Functional Programming

Lecture 6: Imperative scheme and parallelism

Viliam Lisý
Rostislav Horčík
Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague
viliam.lisy@fel.cvut.cz
xhorcik@fel.cvut.cz

Last lecture

- We do not need to modify the state if we compute a function
- States break nice properties of pure FP
- Make the pure part of programs as large as possible
- States can sometimes be useful
  - random access in O(1)
  - memoization

"Classes and objects"

```
(define (make-account balance)
  (define (withdraw x)
    (if (>= balance x)
        (begin (set! balance (- balance x))
               balance)
        (error "Not enough money!!!")))
  (define (deposit x)
    (set! balance (+ balance x))
    balance)
  (define (dispatch name)
    (cond ((eq? name 'withdraw) withdraw)
          ((eq? name 'deposit) deposit)
          (else (error "Unknown request"))))
  dispatch)
```

Lists modifications

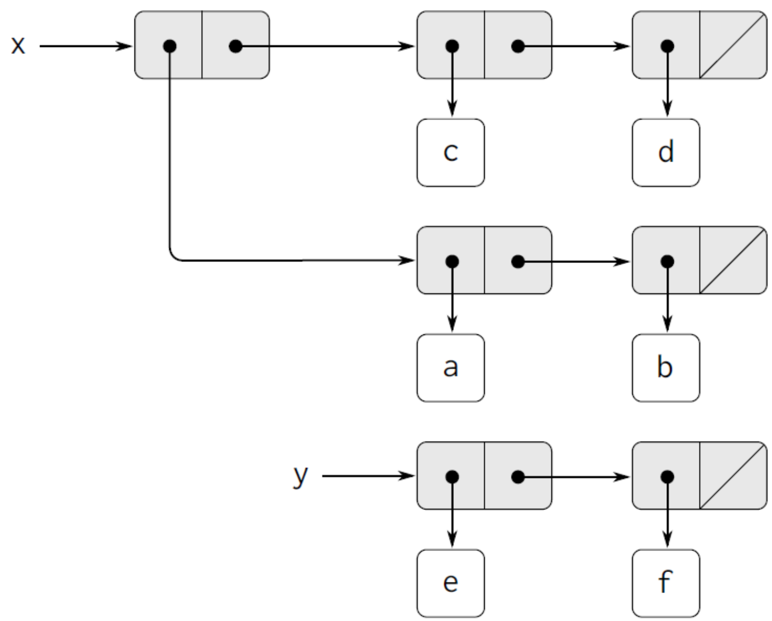
In R5RS, we can modify lists using

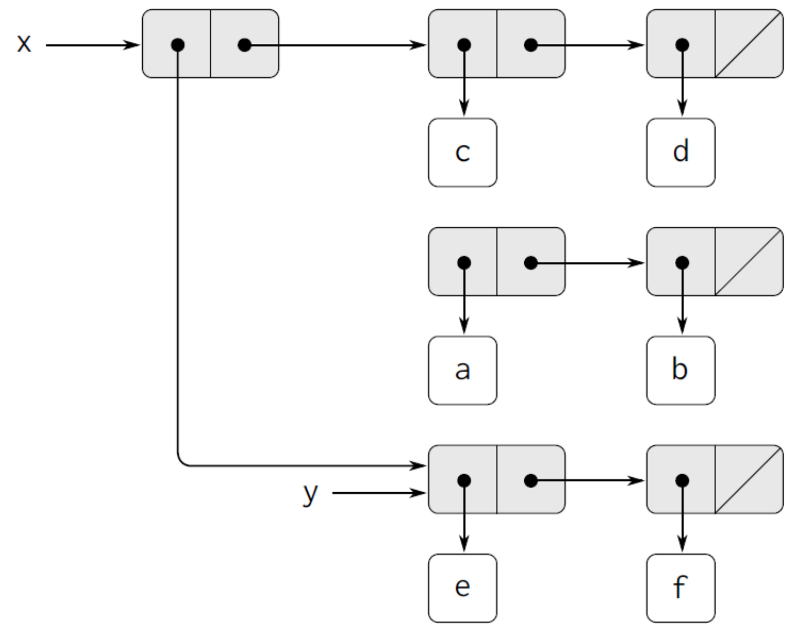
`set-car!, set-cdr!`

List are immutable by default with `#lang scheme`

Need to use `mcons, mcar, set-mcar!,`…

(set-car! x y)

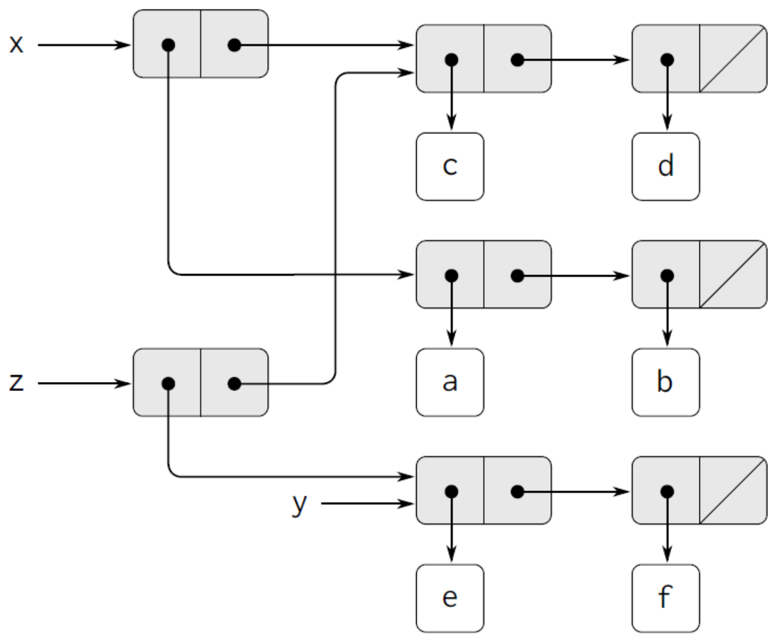


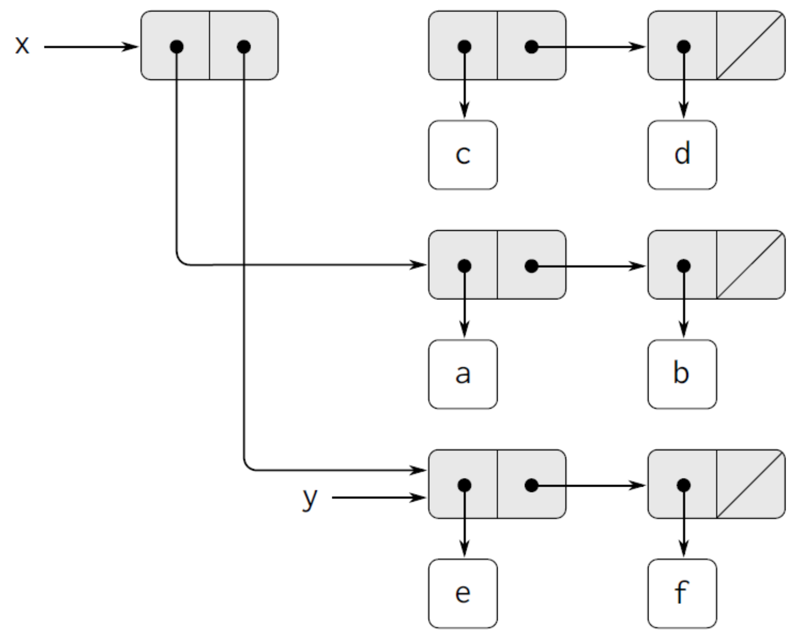


x = ((a b) c d), y = (e f)

x = ((e f) c d), y = (e f)

(set-cdr! x y)

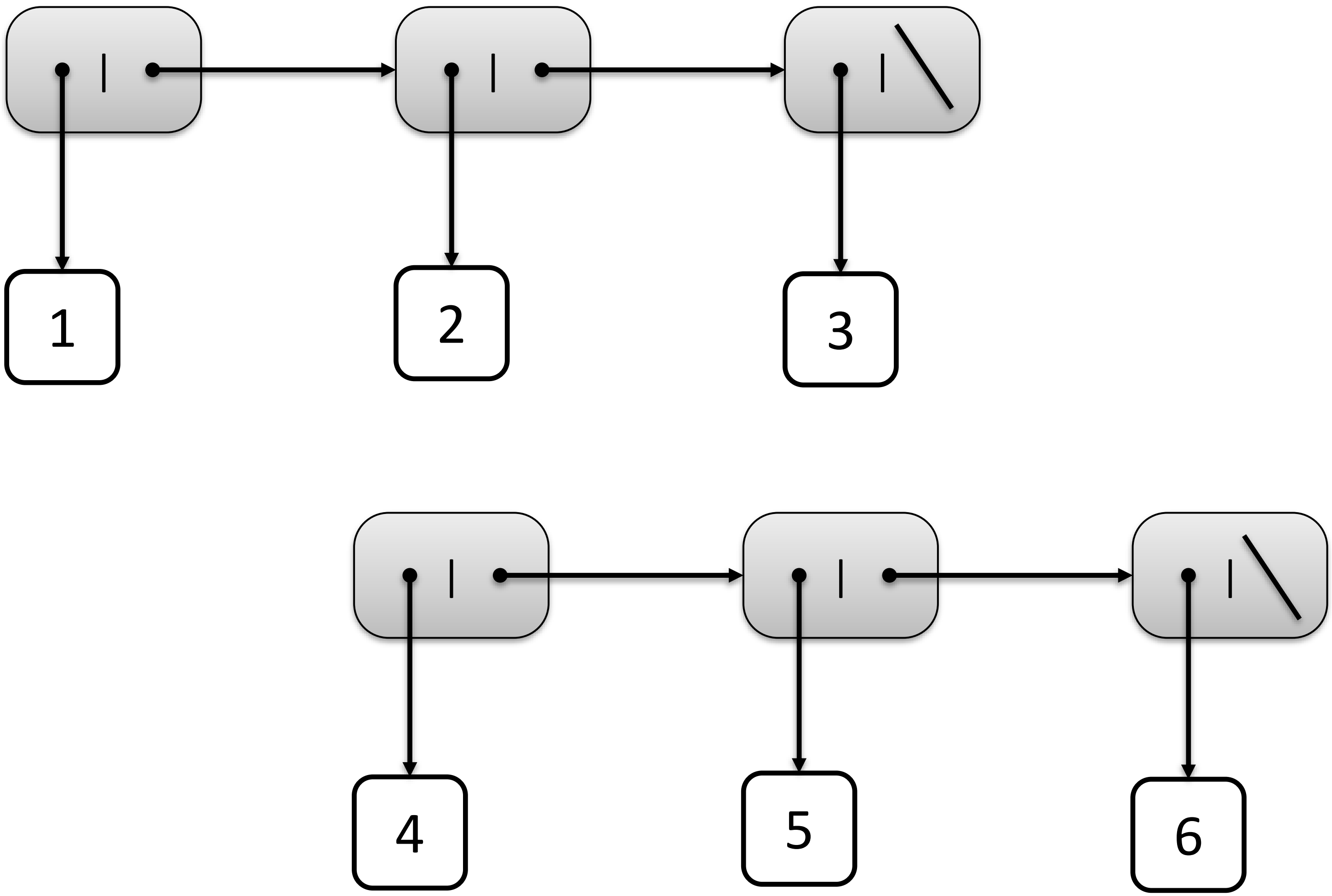




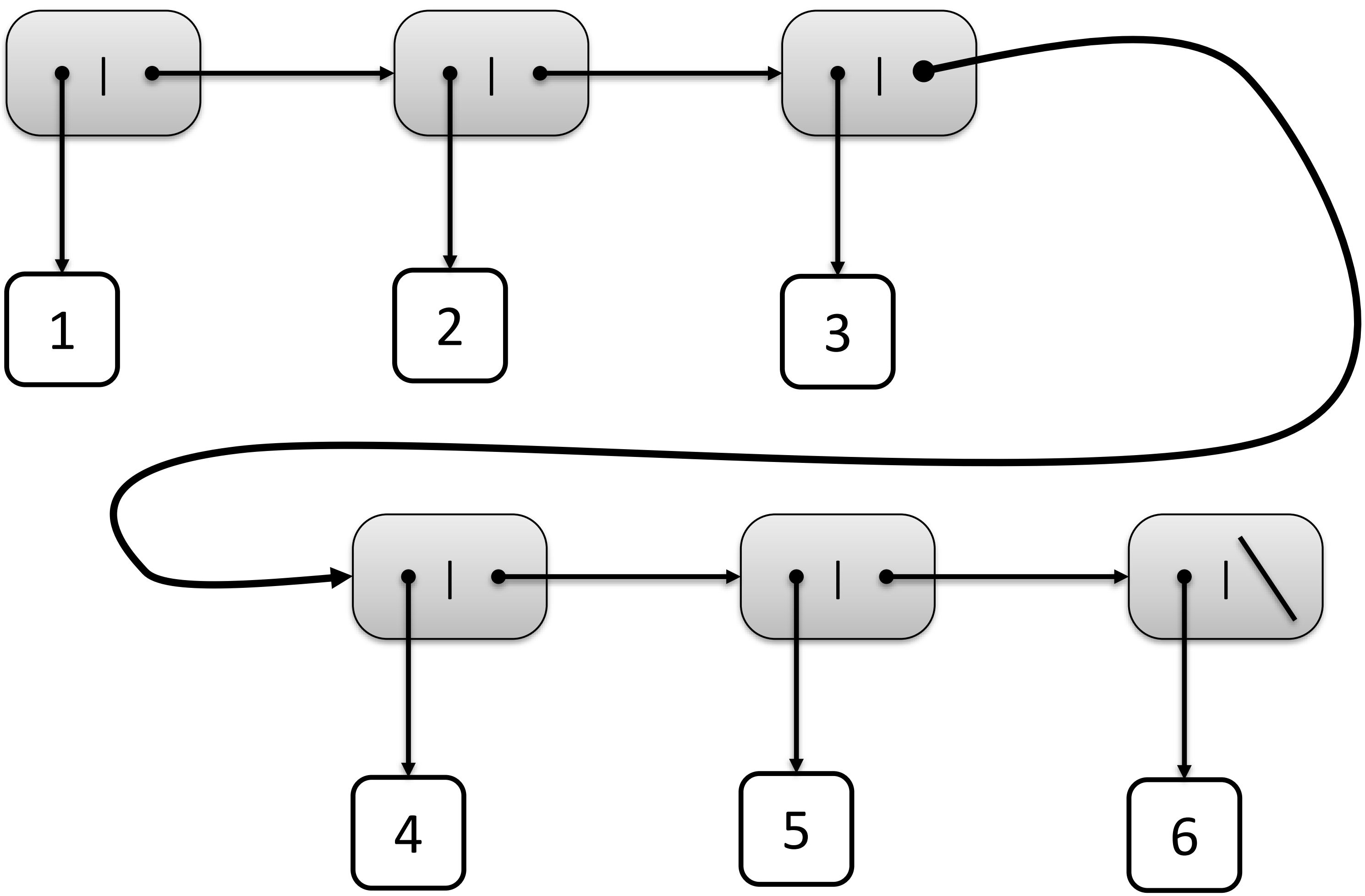
x = ((a b) c d), y = (e f)

x = ((a b) e f), y = (e f)

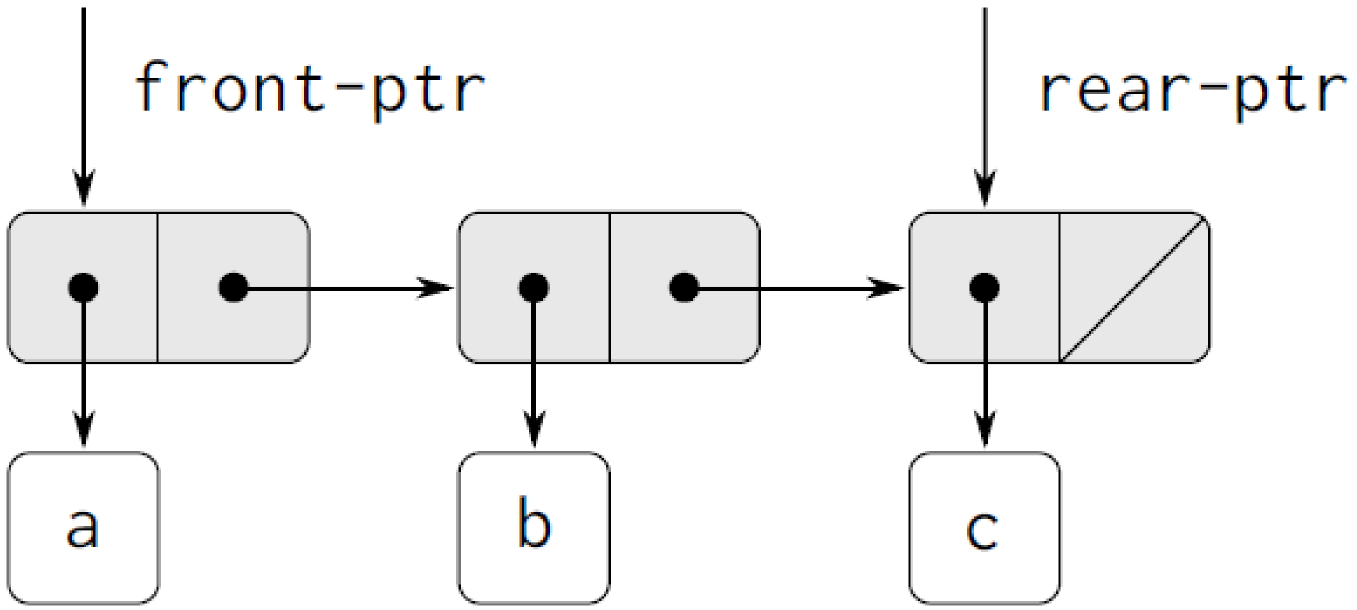
Append!



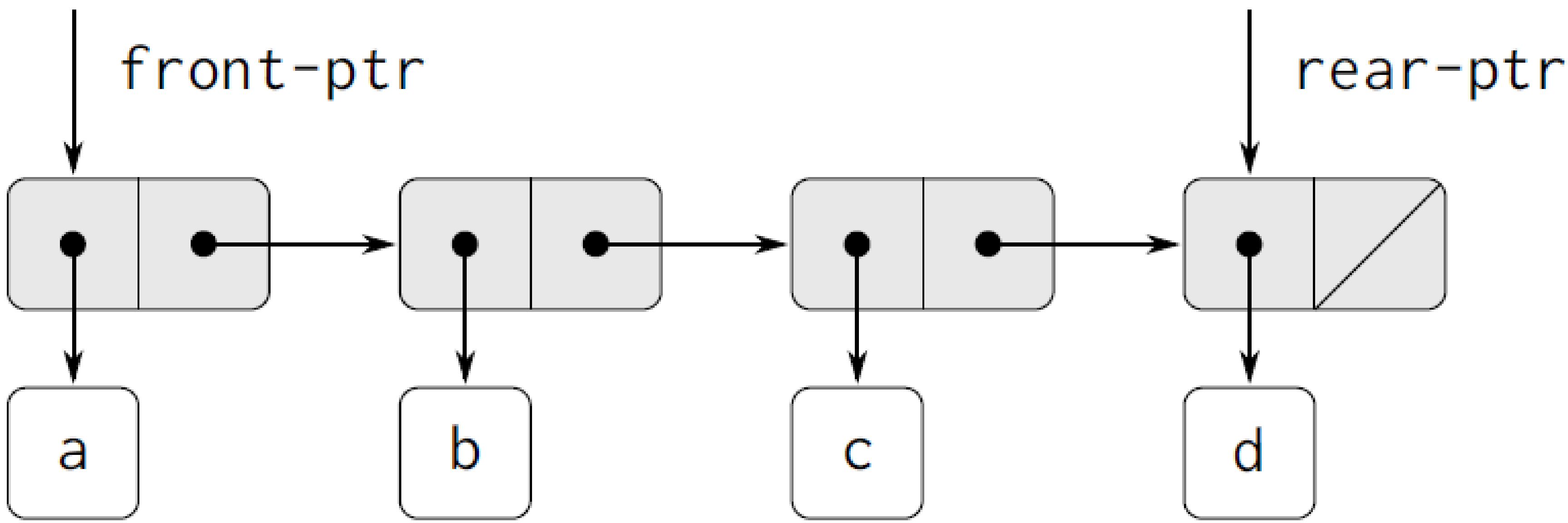
Append!



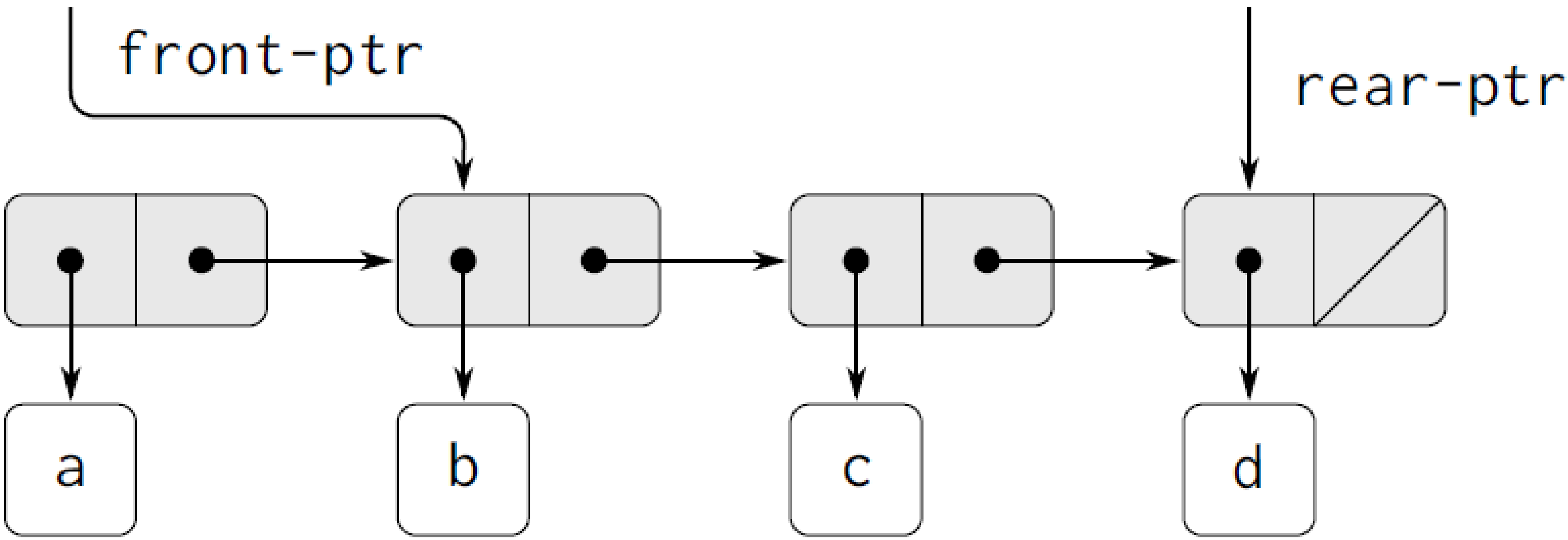
Queue



Insert



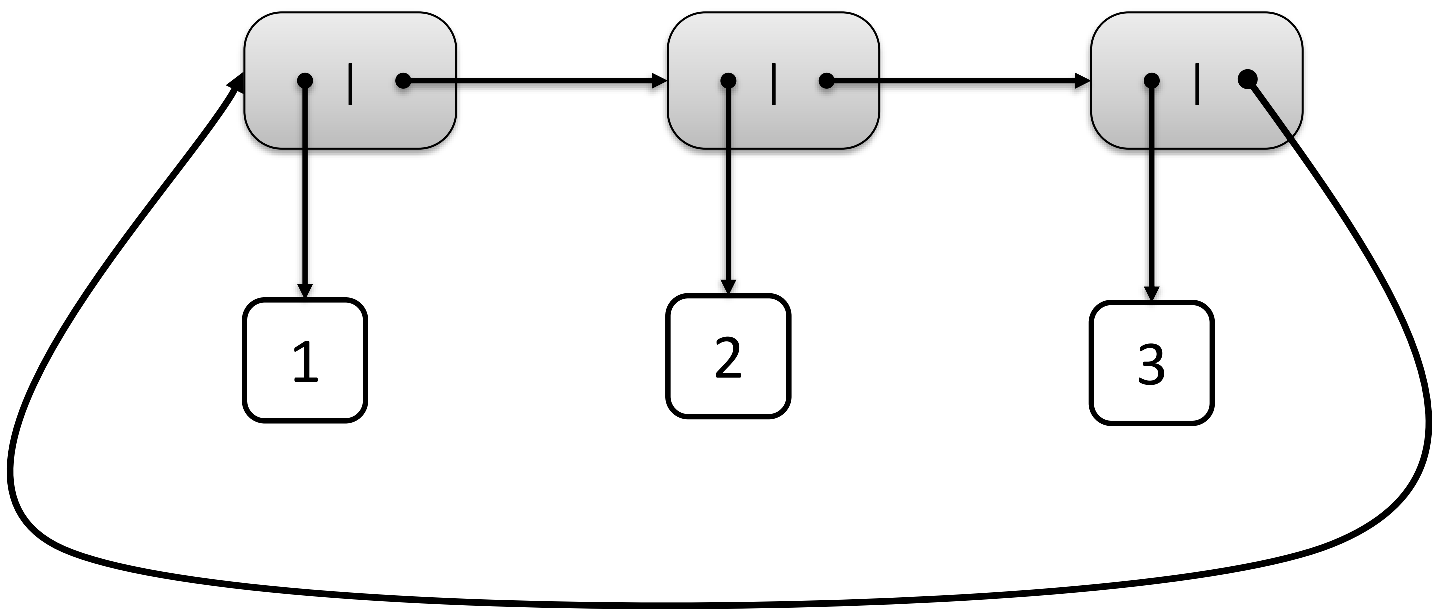
Delete



Queue

```
(define (make-q)
  (let ((front '()) (rear '()))
    (define (in x)
      (let ((new (list x)))
        (if (null? front)
            (begin (set! front new) (set! rear new))
            (begin (set-cdr! rear new) (set! rear new)))))
    (define (out)
      (let ((x (car front)))
        (set! front (cdr front))
        x))
    (define (dispatch name)
      (cond
        ((eq? name 'in) in)
        ((eq? name 'out) out)))
    dispatch))
```

Circular “lists”



Circular “lists”

```
(define (make-cyclic-list! ls)
  (define (cyc! xs)
    (if (null? (cdr xs))
        (begin (set-cdr! xs ls) ls)
        (cyc! (cdr xs))
        ))
  (cyc! ls))

(define week (make-cyclic-list!
        '(mon tue wed thu fri sat sun)))
```

Hash tables in Racket

There are many variants of hash tables

Create a hash table comparing with `equal?`

```
(make-hash)
```

Associate `v` with `key` in `hash`

```
(hash-set! hash key v)
(hash-ref hash key [failure-result])
(hash-ref! hash key to-set)
```

```
hash-remove!, hash-update!
```

Memoization

```
(define (memoize f)
  (let ((table (make-hash)))
    (lambda args
      (hash-ref! table
                 args
                 (lambda ()
                   (display "X")
                   (apply f args))))))
```

Concurrency and Parallelism in Racket

- Thread (concurrency)
  - preempt each other without cooperation
  - share state: variables, function definitions, etc.
  - in Racket, they run on one OS thread
- Futures (parallelism)
  - evaluate an expression in parallel to the main program
  - block on operations that may not run safely in parallel
- Places (parallelism)
  - separate instances of scheme
  - communicate using message passing

Threads

Run on single OS thread

- No speed-up
- Waiting for slow/external event: I/O, sockets, etc.

Operations on threads

- `(thread thunk)` returns thread descriptor
- `thread-suspend, thread-resume, kill-thread`

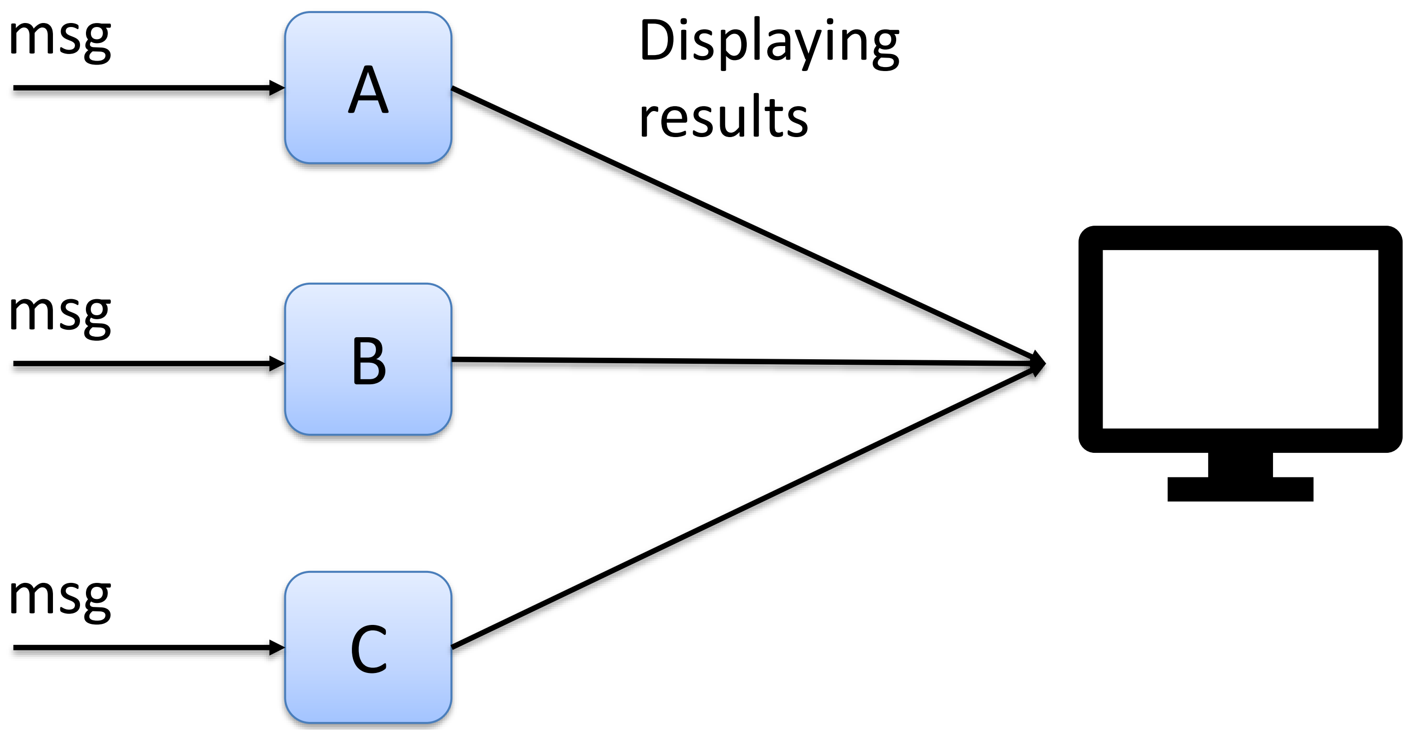
Thread mailboxes

- `(thread-send th msg), (thread-receive)`

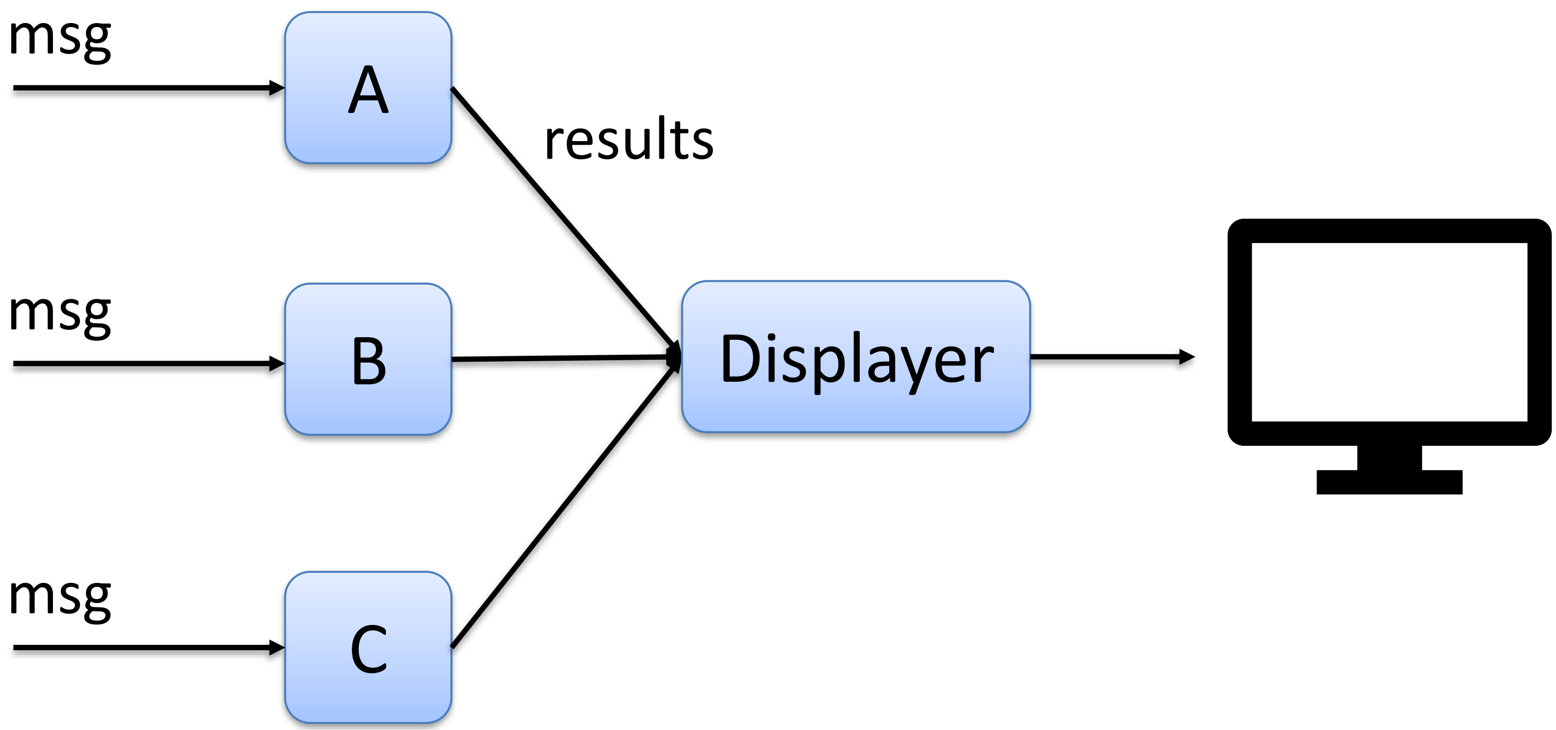
Channels

- `(make-channel), (channel-put ch v)`
- `(channel-get ch), (channel-try-get ch)`

Threads example



Threads example



Futures

`(require racket/future)`

`(future thunk)`

Starts evaluating an expression (given as thunk)

Blocks when an operation may not be safely executed

Returns a "future"

`(touch future)`

Finish evaluating the expression in the main thread

If the expression is already evaluated, return the result

As in *promise*, additional touches just return the result

Future map

Executes a given function on each element of a list in parallel and returns the results

```
(define (future-map f list)
  (let ((res
         (map (lambda (x)
                (future (lambda () (f x))))
              list)))
    (map touch res)))
```

Futures can be visualized and analyzed using

```
(require future-visualizer)
(visualize-futures expr)
```

Home assignment 3

Genetic programming

- Evolution inspired local search in structured data
- Survival of the fittest!!!

- Individual: program for the robot in the maze
- Population: collection of the programs
- New generation: selection, mutation, cross-over
- Fitness function: see Home assignment 2

Summary

- We do not need to modify the state
- It breaks nice properties of FP
- It can sometimes be useful
  - random access in O(1)
  - objects
  - circular data structures
  - memoization
- Concurrency and parallelism