CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

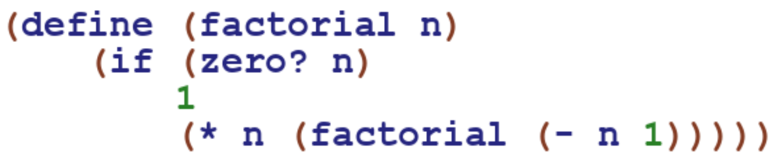
Lenguajes de Programación

- Introducción a Racket

Federico Olmedo
Ismael Figueroa

Racket is a functional language

Functional style

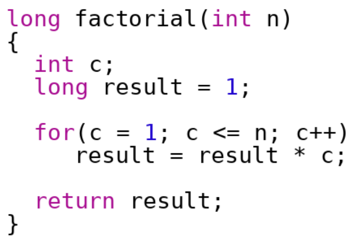


```
(define (factorial n)
    (if (zero? n)
        1
        (* n (factorial (- n 1)))))
```

Factorial program in Racket

- Built around the evaluation of expressions and the application of functions (on immutable data)
- Closer to expressing what to compute
- Closer to mathematics

Imperative style

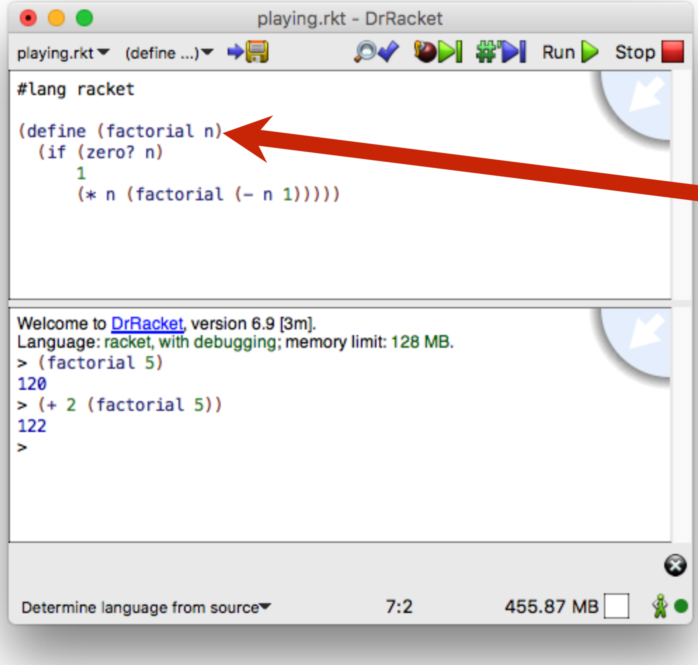


```
long factorial(int n)
{
  int c;
  long result = 1;

  for(c = 1; c <= n; c++)
     result = result * c;

  return result;
}
```

Factorial program in C

- Built around the execution of sequences of commands for their effects on mutable storage
- Closer to expressing how to compute it
- Closer to computer hardware

DrRacket: Our Development Environment



```
#lang racket

(define (factorial n)
  (if (zero? n)
      1
      (* n (factorial (- n 1)))))
```

```
Welcome to DrRacket, version 6.9 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> (factorial 5)
120
> (+ 2 (factorial 5))
122
>
```

Evaluates programs in the DW, making these defs available in the IW (also removes old defs)

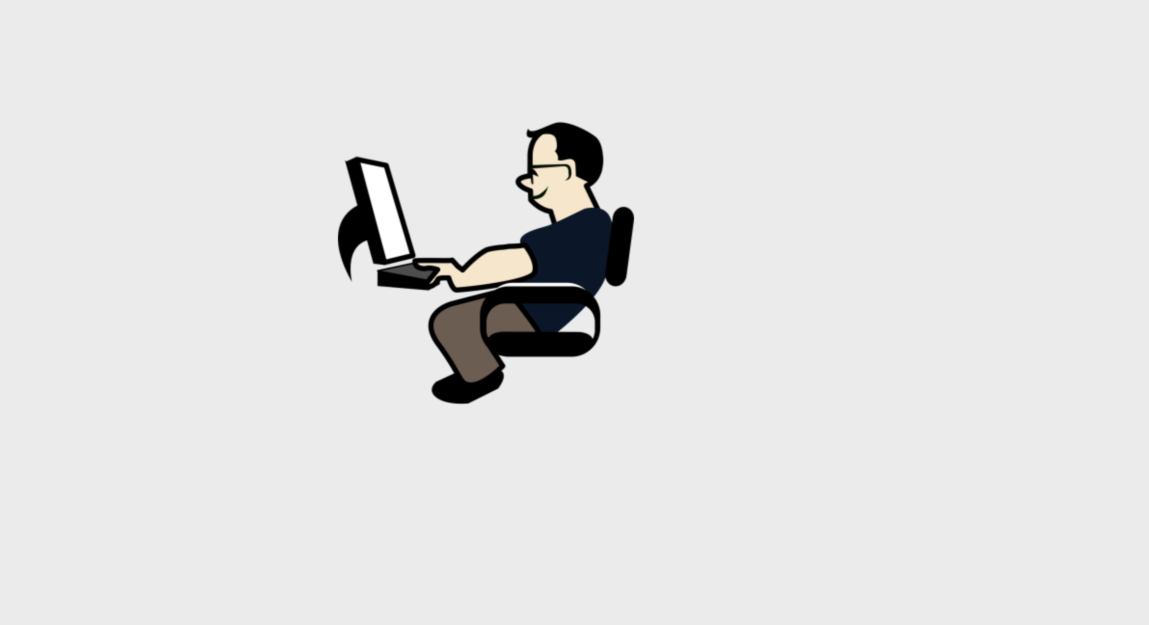
Language selection: Determines available primitives

Definition window (DW)

Interaction window (IW) REPL: read–evaluate–print loop

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

4 Introducing basic elements



Primitive datatypes, conditionals, definitions

Primitive datatypes and operators

- number: `+`, `-`, `*`, `/`, `quotient`, `sqrt`, ..., `<`, `<=`, `=`, `zero?`, ...
- boolean: `and`, `or`, `not`, …
- string: `string-length`, `string-append`, `substring`, …
- symbol: `equal?`, `string->symbol`, …

Conditionals

- (**if** *guard true-brach false-branch*)
- (**cond** [*guard*$_1$ *expr*$_1$] … [*guard*$_n$ *expr*$_n$])

Global and local definition of identifiers

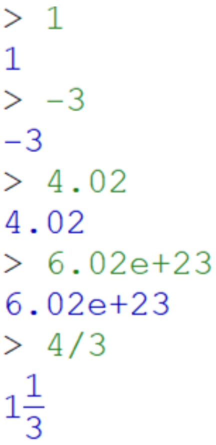
- (**define** *id expr*)
- (**let** ([*id*$_1$ *expr*$_1$] … [*id*$_n$ *expr*$_n$]) *body*)
- (**let*** ([*id*$_1$ *expr*$_1$] … [*id*$_n$ *expr*$_n$]) *body*)

Function definition

- (**define** (*func-name arg*$_1$ … *arg*$_n$) *func-body*)

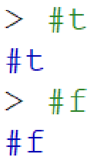
Examples in REPL interaction

numbers



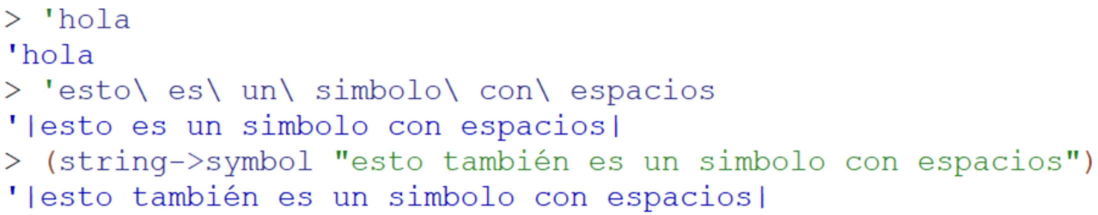
```
> 1
1
> -3
-3
> 4.02
4.02
> 6.02e+23
6.02e+23
> 4/3
1 1/3
```

booleans



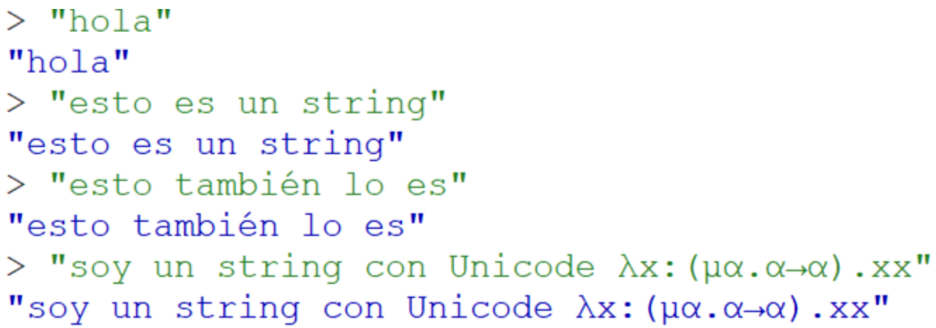
```
> #t
#t
> #f
#f
```

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|
```

strings

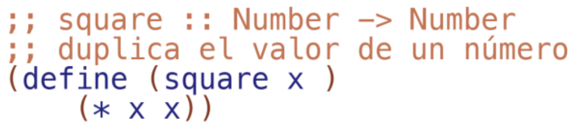


```
> "hola"
"hola"
> "esto es un string"
"esto es un string"
> "esto también lo es"
"esto también lo es"
> "soy un string con Unicode λx:(μα.α→α).xx"
"soy un string con Unicode λx:(μα.α→α).xx"
```

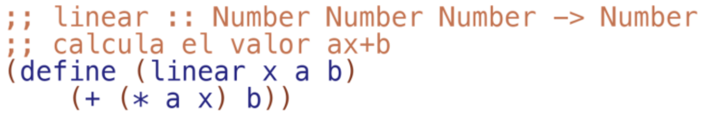
symbols and strings are different!

- **symbols are atomic values, strings are sequences of characters.**
- **equality comparison is O(1) for symbols and O(n) for strings.**

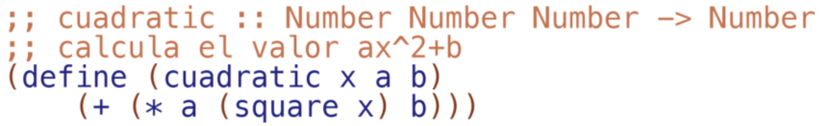
Example functions, in Definitions Window



```
;; square :: Number -> Number
;; duplica el valor de un número
(define (square x )
    (* x x))
```

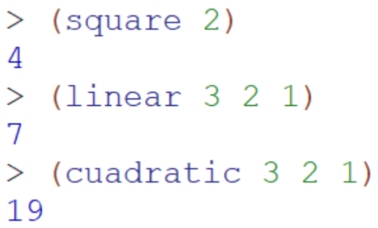


```
;; linear :: Number Number Number -> Number
;; calcula el valor ax+b
(define (linear x a b)
    (+ (* a x) b))
```



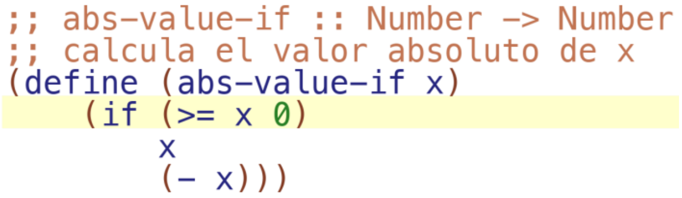
```
;; cuadratic :: Number Number Number -> Number
;; calcula el valor ax^2+b
(define (cuadratic x a b)
    (+ (* a (square x) b)))
```

Usage (in REPL)



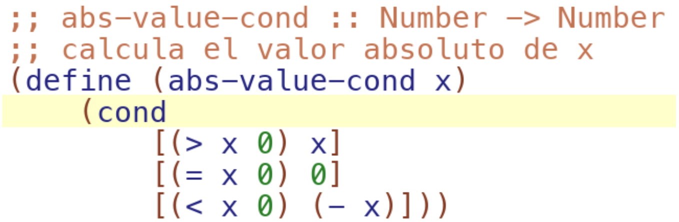
```
> (square 2)
4
> (linear 3 2 1)
7
> (cuadratic 3 2 1)
19
```

Example of function using conditionals



```
;; abs-value-if :: Number -> Number
;; calcula el valor absoluto de x
(define (abs-value-if x)
    (if (>= x 0)
        x
        (- x)))
```

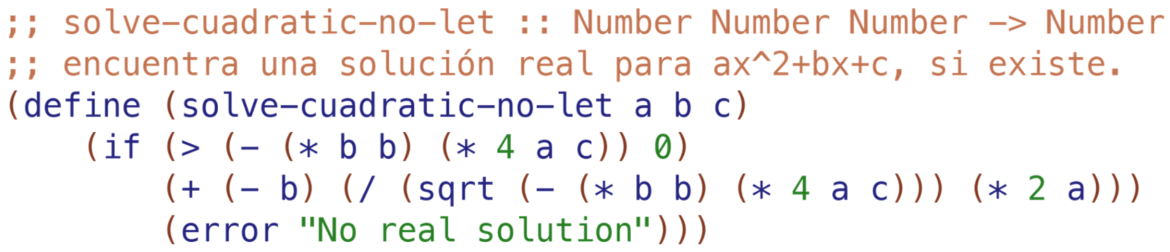
$$|x| = \begin{cases} x & \text{if} \quad x > 0, \\ 0 & \text{if} \quad x = 0, \\ -x & \text{if} \quad x < 0. \end{cases}$$



```
;; abs-value-cond :: Number -> Number
;; calcula el valor absoluto de x
(define (abs-value-cond x)
    (cond
        [(> x 0) x]
        [(= x 0) 0]
        [(< x 0) (- x)]))
```

Example function ***without*** local let binding

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

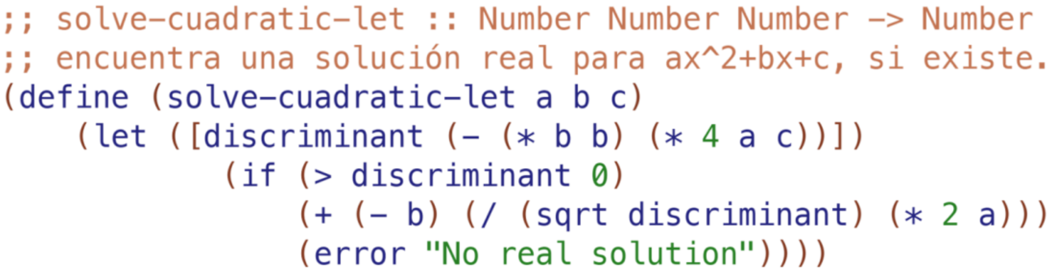


```
;; solve-cuadratic-no-let :: Number Number Number -> Number
;; encuentra una solución real para ax^2+bx+c, si existe.
(define (solve-cuadratic-no-let a b c)
    (if (> (- (* b b) (* 4 a c)) 0)
        (+ (- b) (/ (sqrt (- (* b b) (* 4 a c))) (* 2 a)))
        (error "No real solution")))
```

How ***legible*** is this solution?

Example function *with* let binding

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$



```
;; solve-cuadratic-let :: Number Number Number -> Number
;; encuentra una solución real para ax^2+bx+c, si existe.
(define (solve-cuadratic-let a b c)
    (let ([discriminant (- (* b b) (* 4 a c))])
            (if (> discriminant 0)
                (+ (- b) (/ (sqrt discriminant) (* 2 a)))
                (error "No real solution"))))
```

How ***legible*** is this solution?

Exercises

Function `my-max`

Define function `(my-max a b)`, which returns the greatest value between a and b.

Function `pick-random`

Define function `(pick-random a b)`, which randomly returns either `a` or `b`. Hint: simulate a coin flip with probability 0.5.

Use function `(random)` from the standard library to generate a random number between 0 and 1.

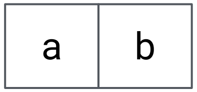
Function `pick-random-in-interval`

Define function `(pick-random-in-interval a b)`, which returns a random value in the interval `[a, b]`.

Basic data structures: pairs, list and vectors

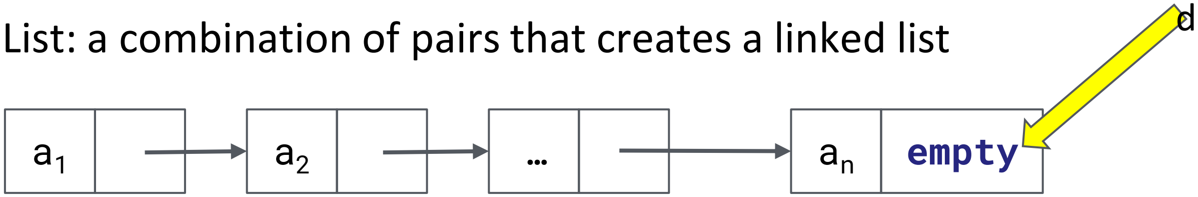
Immutable data structures and operators: pairs & lists ℹ

Pairs: join two arbitrary values



- (cons a b)   car, cdr ?

List: a combination of pairs that creates a linked list



special singleton value denoting empty list, also noted as null, or '()

- (cons a_1 (cons a_2 (… (cons a_n empty)…)))
- (list a_1 a_2 … a_n)

car, cdr, first, rest, append, length, empty?, reverse, list-ref, …

Mutable data structures: vectors ℹ

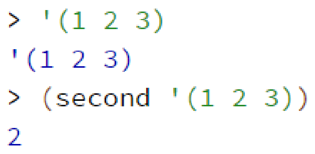
Vectors: fixed-length array with direct access/update

- (vector a_1 a_2 … a_n)

vector-ref, vector-set!, vector-length, …

Lists & Quotation

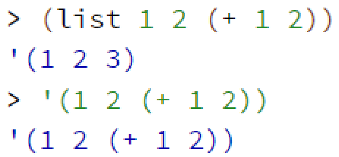
A literal list value is created using the *quote* operator.



```
> '(1 2 3)
'(1 2 3)
> (second '(1 2 3))
2
```

Quote tells Racket to consider everything after it as a *data*. This opens the door to representing the source code of our interpreters' languages as quoted Racket elements (more on this in the next lectures…)

Using *quote* is not the same as using the **`list`** constructor



```
> (list 1 2 (+ 1 2))
'(1 2 3)
> '(1 2 (+ 1 2))
'(1 2 (+ 1 2))
```

Exercises

Function **solve-cuadratic**

Define function `(solve-cuadratic a b c)`, which returns a pair with the two real solutions to equation ax^2+bx+c, if they exist. If there is only one real solution, return a `void` value as the second element. Otherwise raise an error.

Function **pick-random-vector**

Define function `(pick-random-from-vector v)`, which returns an element from a random position in vector `v`.

Use function `(random k)` from the standard library to generate a random integer between 0 and k-1.

Takeaways

- Prefix notation
- Dynamically type-checked
- Standard primitive and compound datatypes
- Difference between mutable and immutable data structures
- Secure access to list and vectors
- Quotation opens the door to the representation of code as Racket data

Lecture material

Bibliography

- PrePLAI: Introduction to functional programming in Racket [Sections 1-2]

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

- Racket Guide: tutorial
- Racket Reference: reference manual