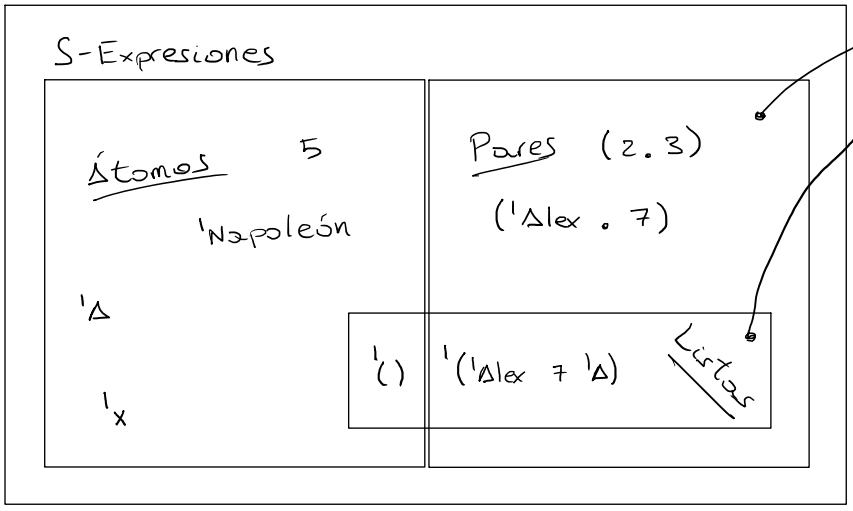
Anotaciones

sábado, 5 de marzo de 2022 19:20

(función parámetros) → 1er elemento tras paréntesis ⇒ función.



→ Pares = 2 Átomos $(\Delta_1 . \Delta_2)$

→ Listas = Pares donde el segundo elemento es otro par hasta que el último elemento del último par es una lista vacía '().

('Δ . ('B . ('C . ('D . '()))))

('Δ 'B 'C 'D)

'(Δ B C D)

Nota: Excepto '(), todas las listas son pares

Δ la comilla se le puede sacar "factor común" y la '() se puede omitir al final de una lista.

Car / Cdr ~ Operaciones sobre pares → y la mayoría de listas

Car → Retorna el primer elemento del par.

> (car '(Δ . B)) → 'Δ

> (car '(Δ . (B . (C . ()))) → 'Δ

1° 2° elemento ⇝ ¿Qué más da si el segundo es otro par?

> (car '(Δ B C)) → 'Δ

↳ Esta lista es lo mismo que el par anterior.

Cdr → Retorna el segundo elemento del par.

> (cdr '(Δ . B)) → 'B

> (cdr '(Δ . (B . (C . ()))) → '(B . (C . ()))

1° 2° elemento ⇝ ¿Qué más da si el segundo es otro par?

> (car '(Δ B C)) → '(B C)

↳ Esta lista es lo mismo que el par anterior.

Datos

Numéricos → 0, 1, 7, -3

Strings -> 'Alex , 'A . // Necesitan ' para ser reconocidos.
'Alex es equivalente a (quote Alex)
Si no hay ' , se interpreta que es una función, y si la función no existe explota.

Boolean -> #t -> True
#f -> False

Nota: Todo valor que no sea falso es verdadero.

Funciones

(+ a b ...) -> a + b + ...
(- a b ...) -> a - b - ...
(* a b ...) -> a * b * ...
(/ a b ...) -> a / b / ...

(and a b ...) -> a & b & ...
(or a b ...) -> a || b || ...
} No son pasables como parámetros!

(null? a) -> Revisa si es nulo
(zero? n) -> Revisa si es 0
(positive? n) -> Revisa si es >0
(negative? n) -> Revisa si es <0
(empty? lista) -> Revisa si el par/lista es '()
(even? n) -> Revisa si es par
(eq? a b) -> Revisa si (a == b) => Sólo átomos
(equal? a b) -> Revisa si (a.equals(b)) => Para todo
(member? e lista) -> Revisa si e pertenece a lista.
↳ #f si no
↳ lista si sí.

Nota: Todos llevan ? al final menos member. Booleanos

Funciones Lambda

(lambda (<parámetros>) <órden>)
(λ (<parámetros>) <órden>)
La órden será siempre una función => irá entre paréntesis

((λ (x) (if (not (zero? x)) x 'ERROR!)) 7)
Inicio, Parámetro, #t <=> x=∅, #t <=> x≠∅, #t, #f, Fin, Parámetro que se le pasa

λ Retorna x si x≠∅, si no retorna 'ERROR!

Definiciones y Recursividad

(define (<nombre> <parámetros?>) <órden>)
Órden => Sólo 1, if-else cuentan como una órden

(define (factorial x)
(if (zero?) 1 -> Si x = ∅ => 1

```
(define (factorial x)
        (if (zero? x) 1 -> Si x=0 => 1
            (* x (factorial (- x 1)))
        )                 ↳> Else  x * factorial (x-1)
)
```

```
(define (foo x)                    ↗ Switch
        (cond [(zero? x) => 1]
              [(even? x) => (+ x (foo (- x 1)))]
              [else      => (- x (foo (- x 1)))]
        )
)
```

↳ Si se cumple uno, no se evalúan los demás.

↳ Sin ( )

↳ Si no se cumplió ninguno se hace éste

No es necesario definir una recursividad para que sea recursiva.

```
(define (len lista)
        (length lista)
)
```

length es recursiva (librería de Racket)

⇓

mi len es recursiva

Funciones de Racket (algunas)

(cons a b) -> (a . b) -> Crea par ↗ Crea lista

(list a b ...) -> (a . (b . ... ())) -> (a b ...)

(reverse lista) -> lista pero al revés

```
((λ x (car x)) 2 4 6)
```

↗ 1 parámetro

Sin ()? ↙

↳ Se le pasan 3 parámetros? → Ninguno es lista?

↳ Retorna el primer elemento del par x entonces x debería ser un par/lista.

Δ una función cuyo parámetro no tenga () se le podrán pasar varios elementos, los convertirá en una lista cuyo nombre será el del parámetro.

```
((λ x (car x)) 2 4 6) -> (car '(2 4 6)) -> 2
```

Funciones de órden superior (FOS)

↳ Recibe una función como parámetro
↳ Retorna una función

```
(define (foo oper n)
```

(oper n ∅)
)
↳ Oper está al principio => Debe ser una función

(foo eq? 7) -> #f (7 ≠ ∅)
(foo > 7) -> #t (7 > ∅)

FOS de Racket

Función Datos
↑ ↑
(filter <foo_test> <lista>) -> Retorna los elementos de <lista> que cumplen <foo_test>

(filter positive? '(-3 -2 -1 1 2 3))
'(1 2 3)

-> Función que retorne los elementos de una lista que son pares y positivos.
↳ (and (even? x) (positive? x)) , siendo x el elemento

(filter (λ (x) (and (even? x) (positive? x))) '(-3 -2 -1 1 2 3))
<foo_test> ¿Qué más da si es λ o no? Datos

(drop-until <foo_test> <lista>) ~> Y todas las similares.

drop -> Borrar until -> Hasta que
take -> Coger while -> Mientras

(drop-until <foo_test> <data>) ~> Borra elementos hasta que <foo_test> se cumple

(drop-until even? '(1 3 7 12 9 23)) -> '(12 9 23)

Plantilla: <Primero> elementos <Segundo> foo_test se cumpla

Drop	Take	Until	While
↓	↓	↓	↓
Borrar	Coger	Hasta que	Mientras

(apply <foo> <lista>) -> Aplica <foo> a los elementos de <lista>.

(apply even? '(1 2 3 4 5 6))
'(#f #t #f #t #f #f)

Nota: Estoy diciendo que toda función es sustituible por

'(#f #t #f #t #f #f)

(apply (λ (x) (+ x 2)) '(1 2 3))

que toda función es sustituible por una λ porque todas las λ son funciones. ¿Se pilla?

(3 4 5) <foo>

1+2 2+2 3+2

(map <foo> <lista$_1$> ... <lista$_n$>)

→ Se necesitan tantas <list> como parámetros tenga <foo>.

Aplica <foo> tantas veces como longitud tengan las listas.
Los parámetros de la iteración i son los elementos i de cada lista.

(map + '(1 2 3) '(4 5 6))

'((+ 1 4) (+ 2 5) (+ 3 6)) → '(5 7 9)

Conclusión$_1$: Se necesitan tantas listas como parámetros tenga <foo>
Conclusión$_2$: Todas las listas deben tener la misma longitud excepto si <foo> admite un número variable de parámetros

((compose <f> <g>) <dato> ... <dato$_n$>) ⇝ f (apply g datos)

Aplica primero g a todos los datos. Aplica a la lista resultante.

Definiciones locales let, let* y letrec

(let ((x 1) (y 2) (z +))
(z x y)
)

Definiciones

+ 1 2

↳ (+ 1 2) -> 3

(let ((x 1) (y (+ x 2)))
(+ x y)
)

No se puede utilizar una variable definida en este mismo let

↳ ERR.

(let ((x 1))
(let ((x 2) (y (+ x 2)))

No definida aún

```
            (+ x y)
        )      2 3
  )
    └> (+ 2 3) --> 5
```

```
(let* ( (<var> <valor>) ... ) <código>)
```

Nota: let* permite usar variables definidas en sí mismo

```
(let* ( (x 1) (y (+ x 2)))
                   1
      (+ x y)
         1 2
 ) └> (+ 1 2) --> 3
```

```
(letrec ( (<var> <valor>) ... ) <código>)
```

Nota: letrec permite definiciones recursivas.

Curry ~ Permite bloquear algunos parámetros de una función

```
(define (curry+ m)
        (λ (n) (+ n m))
)
((curry+ 7) 5) --> 12
```