Lenguajes de Programación

- Functions as values & higher-order functions
- Methodology to design functions

Federico Olmedo
Ismael Figueroa

- Functions as values & higher-order functions

Functions as values

numbers

```
> 1
1
> -3
-3
> 4.02
4.02
> 6.02e+23
6.02e+23
> 4/3
1 1/3
```

booleans

```
> #t
#t
> #f
#f
```

strings

```
> "hola"
"hola"
> "esto es un string"
"esto es un string"
> "esto también lo es"
"esto también lo es"
> "soy un string con Unicode λx:(μα.α→α).xx"
"soy un string con Unicode λx:(μα.α→α).xx"
```

symbols

```
> 'hola
'hola
> 'esto\ es\ un\ simbolo\ con\ espacios
'|esto es un simbolo con espacios|
> (string->symbol "esto también es un simbolo con espacios")
'|esto también es un simbolo con espacios|
```

procedures

```
> +
#<procedure:+>
> -
#<procedure:->
> list
#<procedure:list>
> cons
#<procedure:cons>
> car
#<procedure:car>
>
```

Function as values

procedures

```
> +
#<procedure:+>
> -
#<procedure:->
> list
#<procedure:list>
> cons
#<procedure:cons>
> car
#<procedure:car>
>
```

Can be defined anonymously.

Can be stored in any data structure.

Can be returned as the result of another function.

Can be passed as parameters to other functions.

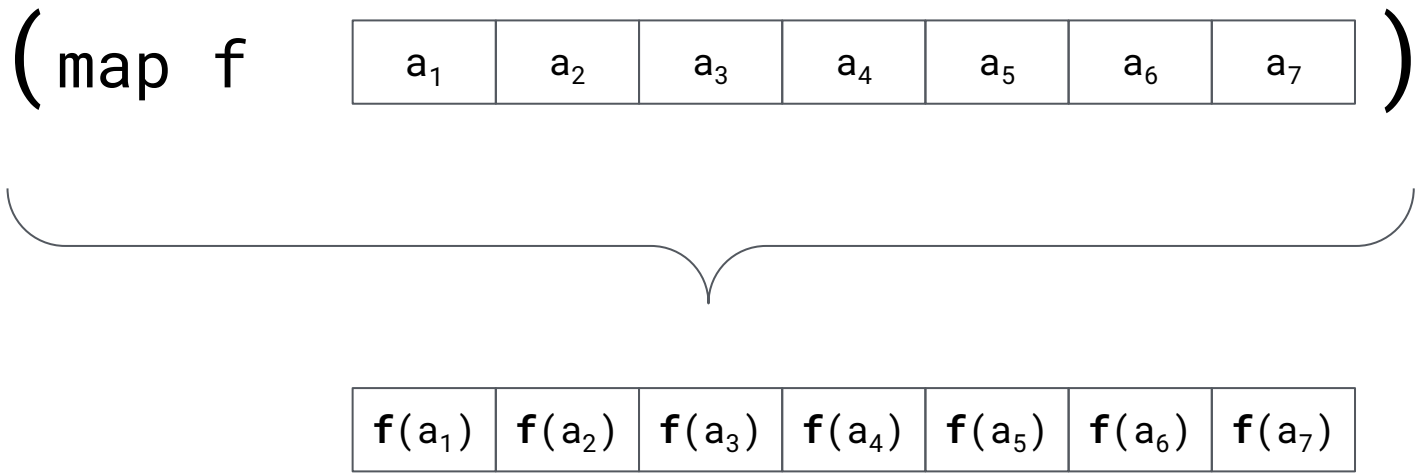
Higher-order functions

As functions are regular values, just like numbers, strings, etc; we can use them whenever we can use values of such types. This opens the door to *advanced programming patterns* in a straightforward manner.

Higher-order functions in list processing: map

Given that lists are one of the core data structures in Racket, and functional programming in general, there are standard manipulation patterns expressed using higher-order functions.

MAP FUNCTION: constructs a list which results from the element-wise application of a function to an initial list of values.



Higher-order functions in list processing: map

Given that lists are one of the core data structures in Racket, and functional programming in general, there are standard manipulation patterns expressed using higher-order functions.

MAP FUNCTION: constructs a list which results from the element-wise application of a function to an initial list of values.

```
> (map abs '(1 -1 0 -32 14))
'(1 1 0 32 14)
```

```
> (define (add1 n) (+ n 1))
> (map add1 '(1 2 3 4 5))
'(2 3 4 5 6)
```

```
> (define (add124 n) (+ n 124))
> (map add124 '(1 2 3 4 5))
'(125 126 127 128 129)
```

Anonymous functions

Sometimes we don't need to assign names at all.

- As parameters to higher-order functions, which are too specific and will likely never be reused in other contexts.

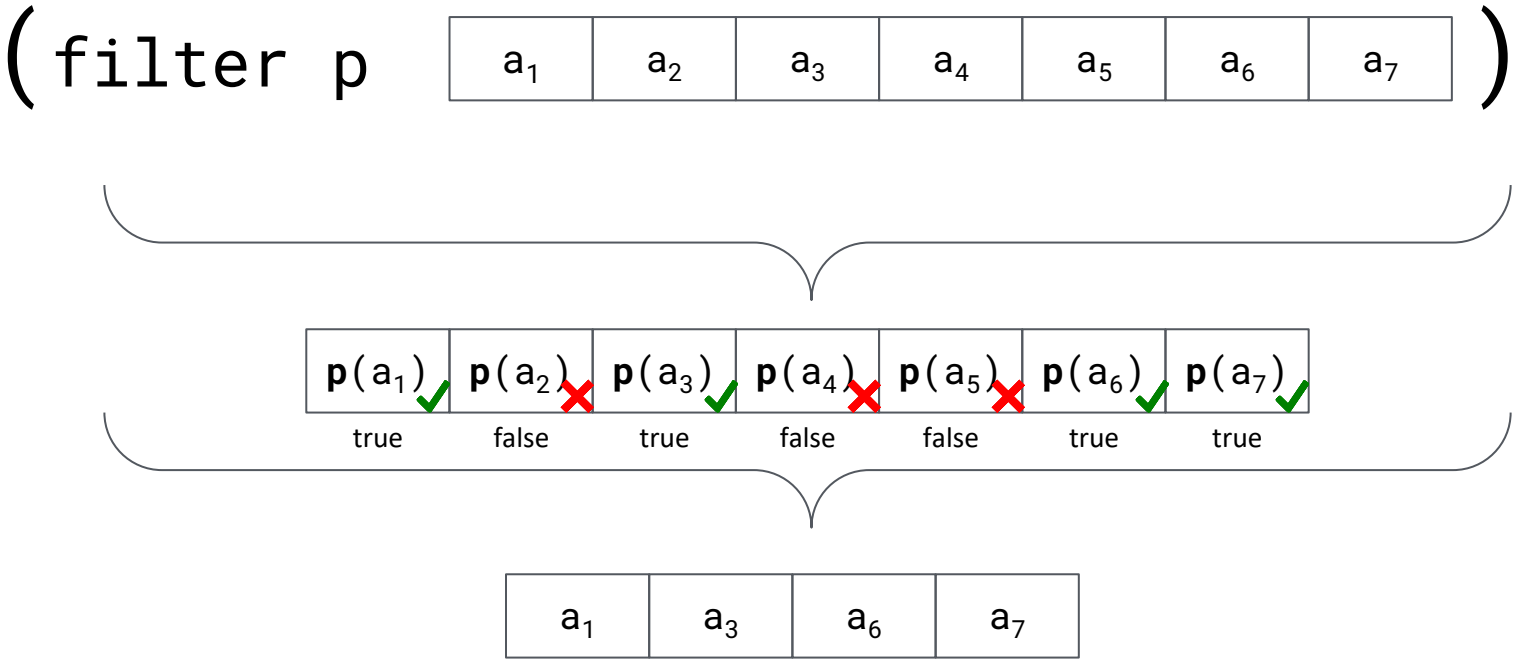
SYNTAX: (λ (arg_1 … arg_n) *func-body*)

```
> (lambda (x) (+ x 1))
#<procedure>

> ((lambda (x) (+ x 1)) 10)
11

> (map (lambda (n) (+ n 1)) '(1 2 3 4 5))
'(2 3 4 5 6)

> (map (lambda (n) (+ n 124)) '(1 2 3 4 5))
'(125 126 127 128 129)
```

Unlike other languages, in Racket anonymous and named functions have the same expressiveness as named functions.

Higher-order functions in list processing: filter

FILTER FUNCTION: constructs a list that only contains the elements of an initial list that satisfy a boolean ***predicate function***. The relative order of elements is maintained.



Higher-order functions in list processing: filter

FILTER FUNCTION: constructs a list that only contains the elements of an initial list that satisfy a boolean ***predicate function***. The relative order of elements is maintained.

```
> (filter even? '(1 2 3 4 5 6 7 8))
'(2 4 6 8)

> (filter odd? '(1 2 3 4 5 6 7 8))
'(1 3 5 7)

> (filter (lambda (n) (> n 4)) '(1 2 3 4 5 6 7 8))
'(5 6 7 8)


> (filter string? '(1 2 3 4 5 6 7 8))
'()
```

A predicate function is any function that returns a boolean, and is usually called inside the condition section of a conditional expression. In Racket's naming convention is to end predicate function names with a **?** sign: `even?`, `odd?`, `string?`, etc.

Higher-order functions that return functions

Any function can return another function as a value. This pattern is usually known as a *factory* in object-oriented settings.*

```
(define (addn n)
  (λ (x)
    (+ x n)))
```

```
> ((addn 1) 2)
3
```

```
> (map (addn 124) '(1 2 3 4 5))
'(125 126 127 128 129)
```

```
(define (less-than n)
  (λ (m)
    (< m n)))
```

```
> (filter (less-than 10) '(1 2 3 4 5 6 7 8))
'(1 2 3 4 5 6 7 8)
```

```
(define (more-than n)
  (λ (m)
    (> m n)))
```

```
> (filter (more-than 10) '(1 2 3 4 5 6 7 8))
'()
```

Another pattern to return functions is to create compositions or modifications over existing functions, using logical operators.

*We exploit this pattern of function construction when studying object-oriented programming (OOPLAI)

Currying

In Python the partial function is a kind of currying for partial application.


```
(define (curry f)
  (λ (a)
    (λ (b)
      (f a b))))
```

Currying is the technique that *transforms a function* that takes simultaneously two (or more) arguments into a function that takes them one by one.

BENEFIT: allows for partial application, improving reusability and composition of functions.

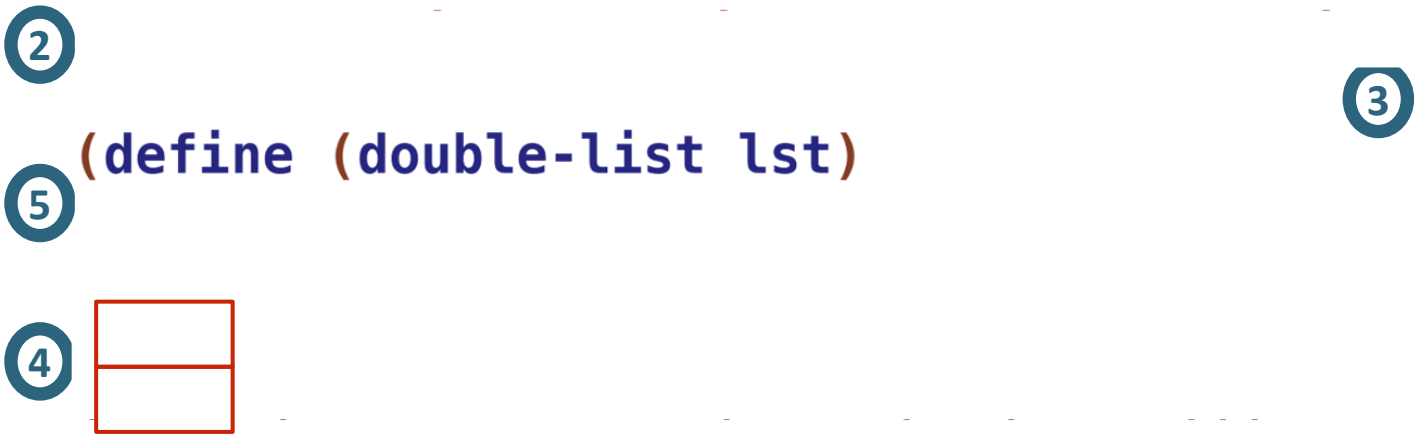
```
> (map ((curry +) 1) '(1 2 3))
'(2 3 4)
```
addn

```
> (filter ((curry <) 1) '(1 2 3))
'(2 3)
```
less-than

```
((curry <) 1)
   [def. curry]
((λ (a) (λ (b) (< a b))) 1)
   [funct. application]
(λ (b) (< 1 b))
```

→ How to design programs

Methodology for defining functions: double-list example



Provided by **`#lang play`**

- Implementation should be the last step!!!
- Tests should cover all “significant” cases
- GIGO: garbage-in, garbage-out

Methodology for defining functions: mysqrt example

```
#lang play
```

```
(define (mysqrt x)
```

Exercises

Function **negate**

Define function `(negate p)`, which takes a single-argument predicate, and returns its negation.*

Function **reject**

Define function `(reject p l)`, which takes a list and a predicate and returns a list where all elements that satisfy the predicate are removed.

Function **compose**

Define function `(compose f g)`, which takes two functions `f` and `g`, and returns a function that performs the composition `f ∘ g` between them.

Function **apply-f-n**

Define function `(apply-f-n f n)`, which returns a function that composes `n` times the function `f`. Assume `f` takes only a single argument.

*Use the **not** logical operator.

Apply the function design methodology to all these exercises

Takeaways

Javascript also provides functions as values. Python has weaker support for this, with a bias towards named functions.

- Functions *are* values.

  And therefore they can:

  - Be defined anonymously.
  - Be the argument of other functions.
  - Be the return value of other functions.
  - Be stored in data structures.

  This opens the door to powerful programming patterns, commonly known as higher-order functions.

- We will use a methodology for defining functions, with an emphasis on specification and tests before implementation.

- The Play language provides simple constructs to help us specify test cases.

Lecture material

Bibliography

- PrePLAI: Introduction to functional programming in Racket [Sections 3, 4.1, 5.1 and 5.2]

For a more detailed reference, see the online Racket documentation:

- Racket Guide: tutorial
- Racket Reference: reference manual