

SNU 4190.210 프로그래밍
원리(Principles of Programming)
Part II

Prof. Kwangkeun Yi

차례

- 1 데이터 구현하기 (data implementation)
- 2 데이터 속구현 감추기 (data abstraction)
- 3 여러 구현 동시 지원하기 (multiple implemenations)
- 4 각 계층별로 속구현 감추기(data abstraction hierarchy)

다음

- 1 데이터 구현하기 (data implementation)
- 2 데이터 속구현 감추기 (data abstraction)
- 3 여러 구현 동시 지원하기 (multiple implemenations)
- 4 각 계층별로 속구현 감추기(data abstraction hierarchy)

데이터 구현하기 (data implementation)

새로운 타입의 데이터/값 구현하기

- ▶ 기억하는가: 타입들(types)

$\tau ::= \iota$	primitive type
$\tau \times \tau$	pair(product) type
$\tau + \tau$	or(sum) type
$\tau \rightarrow \tau$	ftn type, single param
$\tau * \dots * \tau \rightarrow \tau$	ftn type, multi params
\top	any type
t	user-defined type's name
τt	user-defined type's name, with param
$\iota ::= \textit{int} \mid \textit{real} \mid \textit{bool} \mid \textit{string} \mid \textit{symbol} \mid \textit{unit}$	

- ▶ 새로운 타입(t 혹은 τt)의 데이터 구현해야

새로운 데이터 타입

- ▶ 기본으로 제공되는 타입들

int, real, bool, string, symbol, unit

이 아닌

- ▶ 소프트웨어마다 새로운 타입의 데이터/값이 필요
 - ▶ 예) 짝, 집합, 리스트, 가지, 나무, 숲, 땅, 감정, 관계, 지식, 자동차, 목록표, 하늘, 바람, 원자, 분자, 세포, 뉴런, 책, 색깔, 종이, 건물, 층, 벽, 기둥, 사람, 테란, 젤-나가, 저그, 프로토스, 등등

새로운 데이터 타입 구현하기

기억하는가:



모든 프로그래밍 언어에는 각 타입마다 그 타입의 값을 만드는 식과 사용하는 식을 구성하는 방법이 제공된다.

새로운 데이터 타입 구현하기

기억하는가:



모든 프로그래밍 언어에는 각 타입마다 그 타입의 값을 만드는 식과 사용하는 식을 구성하는 방법이 제공된다.

구현할 새로운 타입의 데이터/값에 대해서도

- ▶ 만드는(introduction, construction) 방법과
- ▶ 사용하는(elimination, use) 방법을

함수로 구현해서 사용하면 된다.

새로운 데이터 타입 구현하기



- ▶ 새로운 타입을 τ 라고 하면
 - ▶ 만들기 함수들의 타입은

$\dots \rightarrow \tau$

- ▶ 사용하기 함수들의 타입은

$\tau \rightarrow \dots$

예) 짝(pair) $\tau \times \tau'$

- ▶ 만들기

$$\text{pair} : \quad \rightarrow \tau \times \tau'$$

- ▶ 사용하기

$$\text{l} : \tau \times \tau' \rightarrow$$

$$\text{r} : \tau \times \tau' \rightarrow$$

그 함수들의 구현:

예) 짝(pair) $\tau \times \tau'$

- ▶ 만들기

$$\text{pair} : \tau * \tau' \rightarrow \tau \times \tau'$$

- ▶ 사용하기

$$\text{l} : \tau \times \tau' \rightarrow$$

$$\text{r} : \tau \times \tau' \rightarrow$$

그 함수들의 구현:

예) 짝(pair) $\tau \times \tau'$

- ▶ 만들기

$$\text{pair} : \tau * \tau' \rightarrow \tau \times \tau'$$

- ▶ 사용하기

$$\text{l} : \tau \times \tau' \rightarrow \tau$$

$$\text{r} : \tau \times \tau' \rightarrow \tau'$$

그 함수들의 구현:

예) 짝(pair) $\tau \times \tau'$

- ▶ 만들기

$$\text{pair} : \tau * \tau' \rightarrow \tau \times \tau'$$

- ▶ 사용하기

$$\text{l} : \tau \times \tau' \rightarrow \tau$$

$$\text{r} : \tau \times \tau' \rightarrow \tau'$$

그 함수들의 구현:

예) 짝(pair) $\tau \times \tau'$

- ▶ 만들기

$$\text{pair} : \tau * \tau' \rightarrow \tau \times \tau'$$

- ▶ 사용하기

$$\text{l} : \tau \times \tau' \rightarrow \tau$$

$$\text{r} : \tau \times \tau' \rightarrow \tau'$$

그 함수들의 구현:

```
(define pair cons)
```

```
(define l car)
```

```
(define r cdr)
```

예) 리스트(list) $\tau list$

▶ 만들기

`empty` : $\rightarrow \tau list$

`link` : $\rightarrow \tau list$

▶ 사용하기

`is-empty?` : $\tau list \rightarrow$

`fst` : $\tau list \rightarrow$

`rest` : $\tau list \rightarrow$

그 함수들의 구현:

예) 리스트(list) $\tau list$

▶ 만들기

`empty` : $\tau list$
`link` : $\rightarrow \tau list$

▶ 사용하기

`is-empty?` : $\tau list \rightarrow$
`fst` : $\tau list \rightarrow$
`rest` : $\tau list \rightarrow$

그 함수들의 구현:

예) 리스트(list) $\tau list$

▶ 만들기

`empty` : $\tau list$

`link` : $\tau * \tau list \rightarrow \tau list$

▶ 사용하기

`is-empty?` : $\tau list \rightarrow$

`fst` : $\tau list \rightarrow$

`rest` : $\tau list \rightarrow$

그 함수들의 구현:

예) 리스트(list) $\tau list$

▶ 만들기

`empty` : $\tau list$

`link` : $\tau * \tau list \rightarrow \tau list$

▶ 사용하기

`is-empty?` : $\tau list \rightarrow bool$

`fst` : $\tau list \rightarrow$

`rest` : $\tau list \rightarrow$

그 함수들의 구현:

예) 리스트(list) τ list

▶ 만들기

`empty` : τ list

`link` : $\tau * \tau$ list \rightarrow τ list

▶ 사용하기

`is-empty?` : τ list \rightarrow *bool*

`fst` : τ list \rightarrow τ

`rest` : τ list \rightarrow

그 함수들의 구현:

예) 리스트(list) τ list

▶ 만들기

`empty` : τ list

`link` : $\tau * \tau$ list \rightarrow τ list

▶ 사용하기

`is-empty?` : τ list \rightarrow *bool*

`fst` : τ list \rightarrow τ

`rest` : τ list \rightarrow τ list

그 함수들의 구현:

예) 리스트(list) τ list

▶ 만들기

`empty` : τ list

`link` : $\tau * \tau$ list \rightarrow τ list

▶ 사용하기

`is-empty?` : τ list \rightarrow *bool*

`fst` : τ list \rightarrow τ

`rest` : τ list \rightarrow τ list

그 함수들의 구현:

```
(define empty ())
```

```
(define is-empty? null?)
```

```
(define link pair)
```

```
(define fst l)
```

```
(define rest r)
```

예) 두갈래 가지구조(binary tree) τ bintree

▶ 만들기

```
leaf :      →  $\tau$  bintree
node-l :    →  $\tau$  bintree
node-r :    →  $\tau$  bintree
node-lr :   →  $\tau$  bintree
```

▶ 사용하기

```
node-val :  $\tau$  bintree →
is-leaf? :  $\tau$  bintree →
is-ltree? :  $\tau$  bintree →
is-rtree? :  $\tau$  bintree →
is-lrtree? :  $\tau$  bintree →
l-subtree :  $\tau$  bintree →
r-subtree :  $\tau$  bintree →
```

그 함수들의 구현:

예) 두갈래 가지구조(binary tree) τ *bintree*

▶ 만들기

```
leaf   :  $\tau \rightarrow \tau$  bintree
node-l :                 $\rightarrow \tau$  bintree
node-r :                 $\rightarrow \tau$  bintree
node-lr :                 $\rightarrow \tau$  bintree
```

▶ 사용하기

```
node-val :  $\tau$  bintree  $\rightarrow$ 
is-leaf? :  $\tau$  bintree  $\rightarrow$ 
is-ltree? :  $\tau$  bintree  $\rightarrow$ 
is-rtree? :  $\tau$  bintree  $\rightarrow$ 
is-lrtree? :  $\tau$  bintree  $\rightarrow$ 
l-subtree :  $\tau$  bintree  $\rightarrow$ 
r-subtree :  $\tau$  bintree  $\rightarrow$ 
```

그 함수들의 구현:

예) 두갈래 가지구조(binary tree) τ *bintree*

▶ 만들기

```
leaf   :  $\tau \rightarrow \tau$  bintree  
node-l :  $\tau * \tau$  bintree  $\rightarrow \tau$  bintree  
node-r :                 $\rightarrow \tau$  bintree  
node-lr :                 $\rightarrow \tau$  bintree
```

▶ 사용하기

```
node-val :  $\tau$  bintree  $\rightarrow$   
is-leaf? :  $\tau$  bintree  $\rightarrow$   
is-ltree? :  $\tau$  bintree  $\rightarrow$   
is-rtree? :  $\tau$  bintree  $\rightarrow$   
is-lrtree? :  $\tau$  bintree  $\rightarrow$   
l-subtree :  $\tau$  bintree  $\rightarrow$   
r-subtree :  $\tau$  bintree  $\rightarrow$ 
```

그 함수들의 구현:

예) 두갈래 가지구조(binary tree) τ *bintree*

▶ 만들기

```
leaf :  $\tau \rightarrow \tau$  bintree  
node-l :  $\tau * \tau$  bintree  $\rightarrow \tau$  bintree  
node-r :  $\tau * \tau$  bintree  $\rightarrow \tau$  bintree  
node-lr :  $\tau * \tau * \tau$  bintree  $\rightarrow \tau$  bintree
```

▶ 사용하기

```
node-val :  $\tau$  bintree  $\rightarrow \tau$   
is-leaf? :  $\tau$  bintree  $\rightarrow \text{bool}$   
is-ltree? :  $\tau$  bintree  $\rightarrow \text{bool}$   
is-rtree? :  $\tau$  bintree  $\rightarrow \text{bool}$   
is-lrtree? :  $\tau$  bintree  $\rightarrow \text{bool}$   
l-subtree :  $\tau$  bintree  $\rightarrow \tau$  bintree  
r-subtree :  $\tau$  bintree  $\rightarrow \tau$  bintree
```

그 함수들의 구현:

예) 두갈래 가지구조(binary tree) τ *bintree*

▶ 만들기

leaf : $\tau \rightarrow \tau$ *bintree*

node-l : $\tau * \tau$ *bintree* $\rightarrow \tau$ *bintree*

node-r : $\tau * \tau$ *bintree* $\rightarrow \tau$ *bintree*

node-lr : $\tau * \tau$ *bintree* * τ *bintree* $\rightarrow \tau$ *bintree*

▶ 사용하기

node-val : τ *bintree* \rightarrow

is-leaf? : τ *bintree* \rightarrow

is-ltree? : τ *bintree* \rightarrow

is-rtree? : τ *bintree* \rightarrow

is-lrtree? : τ *bintree* \rightarrow

l-subtree : τ *bintree* \rightarrow

r-subtree : τ *bintree* \rightarrow

그 함수들의 구현:

예) 두갈래 가지구조(binary tree) τ *bintree*

▶ 만들기

leaf : $\tau \rightarrow \tau$ *bintree*

node-l : $\tau * \tau$ *bintree* $\rightarrow \tau$ *bintree*

node-r : $\tau * \tau$ *bintree* $\rightarrow \tau$ *bintree*

node-lr : $\tau * \tau$ *bintree* * τ *bintree* $\rightarrow \tau$ *bintree*

▶ 사용하기

node-val : τ *bintree* $\rightarrow \tau$

is-leaf? : τ *bintree* \rightarrow

is-ltree? : τ *bintree* \rightarrow

is-rtree? : τ *bintree* \rightarrow

is-lrtree? : τ *bintree* \rightarrow

l-subtree : τ *bintree* \rightarrow

r-subtree : τ *bintree* \rightarrow

그 함수들의 구현:

예) 두갈래 가지구조(binary tree) τ *bintree*

▶ 만들기

`leaf` : $\tau \rightarrow \tau$ *bintree*

`node-l` : $\tau * \tau$ *bintree* $\rightarrow \tau$ *bintree*

`node-r` : $\tau * \tau$ *bintree* $\rightarrow \tau$ *bintree*

`node-lr` : $\tau * \tau$ *bintree* * τ *bintree* $\rightarrow \tau$ *bintree*

▶ 사용하기

`node-val` : τ *bintree* $\rightarrow \tau$

`is-leaf?` : τ *bintree* \rightarrow *bool*

`is-ltree?` : τ *bintree* \rightarrow *bool*

`is-rtree?` : τ *bintree* \rightarrow *bool*

`is-lrtree?` : τ *bintree* \rightarrow *bool*

`l-subtree` : τ *bintree* \rightarrow

`r-subtree` : τ *bintree* \rightarrow

그 함수들의 구현:

예) 두갈래 가지구조(binary tree) τ *bintree*

▶ 만들기

`leaf` : $\tau \rightarrow \tau$ *bintree*

`node-l` : $\tau * \tau$ *bintree* $\rightarrow \tau$ *bintree*

`node-r` : $\tau * \tau$ *bintree* $\rightarrow \tau$ *bintree*

`node-lr` : $\tau * \tau$ *bintree* * τ *bintree* $\rightarrow \tau$ *bintree*

▶ 사용하기

`node-val` : τ *bintree* $\rightarrow \tau$

`is-leaf?` : τ *bintree* \rightarrow *bool*

`is-ltree?` : τ *bintree* \rightarrow *bool*

`is-rtree?` : τ *bintree* \rightarrow *bool*

`is-lrtree?` : τ *bintree* \rightarrow *bool*

`l-subtree` : τ *bintree* $\rightarrow \tau$ *bintree*

`r-subtree` : τ *bintree* \rightarrow

그 함수들의 구현:

예) 두갈래 가지구조(binary tree) τ *bintree*

▶ 만들기

`leaf` : $\tau \rightarrow \tau$ *bintree*

`node-l` : $\tau * \tau$ *bintree* $\rightarrow \tau$ *bintree*

`node-r` : $\tau * \tau$ *bintree* $\rightarrow \tau$ *bintree*

`node-lr` : $\tau * \tau$ *bintree* * τ *bintree* $\rightarrow \tau$ *bintree*

▶ 사용하기

`node-val` : τ *bintree* $\rightarrow \tau$

`is-leaf?` : τ *bintree* \rightarrow *bool*

`is-ltree?` : τ *bintree* \rightarrow *bool*

`is-rtree?` : τ *bintree* \rightarrow *bool*

`is-lrtree?` : τ *bintree* \rightarrow *bool*

`l-subtree` : τ *bintree* $\rightarrow \tau$ *bintree*

`r-subtree` : τ *bintree* $\rightarrow \tau$ *bintree*

그 함수들의 구현:

예) 두갈래 가지구조(binary tree) τ bintree

▶ 만들기

```
leaf :  $\tau \rightarrow \tau$  bintree  
node-l :  $\tau * \tau$  bintree  $\rightarrow \tau$  bintree  
node-r :  $\tau * \tau$  bintree  $\rightarrow \tau$  bintree  
node-lr :  $\tau * \tau$  bintree *  $\tau$  bintree  $\rightarrow \tau$  bintree
```

▶ 사용하기

```
node-val :  $\tau$  bintree  $\rightarrow \tau$   
is-leaf? :  $\tau$  bintree  $\rightarrow$  bool  
is-ltree? :  $\tau$  bintree  $\rightarrow$  bool  
is-rtree? :  $\tau$  bintree  $\rightarrow$  bool  
is-lrtree? :  $\tau$  bintree  $\rightarrow$  bool  
l-subtree :  $\tau$  bintree  $\rightarrow \tau$  bintree  
r-subtree :  $\tau$  bintree  $\rightarrow \tau$  bintree
```

그 함수들의 구현:

```
(define (leaf x) (pair 'leaf x))  
(define (node-l x t) (pair 'l (pair x t)))  
(define (node-r x t) (pair 'r (pair x t)))
```

...

예) 두갈래 가지구조'(binary tree) τ *bintree*

▶ 만들기

`empty` : $\rightarrow \tau$ *bintree*

`node` : $\rightarrow \tau$ *bintree*

▶ 사용하기

`node-val` : τ *bintree* \rightarrow

`is-empty?` : τ *bintree* \rightarrow

`l-subtree` : τ *bintree* \rightarrow

`r-subtree` : τ *bintree* \rightarrow

그 함수들의 구현:

예) 두갈래 가지구조'(binary tree) τ *bintree*

▶ 만들기

`empty` : τ *bintree*

`node` : $\rightarrow \tau$ *bintree*

▶ 사용하기

`node-val` : τ *bintree* \rightarrow

`is-empty?` : τ *bintree* \rightarrow

`l-subtree` : τ *bintree* \rightarrow

`r-subtree` : τ *bintree* \rightarrow

그 함수들의 구현:

예) 두갈래 가지구조'(binary tree) τ *bintree*

▶ 만들기

`empty` : τ *bintree*

`node` : $\tau * \tau$ *bintree* * τ *bintree* \rightarrow τ *bintree*

▶ 사용하기

`node-val` : τ *bintree* \rightarrow

`is-empty?` : τ *bintree* \rightarrow

`l-subtree` : τ *bintree* \rightarrow

`r-subtree` : τ *bintree* \rightarrow

그 함수들의 구현:

예) 두갈래 가지구조'(binary tree) τ *bintree*

▶ 만들기

`empty` : τ *bintree*

`node` : $\tau * \tau$ *bintree* * τ *bintree* \rightarrow τ *bintree*

▶ 사용하기

`node-val` : τ *bintree* \rightarrow τ

`is-empty?` : τ *bintree* \rightarrow

`l-subtree` : τ *bintree* \rightarrow

`r-subtree` : τ *bintree* \rightarrow

그 함수들의 구현:

예) 두갈래 가지구조'(binary tree) τ *bintree*

▶ 만들기

`empty` : τ *bintree*

`node` : $\tau * \tau$ *bintree* * τ *bintree* \rightarrow τ *bintree*

▶ 사용하기

`node-val` : τ *bintree* \rightarrow τ

`is-empty?` : τ *bintree* \rightarrow *bool*

`l-subtree` : τ *bintree* \rightarrow

`r-subtree` : τ *bintree* \rightarrow

그 함수들의 구현:

예) 두갈래 가지구조'(binary tree) τ *bintree*

▶ 만들기

`empty` : τ *bintree*

`node` : $\tau * \tau$ *bintree* * τ *bintree* \rightarrow τ *bintree*

▶ 사용하기

`node-val` : τ *bintree* \rightarrow τ

`is-empty?` : τ *bintree* \rightarrow *bool*

`l-subtree` : τ *bintree* \rightarrow τ *bintree*

`r-subtree` : τ *bintree* \rightarrow

그 함수들의 구현:

예) 두갈래 가지구조'(binary tree) τ *bintree*

▶ 만들기

`empty` : τ *bintree*

`node` : $\tau * \tau$ *bintree* * τ *bintree* \rightarrow τ *bintree*

▶ 사용하기

`node-val` : τ *bintree* \rightarrow τ

`is-empty?` : τ *bintree* \rightarrow *bool*

`l-subtree` : τ *bintree* \rightarrow τ *bintree*

`r-subtree` : τ *bintree* \rightarrow τ *bintree*

그 함수들의 구현:

예) 두갈래 가지구조'(binary tree) τ *bintree*

▶ 만들기

`empty` : τ *bintree*

`node` : $\tau * \tau$ *bintree* * τ *bintree* \rightarrow τ *bintree*

▶ 사용하기

`node-val` : τ *bintree* \rightarrow τ

`is-empty?` : τ *bintree* \rightarrow *bool*

`l-subtree` : τ *bintree* \rightarrow τ *bintree*

`r-subtree` : τ *bintree* \rightarrow τ *bintree*

그 함수들의 구현:

```
(define empty 'empty)
```

```
(define (node x lt rt) (pair x (pair lt rt)))
```

```
(define (node-val t) ...)
```

```
(define (is-empty? t) ...)
```

...

예) 일반 가지구조(tree) τ tree

▶ 만들기

empty : $\rightarrow \tau$ tree

node : $\rightarrow \tau$ tree

▶ 사용하기

node-val : τ tree \rightarrow

is-empty? : τ tree \rightarrow

num-subtrees : τ tree \rightarrow

nth-subtree : τ tree * int \rightarrow

그 함수들의 구현:

예) 일반 가지구조(tree) τ tree

▶ 만들기

```
empty :  $\tau$  tree
node  :  $\tau$  tree  $\rightarrow$   $\tau$  tree
```

▶ 사용하기

```
node-val :  $\tau$  tree  $\rightarrow$ 
is-empty? :  $\tau$  tree  $\rightarrow$ 
num-subtrees :  $\tau$  tree  $\rightarrow$ 
nth-subtree :  $\tau$  tree * int  $\rightarrow$ 
```

그 함수들의 구현:

예) 일반 가지구조(tree) τ tree

▶ 만들기

```
empty :  $\tau$  tree  
node :  $\tau * (\tau tree) list \rightarrow \tau tree$ 
```

▶ 사용하기

```
node-val :  $\tau tree \rightarrow$   
is-empty? :  $\tau tree \rightarrow$   
num-subtrees :  $\tau tree \rightarrow$   
nth-subtree :  $\tau tree * int \rightarrow$ 
```

그 함수들의 구현:

예) 일반 가지구조(tree) τ tree

▶ 만들기

```
empty :  $\tau$  tree  
node :  $\tau * (\tau tree) list \rightarrow \tau tree$ 
```

▶ 사용하기

```
node-val :  $\tau tree \rightarrow \tau$   
is-empty? :  $\tau tree \rightarrow$   
num-subtrees :  $\tau tree \rightarrow$   
nth-subtree :  $\tau tree * int \rightarrow$ 
```

그 함수들의 구현:

예) 일반 가지구조(tree) τ tree

▶ 만들기

```
empty :  $\tau$  tree  
node :  $\tau * (\tau$  tree) list  $\rightarrow \tau$  tree
```

▶ 사용하기

```
node-val :  $\tau$  tree  $\rightarrow \tau$   
is-empty? :  $\tau$  tree  $\rightarrow$  bool  
num-subtrees :  $\tau$  tree  $\rightarrow$   
nth-subtree :  $\tau$  tree * int  $\rightarrow$ 
```

그 함수들의 구현:

예) 일반 가지구조(tree) τ tree

▶ 만들기

```
empty :  $\tau$  tree  
node  :  $\tau * (\tau tree) list \rightarrow \tau tree$ 
```

▶ 사용하기

```
node-val :  $\tau tree \rightarrow \tau$   
is-empty? :  $\tau tree \rightarrow bool$   
num-subtrees :  $\tau tree \rightarrow int$   
nth-subtree :  $\tau tree * int \rightarrow$ 
```

그 함수들의 구현:

예) 일반 가지구조(tree) τ tree

▶ 만들기

```
empty :  $\tau$  tree  
node :  $\tau * (\tau$  tree) list  $\rightarrow \tau$  tree
```

▶ 사용하기

```
node-val :  $\tau$  tree  $\rightarrow \tau$   
is-empty? :  $\tau$  tree  $\rightarrow bool$   
num-subtrees :  $\tau$  tree  $\rightarrow int$   
nth-subtree :  $\tau$  tree * int  $\rightarrow \tau$  tree
```

그 함수들의 구현:

예) 일반 가지구조(tree) τ tree

▶ 만들기

```
empty :  $\tau$  tree  
node :  $\tau * (\tau tree) list \rightarrow \tau tree$ 
```

▶ 사용하기

```
node-val :  $\tau tree \rightarrow \tau$   
is-empty? :  $\tau tree \rightarrow bool$   
num-subtrees :  $\tau tree \rightarrow int$   
nth-subtree :  $\tau tree * int \rightarrow \tau tree$ 
```

그 함수들의 구현:

```
(define empty 'empty)  
(define (node x trees) (pair x trees))  
(define (node-val t) ...)  
(define (is-empty? t) ...)  
(define (nth-subtree t n) ...)
```

데이터 타입 구현 이슈들

하나의 테이타 타입에 대해서

데이터 타입 구현 이슈들

하나의 테이타 타입에 대해서

- ▶ 여러버전 가능: 만들기/사용하기 함수들의 기획

$$\left. \begin{array}{l} \text{leaf, node-l, node-r, node-lr} \\ \text{node-val, is-leaf?, is-rtree?, is-ltree?, is-lrtree?} \\ \text{l-subtree, r-subtree} \end{array} \right\} \text{v.s.} \left\{ \begin{array}{l} \text{empty, node} \\ \text{node-val, is-empty?} \\ \text{l-subtree, r-subtree} \end{array} \right.$$

데이터 타입 구현 이슈들

하나의 테이타 타입에 대해서

- ▶ 여러버전 가능: 만들기/사용하기 함수들의 기획

$$\left. \begin{array}{l} \text{leaf, node-l, node-r, node-lr} \\ \text{node-val, is-leaf?, is-rtree?, is-ltree?, is-lrtree?} \\ \text{l-subtree, r-subtree} \end{array} \right\} \text{v.s.} \left\{ \begin{array}{l} \text{empty, node} \\ \text{node-val, is-empty?} \\ \text{l-subtree, r-subtree} \end{array} \right.$$

- ▶ 여러버전 가능: 만들기/사용하기 함수들의 구현

$$\left. \begin{array}{l} (\text{define empty } A) \\ (\text{define node } B) \\ (\text{define node-val } C) \\ (\text{define l-subtree } D) \\ (\text{define r-subtree } E) \end{array} \right\} \text{v.s.} \left\{ \begin{array}{l} (\text{define empty } \neg) \\ (\text{define node } \sqcup) \\ (\text{define node-val } \sqsupset) \\ (\text{define l-subtree } \sqcap) \\ (\text{define r-subtree } \sqbox) \end{array} \right.$$

데이터 타입 기획과 구현 원리



새로운 데이터 타입의 구현은 만들기(*introduction*) 함수와 사용하기(*elimination*) 함수들을 정의하면 된다.



이 때, 만들기/사용하기 함수들의 **기획**은 드러내고 **구현**은 감추어서, 외부에서는 기획이 드러난(*interface*, *겉*) 내용만 이용해서 프로그램을 작성하도록 한다.

데이터 타입 기획과 구현 원리



새로운 데이터 타입의 구현은 만들기(*introduction*) 함수와 사용하기(*elimination*) 함수들을 정의하면 된다.



이 때, 만들기/사용하기 함수들의 **기획**은 드러내고 **구현**은 감추어서, 외부에서는 기획이 드러난(*interface*, *겉*) 내용만 이용해서 프로그램을 작성하도록 한다.

- ▶ 기획을 작성하는 언어: 이름, 타입, 하는일
프로그램밍언어 자연어, 수학

데이터 타입 기획과 구현 원리



새로운 데이터 타입의 구현은 만들기(*introduction*) 함수와 사용하기(*elimination*) 함수들을 정의하면 된다.



이 때, 만들기/사용하기 함수들의 **기획**은 드러내고 **구현**은 감추어서, 외부에서는 기획이 드러난(*interface*, **겉**) 내용만 이용해서 프로그램을 작성하도록 한다.

- ▶ 기획을 작성하는 언어: **이름, 타입, 하는일**
프로그래밍언어 자연어, 수학
- ▶ 기획(*interface*)과 구현(*implementation*)은 독립적으로:
데이터 속구현 감추기(*data abstraction*)

다음

- 1 데이터 구현하기 (data implementation)
- 2 데이터 속구현 감추기 (data abstraction)
- 3 여러 구현 동시 지원하기 (multiple implemenations)
- 4 각 계층별로 속구현 감추기(data abstraction hierarchy)

데이터 속구현 감추기(data abstraction)

분리: 외부와 데이터 구현의 속내용

- ▶ 외부에 알릴것. 변하지 말것. (interface)

- ▶ 외부에 알리지 말것. 변해도 될것. (implementation)

장점: 외부는 데이터 구현과 무관

데이터 속구현 감추기(data abstraction)

분리: 외부와 데이터 구현의 속내용

- ▶ 외부에 알릴것. 변하지 말것. (interface)
 - ▶ 만들기/사용하기 함수들의 **기획**: 이름, 타입, 하는일
- ▶ 외부에 알리지 말것. 변해도 될것. (implementation)

장점: **외부는** 데이터 구현과 무관

데이터 속구현 감추기(data abstraction)

분리: 외부와 데이터 구현의 속내용

- ▶ 외부에 알릴것. 변하지 말것. (interface)
 - ▶ 만들기/사용하기 함수들의 **기획**: 이름, 타입, 하는일
 - ▶ 외부에서는 만들기/사용하기 함수들만 **이용하기**
- ▶ 외부에 알리지 말것. 변해도 될것. (implementation)

장점: **외부는 데이터 구현과 무관**

데이터 속구현 감추기(data abstraction)

분리: 외부와 데이터 구현의 속내용

- ▶ 외부에 알릴것. 변하지 말것. (interface)
 - ▶ 만들기/사용하기 함수들의 **기획**: 이름, 타입, 하는일
 - ▶ 외부에서는 만들기/사용하기 함수들만 **이용하기**
- ▶ 외부에 알리지 말것. 변해도 될것. (implementation)
 - ▶ 만들기/사용하기 함수들의 **구현**

장점: **외부는 데이터 구현과 무관**

데이터 속구현 감추기(data abstraction)

분리: 외부와 데이터 구현의 속내용

- ▶ 외부에 알릴것. 변하지 말것. (interface)
 - ▶ 만들기/사용하기 함수들의 **기획**: 이름, 타입, 하는일
 - ▶ 외부에서는 만들기/사용하기 함수들만 **이용하기**
- ▶ 외부에 알리지 말것. 변해도 될것. (implementation)
 - ▶ 만들기/사용하기 함수들의 **구현**

장점: **외부는** 데이터 **구현과 무관**

- ▶ 데이터구현 변경과 외부사용 코드변경이 독립됨

데이터 속구현 감추기(data abstraction)

분리: 외부와 데이터 구현의 속내용

- ▶ 외부에 알릴것. 변하지 말것. (interface)
 - ▶ 만들기/사용하기 함수들의 **기획**: 이름, 타입, 하는일
 - ▶ 외부에서는 만들기/사용하기 함수들만 **이용하기**
- ▶ 외부에 알리지 말것. 변해도 될것. (implementation)
 - ▶ 만들기/사용하기 함수들의 **구현**

장점: **외부는 데이터 구현과 무관**

- ▶ 데이터구현 변경과 외부사용 코드변경이 독립됨
- ▶ 데이터구현과 외부사용 프로그래밍 동시진행 가능

예) 짝 $\tau \times \tau'$

- ▶ 기획 (interface)

`pair` : $\tau * \tau' \rightarrow \tau \times \tau'$

`l` : $\tau \times \tau' \rightarrow \tau$

`r` : $\tau \times \tau' \rightarrow \tau'$

- ▶ 구현은 기획에 맞기만하면 다양하게 가능
 - ▶ 안1: `cons`, `car`, `cdr`
 - ▶ 안2: `lambda`
- ▶ 외부사용

예) 짝 $\tau \times \tau'$

- ▶ 기획 (interface)

$$\begin{aligned}\text{pair} &: \tau * \tau' \rightarrow \tau \times \tau' \\ \text{l} &: \tau \times \tau' \rightarrow \tau \\ \text{r} &: \tau \times \tau' \rightarrow \tau'\end{aligned}$$

- ▶ 구현은 기획에 맞기만하면 다양하게 가능

- ▶ 안1: cons, car, cdr

- ▶ 안2: lambda

- ▶ 외부사용

```
(define (leaf x) (pair 'leaf x))
```

```
(define (node-l x t) (pair 'r (pair x t)))
```

```
(define (node-r x t) (pair 'l (pair x t)))
```

```
(define (l-subtree t) (if (equal (l t) 'l) (r (r t))))
```

```
(define (r-subtree t) (if (equal (l t) 'r) (r (r t))))
```

예) 두갈래 가지구조'(tree) τ *bintree*

- ▶ 기획 (interface)

`empty` : τ *bintree*

`node` : $\tau * \tau$ *bintree* * τ *bintree* \rightarrow τ *bintree*

`node-val` : τ *bintree* \rightarrow τ

`is-empty?` : τ *bintree* \rightarrow *bool*

`l-subtree` : τ *bintree* \rightarrow τ *bintree*

`r-subtree` : τ *bintree* \rightarrow τ *bintree*

- ▶ 외부사용

예) 두갈래 가지구조'(tree) τ *bintree*

- ▶ 기획 (interface)

`empty` : τ *bintree*

`node` : $\tau * \tau$ *bintree* * τ *bintree* \rightarrow τ *bintree*

`node-val` : τ *bintree* \rightarrow τ

`is-empty?` : τ *bintree* \rightarrow *bool*

`l-subtree` : τ *bintree* \rightarrow τ *bintree*

`r-subtree` : τ *bintree* \rightarrow τ *bintree*

- ▶ 외부사용

(node 10 (node 8 (node 5 empty empty) (node 9 empty empty)) empty)

(node 1 empty (node 2 empty (node 3 empty empty)))

(define (traverse t)

 (if (is-empty? t) ()

 (begin (print (node-val t))

 (traverse (l-subtree t)) (traverse (r-subtree t)))

))

연습: 데이터 구현하기 + 속구현 감추기

- ▶ 대수식(algebraic expression) 데이터
 - ▶ 미분함수(symbolic differentiation)

$$\text{diff} : ae * string \rightarrow ae$$

- ▶ 부분계산함수(partial evaluation)

$$\text{eval} : ae * string * real \rightarrow ae$$

다음

- 1 데이터 구현하기 (data implementation)
- 2 데이터 속구현 감추기 (data abstraction)
- 3 여러 구현 동시 지원하기 (multiple implemenations)
- 4 각 계층별로 속구현 감추기(data abstraction hierarchy)

여러 구현 동시지원: 예) 복소수 데이터

“표면” 단계

▶ 기획(interface)

`make-from-real-imag` : $real * real \rightarrow complex$

`make-from-mag-angle` : $real * real \rightarrow complex$

`is-complex?` : $\top \rightarrow bool$

`real` : $complex \rightarrow real$

`imag` : $complex \rightarrow real$

`mag` : $complex \rightarrow real$

`angle` : $complex \rightarrow real$

외부 사용

`add-complex` : $complex * complex \rightarrow complex$

`mul-complex` : $complex * complex \rightarrow complex$

...

여러 구현 동시지원: 예) 복소수 데이터

한가지 방식만 구현한 경우

add-complex mul-complex

make-from-real-imag make-from-mag-angle real img mag angle is-complex?
rectangular representation

여러 구현 동시지원: 예) 복소수 데이터

한가지 방식만 구현한 경우

add-complex mul-complex

make-from-real-imag make-from-mag-angle real imag mag angle is-complex?
rectangular representation

```
(define (make-from-real-imag r i) ...)  
(define (make-from-mag-angle m a) ...)  
(define (is-complex? x) ...)  
(define (real x) ...)  
(define (imag x) ...)  
(define (mag x) ...)  
  
(define (angle x) ...)
```

여러 구현 동시지원: 예) 복소수 데이터

두가지 방식의 구현을 같이 이용하는 경우

add-complex mul-complex

make-from-real-imag make-from-mag-angle real imag mag angle is-complex?	
*-rectangular	*-polar
rectangular representation	polar representation

여러 구현 동시지원: 예) 복소수 데이터

두가지 방식의 구현을 같이 이용하는 경우

add-complex mul-complex

make-from-real-imag make-from-mag-angle real imag mag angle is-complex?	
*-rectangular	*-polar
rectangular representation	polar representation

```
(define (make-from-real-imag r i) ...)  
(define (make-from-mag-angle m a) ...)  
(define (is-complex? x) ...)  
(define (real x) ...)  
(define (imag x) ...)  
(define (mag x) ...)
```

```
(define (angle x) ...)
```

예) 복소수 데이터 구현A

“rectangular representation”

▶ 기획 (interface)

`make-from-real-imag-rectangular` : $real * real \rightarrow complex$

`make-from-mag-angle-rectangular` : $real * real \rightarrow complex$

`is-rectangular?` : $complex \rightarrow bool$

`real-rectangular` : $complex \rightarrow real$

`imag-rectangular` : $complex \rightarrow real$

`mag-rectangular` : $complex \rightarrow real$

`angle-rectangular` : $complex \rightarrow real$

예) 복소수 데이터 구현B

“polar representation”

▶ 기획 (interface)

`make-from-real-imag-polar` : $real * real \rightarrow complex$

`make-from-mag-angle-polar` : $real * real \rightarrow complex$

`is-polar?` : $complex \rightarrow bool$

`real-polar` : $complex \rightarrow real$

`imag-polar` : $complex \rightarrow real$

`mag-polar` : $complex \rightarrow real$

`angle-polar` : $complex \rightarrow real$

여러 구현방식을 지원할 때의 원리



데이터 속구현 감추기(*data abstraction*) 원리를 유지한다.

- ▶ 즉, 각 구현방식마다 만들기와 사용하기 함수를 제공한다.
- ▶ 단, 그 함수들의 이름이 다른 구현방식과 구별되게 한다.
- ▶ 그리고, “표면” 단계의 만들기와 사용하기 함수들을 아랫단계의 여러 구현방식들을 이용해서 정의한다.

새로운 구현방식도 사용하도록 확장하려면?

같은 원리로:

add-complex mul-complex

make-from-real-imag make-from-mag-angle real imag mag angle is-complex?		
*-rectangular	*-polar	*-xyz
rectangular representation	polar representation	xyz representation

그리고, 바꿀 것은?

새로운 구현방식도 사용하도록 확장하려면?

같은 원리로:

add-complex mul-complex

make-from-real-imag make-from-mag-angle real imag mag angle is-complex?		
*-rectangular	*-polar	*-xyz
rectangular representation	polar representation	xyz representation

그리고, 바꿀 것은?

- ▶ “표면” 단계의 만들기와 사용하기 함수들:

make-from-real-imag, ..., angle, is-complex?

새로운 구현방식을 첨가/삭제하기 더 쉬운 방법?

“표면” 단계의 함수들 모두를 **매번 변경**해주는 번거로움:

```
(define (real x)
  (cond ((is-rectangular? x) (real-rectangular x))
        ((is-polar? x) (real-polar x))
        (else (error))))
```

```
(define (angle x)
  (cond ((is-rectangular? x) (angle-rectangular x))
        ((is-polar? x) (angle-polar x))
        (else (error))))
```

새로운 구현방식을 첨가/삭제하기 더 쉬운 방법?

“표면” 단계의 함수들 모두를 **매번 변경**해주는 번거로움:

```
(define (real x)
  (cond ((is-rectangular? x) (real-rectangular x))
        ((is-polar? x) (real-polar x))
        ((is-xyz? x) (real-xyz x))
        (else (error))))
```

```
(define (angle x)
  (cond ((is-rectangular? x) (angle-rectangular x))
        ((is-polar? x) (angle-polar x))
        ((is-xyz? x) (angle-xyz x))
        (else (error))))
```

새로운 구현방식을 첨가/삭제하기 더 쉬운 방법?

- ▶ 2차원의 함수 테이블 사용. (테이블은 변환 수 있는 물건으로)
 - ▶ 가로: 함수이름 태그 ('real', 'imag, 등등)
 - ▶ 세로: 구현방식 태그 ('rectangular', 'polar, 등등)

	'real	'imag	...
'rectangular	real-rectangular	imag-rectangular	...
'polar	real-polar	imag-polar	...

||

새로운 구현방식을 첨가/삭제하기 더 쉬운 방법?

- ▶ 2차원의 함수 테이블 사용. (테이블은 변환 수 있는 물건으로)
 - ▶ 가로: 함수이름 태그 ('real, 'imag, 등등)
 - ▶ 세로: 구현방식 태그 ('rectangular, 'polar, 등등)

	'real	'imag	...
'rectangular	real-rectangular	imag-rectangular	...
'polar	real-polar	imag-polar	...

- ▶ 새로운 구현방식? 해당 함수들을 테이블에 등록

새로운 구현방식을 첨가/삭제하기 더 쉬운 방법?

- ▶ 2차원의 함수 테이블 사용. (테이블은 변환 수 있는 물건으로)
 - ▶ 가로: 함수이름 태그 ('real', 'imag, 등등)
 - ▶ 세로: 구현방식 태그 ('rectangular', 'polar, 등등)

	'real	'imag	...
'rectangular	real-rectangular	imag-rectangular	...
'polar	real-polar	imag-polar	...
'xyz	real-xyz	imag-xyz	...

- ▶ 새로운 구현방식? 해당 함수들을 테이블에 등록

새로운 구현방식을 첨가/삭제하기 더 쉬운 방법?

- ▶ 2차원의 함수 테이블 사용. (테이블은 변환 수 있는 물건으로)
 - ▶ 가로: 함수이름 태그 ('real, 'imag, 등등)
 - ▶ 세로: 구현방식 태그 ('rectangular, 'polar, 등등)

	'real	'imag	...
'rectangular	real-rectangular	imag-rectangular	...
'polar	real-polar	imag-polar	...
'xyz	real-xyz	imag-xyz	...

- ▶ 새로운 구현방식? 해당 함수들을 테이블에 등록
- ▶ “표면”의 함수들은 고정됨
 - ▶ 두 개의 태그(함수이름, 구현방식)로 테이블에서 가져온다

```
(define (real x) ((lookup ftn-tbl 'real (rep-tag x)) x))
```

```
(define (imag x) ((lookup ftn-tbl 'imag (rep-tag x)) x))
```

다음

- 1 데이터 구현하기 (data implementation)
- 2 데이터 속구현 감추기 (data abstraction)
- 3 여러 구현 동시 지원하기 (multiple implemenations)
- 4 각 계층별로 속구현 감추기(data abstraction hierarchy)

계층별로 속구현 감추기 원리

여러 구현방식을 지원할 때의 원리와 같다:



데이터 속구현 감추기(*data abstraction*) 원리를 유지한다.

- ▶ 즉, 각 구현방식마다 만들기와 사용하기 함수를 제공한다.
- ▶ 단, 그 함수들의 이름이 다른 구현방식과 구별되게 한다.
- ▶ 그리고, “표면” 단계의 만들기와 사용하기 함수들을 아랫단계의 여러 구현방식들을 이용해서 정의한다.