



Tecnologías de Programación

Programación Funcional



Lenguajes Funcionales

- Categorías de Lenguajes:
 - **Puros**: tienen una mayor potencia expresiva, conservando a la vez su transparencia referencial.
 - Ejemplo: Haskel, Miranda.
 - **Híbridos**: admiten secuencias de instrucciones o la asignación de variables.
 - Ejemplo: Scala, Lisp, Scheme, Ocaml y Standard ML.



Tecnologías de Programación

Racket



Sintaxis

- Palabras clave, variables y los símbolos son llamados Identificadores. Los identificadores pueden estar formados por:
 - [a-z]
 - [A-Z]
 - [0-9]
 - ? ! . + - * / < = > : \$ % ° & _ ~ @
 - Ejemplo: Hola, n, x, x3, ?\$&*!!!



Sintaxis

- Todos los identificadores deben estar delimitados por:
 - Un espacio en blanco
 - Comillas dobles (“ ”)
 - Paréntesis
 - carácter de comentario (;)
- No hay límite de longitud
- No es case-sensitive:
 - Abc, abc, aBc son todos el mismo identificador



Sintaxis

- Las estructuras y las Listas se encierran entre paréntesis:
 - Ejemplo: (a b c) o (* (- x 2) y)
- () => Lista vacía
- Valores Booleanos:
 - #t → verdadero
 - #f → falso



Sintaxis

- Vectores:
 - comienzan con #(
 - Finalizan con)
 - Ejemplo: #(esto es un vector de símbolos)
- String: encerrados en “”
- Caracteres: precedidos por #\
 - Ej: #\a



Sintaxis

- Números:
 - Enteros: -123
 - Racionales: $1/2$
 - En punto flotante: 1.3
 - Notación Científica: $1e12$
 - Complejos en Notación Rectangular: $1.3-2.7i$
 - Complejos en Notación Polar: $-1.2@73$



Convención de Nombres

- Predicados finalizan en ?: retornan #t o #f
 - Ejemplo: eq?, zero?, string=?
- El nombre de la mayoría de los procedimientos de string, caracteres y vectores comienzan con el prefijo string-, char- y vector-
 - Ejemplo: string-append
- Conversión entre tipos de objetos se escriben como *tipo1->tipo2*.
 - Ej: *vector->list*



Interactuando con Racket

- Probar:
 - "hola"
 - 42
 - $22/7$
 - 3.141592653
 - +
 - (+ 76 31)
 - '(a b c d)



Interactuando con Racket

- Resultados:
 - "hola" \Rightarrow "hola"
 - 42 \Rightarrow 42
 - $22/7 \Rightarrow 3 \frac{1}{7}$
 - 3.141592653 \Rightarrow 3.141592653
 - + \Rightarrow #<primitive:+>
 - (+ 76 31) \Rightarrow 107
 - '(a b c d) \Rightarrow (a b c d)



Interactuando con Racket

- Identifique que hacen las funciones:
 - `(car '(a b c))`
 - `(cdr '(a b c))`
 - `(cons 'a '(b c))`
 - `(cons (car '(a b c))
 (cdr '(d e f)))`



Interactuando con Racket

- Definir un procedimiento:

```
(define cuadrado  
  (lambda (n)  
    (* n n)))
```

- Y usarlo:

- $(\text{cuadrado } 5) \Rightarrow 25$
- $(\text{cuadrado } -200) \Rightarrow 40000$
- $(\text{cuadrado } 0.5) \Rightarrow 0.25$
- $(\text{cuadrado } -1/2) \Rightarrow 1/4$



Expresiones simples

- Notación prefija
 - $(+ 2 2) \Rightarrow 4$
 - $(+ (+ 2 2) (+ 2 2)) \Rightarrow 8$
 - $(- 2 (* 4 1/3)) \Rightarrow 2/3$
 - $(* 2 (* 2 (* 2 (* 2 2)))) \Rightarrow 32$
 - $(/ (* 6/7 7/2) (- 4.5 1.5)) \Rightarrow 1.0$



Expresiones simples

- Estructura de agregación: Listas (list)
 - `(quote (1 2 3 4 5))` → lista de números
 - `'("esto" "es" "una" "lista")` → lista de strings
 - `'(4.2 "hola")` → lista de múltiples tipos
 - `'((1 2) ("hola" "Racket"))` → lista de listas



Expresiones Simples

- Quote (') le dice a Racket que trate un identificador como símbolo y no como variable
- Los símbolos y variables en Racket son similares a los símbolos y variables en expresiones matemáticas y ecuaciones
 - En expresiones Matemáticas:
 - $1 - x \rightarrow$ pensamos en x como variable
 - En expresiones Algebraicas:
 - $X^2 - 2 \rightarrow$ pensamos en x como símbolo




Expresiones Simples

- Operadores de Listas
 - car: retorna el primer elemento de la lista
 - cdr (could-er): retorna el resto de la lista
 - $(\text{car } '(a\ b\ c)) \Rightarrow a$
 - $(\text{cdr } '(a\ b\ c)) \Rightarrow (b\ c)$
 - $(\text{cdr } '(a)) \Rightarrow ()$



Expresiones Simples

- $(\text{car } '(a\ b\ c)) \Rightarrow a$
- $(\text{cdr } '(a\ b\ c)) \Rightarrow (b\ c)$
- $(\text{cons } 'a\ '(b\ c)) \Rightarrow (a\ b\ c)$
- $(\text{cons } (\text{car } '(a\ b\ c))$
 $(\text{cdr } '(d\ e\ f))) \Rightarrow (a\ e\ f)$




Expresiones Simples

- Operadores de Listas: car – cdr
 - Resuelva:
 - (car (cdr '(a b c)))
 - (cdr (cdr '(a b c)))
 - (car '((a b) (c d)))
 - (cdr '((a b) (c d)))



Expresiones Simples

- Operadores de Listas: car – cdr
 - Resultados:
 - $(\text{car } (\text{cdr } '(a\ b\ c))) \Rightarrow b$
 - $(\text{cdr } (\text{cdr } '(a\ b\ c))) \Rightarrow (c)$
 - $(\text{car } '((a\ b)\ (c\ d))) \Rightarrow (a\ b)$
 - $(\text{cdr } '((a\ b)\ (c\ d))) \Rightarrow ((c\ d))$



Expresiones Simples

- Operadores de Listas:
 - cons: construye listas. Recibe dos argumentos. Usualmente el segundo es una lista y en ese caso retorna una lista
 - $(\text{cons } 'a \ '()) \Rightarrow (a)$
 - $(\text{cons } 'a \ '(b \ c)) \Rightarrow (a \ b \ c)$
 - $(\text{cons } 'a \ (\text{cons } 'b \ (\text{cons } 'c \ '()))) \Rightarrow (a \ b \ c)$
 - $(\text{cons } '(a \ b) \ '(c \ d)) \Rightarrow ((a \ b) \ c \ d)$



Expresiones Simples

- Operadores de Listas:
 - cons: construye listas. Recibe dos argumentos. Usualmente el segundo es una lista y en ese caso retorna una lista
 - $(\text{cons } 'a \ '()) \Rightarrow (a)$
 - $(\text{cons } 'a \ '(b \ c)) \Rightarrow (a \ b \ c)$
 - $(\text{cons } 'a \ (\text{cons } 'b \ (\text{cons } 'c \ '()))) \Rightarrow (a \ b \ c)$
 - $(\text{cons } '(a \ b) \ '(c \ d)) \Rightarrow ((a \ b) \ c \ d)$



Expresiones Simples

- Operadores de Listas:
 - Resuelva:
 - `(car (cons 'a '(b c)))`
 - `(cdr (cons 'a '(b c)))`
 - `(cons (car '(a b c))
 (cdr '(d e f)))`
 - `(cons (car '(a b c))
 (cdr '(a b c)))`



Expresiones Simples

- Operadores de Listas:
 - Resultados:
 - $(\text{car } (\text{cons } 'a \ '(b \ c))) \Rightarrow a$
 - $(\text{cdr } (\text{cons } 'a \ '(b \ c))) \Rightarrow (b \ c)$
 - $(\text{cons } (\text{car } '(a \ b \ c))$
 $(\text{cdr } '(d \ e \ f))) \Rightarrow (a \ e \ f)$
 - $(\text{cons } (\text{car } '(a \ b \ c))$
 $(\text{cdr } '(a \ b \ c))) \Rightarrow (a \ b \ c)$



Expresiones Simples

- Cons: construye pares. Cuando el segundo parámetro es una lista devuelve una lista *Propia*.
 - Lista Propia: Una lista vacía es una lista propia, y toda lista cuyo **cdr** sea una lista propia es una lista propia.
 - Listas impropias: compuestas por pares donde se marca la separación de elementos por puntos.



Expresiones simples

- $(\text{cons } 'a \ 'b) \rightarrow (a . b)$
- $(\text{cdr } '(a . b)) \rightarrow b$
 - A diferencia de $(\text{cdr } '(a b)) \rightarrow (b)$
- $(\text{cons } 'a \ '(b . c)) \rightarrow (a b . c)$



Evaluado de Expresiones

- (procedimiento arg1 arg2 ...)
 - Buscar el valor de *procedimiento*
 - Buscar el valor de *arg1*
 - Buscar el valor de *arg2*
 -
 - Aplicar el valor de *procedimiento* a los valores de *arg1*, *arg2*, ...



Evaluado de Expresiones

- Ejemplo:
 - $(+ \ 3 \ 4)$
 - el valor de $+$ es el procedimiento *adición*
 - el valor de 3 es el número 3
 - el valor de 4 es el número 4
 - aplicando el procedimiento *adición* a los números 3 y 4 devuelve 7
 - pruebe: `((car (cdr (list + - * /))) 17 5)`



Programación Funcional


to be continued...



Variables y Expresiones Let


- La forma sintáctica Let incluye pares de *variables-expresiones* junto con una secuencia de expresiones que representan el cuerpo del Let

(let ((var1 val) [(var2 val)...]) exp1 exp2 ...)



Variables y Expresiones Let

- También se usa para simplificar expresiones
 - $(+ (* 4 4) (* 4 4)) \Rightarrow 32$
 - $(\text{let } ((a (* 4 4)))$
 $(+ a a)) \Rightarrow 32$
 - $(\text{let } ((\text{list1 } '(a b c)) (\text{list2 } '(d e f))))$
 $(\text{cons } (\text{cons } (\text{car list1}) (\text{car list2}))$
 $(\text{cons } (\text{car } (\text{cdr list1})) (\text{car } (\text{cdr list2}))))$
 $\Rightarrow ((a . d) b . e)$



Variables y Expresiones Let

- Es posible anidar Lets
 - ```
(let ((a 4) (b -3))
 (let ((a-squared (* a a))
 (b-squared (* b b)))
 (+ a-squared b-squared)))
```

 → 25
  - ```
(let ((x 1))  
  (let ((x (+ x 1)))  
    (display x)))
```

 → 2



Expresiones Lambda

- Lambda permite crear un nuevo procedimiento
- Expresión general:
 - $(\text{lambda } (\text{var } \dots) \text{exp1 exp2 } \dots)$
 - Ejemplo: $(\text{lambda } (x) (+ x x)) \Rightarrow \#<\text{procedure}>$
- Uso:
 - $((\text{lambda } (x) (+ x x)) (* 3 4)) \Rightarrow 24$



Expresiones Lambda

- Como los procedimientos son objetos los podemos asignar a variables
 - (let ((double (lambda (x) (+ x x))))
 (list (double (* 3 4))
 (double (/ 99 11))
 (double (- 2 7)))) ⇒ (24 18 -10)
 - (let ((double-cons (lambda (x) (cons x x))))
 (double-cons 'a)) ⇒ (a . a)



Definiciones de alto nivel

- Las definiciones de **Alto Nivel** son vistas desde todos los procedimientos
- se declaran a partir de la cláusula *define*
 - $(\text{define double-any}$
 $(\text{lambda } (f\ x)$
 $(f\ x\ x)))$
 - $(\text{double-any } +\ 10) \Rightarrow 20$
 - $(\text{double-any cons 'a}) \Rightarrow (a . a)$



Definiciones de Alto Nivel

- se pueden utilizar para cualquier tipo de objetos, no solo procedimientos
(define sandwich "milanesa-tomate-y-lechuga")
sandwich \Rightarrow "milanesa-tomate-y-lechuga"



Definiciones de Alto Nivel

- Scheme provee abreviaturas llamadas `cadr` y `cddr` que son composiciones de
 - `(car (cdr list))`
 - `(cdr (cdr list))`
 - `(define cadr
 (lambda (x)
 (car (cdr x))))`
 - `(define cddr
 (lambda (x)
 (cdr (cdr x))))`



Expresiones Condicionales

- (if (<test>
 (<verdad>
 (<falso>
)
)
- Ej: (define abs
 (lambda (n)
 (if (< n 0)
 (- 0 n)
 n)
))



Expresiones Condicionales

- not: devuelve el inverso del parámetro dado
 - $(\text{not } \#t) \rightarrow \#f$
 - $(\text{not "false"}) \rightarrow \#f$
 - $(\text{not } \#f) \rightarrow \#t$
- or/and: realiza la comparación lógica y devuelve el resultado
 - $(\text{or}) \rightarrow \#f$
 - $(\text{or } \#t \#f) \rightarrow \#t$
 - $(\text{and } \#t \#f) \rightarrow \#f$



Expresiones Condicionales

- $=$, $<$, $>$, \leq , y \geq son todos predicados y responden a preguntas específicas sobre sus argumentos devolviendo un valor de verdad
- los nombres de los predicados normalmente finalizan en $?$ excepto los anteriores



Expresiones Condicionales

- `null?` : devuelve `#t` si el argumento es una lista vacía
 - `(null? ())` \rightarrow `#t`
 - `(null? 'abc)` \Rightarrow `#f`
 - `(null? '(x y z))` \Rightarrow `#f`



Expresiones Condicionales

- `eqv?`: requiere dos argumentos y devuelve `#t` si son equivalentes
 - `(eqv? 'a 'a) ⇒ #t`
 - `(eqv? 'a 'b) ⇒ #f`
 - `(eqv? #f #f) ⇒ #t`
 - `(eqv? #t #t) ⇒ #t`
 - `(eqv? #f #t) ⇒ #f`
 - `(eqv? 3 3) ⇒ #t`
 - `(eqv? 3 2) ⇒ #f`



Expresiones Condicionales

- Otros predicados son:
 - pair?
 - symbol?
 - number?
 - string?



Expresiones Condicionales

- cond: permite realizar múltiples test/acciones. Su forma general es la siguiente:
 - `(cond (test exp) ... (else exp))`
 - Ej: `(define sign
 (lambda (n)
 (cond
 ((< n 0) -1)
 ((> n 0) +1)
 (else 0))))`



Expresiones Condicionales

- ```
(define income-tax
 (lambda (income)
 (cond
 ((<= income 10000) (* income .05))
 ((<= income 20000) (+ (* (- income 10000) .08) 500.00))
 ((<= income 30000) (+ (* (- income 20000) .13) 1300.00))
 (else (+ (* (- income 30000) .21) 2600.00)))))
```
- $(\text{income-tax } 5000) \Rightarrow 250.0$
- $(\text{income-tax } 15000) \Rightarrow 900.0$
- $(\text{income-tax } 25000) \Rightarrow 1950.0$
- $(\text{income-tax } 50000) \Rightarrow 6800.0$



# Paradigma Funcional

continuará...



# Tecnologías de Programación

Programación Funcional



# Asignaciones

- Let\*: permite realizar asignaciones secuenciales, donde la definición de las variables internas pueden ver a las variables externas.
- ```
(let* ((x (* 5.0 5.0))  
      (y (- x (* 4.0 4.0))))  
  (sqrt y)) ⇒ 3.0
```



Recursividad

- la recursividad se produce cuando un procedimiento se llama a si mismo.
- Ej:
 - ```
(define length
 (lambda (ls)
 (if (null? ls)
 0
 (+ (length (cdr ls)) 1))))
```
  - $(\text{length } '()) \Rightarrow 0$
  - $(\text{length } '(a)) \Rightarrow 1$
  - $(\text{length } '(a\ b)) \Rightarrow 2$



# Recursividad

- ¿es posible hacer un let-bound recursivo?
  - EJ:

```
(let ((sum (lambda (ls)
 (if (null? ls)
 0
 (+ (car ls) (sum (cdr ls)))))))
 (sum '(1 2 3 4 5)))
```
- NO!!!!
- sum solo existe en el cuerpo del LET y no en la definición de las variables





# Recursividad

- *letrec*: al igual que *let* permite definir un conjunto de pares variable-valor y un conjunto de sentencias que las referencian.
- A diferencia de *let*, las variables son vistas en la cabecera también.
  - EJ:

```
(letrec ((sum (lambda (ls)
 (if (null? ls)
 0
 (+ (car ls) (sum (cdr ls)))))))
 (sum '(1 2 3 4 5)))
```



# Vectores

- un vector es una secuencia de objetos separados por un blanco y precedidos por un # o con la siguiente sintaxis:
  - (vector obj ...)
- Ejemplos:
  - #(a b c) → vector de elementos a, b y c
  - (vector) ⇒ #()
  - (vector 'a 'b 'c) ⇒ #(a b c)



# Vectores

- `(make-vector n)`  
`(make-vector n obj)`: retornan un vector de `n` posiciones. Si se provee *obj* se llenaran las posiciones con *obj*, en caso contrario permanecerán como *indefinido*
- `(make-vector 0) ⇒ #()`
- `(make-vector 0 'a) ⇒ #()`
- `(make-vector 5 'a) ⇒ #(a a a a a)`



# Vectores

- `(vector-length vector)` : retorna la cantidad de elementos de un vector.
  - `(vector-length #())`  $\Rightarrow$  0
  - `(vector-length #(a b c))`  $\Rightarrow$  3
  - `(vector-length (vector 1 2 3 4))`  $\Rightarrow$  4
  - `(vector-length (make-vector 300))`  $\Rightarrow$  300



# Vectores

- (vector-ref vector n) : retorna la enésima posición de un vector
  - (vector-ref #(a b c) 0)  $\Rightarrow$  a
  - (vector-ref #(a b c) 1)  $\Rightarrow$  b
  - (vector-ref #(x y z w) 3)  $\Rightarrow$  w



# Vectores

- (vector-ref vector n) : retorna la enésima posición de un vector
  - (vector-ref #(a b c) 0)  $\Rightarrow$  a
  - (vector-ref #(a b c) 1)  $\Rightarrow$  b
  - (vector-ref #(x y z w) 3)  $\Rightarrow$  w



# Vectores

- `(vector-set! vector n obj )`: establece el valor de la enésima posición del vector a *obj*
  - `(let ((v (vector 'a 'b 'c 'd 'e)))`  
    `(vector-set! v 2 'x)`  
    `v) ⇒ #(a b x d e)`





# Vectores

- `(vector-set! vector n obj )`: establece el valor de la enésima posición del vector a *obj*
  - `(let ((v (vector 'a 'b 'c 'd 'e)))`  
    `(vector-set! v 2 'x)`  
    `v) ⇒ #(a b x d e)`



# Vectores

- `(vector-fill! vector obj)` reemplaza cada elemento del vector *obj*
- `(vector->list vector)` devuelve una lista a partir de un vector
- `(list->vector list)` convierte una lista en vector



# Programación Funcional

continuará...



# Tecnologías de Programación

Programación Funcional



# Mapeo de Procedimientos a Listas

- Repetición de un procedimiento a cada elemento de una lista:
  - MAP
  - FOR-EACH



# Mapeo de Procedimientos a Listas

- MAP: aplica el procedimiento a cada elemento de la lista y devuelve una lista con los resultados.
  - $(\text{map } (\text{lambda } (x) (+ x 2)) '(1\ 2\ 3)) \rightarrow (3\ 4\ 5)$
- También es posible tener múltiples argumentos
  - $(\text{map cons } '(1\ 2\ 3) '(10\ 20\ 30)) \rightarrow ((1\ .\ 10) (2\ .\ 20) (3\ .\ 30))$



# Mapeo de Procedimientos a Listas

- FOR-EACH: aplica el procedimiento a cada elemento de la lista pero devuelve <void>.
  - (for-each display  
    (list "un" "dos " "tres"))





# Ingreso y Salida de datos

- read-char: permite leer un carácter desde el puerto indicado. Por defecto, la consola
- read-line: lee toda una línea de caracteres desde el puerto indicado y devuelve un string
- write-char: escribe un carácter al puerto indicado
- write: escribe la expresión en el puerto indicado en formato de máquina. Ej: los strings con comillas dobles y los caracteres precedidos con #\
- display: muestra el resultado de la expresión



# Puertos de Archivos

- `open-input-file`: abre un archivo y devuelve un puerto de lectura
- `open-output-file`: abre un archivo y devuelve un puerto de escritura
- `close-input-port` / `close-output-port`: cierran los puertos.



# Puertos de Archivos

- `Hola.txt`  
`hola!!`
- `(define i (open-input-file "hola.txt"))`
- `(read-char i) → #\h`
- `(define j (read i))`
- `j → ola!!`
- `(close-input-file i)`



# Puertos de Archivos

- (define o (open-output-file "saludo.txt"))
- (display "hola" o)
- (write-char #\space o)
- (display 'mundo! o)
- (newline o)
- (close-output-port o)

→ saludo.txt  
hola mundo!



# Puertos de Strings

- Las lecturas sobre puertos de string finalizan en los separadores de las mimas.
- (define i (open-input-string "hola mundo"))
- (read-char i)  $\rightarrow$  #\h
- (read i)  $\rightarrow$  ola
- (read i)  $\rightarrow$  mundo