Racket

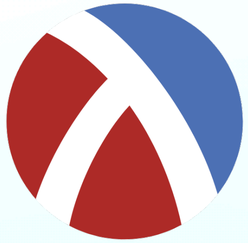
A.Y. 2023-24
Principles of Programming Languages

Andrea Franchini — andrea.franchini@polimi.it

Let’s code!

You should have already **downloaded** Racket (~200 MB)!

It contains DrRacket, the IDE we’re using for the exercises! It has everything we need.



https://download.racket-lang.org/

Let’s code!

The exercises will be available on my GitHub repository.

👇

https://github.com/andreafra/ppl

Any doubt?

You can write me an email at:

andrea.franchini@polimi.it

Racket in 5 minutes

Types

NUMBERS

1 (integer)

3.14 (floating)

1+2i 6.02e+23

#x29 #o32

#b010101

STRINGS

“Hello world!”

CHARACTERS

#\a #\B #\5

BOOLEANS

#t #f

SYMBOLS

apple a-10!

really? hey/you

VECTORS

#(1 2 3 4)

LISTS

'(1 2 #\3 “4”) '()

Racket in 5 minutes

Top facts about Racket

- Prefix notation
- Dynamically-typed
- (Mostly) functional programming
- *Homoiconic* -> Code is Data
- Call-by-Value

Racket in 5 minutes

Dynamically-typed

In **statically-typed** languages such as C and Java the type of a variable is checked at **compile time**, and we usually **specify** it when declaring something.

```
int sumFive(int n) {
    return n + 5
}
```

In **Racket**, just like in Python and JavaScript, the type is **unknown** before runtime. You can **reassign** values of any kind to a variable.

```
; error if n is not a number
(define (sum-five n)
  (+ n 5))
```

We can use **(number? n)** to check if 'n' is a number.

Racket in 5 minutes

Prefix notation

```
3 + 2 - 5 = 0
```

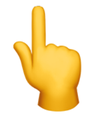
Infix

```
(= (- (+ 3 2) 5) 0)
```

or

```
(= (+ 3 (- 2 5) 0))
```

Prefix



Racket in 5 minutes

Functional programming

Procedures are **first-class** citizens. We can assign them to an identifier just like any other value.

'lambda's (also 'λ') are **unnamed** procedures:

```
(λ (x) (+ x 5))
```

Btw, we can assign a value to an identifier with:

```
(define two 2)
```

We can make a named procedure 'sum-five':

```
(define sum-five
  (λ (x) (+ x 5)))
```

Racket in 5 minutes
Functional programming

```
(define sum
  (λ (x y) (+ x y)))
```

Looks ugly? Here's some syntactic sugar:

Also note that there is no '**return**' keyword: the result of the last expression is returned.

```
(define (sum x y) (+ x y))
```

A list of (**id** *arg1 arg2* …)

body (one or more expressions)

Racket in 5 minutes

Homoiconic: Code is Data

- In (f x y) we're trying to call a function 'f' with 'x' and 'y' as arguments. *What if we want them as a list?*
- We use 'quote' (aka ' – single quote)

```
'(f x y) is now a list

'(+ 1 2) ; => (+ 1 2)
 (+ 1 2) ; => 3
```

- 'quote' prevents evaluation. Remember that S-expressions are lists, where the first elements is the id of a procedure.

Racket in 5 minutes

Homoiconic: Code is Data

- If we want to partially evaluate an expression, like:

```
(1 (+ 2 3) 4) ; we want '(1 5 4)

'(1 (+ 2 3) 4) ; => '(1 (+ 2 3) 4)
```

- 'quote' does not work: it makes the expression a constant.

Racket in 5 minutes

Homoiconic: Code is Data

- We can use ‘quasiquote’ (aka ` – backtick) and ‘unquote’ (aka , – comma):

```
`(1 ,(+ 2 3) 4) ; => ‘(1 5 4)
```

- ‘unquote’ must be used in a ‘quasiquote’
- To evaluate a list such as ‘(+ 1 2 3) we can use ‘eval’:

```
(eval `(+ 1 2 3))
```

Racket in 5 minutes

Call-by-Value

- Objects are allocated on the **heap!**
- **References** are passed by value.
- It behaves just like in Java.
- ‘x’ in ‘set-local’ is not the same as the ‘x’ outside. It is just a **copy.**

```
(define (set-local x)
  (set! x 3)
  (displayln x)) ; => 3

(define x 1)
(set! x 2)
(displayln x) ; => 2
(set-local x)
(displayln x) ; => 2
```

Racket in 5 minutes
Call-by-Value

- In this case, ‘v’ is just the reference to a vector #(1 2 3).
- Therefore, if we modify it in ‘set-vector’ we’re actually affecting the real (and only) one.

```
(define (set-vector v)
  (vector-set! v 1 "Hi")
  (displayln v)) ; => #(1 "Hi" 3)

(define v (vector 1 2 3))
(set-vector v)
(displayln v) ; => #(1 "Hi" 3)
```

** Vectors are fixed-length arrays.*

Racket in 5 minutes
Declaring local variables with let

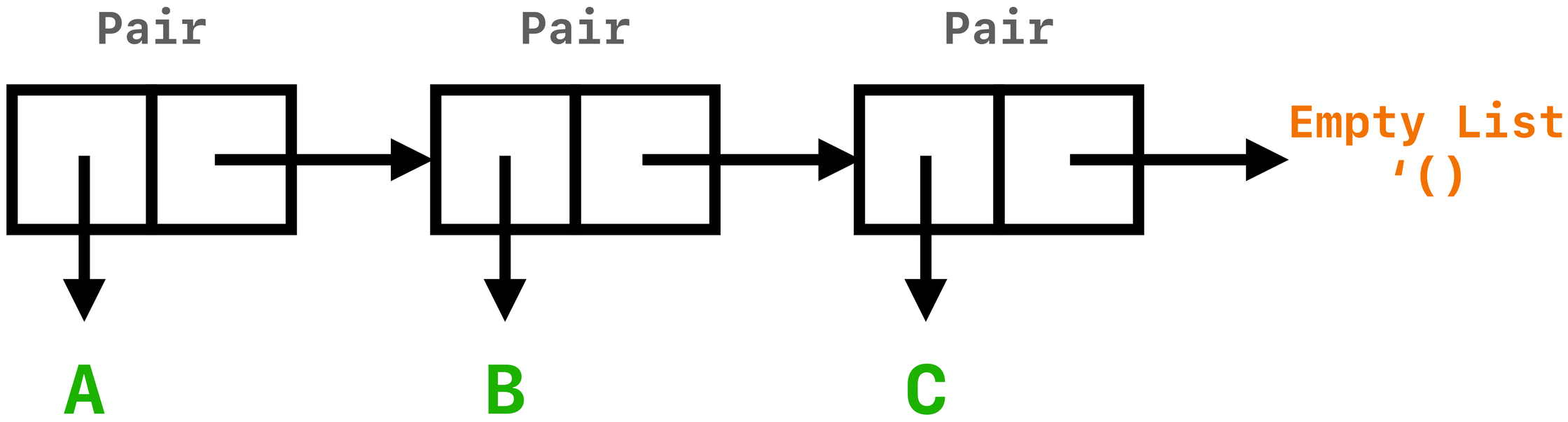
- ‘**let**’ ‘**let***’ allow us to **bind** variables **locally**, (‘letrec’ ‘letrec*’, if you need mutual recursion).

```
; binds in parallel
(let ((x 2)
      (y 3)) ← You can’t use ‘x’ in
  . . .)       the ‘y’ expression

; binds in sequence
(let* ((x 2)
       (y (add1 x)))
  . . .)
```

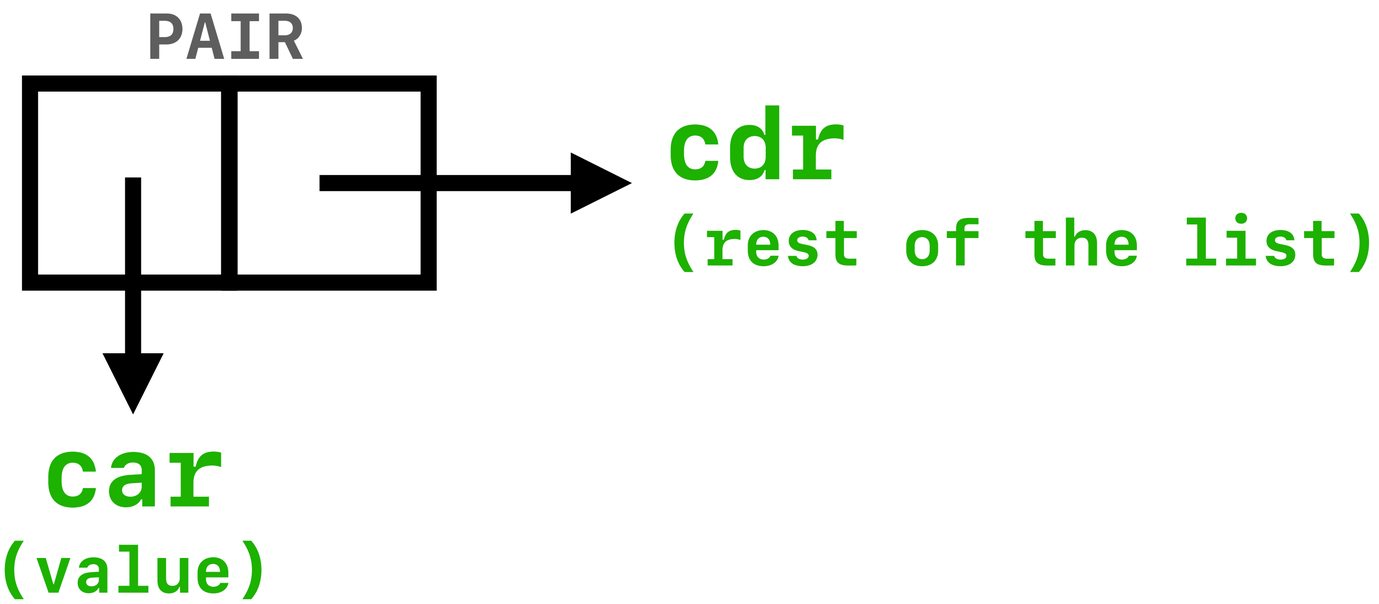
Racket in 5 minutes
Lists & Pairs

- Lists in Racket are a linked list, made up of pairs.



Racket in 5 minutes
Lists & Pairs

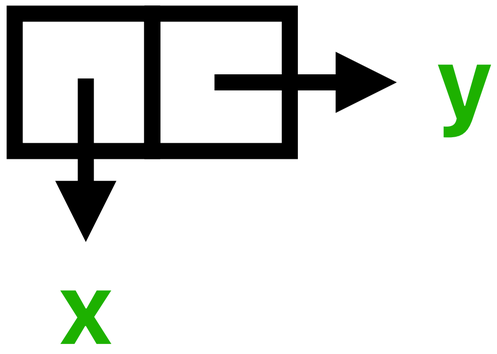
- Lists in Racket works a little different from what you are used to:



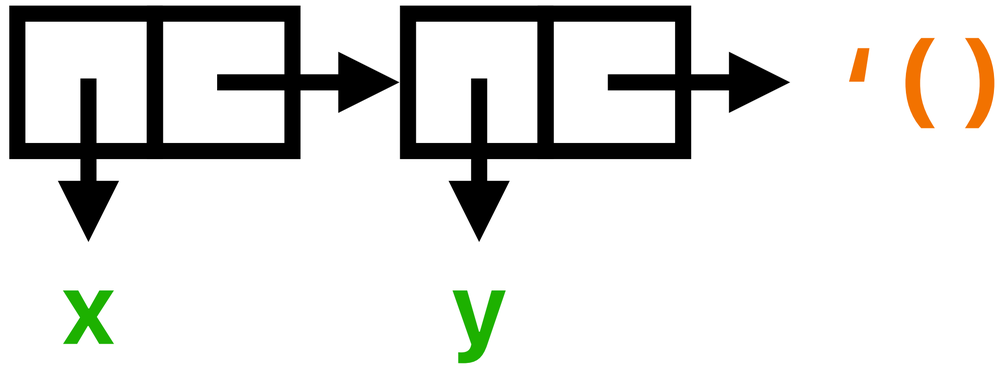
Racket in 5 minutes
Lists and Pairs

- A **pair** is written as (x . y) and we use '**(cons x y)**' to create it.
- Pairs are **immutable**.
- (1 2 3) is stored as (1 . (2 . (3 . ()))) where **()** is the empty list.

```
(cons x y) ; => '(x . y)
```




```
(cons x '(y)) ; => '(x y)
= (cons x (cons y '()))
```



Racket in 5 minutes
Lists & Pairs

- We can access the first element of a list with ‘car’ and the rest of it with ‘cdr’.

```
(car ‘(1 2 3)) ; => 1

(cdr ‘(1 2 3)) ; => ‘(2 3)
```

- If we have a procedure that accepts a variable number of arguments, we can use:

```
(define (f x . xs) ; e.g.: (f 1 2 3)
  (displayln x) ; => 1
  (displayln xs)) ; => (2 3)
```

Racket in 5 minutes
Control Flow

- ‘**if**’ is a syntactic form (not a *procedure* nor a *value*):

  (**if** <condition> <then-body> <else-body>)

- If we want just the ‘then’ branch we have:

  (**when** <condition> <then-body>)

- Of course, there’s just the ‘else’ branch, too:

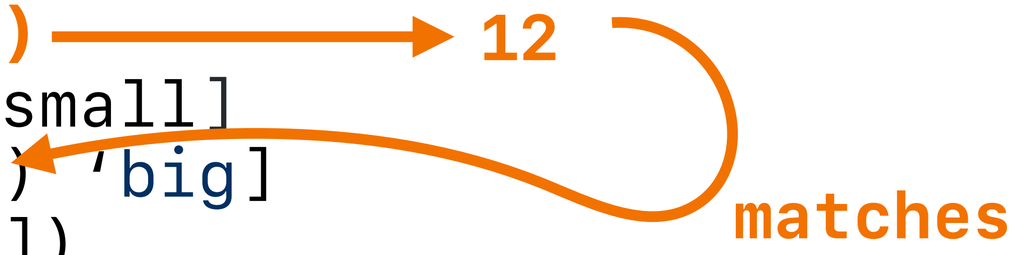
  (**unless** <condition> <else-body>)

Racket in 5 minutes

Control Flow

- ‘**case**’ evaluates ***value*** and finds the first *datum* (*e.g.* *‘1’*) for which *(equal? value ‘datum)* is true.

```
(case (+ 7 5)
  [(1 2 3) 'small]
  [(10 11 12) 'big]
  [else 'idk])
```



Racket in 5 minutes
Control Flow

- ‘**cond**’ evaluates each test expression until one is true, then evaluates and returns its body.

```
(cond
  [(positive? -5) (error "doesn't get here")]
  [(zero? -5) (error "doesn't get here, either")]
  [else ‘here])
```

Racket in 5 minutes

Loops

- Loops are achieved through **recursion**. Either:
  - Recursively **call** a procedure
  - Use a **named** let
  - On lists (and vectors), you can use ‘**for-each**’
- Use **tail-recursion** whenever possible! The last called thing in a recursive procedure should be the procedure itself!