CS339: Abstractions and Paradigms for Programming

Data Abstraction: Pairs and Lists

Manas Thakur
CSE, IIT Bombay

Autumn 2024

Let's work with "rational" numbers

- Say we have the following procedures available (*wishful thinking*):
  - `(make-rat n d)`, which gives a rational number with numerator `n` and denominator `d`
  - `(numer x)`, which gives the numerator of a rational number `x`
  - `(denom x)`, which gives the denominator of a rational number `x`

- Now we can play with rational numbers!
- PC: What would be the problem if we didn't make an explicit `rat`?

Operations with rational numbers

- How do we add two rational numbers x and y?

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

- Multiply them?

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

- Print them nicely:

```
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

Interestingly, we don't yet know how to represent them!

One way to represent rational numbers

```
(define (make-rat n d)
  (lambda (choose)
    (if (= select 0)
        n
        d)))

(define (numer x)
  (x 0))

(define (denom x)
  (x 1))
```

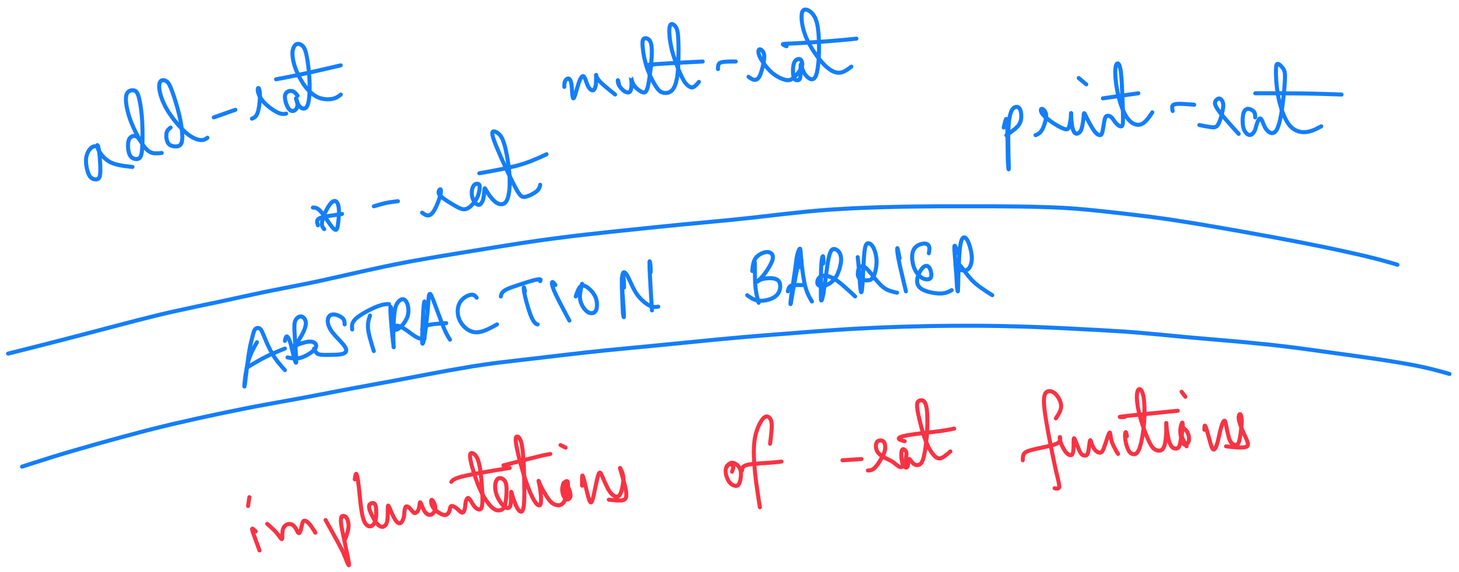
- Let's check if this works:

But pairing data is quite a popular operation!

- Scheme provides the following special forms to work with ***pairs***:
  - `(cons x y)` puts `x` and `y` together and returns a pair
  - `(car x)` returns the first element of a pair `x`
  - `(cdr x)` returns the second element of a pair `x`

Data Abstraction

- Representation is separate from usage; designing/changing the former should not affect the latter.



Updated rational-number implementation

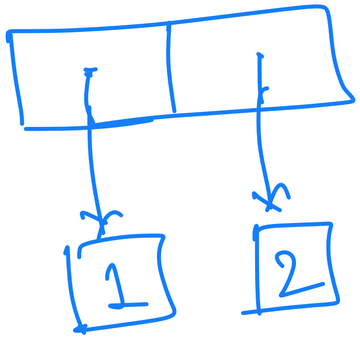
```
(define (make-rat n d)
  (cons n d))

(define (numer x)
  (car x))

(define (denom x)
  (cdr x))
```

- Notice that the usage need not change:

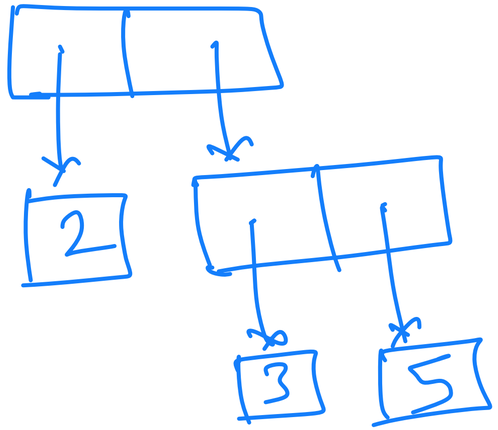
```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (mult-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))

(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

Box-Pointer Notation of Pairs

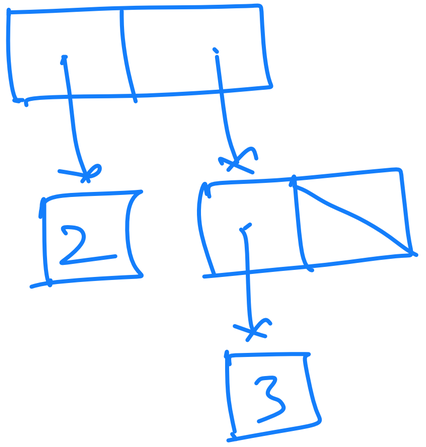
(cons 1 2)



(cons 2
(cons 3 5))



(cons 2
(cons 3 nil)



- Advantage: Implicit that pairs can have pairs as elements!

But chaining pairs to form a list is quite a popular operation!

- Scheme provides the following special forms to work with ***lists***:
  - `(list e1 e2 e3)` is a syntactic sugar for

    ```
    (cons (e1 (cons e2 (cons e3 nil))))
    ```

  - `nil` represents an empty list
  - `(null? l)` checks whether a list `l` is empty and returns a boolean

And we often overdo stuff :D