VL Nichtprozedurale Programmierung

Funktionale Programmierung

Andreas Naderlinger
Fachbereich Computerwissenschaften
Software Systems Center
Universität Salzburg

UNIVERSITÄT SALZBURG

Some good news…

- We are done with the syntax !
  (more or less)

- Now: time for some **big ideas**

  - **Today**
    - **Procedures (cont'd)**
    - **Data**

Procedures (cont'd)

- Last lecture we covered
  - Recursion
    - elegant
    - avoids mutation
    - can be efficiently implemented (tail recursion)
  - Higher-order procedures
    - take a procedure as parameter OR
    - return a procedure (covered today)
  - Lambda
    - anonymous procedures

FuncProg_02_higherOrderFuncs.java

BUILDING ABSTRACTIONS WITH DATA

Building Abstractions with Data

- Today: excerpt from SICP chapter 2
  - Use functions to create and encapsulate data structures.

Encapsulation:
bundling of data with the methods that operate on that data

(also a mechanism to restrict access to some elements within a component → *information hiding*)

  - Example rational numbers
    - $\frac{x}{y}$
      - numerator x (integer)
      - denominator y (integer)
    - We'll write code to do rational arithmetic

Addition (+): A naïve approach (with what we have learned so far)

- $\frac{n1}{d1}$ …….. two integers

$$\frac{n1}{d1} + \frac{n2}{d2} = \frac{n1 * d2 + n2 * d1}{d1 * d2}$$

- Addition: Implement two procedures
  - One to get the numerator of the sum

```
(define (addRatNum n1 d1 n2 d2)
  (+ (* n1 d2)
     (* n2 d1)))
```

  - One to get the denominator of the sum

```
(define (addRatDenom n1 d1 n2 d2)
  (* d1 d2))
```

- Usage: $\frac{1}{2} + \frac{1}{2}$

```
(addRatNum 1 2 1 2)   ;4
(addRatDenom 1 2 1 2) ;4
```

$\rightarrow \frac{4}{4} = 1$ (Demo3a)

Data Abstraction → **Compound Data**

- Wishful thinking
  - We assume that we already have a way of **constructing** a rational number from a numerator and a denominator.
  - We further assume that, given a rational number, we have a way of extracting (or **selecting**) its numerator and denominator.

- Constructor

```
(make-rat <n> <d>)
```

returns the rational number whose numerator is the integer <n> and whose denominator is the integer <d>.

- Selectors

```
(numer <x>)
```

returns the numerator of the rational number <x>.

```
(denom <x>)
```

returns the denominator of the rational number <x>

½ ½

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
```

$$\frac{n1}{d1} + \frac{n2}{d2} = \frac{n1*d2+n2*d1}{d1*d2}$$

```
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

But we haven't yet defined make-rat, numer, denom. What do we need for this?

Pairs

- Racket provides a compound structure ‘pair’.

- Constructor
  - `cons`

- Selector (names have historical reasons)
  - `car` →1st element
  - `cdr` →2nd element (pronounced “could-er”)

  - `(define p (cons 1 2))`
  - `(car p)`
  - `(cdr p)`

Pairs (2)

- `(define x (cons 1 2))`
  `(define y (cons 3 4))`
  `(define z (cons x y))`

- z: ((1,2),(3,4))
  - `(car (car z))`

    1
  - `(car (cdr z))`

    3

Demo3b

Example: Rational numbers (cont'd)

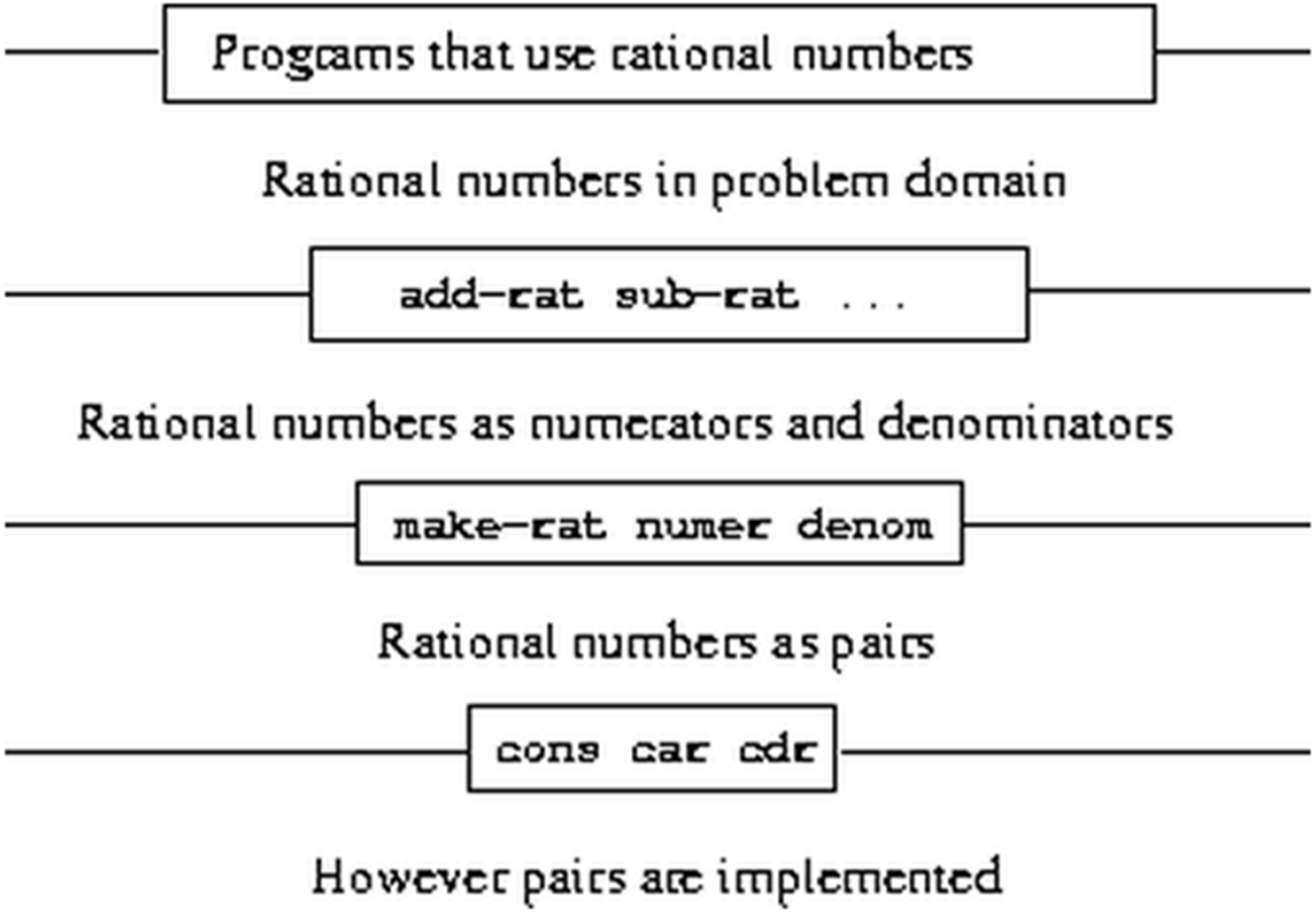
- `(define (make-rat n d)                    )`
- `(define (numer x)            )`
- `(define (denom x)            )`

Example: Rational numbers (cont'd)

- `(define (make-rat n d) (cons n d))`
- `(define (numer x) (car x))`
- `(define (denom x) (cdr x))`

Demo3c

Data-abstraction barriers



Sample Benefit

- Reducing the fraction in constructor

```
(define (make-rat n d)
  (define g (gcd n d))
  (cons (/ n g) (/ d g)))
```

- Reducing the fraction in selectors

```
(define (numer x)
  (define g (gcd (car x) (cdr x)))
  (/ (car x) g))

(define (denom x)
  (define g (gcd (car x) (cdr x)))
  (/ (cdr x) g))
```

What is meant by Data ?

- We don't know what pairs are. But we know that Racket provides `cons`, `car`, `cdr` to work with them.

- They are built-in but we can also define them ourselves. (→ <u>really cool. See next slide</u>)

- The only thing we need to know about these three operations is that if we glue two objects together using `cons` we can retrieve the objects using `car` and `cdr`.
- → for any objects `x` and `y`,
  if `z` is `(cons x y)` then
  `(car z)` is `x` and `(cdr z)` is `y`.

How to define **cons/car/cdr** ourselves?

cons | car | cdr

Example:

(define r (mycons 3 4))

(mycar r) → 3

```
(define (mycons x y)
  (define (dispatch m)
    (cond [(= m 0) x]
          [(= m 1) y]
          [else (error "argument not 0 or 1 -- in mycons" m)]))
  dispatch)
```

Procedure as Returned Value

```
(define (mycar z) ______)

(define (mycdr z) ______)
```

cons | car | cdr

```
(define (mycons x y)
  (define (dispatch m)
    (cond [(= m 0) x]
          [(= m 1) y]
          [else (error "argument not 0 or 1 -- in mycons" m)]))
  dispatch)
```

Example:

```
(define r (mycons 3 4))
(mycar r)    → 3
```

```
(define (mycar z) (z 0))

(define (mycdr z) (z 1))
```

- → no data structures, just procedures
- → the ability to manipulate procedures as objects automatically provides the ability to represent compound data
- Note: Racket implements pairs directly (performance)

Demo3d

cons | car | cdr with λ

```
(define (cons2 x y)
  (λ (m)
    (cond [(= m 0) x]
          [(= m 1) y]
          [else (error "Argument not 0 or 1 -- in cons2" m)])))
```

```
(define (cons3 x y)
    (λ (m) (m x y)))

(define (car3 z)
    (z (λ (p q) p)))

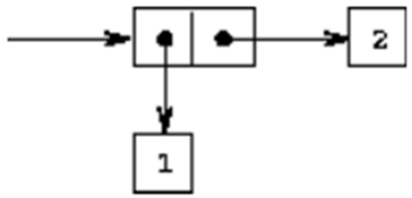
(define (cdr3 z)
    (z (λ (p q) q)))
```

What is going on here?

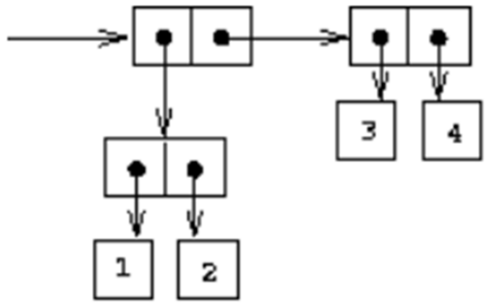
Demo3e

Hierarchical Data (box-pointer-representation)

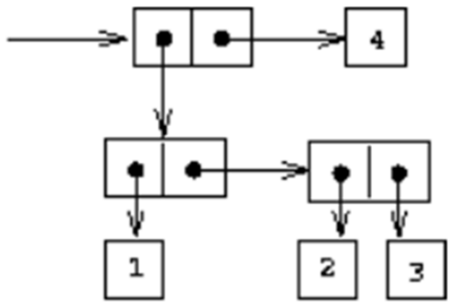
- (cons 1 2)



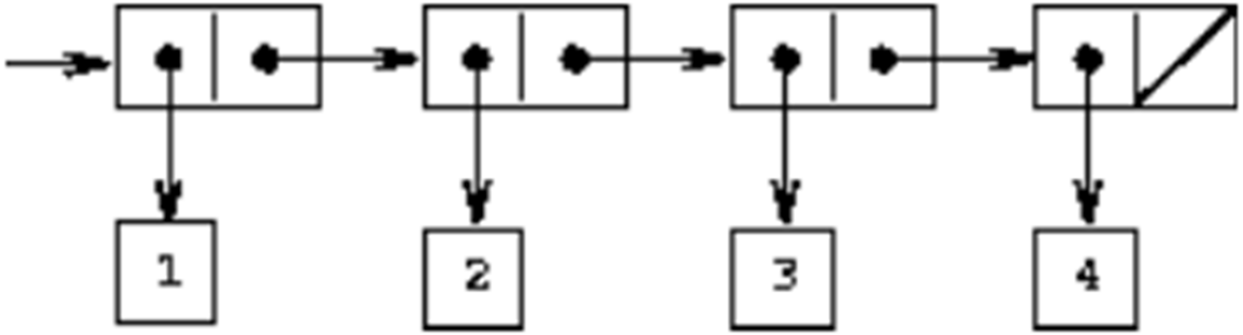
- (cons (cons 1 2)
          (cons 3 4))



- (cons (cons 1
                    (cons 2 3))
          4)



Sequences (by chaining pairs)



- (cons 1
        (cons 2
              (cons 3
                    (cons 4 null))))

(cons <a_1> (cons <a_2> (cons ...(cons <a_n> null) ...)))

is the same as

(list <a_1> <a_2> ... <a_n>)

- (list 1 2 3 4)

List operations

- `(define mylist (list 1 2 3 4))`

- `(car mylist)`
- `(cdr mylist)`
- `(car (cdr mylist))`
- `(car (cdr (cdr mylist)))`

Demo3f

Lists

- To form a list, we always start with the **empty list**:
  - `` `() ``
  - `null`
  - `empty`
  - `(list)`

In Lisp/Scheme the empty list was ***`nil`*** (from latin *nihil*).

For compatibility:

`(define nil null)`

- `(cons 1 null)`
- `(cons 1 (cons 2 (cons 3 null)))`

- Check for emptiness
  - `null?`
  - `empty?`

Lists 2: **car|cdr** vs. **first|rest**

- `(define mylist (list 1 2 3 4))`

- (**car** mylist)
- (**cdr** mylist)
- `(car (cdr mylist))`
- `(car (cdr (cdr mylist)))`

vs.

- (**first** mylist)
- (**rest** mylist)
- `(first (rest mylist))`
- `(first (rest (rest mylist)))`

- (c**add**r mylist) ← (c**a**r (c**d**r (c**d**r mylist)))

(note: there are only a few shortcuts implemented)

Lists 3

- Lists may contain different types of data:

Symbol: a sequence of characters preceded by a single quotation mark (no blanks)

  - (list #t 4 'aSymbol "a String" 5)

true

  - (list (list 1 2) (list 3 4))
  - (list (list #t 2) (list #f 4) 'S)

false

Examples [cons | list]

(see Demo3f for
the code up to this slide)

- `(cons 1 2 3)`
  - `;error: cons only takes two arguments`
- `(cons 1 2)`
  - `;makes a pair '(1 . 2)`
- `(cons (list 1) 2)`
  - `;makes a pair '((1) . 2)`
- `(cons 1 (cons 2 null))`
  - `;makes a list (because of the empty list on the right)   '(1 2)`
- `(cons 1 (list 2))`
  - `;makes a list '(1 2)   ... so it takes the first argument, ie. 1, and makes it the first entry of the list`
- `(cons (list 1) (list 2))`
  - `;makes a list '((1) 2)   ... again it takes the first argument, ie. a list, and makes it the first entry in the list`
- `(list (list 1) (list 2))`
  - `;makes a list of two elements (namely the original lists)   '((1) (2))`
- `(append (list 1 2) (list 3 4))`   (just for ‘completeness’ … see later)
  - `; '(1 2 3 4)`

List operations

- Get the n-th element of the list
- ```
  (define (list-ref items n)
     (if (= n 0)
         (first items)
         (list-ref (rest items) (- n 1))))
  ```
  ```

List operations

- Get the n-th element of the list
- `(define (list-ref items n)`

```
conditional (if (= n 0)
                (first items) base case
                (list-ref (rest items) (- n 1))))
                                          self-referential case
```

List operations (2)

- Get the n-th element of the list
- `(define (list-ref items n)`

```
conditional (if (= n 0)
                (first items)  base case
                (list-ref (rest items) (- n 1))))
                                         self-referential case
```

- Get the length of the list
- `(define (length items)`

```
    (if (null? items)
        0
        (+ 1 (length (rest items)))))
```

Demo3g

That’s it for today

References
-SICP
-HTDP

Thank you!