Aritmética y abstracción de datos

José A. Alonso y María J. Hidalgo

Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Operaciones numéricas

```
;;; (+ n1 ... np)     =>  Suma de n1, n2, ..., np
(+ 3 4)               =>  7
(+ 3)                 =>  3
(+)                   =>  0

;;; (- n1 ... np)     =>  Resta de n1, n2, ..., np
(- 3 4)               =>  -1
(- 3 4 5)             =>  -6
(- 3)                 =>  -3

;;; (* n1 ... np)     =>  Producto de n1, n2, ..., np
(* 2 3 5)             =>  30
(* 4)                 =>  4
(*)                   =>  1

;;; (/ n1 ... np)     =>  Cociente de n1, n2, ..., np
(/ 20 2 5)            =>  2
(/ 2)                 =>  0.5
```

Operaciones numéricas

```
;;; (quotient n k)   =>  El cociente entero de n entre k.
;;; (remainder n k)  =>  El resto de n entre k con el signo de n.
(quotient 13 4)      =>  3
(remainder 13 4)     =>  1
(quotient -13 4)     =>  -3
(remainder -13 4)    =>  -1
(quotient 13 -4)     =>  -3
(remainder 13 -4)    =>  1
(quotient -13 -4)    =>  3
(remainder -13 -4)   =>  -1

;;; (max n1 ... np)  =>  Máximo de n1, n2, ..., np
(max 3 4)            =>  4
(max 3.9 4)          =>  4.0

;;; (min n1 ... np)  =>  Mínimo de n1, n2, ..., np
(min 3 4)            =>  3
(min 3 4.6)          =>  3.0
```

Operaciones numéricas

```
;;; (abs n)           =>  Valor absoluto de n
(abs -7)              =>  7

;;; (gcd n1 ... np)   =>  Máximo común divisor de n1, ..., np
(gcd 10 -15)          =>  5
(gcd)                 =>  0

;;; (lcd n1 ... np)   =>  Mínimo común múltiplo de n1, ..., np
(lcm 10 -15)          =>  30
(lcm)                 =>  1

;;; (floor n)         =>  El mayor entero que es <= n.
(floor 4.3)           =>  4.0
(floor -4.3)          =>  -5.0
(floor 3.5)           =>  3.0
(floor 4.7)           =>  4.0
(floor -4.7)          =>  -5.0
```

Operaciones numéricas

```
;;; (ceiling n)      =>  El menor entero que es >= n.
(ceiling 4.3)        =>  5.0
(ceiling -4.3)       =>  -4.0
(ceiling 3.5)        =>  4.0
(ceiling 4.7)        =>  5.0
(ceiling -4.7)       =>  -4.0
;;; (truncate n)     =>  El entero obtenido quitándole a n la parte decimal.
(truncate 4.3)       =>  4.0
(truncate -4.3)      =>  -4.0
(truncate 3.5)       =>  3.0
(truncate 4.7)       =>  4.0
(truncate -4.7)      =>  -4.0
;;; (Round n)        =>  El entero más próximo a n.
(round 4.3)          =>  4.0
(round -4.3)         =>  -4.0
(round 3.5)          =>  4.0
(round 4.7)          =>  5.0
(round -4.7)         =>  -5.0
```

Operaciones numéricas

```
;;; (sqrt n)          =>  La raiz cuadrada de n.
(sqrt 9)              =>  3.0

;;; (expt n k)        =>  Eleva n a la potencia k.
(expt 2 3)            =>  8

;;; (exp n)           =>  La exponencial de n en base e.
(exp 1)               =>  2.71828182845905

;;; (log n)           =>  El logaritmo de n en base e.
(log (exp 1))         =>  1.0
```

Operaciones numéricas

```
;;; (sin n)          =>  Seno de n
;;; (cos n)          =>  Coseno de n
;;; (tan n)          =>  Tangente de n
;;; (asin n)         =>  Arco seno de n
;;; (acos n)         =>  Arco coseno de n
;;; (atan n)         =>  Arco tangente de n
(* 2 (asin 1))       =>  3.14159265358979
(sin 3.14)           =>  0.0015926529164872 7
```

Relaciones numéricas

```
;;; (= m n)          =>  #t, si m es igual que n; #f, e.c.c.
;;; (< m n)          =>  #t, si m es menor que n; #f, e.c.c.
;;; (<= m n)         =>  #t, si m es menor o igual que n; #f, e.c.c.
;;; (> m n)          =>  #t, si m es mayor que n; #f, e.c.c.
;;; (>= m n)         =>  #t, si m es mayor o igual que n; #f, e.c.c.
(> 4 (+ 1 2))       =>  #t
(> 4 (+ 1 5))       =>  #f

;;; (zero? n)        =>  #t, si n es cero; #f, e.c.c.
;;; (positive? n)    =>  #t, si n es positivo; #f, e.c.c.
;;; (negative? n)    =>  #t, si n es negativo; #f, e.c.c.
(zero? 7)           =>  #f
(positive? 7)       =>  #t
(negative? 7)       =>  #f
```

Recursión sobre números: suma-armonica

- **Problema:**

```
suma-armonica(n) = 1 + 1/2 + 1/3 + ··· + 1/n
```

- **Esquema:**

```
suma-armonica(0) = 1
suma-armonica(n) = 1 + 1/2 + 1/3 + ··· + 1/(n-1) + 1/n =
                 = suma-armonica(n-1) + 1/n
```

- **Procedimiento:**

```
(define suma-armonica
  (lambda (n)
    (if (zero? n)
        0
        (+ (/ 1 n) (suma-armonica (- n 1))))))
```

- **Ejemplos:**

```
(suma-armonica 2)    =>  1.5
(suma-armonica 500)  =>  6.79282342999052
```

Recursión sobre listas: `longitud`

- **Ejemplos:**

```
(longitud '(a b c))    =>  3
(longitud '(a (b c)))  =>  2
(longitud ())          =>  0
```

- **Esquema:**

```
(a b c)  ---> (b c) --------> 2 --> 3
         cdr        longitud    +1
```

- **Procedimiento:**

```
(define longitud
  (lambda (l)
    (if (null? l)
        0
        (+ 1 (longitud (cdr l))))))
```

Recursión sobre listas: `longitud`

- **Cálculo con traza**

```
> (trace longitud)
#<unspecified>
> (longitud '(a b c))
"CALLED" longitud (a b c)
 "CALLED" longitud (b c)
  "CALLED" longitud (c)
   "CALLED" longitud ()
   "RETURNED" longitud 0
  "RETURNED" longitud 1
 "RETURNED" longitud 2
"RETURNED" longitud 3
3
> (untrace)
(longitud)
```

- **Procedimiento** `length`

Recursión numérica: `list-ref`

- **Procedimiento `list-ref`:**

```
(list-ref '(a b c d) 3)      =>  d
(list-ref '((a b) (c d)) 1)  =>  (c d)
```

- **Esquema:**

```
(list-ref '(a b c d) 0)  =>  a
(list-ref '(a b c d) 2)  =>  (list-ref '(b c d) 1)  =>  c
```

- **Procedimiento `n-list-ref`:**

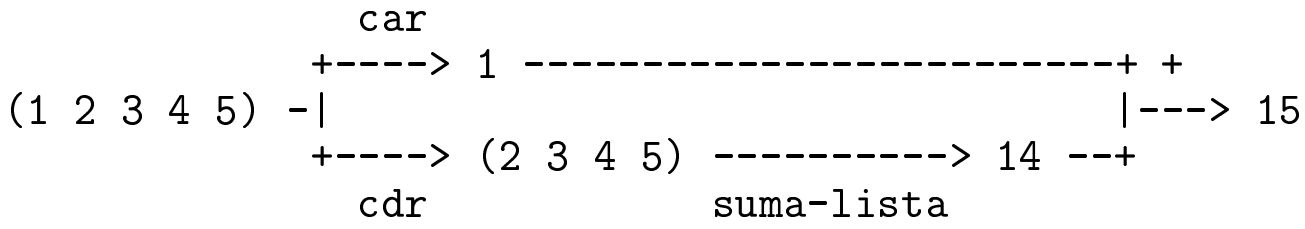
```
(define n-list-ref
  (lambda (l n)
    (if (zero? n)
        (car l)
        (n-list-ref (cdr l) (- n 1)))))
```

Recursión sobre lista de números: `suma-lista`

- **Ejemplos:**

```
(suma-lista '(1 2 3 4 5))  =>  15
(suma-lista '(7))          =>  7
(suma-lista '())           =>  0
```

- **Esquema:**



- **Procedimiento:**

```
(define suma-lista
  (lambda (l)
    (if (null? l)
        0
        (+ (car l) (suma-lista (cdr l))))))
```

Recursión sobre lista de números: multiplica-elementos

- **Ejemplos:**

```
(multiplica-elementos '(2 3 5) '(1 -1 2))  =>  (2 -3 10)
(multiplica-elementos '(2 5) '(1.2 3.2))   =>  (2.4 16.0)
(multiplica-elementos () ())               =>  ()
```

- **Procedimiento:**

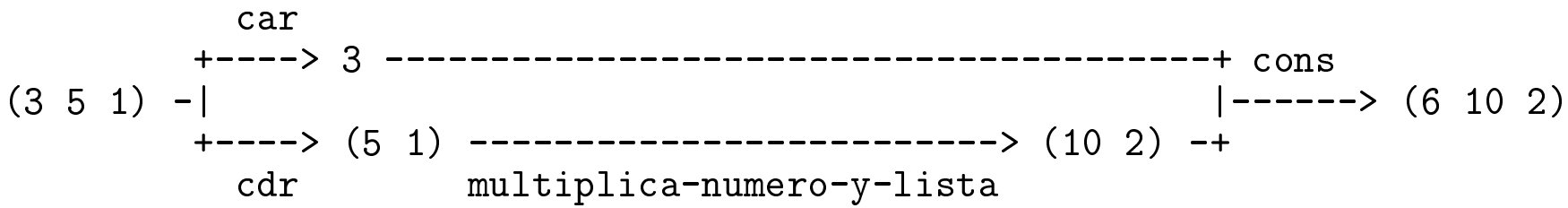
```
(define multiplica-elementos
  (lambda (l1 l2)
    (if (null? l1)
        ()
        (cons (* (car l1) (car l2))
              (multiplica-elementos (cdr l1) (cdr l2))))))
```

Recursión sobre listas: multiplica-numero-y-lista

- **Ejemplos:**

```
(multiplica-numero-y-lista 2 '(3 5 1))  =>  (6 10 2)
(multiplica-numero-y-lista 2 '())       =>  ()
```

- **Esquema:**



- **Procedimiento:**

```
(define multiplica-numero-y-lista
  (lambda (n l)
    (if (null? l)
        ()
        (cons (* n (car l))
              (multiplica-numero-y-lista n (cdr l))))))
```

Recursión sobre lista de números: `producto-escalar`

- **Ejemplos:**

```
(producto-escalar '(2 3 5) '(1 -1 2))  =>  9
(producto-escalar () ())               =>  0
```

- **Procedimiento:**

```
(define producto-escalar
  (lambda (l1 l2)
    (suma-lista (multiplica-elementos l1 l2))))
```

Recursión sobre listas: `indice`

- **Ejemplos:**

```
(indice 'c '(a b c d))    =>  2
(indice '(a) '((a) b c))  =>  0
(indice 'a '((a) b c))    =>  -1
(indice 'a '())           =>  -1
```

- **Procedimiento:**

```
(define indice
  (lambda (x l)
    (cond
     ((not (member x l)) -1)
     ((equal? x (car l)) 0)
     (else (+ 1 (indice x (cdr l)))))))
```

Abstracción de datos: Aritmética racional

- **Representación de los racionales mediante pares:**

```
;;; (crea-racional 2 3)  =>  (2 3)
;;; (crea-racional 2 0)  =>  ERROR: el denominador no puede ser cero
(define crea-racional
  (lambda (x y)
    (if (zero? y)
        (error "el denominador no puede ser cero")
        (list x y))))

;;; (numerador (crea-racional 2 3))  =>  2
(define numerador
  (lambda (rac)
    (car rac)))

;;; (denominador (crea-racional 2 3))  =>  3
(define denominador
  (lambda (rac)
    (cadr rac)))
```

Abstracción de datos: Aritmética racional

- **Aritmética racional usando** `crea-racional`, `numerador` **y** `denominador`

```
;;; (rzero? (crea-racional 0 3))  =>  #t
;;; (rzero? (crea-racional 5 3))  =>  #f
(define rzero?
  (lambda (rac)
    (zero? (numerador rac))))

;;; (r+ (crea-racional 1 2) (crea-racional 3 5))  =>  (11 10)
(define r+
  (lambda (x y)
    (crea-racional
      (+ (* (numerador x) (denominador y))
         (* (numerador y) (denominador x)))
      (* (denominador x) (denominador y)))))
```

Abstracción de datos: Aritmética racional

```
;;; (r* (crea-racional 1 2) (crea-racional 3 5))  =>  (3 10)
(define r*
  (lambda (x y)
    (crea-racional
      (* (numerador x) (numerador y))
      (* (denominador x) (denominador y)))))

;;; (r-opuesto (crea-racional 2 3))  =>  (-2 3)
(define r-opuesto
  (lambda (x)
    (r* (crea-racional -1 1) x)))

;;; (r- (crea-racional 1 2) (crea-racional 3 5))  =>  (-1 10)
(define r-
  (lambda (x y)
    (r+ x (r-opuesto y))))
```

Abstracción de datos: Aritmética racional

```
;;; (r-inverso (crea-racional 2 3))  =>  (3 2)
;;; (r-inverso (crea-racional 0 3))  =>  ERROR: El numerador no puede ser cero
(define r-inverso
  (lambda (x)
    (if (rzero? x)
        (error "El numerador no puede ser cero")
        (crea-racional (denominador x) (numerador x)))))

;;; (r/ (crea-racional 1 2) (crea-racional 3 5))  =>  (5 6)
(define r/
  (lambda (x y)
    (r* x (r-inverso y))))

;;; (r= (crea-racional 1 2) (crea-racional 3 6))  =>  #t
;;; (r= (crea-racional 1 2) (crea-racional 3 5))  =>  #f
(define r=
  (lambda (x y)
    (rzero? (r- x y))))
```

Abstracción de datos: Aritmética racional

```
;;; (rpositive? (crea-racional 2 3))   =>  #t
;;; (rpositive? (crea-racional -2 3))  =>  #f
(define rpositive?
  (lambda (x)
    (positive? (* (numerador x) (denominador x)))))

;;; (r> (crea-racional 1 2) (crea-racional 3 5))   =>  #f
;;; (r> (crea-racional 1 2) (crea-racional -3 5))  =>  #t
(define r>
  (lambda (x y)
    (rpositive? (r- x y))))

;;; (rmax (crea-racional 1 2) (crea-racional 3 5))   =>  (3 5)
;;; (rmax (crea-racional 1 2) (crea-racional -3 5))  =>  (1 2)
(define rmax
  (lambda (x y)
    (if (r> x y)
        x
        y)))
```

Abstracción de datos: Aritmética racional

- **Representación de los racionales mediante pares simplificados:**

```
;;; (crea-racional 2 3)  =>  (2 3)
;;; (crea-racional 2 0)  =>  ERROR: el denominador no puede ser cero
;;; (crea-racional 2 6)  =>  (1 3)
(define crea-racional
  (lambda (x y)
    (if (zero? y)
        (error "el denominador no puede ser cero")
        (list (/ x (gcd x y))
              (/ y (gcd x y))))))
```

- **Simplificación en operaciones:**

```
;;; (r* (crea-racional 5 6) (crea-racional 3 5))  =>  (1 2)
;;; (r+ (crea-racional 5 6) (crea-racional 3 2))  =>  (7 3)
```

Bibliografía

- [Abelson–96]
  Cap. 2.1: "Introduction to data abstraction".
- [Springer–94]
  Cap. 3: "Data abstraction and numbers".