CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Lenguajes de Programación

2020 — 2º Semestre

Clase 3:

- Functions as values
- A good programming practice
- Pattern matching

Federico Olmedo

Anonymous and higher-order functions

Anonymous functions

SYNTAX: `(λ (arg₁ … argₙ) func-body)`

EXAMPLE: `(λ (n) (+ n 3))`

Higher-order functions

- That takes as argument a function:

```
(map (λ (n) (+ n 3)) '(1 2 3))    ⇝   '(4 5 6)
(filter even? '(1 2 3))           ⇝    '(2)
```

- That returns a function as output:

```
(define (add-n n) (λ (x) (+ x n)))
```

Anonymous and higher-order functions

Anonymous functions

SYNTAX: `(λ (arg₁ … argₙ) func-body)`

EXAMPLE: `(λ (n) (+ n 3))`

Higher-order functions

- That takes as argument a function:

```
(map (λ (n) (+ n 3)) '(1 2 3))    ⇝    '(4 5 6)
(filter even? '(1 2 3))           ⇝     '(2)
```

- That returns a function as output:

```
(define (add-n n) (λ (x) (+ x n)))
```

1. Define function `(negate p)` that takes a predicate `p` and returns its negation.
2. Define function `(reject p l)` that takes a list `l` and a predicate `p` and removes from the list the elements satisfying `p`.
3. Define a function `(apply-twice f)` that takes a function `f:A → A` and returns the functions that applies `f` twice to its argument.

Currying

$$f : A\ B \to C \quad \xrightarrow{\text{CURRYING}} \quad f^{\sharp} : A \to (B \to C)$$

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

Currying

$$f : A \ B \to C \xrightarrow{\text{CURRYING}} f^{\sharp} : A \to (B \to C)$$

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

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

Currying

$$f : A\ B \to C \xrightarrow{\text{CURRYING}} f^{\sharp} : A \to (B \to C)$$

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

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

BENEFIT: allows for partial application.

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

Currying

$$f : A\ B \to C \xrightarrow{\text{CURRYING}} f^{\sharp} : A \to (B \to C)$$

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

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

Benefit: allows for partial application.

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

```
> (filter ((curry <) 1) '(1 2 3))
'(2 3)
```

Currying

$$f : A\ B \to C \xrightarrow{\text{CURRYING}} f^{\sharp} : A \to (B \to C)$$

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

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

BENEFIT: allows for partial application.

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

```
> (filter ((curry <) 1) '(1 2 3))
'(2 3)
```

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

5 Methodology for defining functions

Methodology for defining functions

1

Understand what the function does

Methodology for defining functions

```
;; double-list :: (listof Number) -> (listof Number)
```

2

Write the function contract

Methodology for defining functions

```
;; double-list :: (listof Number) -> (listof Number)
;; Doubles the elements of the list
```

3

Write the function purpose

5

Methodology for defining functions

```
;; double-list :: (listof Number) -> (listof Number)
;; Doubles the elements of the list
```

```
(test (double-list ‘(1 2 3)) ‘(2 4 6))
(test (double-list empty) empty)
```

Requires
#lang play

4

Provide tests

5

Methodology for defining functions

```
;; double-list :: (listof Number) -> (listof Number)
;; Doubles the elements of the list

(define (double-list l)
  (map (λ (x) (* 2 x)) l))

(test (double-list ‘(1 2 3)) ‘(2 4 6))
(test (double-list empty) empty)
```

5

Write the function implementation

Methodology for defining functions

```
;; double-list :: (listof Number) -> (listof Number)
;; Doubles the elements of the list

(define (double-list l)
  (map (λ (x) (* 2 x)) l))

(test (double-list '(1 2 3)) '(2 4 6))
(test (double-list empty) empty)
```

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

6 What else provides `#lang play`?

What else provides `#lang play`?

- Support for testing functions on inputs that throw an `(error “bla …”)`
  - See PrePLAI [Section 5.2]

What else provides `#lang play`?

- Support for testing functions on inputs that throw an `(error “bla …”)`
    - See PrePLAI [Section 5.2]

- Support for defining data structures (via `typedef`)
    - Next lecture

What else provides `#lang play`?

- Support for testing functions on inputs that throw an `(error “bla …”)`
  - See PrePLAI [Section 5.2]

- Support for defining data structures (via `typedef`)
  - Next lecture

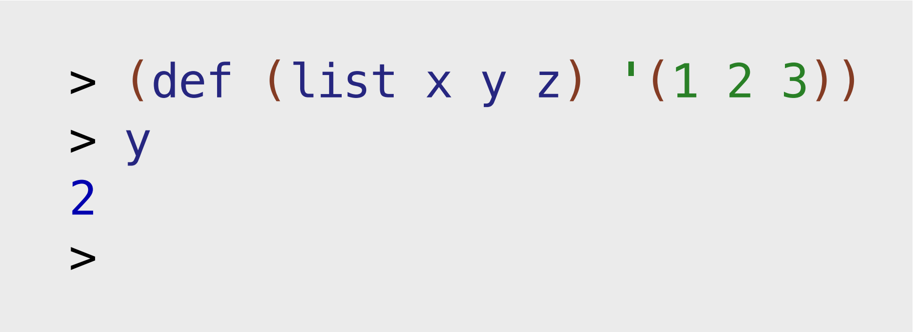
- Pattern matching over expressions via `def`

What else provides `#lang play`?

- Support for testing functions on inputs that throw an `(error “bla …”)`
  - See PrePLAI [Section 5.2]

- Support for defining data structures (via `typedef`)
  - Next lecture

- Pattern matching over expressions via `def`



```
> (def (list x y z) '(1 2 3))
> y
2
>
```

What else provides `#lang play`?

- Support for testing functions on inputs that throw an `(error “bla …”)`
  - See PrePLAI [Section 5.2]

- Support for defining data structures (via `typedef`)
  - Next lecture

- Pattern matching over expressions via `def`

```
> (def (list x y z) '(1 2 3))
> y
2
>
```

```
;; distance :: (cons Int Int) (cons Int Int) -> Float
;; Calcula la distancia entre dos puntos
(define (distance p1 p2)
  (def (cons x1 y1) p1)
  (def (cons x2 y2) p2)
  (sqrt (+ (expt (- x1 x2) 2) (expt (- y1 y2) 2))))
```

Define function `(apply-f-n f n)` that returns the function obtained by composing `f` with itself `n` times

8 Takeaways

Takeaways

Functions are values

And therefore they can:

- Be the argument of other functions
- Be the output (or return value) of other functions

Takeaways

Functions are values

And therefore they can:

- Be the argument of other functions
- Be the output (or return value) of other functions

Function implementation as last step

Lecture material

Bibliography

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

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

- Racket Guide: tutorial
- Racket Reference: reference manual