SNU 4190.210 프로그래밍 원리(Principles of Programming) Part II

Prof. Chung-Kil Hur

차례

다음

데이타 구현하기 (data implementation)

새로운 타입의 데이타/값 구현하기

- 기억하는가: 타입들(types)

τ	::=	ι	primitive type
	|	$\tau \times \tau$	pair(product) type
	|	$\tau + \tau$	or(sum) type
	|	$\tau \to \tau$	ftn type, single param
	|	$\tau * \cdots * \tau \to \tau$	ftn type, multi params
	|	$\top$	any type
	|	t	user-defined type's name
	|	$\tau\ t$	user-defined type's name, with param
ι	::=	*int* | *real* | *bool* | *string* | *symbol* | *unit*	

- 새로운 타입(t 혹은 $\tau\ t$)의 데이타 구현해야

새로운 데이타 타입

- 기본으로 제공되는 타입들

  int, real, bool, string, symbol, unit

  이 아닌
- 소프트웨어마다 새로운 타입의 데이타/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 데이타 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 사용하면된다.

새로운 데이타 타입 구현하기

- 새로운 타입을 τ 라고 하면
  - 만들기 함수들의 타입은

$$\cdots \to \tau$$

  - 사용하기 함수들의 타입은

$$\tau \to \cdots$$

예) 짝(pair) $\tau \times \tau'$

- 만들기

$$\texttt{pair} : \tau * \tau' \to \tau \times \tau'$$

- 사용하기

$$\begin{array}{rcl} \texttt{l} & : & \tau \times \tau' \to \tau \\ \texttt{r} & : & \tau \times \tau' \to \tau' \end{array}$$

그 함수들의 구현:

```
(define pair cons)
(define l car)
(define r cdr)
```

예) 합(sum) $\tau + \tau'$

- 만들기

$$\begin{aligned} \texttt{inl} &: \tau \to \tau + \tau' \\ \texttt{inr} &: \tau' \to \tau + \tau' \end{aligned}$$

- 사용하기

$$\begin{aligned} \texttt{is-left?} &: \tau + \tau' \to \mathit{bool} \\ \texttt{left} &: \tau + \tau' \to \tau \\ \texttt{right} &: \tau + \tau' \to \tau' \end{aligned}$$

그 함수들의 구현:

```
(define (inl x) (pair 'left x))
(define (inr y) (pair 'right y))
(define (is-left? t) (?eq (l t) 'left))
(define (left t) (r t))
(define (right t) (r t))
```


예) 리스트(list) τ *list*

- 만들기

$$\begin{aligned} \texttt{empty} &: \tau\ \mathit{list} \\ \texttt{link} &: \tau * \tau\ \mathit{list} \to \tau\ \mathit{list} \end{aligned}$$

- 사용하기

$$\begin{aligned} \texttt{is-empty?} &: \tau\ \mathit{list} \to \mathit{bool} \\ \texttt{fst} &: \tau\ \mathit{list} \to \tau \\ \texttt{rest} &: \tau\ \mathit{list} \to \tau\ \mathit{list} \\ &\text{vs.} \\ \texttt{length} &: \tau\ \mathit{list} \to \mathit{int} \\ \texttt{nth-elmt} &: \tau\ \mathit{list} * \mathit{int} \to \tau \end{aligned}$$

그 함수들의 구현:

```
(define empty ())
(define is-empty? null?)
(define link pair) → (define link cons)
(define fst l) → (define fst car)
(define rest r) → (define rest cdr)
```

예) 두갈래 가지구조(binary tree) τ *bintree*

- 만들기

$$\begin{aligned} \texttt{leaf} &: \tau \to \tau\ \mathit{bintree} \\ \texttt{node} &: \tau * \tau\ \mathit{bintree} * \tau\ \mathit{bintree} \to \tau\ \mathit{bintree} \end{aligned}$$

- 사용하기

$$\begin{aligned} \texttt{is-leaf?} &: \tau\ \mathit{bintree} \to \mathit{bool} \\ \texttt{node-val} &: \tau\ \mathit{bintree} \to \tau \\ \texttt{l-subtree} &: \tau\ \mathit{bintree} \to \tau\ \mathit{bintree} \\ \texttt{r-subtree} &: \tau\ \mathit{bintree} \to \tau\ \mathit{bintree} \end{aligned}$$

그 함수들의 구현:

~~(define (leaf x) (pair x 'leaf))~~ → (define (leaf x) (pair 'leaf x))
~~(define (node x lt rt) (pair x (pair lt rt)))~~
→ (define (node x lt rt) (pair 'node (pair x (pair lt rt))))
~~(define (is-leaf? t) (?eq (r t) 'leaf))~~ → (define (is-leaf? t) (?eq (l t) 'leaf))
~~(define (node-val t) (l t))~~
→ (define (node-val t) (if (is-leaf? t) (r t) (l (r t))))
...

예) 일반 가지구조(tree) τ *tree*

- 만들기

$$\texttt{node} \quad : \quad \tau * (\tau\ \mathit{tree})\ \mathit{list} \to \tau\ \mathit{tree}$$

- 사용하기

$$\begin{aligned} \texttt{node-val} &: \tau\ \mathit{tree} \to \tau \\ \texttt{node-subtrees} &: \tau\ \mathit{tree} \to (\tau\ \mathit{tree})\ \mathit{list} \end{aligned}$$

그 함수들의 구현:

```
(define (node x trees) (pair x trees)
(define (node-val t) (l t))
(define (node-subtrees t) (r t))
```

데이타 타입 구현 이슈들

하나의 테이타 타입에 대해서

- 여러버전 가능: 만들기/사용하기 함수들의 기획

```
                                leaf, node-l, node-r, node-lr
node-val, is-leaf?, is-rtree?, is-ltree?, is-lrtree?        } v.s. { empty, node
                                l-subtree, r-subtree                 node-val, is-empty?
                                                                     l-subtree, r-subtree
```

- 여러버전 가능: 만들기/사용하기 함수들의 구현

$$\left.\begin{array}{r}(\text{define empty } A)\\(\text{define node } B)\\(\text{define node-val } C)\\(\text{define l-subtree } D)\\(\text{define r-subtree } E)\end{array}\right\} \text{v.s.} \left\{\begin{array}{l}(\text{define empty } \text{ㄱ})\\(\text{define node } \text{ㄴ})\\(\text{define node-val } \text{ㄷ})\\(\text{define l-subtree } \text{ㄹ})\\(\text{define r-subtree } \text{ㅁ})\end{array}\right.$$

데이타 타입 기획과 구현 원리

♣

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

♣

이 때, 만들기/사용하기 함수들의 기획은 드러내고 구현은 감추어서, 외부에서는 기획이 드러낸(interface, 겉) 내용만 이용해서 프로그램을 작성하도록 한다.

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

다음

데이타 속구현 감추기(data abstraction)

분리: 외부와 데이타 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

- 데이타구현 변경과 외부사용 코드변경이 독립됨
- 데이타구현과 외부사용 프로그래밍 동시진행 가능

예) 짝 $\tau \times \tau'$

- 기획 (interface)

$$\begin{array}{rcl} \texttt{pair} & : & \tau * \tau' \to \tau \times \tau' \\ \texttt{l} & : & \tau \times \tau' \to \tau \\ \texttt{r} & : & \tau \times \tau' \to \tau' \end{array}$$

- 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)

$$\begin{array}{rcl} \texttt{leaf} & : & \tau \to \tau \ \textit{bintree} \\ \texttt{node} & : & \tau * \tau \ \textit{bintree} * \tau \ \textit{bintree} \to \tau \ \textit{bintree} \\ \texttt{is-leaf?} & : & \tau \ \textit{bintree} \to \textit{bool} \\ \texttt{node-val} & : & \tau \ \textit{bintree} \to \tau \\ \texttt{l-subtree} & : & \tau \ \textit{bintree} \to \tau \ \textit{bintree} \\ \texttt{r-subtree} & : & \tau \ \textit{bintree} \to \tau \ \textit{bintree} \end{array}$$

- 외부사용

```
(node 10 (node 8 (leaf 5) (leaf 9)) (leaf 7))
(node 1 (leaf 5) (node 2 (leaf 4) (leaf 3)))

(define (traverse t)
  (begin (print (node-val t))
    (if (is-leaf? t) ()
      (begin (traverse (l-subtree t)) (traverse (r-subtree t))))))
```

연습: 데이터 구현하기 + 속구현 감추기

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

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

  - 부분계산함수(partial evaluation)

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

다음

여러 구현 동시지원: 예) 복소수 데이타

"표면" 단계

- 기획(interface)

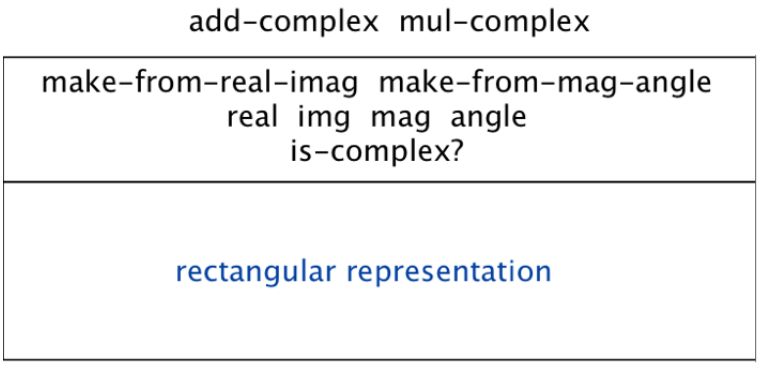
$$
\begin{aligned}
\texttt{make-from-real-imag} &: \mathit{real} * \mathit{real} \to \mathit{complex} \\
\texttt{make-from-mag-angle} &: \mathit{real} * \mathit{real} \to \mathit{complex} \\
\texttt{is-complex?} &: \top \to \mathit{bool} \\
\texttt{real} &: \mathit{complex} \to \mathit{real} \\
\texttt{imag} &: \mathit{complex} \to \mathit{real} \\
\texttt{mag} &: \mathit{complex} \to \mathit{real} \\
\texttt{angle} &: \mathit{complex} \to \mathit{real}
\end{aligned}
$$

외부 사용

$$
\begin{aligned}
\texttt{add-complex} &: \mathit{complex} * \mathit{complex} \to \mathit{complex} \\
\texttt{mul-complex} &: \mathit{complex} * \mathit{complex} \to \mathit{complex} \\
&\ldots
\end{aligned}
$$

여러 구현 동시지원: 예) 복소수 데이타

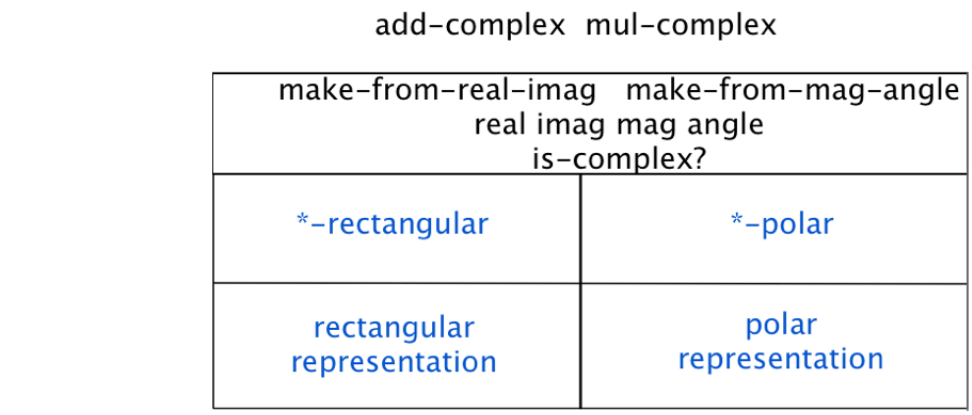
한가지 방식만 구현한 경우




```
(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 데이타

두가지 방식의 구현을 같이 이용하는 경우




```
(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 데이타 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 데이타 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

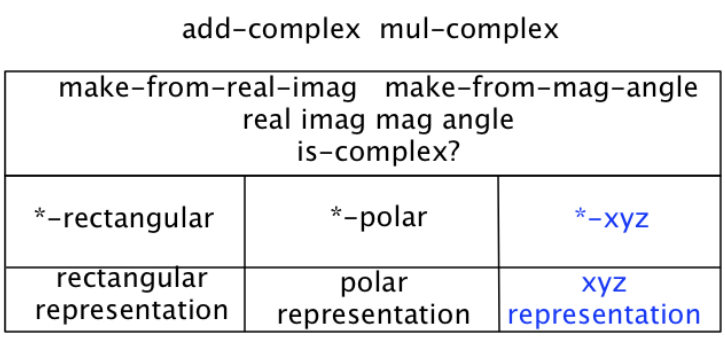
♣

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:



그리고, 바꿀 것은?

- "표면" 단계의 만들기와 사용하기 함수들:
  `make-from-real-imag`, $\cdots$, `angle`, `is-complex?`

새로운 구현방식을 첨가/삭제하기 더 쉬운 방법?

"표면" 단계의 함수들 모두를 매번 변경해주는 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))))
```

$\vdots$

새로운 구현방식을 첨가/삭제하기 더 쉬운 방법?

- 2차원의 함수 테이블 사용. (테이블은 변할 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

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.*