 3
  - 환경 B에서 x: 3
  - 환경 A에서 x: 7 (가리기(shadowing): 바깥 환경의 같은 이름을 가려서 안 보이게 함)

환경과 평가

- 환경은 표현식이 평가되는 문맥을 결정
- 표현식은 평가되는 환경이 있어야만 의미를 가짐
  - e.g. `1 + 1`은 `+`가 덧셈 기호라는 문맥 안에서만 작동
- 전역 환경(global environment): 인터프리터가 작동하는 최상위 환경 (단일 프레임, 상위 환경 없음)

3.2.1. The Rules for Evaluation

- 함수 개체 생성
  - 함수 = (코드, 환경으로의 포인터) 쌍
- e.g. `square <- function(x) { x * x }`
  - 코드: 매개변수 명세(`x`)와 함수 본체 `{ x * x }`
  - 포인터: 무명함수 표현식 `function(x) { x * x }`이 평가된 환경(전역 환경)을 가리킴
  - `<-`로 이름과 값(함수 개체)를 묶음

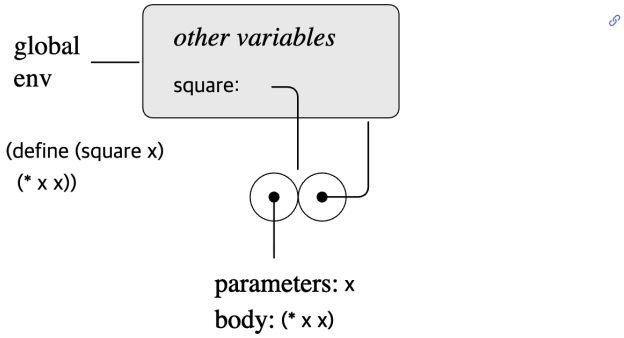

Figure 2: 전역 환경에서 `square <- function(x) { x * x }` 를 평가해서 만들어진 환경 구조

함수 적용

- Remark. 함수 적용의 평가 (1.1.4.)
  1. 적용의 부분 표현식들(함수 표현식 `square`, 전달인자 표현식 `5`)을 각각 평가
  2. 함수(함수 표현식의 값 - 함수 객체)을 전달인자 표현식의 값들에 적용
- 함수를 전달인자에 적용할 때 치환 모형 대신 환경 모형을 사용하면?
  1. 매개변수 `x`를 전달인자 값 `5`에 묶는 프레임을 하나 담은 환경(E1) 생성 + 상위 환경 (함수에서 가리키는 환경)
  2. 새 환경에서 함수 본체 `{ x * x }`를 평가

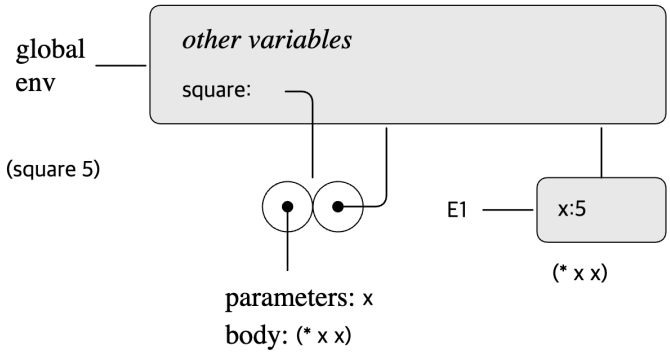

Figure 3: 전역 환경에서 `square(5)` 를 평가해서 얻은 환경

- Notes
  - E1은 전역환경을 가리키는 이유: `square`는 전역 환경에서 정의되어 있음 -> 함수 객체는 코드 + 정의될 때의 환경 포인터로 구성되므로 호출한 곳이 아닌 "정의된 곳"인 전역 환경을 가리킴
  - `square(5)`의 값: E1에서 본체 `{ x * x }`를 평가
    - `x` 찾기: E1에 `x=5` 있음 -> 5
    - `*` 찾기: E1에 없으니 부모(전역)으로 올라가서 찾음
    - 계산 `{ 5 * 5 }` -> 25
- 함수 적용의 환경 모형 규칙
  1. 함수 개체를 전달인자들에 적용할 때는
  - 함수의 형식 매개변수를 전달인자와 묶어서 만든 프레임을 가지고 새 환경을 만들고,
  - 해당 환경의 문맥에서 함수 본체를 평가함
  - 새 프레임은 적용되는 함수 개체가 가리키는 환경을 상위 환경으로 가리킴
  2. 함수는 주어진 환경에서
  - 무명함수 표현식을 평가한 결과로 생성됨
  - 그렇게 만들어진 함수 개체는 무명함수의 코드와 무명함수 표현식이 평가된 환경을 가리키는 포인터로 구성됨

```
assign(<name>, <new_value>, inherits=TRUE )
```

`assign`의 작동 방식

- 위의 표현식을 주어진 환경에서 평가하면
  - 해당 이름 `<name>`의 바인딩을 가진 첫 번째 프레임을 해당 환경(상위 환경 포함)에서 찾아 `<new_value>`와 새로이 묶음
  - 찾는 바인딩이 없으면 새로 만듦
  - `inherits = FALSE` -> 상위 환경을 조사하지 않음 (default)

3.2.2. Applying Simple Procedure

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

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

```
}

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

f(5)
```

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

- 환경 모형을 이용해 분석:

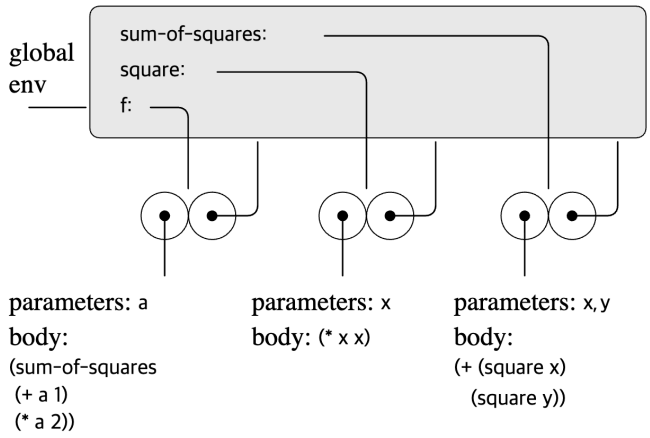

Figure 4: 전역 환경 내의 함수 개체

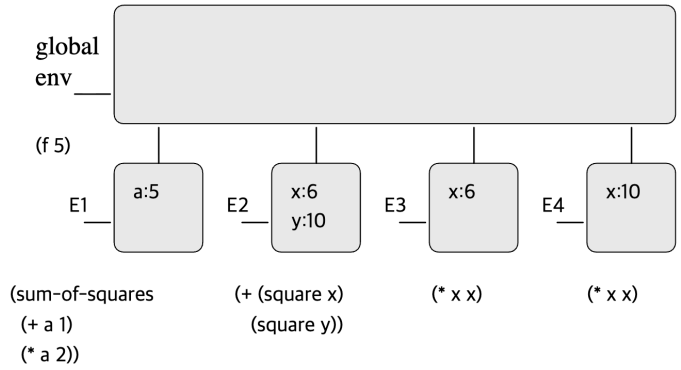

Figure 5: `f(5)` 평가로 생성된 환경

- E1
  - a는 5에 묶임
  - `sum_of_squares(a + 1, a * 2)` 평가
    - 부분 표현식 평가
      - E1의 첫번째 프레임에 `sum_of_squares` 없음 -> 상위 전역 환경에서 찾음
      - `a+1`, `a*2`는 원초 연산자를 적용 -> 각각 6, 10으로 평가
      - 함수 개체 `sum_of_squares`를 전달인자 `(6, 10)`에 적용 -> 새 환경 E2 생성
- E2
  - x는 6, y는 10에 묶임
  - `square(x) + square(y)` 평가

        * 부분 표현식 평가: `square(x)`
            · E2의 첫번째 프레임에 `square`가 없음 -> 상위 전역 환경에서 찾음
            · 함수 개체 `square`를 전달인자 6에 적용 -> 새 환경 E3 생성
        * 부분 표현식 평가: `square(y)`
            · 같은 방식으로 새 환경 E4 생성
- E3
    - `x`가 6에 묶임
    - `x * x` 평가 -> 36
- E4
    - `x`가 10에 묶임
    - `x * x` 평가 -> 100
- 부분 표현식들을 모두 평가한 후 최종 결과값을 반환
    - 두 번의 `square` 호출로 얻어진 값들이 `sum_of_squares`에서 원초 연산자 `+`를 이용하여 더해짐
    - 결과값이 `f(5)`의 값으로 평가
- 주안점
    - `square`를 호출할 때마다 `x`의 바인딩을 포함하는 새로운 환경이 생성
        * 이름은 모두 `x`로 같지만 실제로는 **서로 다른** 지역변수들이 각자 다른 프레임에 담겨서 격리
        * 생성된 프레임들이 모두 전역환경을 가리키는 이유: `square`가 전역에서 정의됨

3.2.3. Frames as the Repository of Local State

인출 처리 장치

- 환경 모형으로 함수, 대입문으로 지역 상태를 갖는 개체 표현하기

```
make_withdraw <- function(balance) {
  function(amount) {
    if (balance >= amount) {
      assign("balance", balance - amount, inherits=TRUE )
      balance
    } else {
      "Insufficient funds"
    }
  }
}
W1 <- make_withdraw(100)
W1(50)
```

```
## [1] 50
```

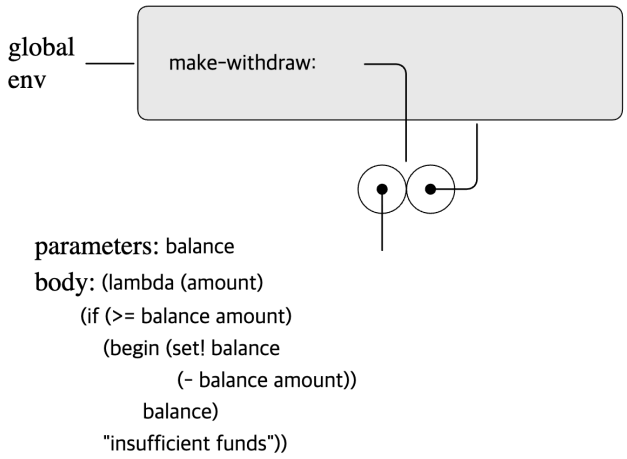

Figure 6: 전역 환경에서 `make_withdraw`를 정의한 결과

- 특기 사항: 함수 본체의 반환 표현식이 무명함수

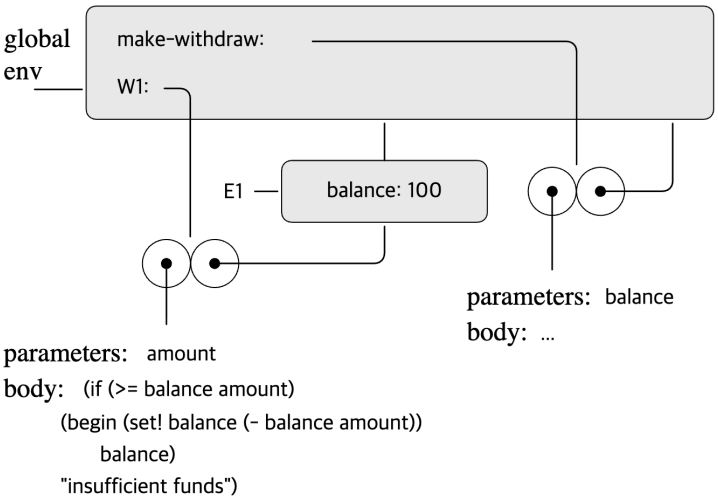

Figure 7: `W1 <- make_withdraw(100)` 의 평가로 생긴 환경 구조

- `W1 <- make_withdraw(100)`의 평가로 새로 생성된 환경 E1
  - 매개변수 `balance`와 전달인자 100을 묶은 바인딩이 있는 프레임 1개
  - E1에서 `make_withdraw`의 본체 (무명함수 `function(amount){...}`의 표현식 평가)
    - 평가 결과: 코드가 무명함수의 매개변수 명세 및 본체 / 상위 환경으로 E1 가리킴
  - 이 개체가 `make_withdraw(100)`의 값으로 반환, `W1`에 묶임

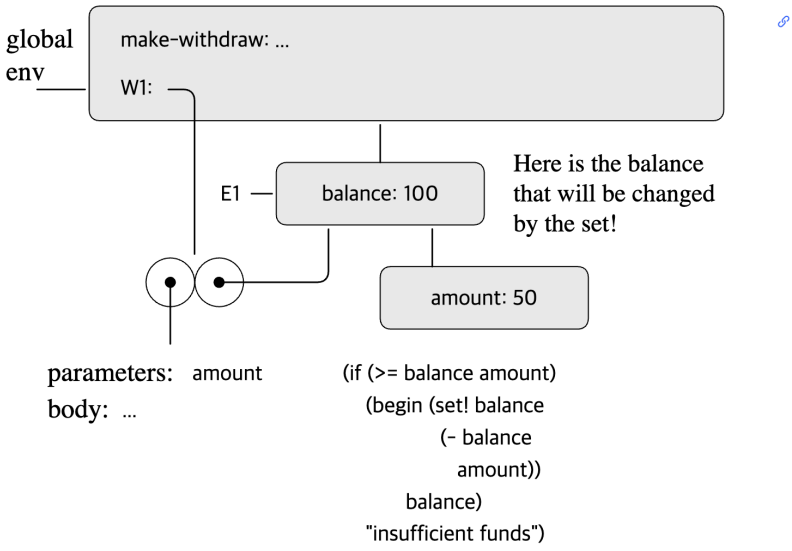

Figure 8: 함수 개체 `W1` 을 전달인자 `50` 에 적용하여 만들어진 환경

- `W1`를 전달인자 50에 적용 -> `W1`의 매개변수 `amount`와 전달인자 50을 묶은 프레임을 담은 새 환경 생성
  - 상위 환경은 E1
  - 이 환경에서 `W1`의 본체를 평가
    - `amount` 바인딩: 첫번째 프레임
    - `balance` 바인딩: E1 (상위 환경)
    - `assign("balance", balance - amount, inherits=TRUE )`: `balance`의 바인딩을 100에서 50으로 바꿈

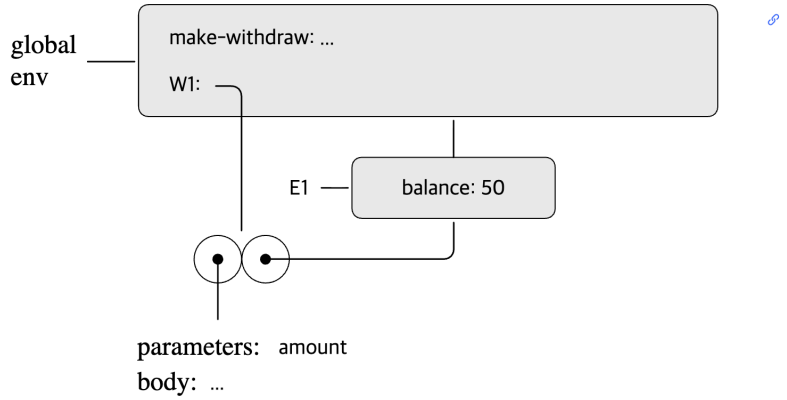

Figure 9: `W1` 호출 종료 후의 환경

- `W1` 호출 종료 후:
  - `amount`: 50이었던 환경은 더 이상 유효하지 않음
  - E1의 역할: 함수 개체 `W1`의 지역 상태 변수 `balance`를 담는 저장소
  - 향후 `W1` 호출은 `amount` 바인딩을 갖고 상위 환경으로 `E1`을 가리키는 새로운 프레임을 생성

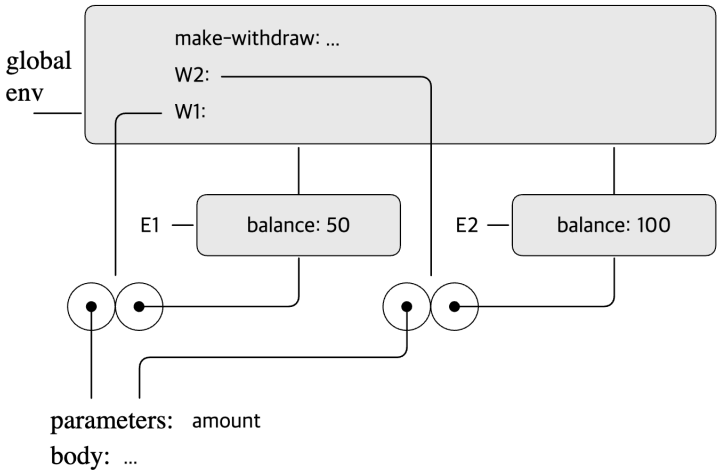

Figure 10: `W2 <- make_withdraw(100)` 으로 또다른 인출 처리 장치 개체를 생성한 후의 환경 구조

- 또 다른 인출 처리 장치 개체 생성 가능

3.2.4. Internal Definitions

```
average <- function(x, y) {
    (x + y) / 2
}

sqrt2 <- function(x) {
  is_good_enough <- function(guess) {
    abs(square(guess) - x) < 0.001
  }
  improve <- function(guess) {
    average(guess, x / guess)
  }
  sqrt_iter <- function(guess) {
    ifelse(is_good_enough(guess),
           guess,
           sqrt_iter(improve(guess))
    )
  }
  sqrt_iter(1)
}

sqrt2(2)
```

제곱근 계산 (1.1.8.)

```
## [1] 1.414216
```

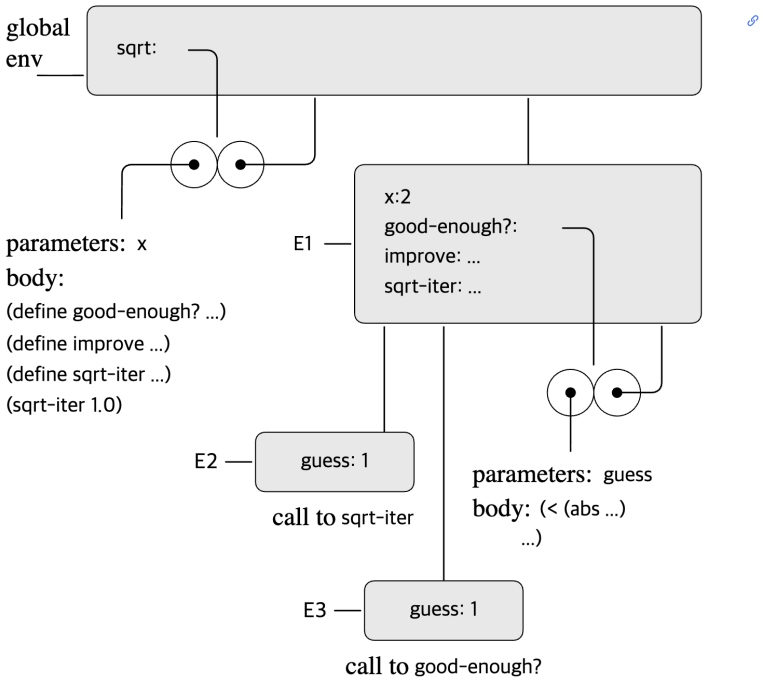

Figure 11: `sqrt2` 함수와 내부 함수

- 지역 함수의 이름은 상위 함수 외부의 이름과 충돌하지 않음
  - 지역 함수 이름은 상위 함수가 수행될 때 생성하는 프레임에 묶임 (전역 환경에 묶이지 않음)
- 지역 함수는 상위 함수의 전달인자에 접근 가능: 매개변수 이름을 자유 변수로 사용
  - 지역 함수의 본체는 상위 함수의 평가 환경에 종속되는 환경에서 평가되기 때문

3.3. Modeling with Mutable Data

- Remark. 데이터 추상화의 원리 (2장): 자료 구조를 생성자와 선택자를 이용하여 기술
- 상태를 가진 개체로 시스템 모형화 -> 변경자(mutator)가 필요
- e.g. 은행 계좌: 지정된 계좌의 잔액 변경하는 `set_balance(<account>, <new_value> )`
- 변경 가능 데이터 개체(mutable data objects): 변경자가 정의된 데이터 개체

3.3.1. Mutable List Structure

- 쌍(pairs) -> 내용 변경 불가
- 변경자 `set_head`와 `set_tail` 도입
  - 첫번째 전달인자는 `pair`
  - 두번째 전달인자가 가리키는 곳으로 `head` 또는 `tail` 포인터를 변경

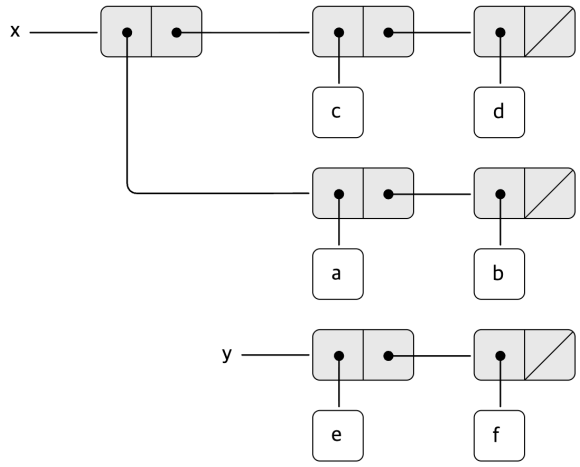

Figure 1: `x <- List(List("a", "b"), "c", "d")` 및 `y <- List("e", "f"))` 로 생성된 리스트 구조

```
(x <- List(List("a", "b"), "c", "d"));
```

```
## [ [ a, [ b, NULL ] ], [ c, [ d, NULL ] ] ]
```

```
(y <- List("e", "f"))
```

```
## [ e, [ f, NULL ] ]
```

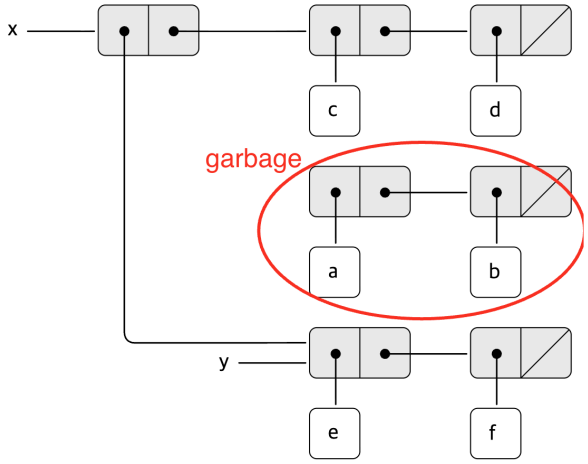

Figure 2: Figure 1 의 리스트에 `set_head(x, y)` 을 수행한 결과

```
set_head(x, y)
x
```

```
## [ [ e, [ f, NULL ] ], [ c, [ d, NULL ] ] ]
```

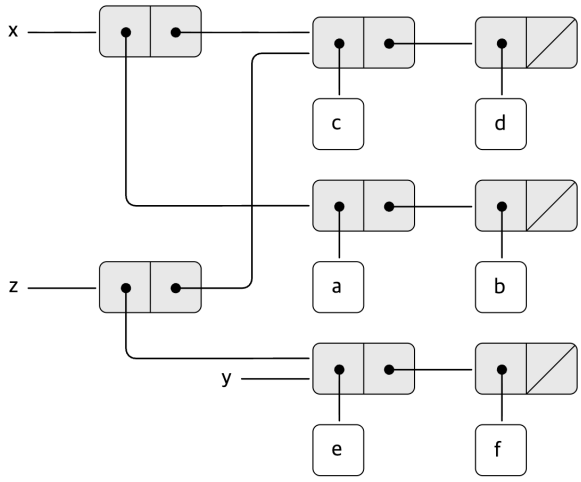

Figure 3: Figure 1 의 리스트에 `z <- pair(y, tail(x))` 을 수행한 결과

```
(z <- pair(y, tail(x)))
```

```
## [ [ e, [ f, NULL ] ], [ c, [ d, NULL ] ] ]
```

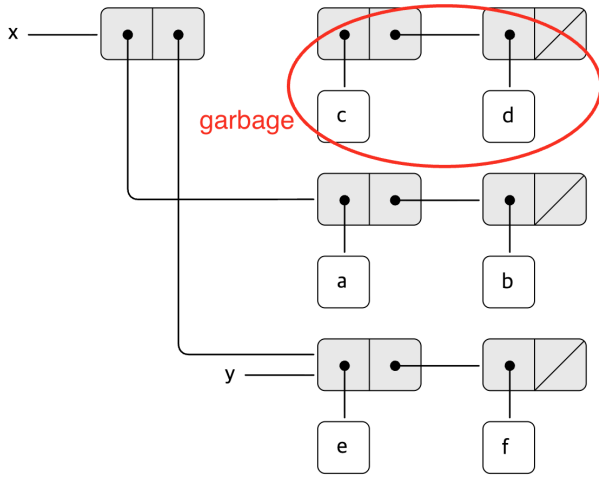

Figure 4: Figure 1 의 리스트에 `set_tail(x, y)` 를 수행한 결과

```
set_tail(x, y)
x
```

```
## [ [ e, [ f, NULL ] ], [ e, [ f, NULL ] ] ]
```

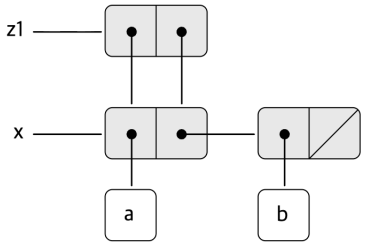

Figure 5: `z1 <- pair(x, x)` 로 생성된 리스트 개체

공유와 독자성

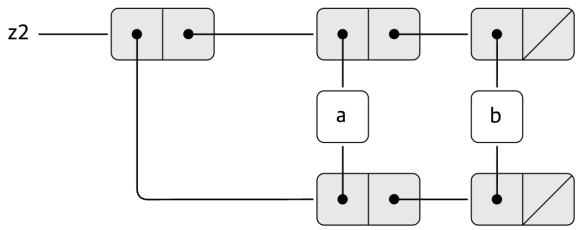

Figure 6: `z2 <- pair(List("a", "b"), List("a", "b"))` 로 생성된 리스트 개체

```
x <- List("a", "b")
(z1 <- pair(x, x));
```

```
## [ [ a, [ b, NULL ] ], [ a, [ b, NULL ] ] ]
```

```
(z2 <- pair(List("a", "b"), List("a", "b")))
```

```
## [ [ a, [ b, NULL ] ], [ a, [ b, NULL ] ] ]
```

```
identical(head(z1), tail(z1));
```

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

```
identical(head(z2), tail(z2))
```

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

```
set_to_wow <- function(x) {
  set_head(head(x), "wow")
  x
}
```

```
set_to_wow(z1);
```

변경자의 효과

```
## [ [ wow, [ b, NULL ] ], [ wow, [ b, NULL ] ] ]
```

```
set_to_wow(z2)
```

```
## [ [ wow, [ b, NULL ] ], [ a, [ b, NULL ] ] ]
```

- 변경은 대입일 뿐

```
pair <- function(x, y) {
  .head <- x
  .tail <- y
  dispatch <- function(message, val = NULL) {
    if (identical(message, "head")) {
      .head
    } else if (identical(message, "tail")) {
      .tail
    } else if (identical(message, "set-head!")) {
      #.head <<- val
      assign(".head", val, inherits=TRUE )
      invisible(NULL)
    } else if (identical(message, "set-tail!")) {
```

```
      #.tail <<- val
      assign(".tail", val, inherits=TRUE )
      invisible(NULL)
    } else {
      "unknown-message"
    }
  }
  class(dispatch) <- "pair" # S3, covered later
  dispatch
}

head.pair <- function(pair) {
  pair("head")
}
tail.pair <- function(pair) {
  pair("tail")
}
set_head <- function(pair, new_value) {
  pair("set-head!", new_value)
}
set_tail <- function(pair, new_value) {
  pair("set-tail!", new_value)
}
```

3.3.2. Representing Queues

- 대기열: 선입선출(first in, first out, FIFO) 자료 구조
  - 줄의 뒷단에 삽입, 앞단에서 삭제

연산	결과 대기열
`q <- make_queue()`	
`insert_queue(q, "a")`	a
`insert_queue(q, "b")`	a b
`delete_queue(q)`	b
`insert_queue(q, "c")`	b c
`insert_queue(q, "d")`	b c d
`delete_queue(q)`	c d

- 대기열은 아래의 연산으로 정의됨

구분	함수 이름
생성자	`make_queue()`
서술어	`is_empty_queue(<queue>)`
선택자	`front_queue(<queue>)`
변경자	`insert_queue(<queue>, <item>)` `delete_queue(<queue>)`

- cf. 리스트: 앞단 = `head`, 삽입 = 리스트 끝에 원소 추가, 삭제 = `tail`?
  - 원소 삽입: 리스트 끝까지 모든 원소를 스캔
  - 스캔 = 연속된 `tail` 연산
  - 원소 개수가 n일 때 $\Theta(n)$번의 단계 필요 -> 비효율적

- 대기열 구현

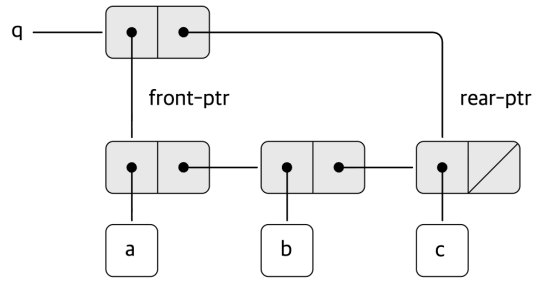

Figure 7: 앞단 포인터와 뒷단 포인터의 리스트로 구현한 대기열

```
front_ptr     <- function(queue) { head(queue) } # 리스트의 첫 페어를 가리키는 포인터
rear_ptr      <- function(queue) { tail(queue) }
set_front_ptr <- function(queue, item) { set_head(queue, item) }
set_rear_ptr  <- function(queue, item) { set_tail(queue, item) }
```

- 대기열 연산

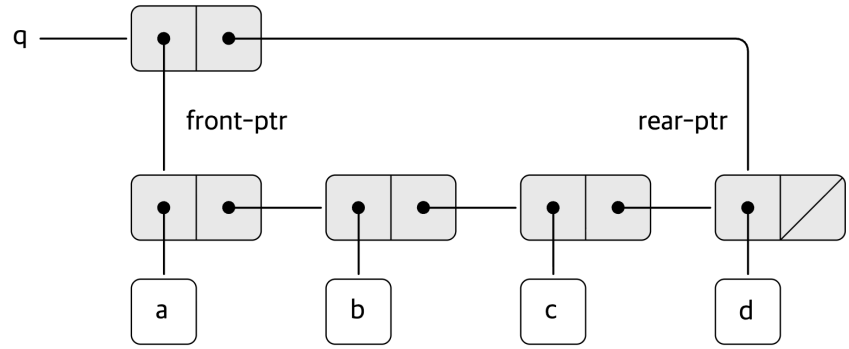

Figure 8: Figure 7 의 대기열에 `insert_queue(q, "d")` 를 적용한 결과

```
is_empty_queue <- function(queue) { is.null(front_ptr(queue)) }

make_queue  <- function() { pair(NULL, NULL) }

front_queue <- function(queue) {
  if (is_empty_queue(queue)) {
    stop("front_queue called with an empty queue")
  } else {
    head(front_ptr(queue))
  }
}

insert_queue <- function(queue, item) {
  new_pair <- pair(item, NULL)
  if (is_empty_queue(queue)) {
    set_front_ptr(queue, new_pair)
    set_rear_ptr(queue, new_pair)
  } else {
    set_tail(rear_ptr(queue), new_pair)
    set_rear_ptr(queue, new_pair)
```

```
  }
  queue
}
```

```
q <- make_queue()
insert_queue(q, "a")
```

```
## [ [ a, NULL ], [ a, NULL ] ]
```

```
insert_queue(q, "b")
```

```
## [ [ a, [ b, NULL ] ], [ b, NULL ] ]
```

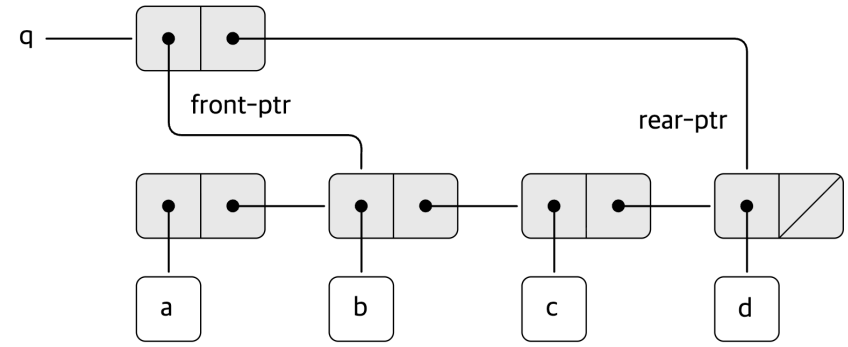

Figure 8 의 대기열에 `delete_queue(q)` 를 적용한 결과

```
delete_queue <- function(queue) {
  if (is_empty_queue(queue)) {
    stop("delete_queue called with an empty queue")
  } else {
    set_front_ptr(queue, tail(front_ptr(queue)))
    queue
  }
}
```

```
delete_queue(q)
```

```
## [ [ b, NULL ], [ b, NULL ] ]
```

3.3.3. Representing Tables

- 키-값 쌍의 저장과 색인

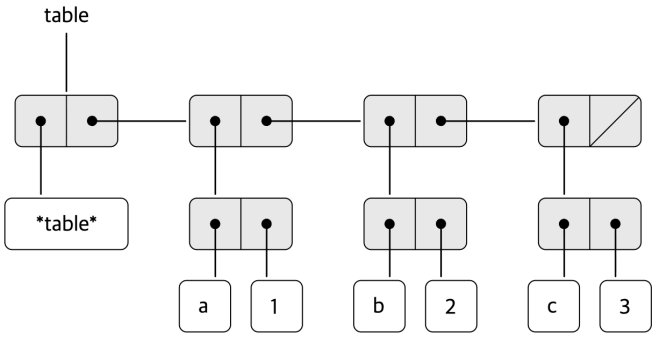

Figure 9: 머리 달린 리스트로 표현한 표

1차원 표 (2.3.3.)

```
# expects a key and a list of key-value pairs as arguments.
# returns the key-value pair that has the given key as its head
assoc <- function(key, pairs) {
  if (is.null(pairs)) {
    FALSE
  } else if (equal(key, head(head(pairs))) ) {
    head(pairs)  # a key-value pair
  } else {
    assoc(key, tail(pairs))
  }
}

equal <- function(xs, ys) {
  if (is_pair(xs)) {
    is_pair(ys) &&
      equal(head(xs), head(ys)) &&
      equal(tail(xs), tail(ys))
  } else {
    identical(xs, ys)
  }
}
```

```
# takes a key as argument and returns the associated value
lookup <- function(key, table) {
  kvpair <- assoc(key, tail(table))
  if (!identical(kvpair, FALSE)) {
    tail(kvpair)  # value
  } else {
    FALSE
  }
}
```

```
insert <- function(key, value, table) {
  kvpair <- assoc(key, tail(table))
  if (identical(kvpair, FALSE)) {
    set_tail(table,
             pair(pair(key, value), tail(table)))
  } else {
    set_tail(kvpair, value)
  }
  "ok"
}
```

```
make_table <- function() {
    List("*table*")
}
```

- 예시

```
t1 <- make_table()
insert("a", 10, t1)
```

```
## [1] "ok"
```

```
lookup("a", t1)
```

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

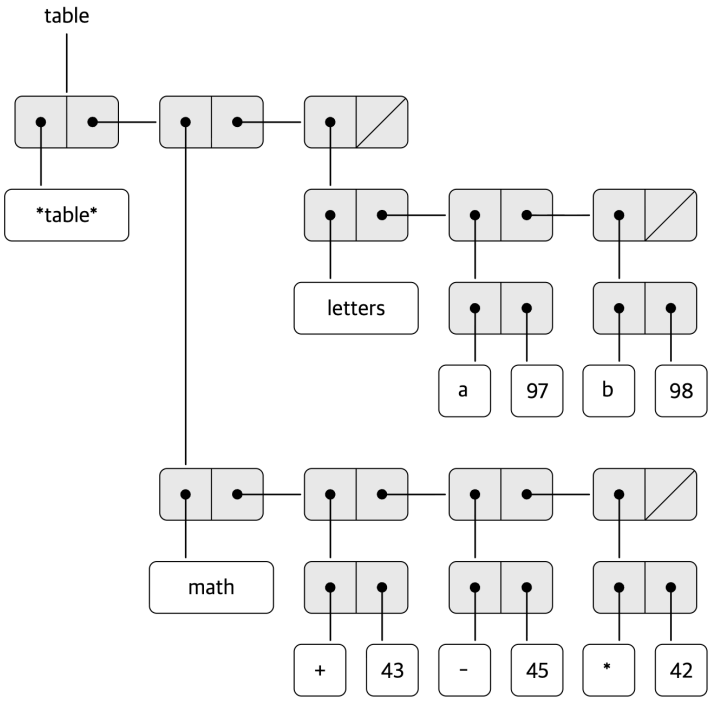

Figure 10: 2차원 표

2차원 표 (2.4.3.)

```
lookup2 <- function(key1, key2, table) {
  subtable <- assoc(key1, tail(table))
  if (!identical(subtable, FALSE)) {
    record <- assoc(key2, tail(subtable))
    if (!identical(record, FALSE)) {
      tail(record)
    } else {
      FALSE
    }
  } else {
    FALSE
  }
}
```

```
insert2 <- function(key1, key2, value, table) {
  subtable <- assoc(key1, tail(table))
  if (!identical(subtable, FALSE)) {
    record <- assoc(key2, tail(subtable))
    if (!identical(record, FALSE)) {
      set_tail(record, value)
    } else {
      set_tail(subtable,
               pair(pair(key2, value), tail(subtable)))
    }
  } else {
    set_tail(table,
             pair(
               List(key1, pair(key2, value)),
               tail(table)
             ))
  }
  "ok"
}
```

```
## [1] "ok"
```

```
## [1] "ok"
```

```
## [1] "ok"
```

```
## [1] "ok"
```

```
## [1] "ok"
```

```
lookup2("math", "-", t2)
```

```
## [1] 45
```

```
lookup2("letters", "a", t2)
```

```
## [1] 97
```

지역 표

- 표마다 lookup과 insert 함수를 따로

```
# table with two keys
make_table2 <- function() {
  local_table <- List("*table*")
  lookup <- function(key1, key2) {
    subtable <- assoc(key1, tail(local_table))
    if (!identical(subtable, FALSE)) {
      record <- assoc(key2, tail(subtable))
      if (!identical(record, FALSE)) {
        tail(record)
      } else {
        FALSE
      }
    } else {
      FALSE
    }
```

```
  }
  insert <- function(key1, key2, value) {
    subtable <- assoc(key1, tail(local_table))
    if (!identical(subtable, FALSE)) {
      record <- assoc(key2, tail(subtable))
      if (!identical(record, FALSE)) {
        set_tail(record, value) # replace existing value
      } else {
        set_tail(subtable,
                 pair(pair(key2, value), tail(subtable)))
      }
    } else {
      set_tail(local_table,
               pair(
                 List(key1, pair(key2, value)),
                 tail(local_table)
               ))
    }
    "ok"
  }
  dispatch <- function(m) {
    if (identical(m, "lookup")) {
      lookup
    } else if (identical(m, "insert!")) {
      insert
    } else if (identical(m, "table")) { # tempoarily for testing
      local_table
    } else {
      stop("Unknown operation -- TABLE")
    }
  }
  dispatch
}
```

- Get과 Put 연산의 구현

```
operation_table <- make_table2()
Get <- operation_table("lookup")
Put <- operation_table("insert!")
```

```
Put("a", "b", 10);
```

```
## [1] "ok"
```

```
Get("a", "b")
```

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