INTELIGENCIA ARTIFICIAL
Introducción al Lenguaje Scheme

Prof. Adrián Domínguez Díez
Prof. José Luis Cuadrado García

Universidad de Alcalá
Departamento de Ciencias de la Computación e Inteligencia Artificial

→ Características de la Programación Funcional

- La **Programación Funcional** es un subtipo de la Programación Declarativa.
- Objetivo: Descripción del Problema.

  ¿"**Qué**" problema debe ser resuelto?

  – No importa "cómo" es resuelto el problema.

  – Evita aspectos o características de implementación.

- Características:

  – Expresividad , Extensible (regla del 10% - 90%) , Protección , Elegancia Matemática,

- Tipos: Programación **Funcional** o Aplicativa (*Lisp, Scheme, Haskell, ...*) y Programación **Lógica** (Prolog)

→ Características de la Programación Funcional

- El paradigma de la Programación Funcional es diferente a Programación Imperativa, y se aleja de la máquina de von-Neumann
- Esta basado en funciones matemática (notación funcional lambda de Church)
  - No existen realmente arquitecturas de computadores que permitan la eficiente ejecución de programas funcionales
- LISP es el primero, del cual derivan Scheme, CommonLISP, ML y Haskell

→ Características de la Programación Funcional

¿Porqué la Programación Funcional es relevante?

- En programación funcional no hay instrucciones de asignación.

- La evaluación de un programa funcional no tiene *efectos colaterales*. (Las funciones matemáticas entregan siempre el mismo valor para el mismo conjunto de argumentos). La evaluación de funciones está controlada por recursiones y condiciones, la imperativa lo hacen normalmente por secuencias e iteraciones)

- Las funciones pueden evaluarse en cualquier orden, por lo que en programación funcional no hay que preocuparse por el flujo de control.

→ Características de la Programación Funcional

Las características principales

- Definiciones de funciones matemáticas puras, sin estado interno ni efectos laterales.
- Valores inmutables.
- Uso profuso de la recursión en la definición de las funciones.
- Uso de listas como estructuras de datos fundamentales.
- Funciones como tipos de datos primitivos:
  - expresiones lambda
  - funciones de orden superior.

→ Características de la Programación Funcional

- Principio de la Programación Funcional "**Pura**"

  "*El valor de la expresión sólo depende del valor de sus sub-expresiones, si las tiene*" *

- No existen efectos colaterales :

  El valor de "a + b" **sólo** depende de "a" y "b".
- El término *funciones* es usado en su sentido matemático.
- No hay instrucciones: programación sin asignaciones
  - La Programación Funcional *impura* permite la "*sentencia de asignación*"

** "En matemáticas una función provee un mapeo entre objetos tomados de un conjunto de valores llamado dominio y objetos en otro conjunto llamado codominio o rango"*

Aplicaciones prácticas de la programación funcional

En la actualidad el paradigma funcional es un *paradigma de moda*, como se puede comprobar observando la cantidad de artículos, charlas y blogs en los que se habla de él, así como la cantidad de lenguajes que están aplicando sus conceptos.

- Andrés Marzal - Por qué deberías aprender programación funcional ya mismo (Charla en YouTube) http://www.decharlas.uji.es/es/programacion-funcional-introduccion-haskell)
- Lupo Montero - Introducción a la programación funcional en JavaScript (Blog)
- Mary Rose Cook - A practical introduction to functional programming (Blog)
- Ben Christensen - Functional Reactive Programming in the Netflix API (Charla en InfoQ)

Estructura de los programas

- El programa es una función compuesta de Funciones más simples

- Ejecución de una Función:
  1. Recibe los datos de entrada:
     - Argumentos o parámetros de las funciones
  2. Evalúa las expresiones
  3. Devuelve el resultado:
     - Valor calculado por la función

```
> (+ 1 2)
3
> (* 2 4 5)
40
> (- 1 1/4)
3/4
```

→ Características de la Programación Funcional

Tipos de lenguajes funcionales

- La mayoría son lenguajes interpretados
- Algunas versiones son lenguajes compilados

Gestión de la memoria

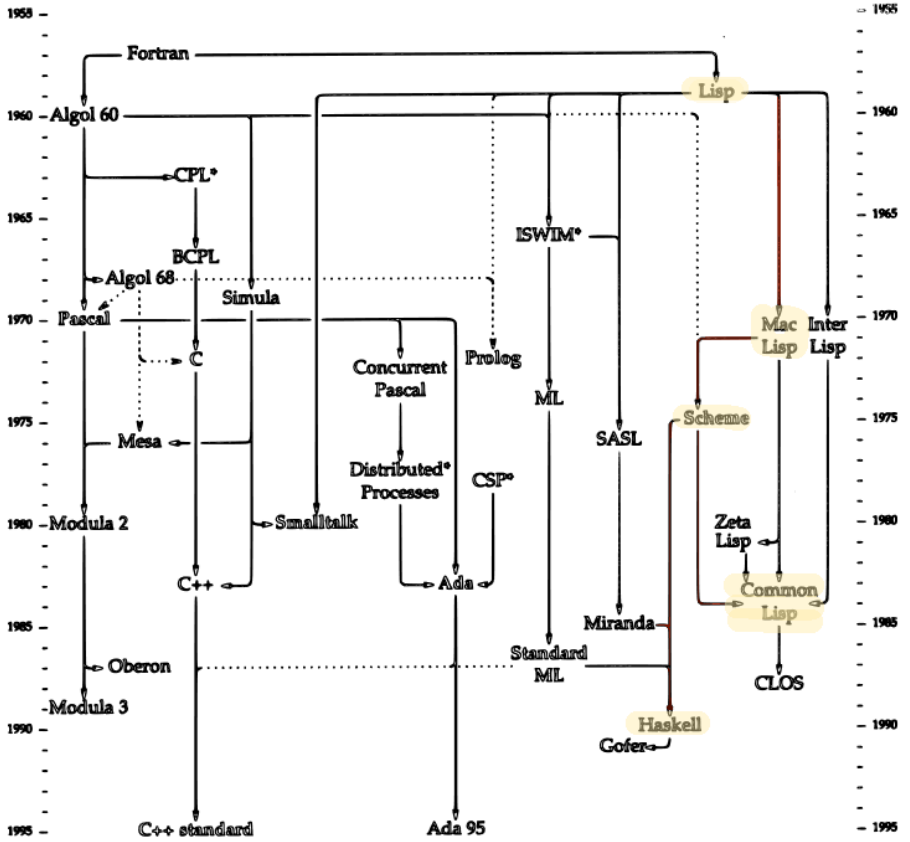
- Gestión implícita de la memoria:
  - La gestión de la memoria es una tarea del intérprete.
  - El programador no debe preocuparse por la gestión de la memoria.
- Recolección de basura: tarea del intérprete.

En resumen:

- El programador sólo se debe de preocupar por la descripción del problema

→ Reseña Histórica de Scheme

LISP → Scheme



→ Reseña Histórica de Scheme

- LISP
  - John McCarthy(MIT)
  - El programa “Advice Taker”:
    - Fundamentos teóricos: Lógica Matemática
    - Objetivo: Deducción e inferencias
  - LISP: **LIS**t **P**rocessing (1956 – 1958)
    - Segundo lenguaje histórico de Inteligencia Artificial (después de IPL)
    - En la actualidad, segundo lenguaje histórico en uso (después de Fortran)
    - LISP está basado en el Lambda Calculus (Alonzo Church)
  - **Scheme** es un dialecto de LISP

→ Reseña Histórica de Scheme

LISP - Características

- Recursión
- Listas
- Gestión implícita de la memoria
- Programas interactivos e interpretados
- Programación simbólica
- Reglas de ámbito dinámico para identificadores no locales
- Funciones “Built – in”
- Recolección de basura
- Lenguaje de definición formal

→ Reseña Histórica de Scheme

Ámbito léxico (o estático) y Dinámico

- Las reglas de ámbito determinan la declaración de identificadores no locales
  - Identificadores no locales: Variables, Funciones o Procedimientos que son usados en otra función o procedimiento donde no han sido declarados.
- Ámbito Léxico o Estático con "estructura de bloques"(Pascal, **Scheme**) o sin "estructura de bloques"(C, Fortran)
- Ámbito Dinámico, siempre con "estructura de bloques"(Lisp, SNOBOL, APL)

→ Reseña Histórica de Scheme

- **Scheme** es un lenguaje de programación funcional (si bien impuro, ya que, por ejemplo, sus estructuras de datos no son inmutables) y un dialecto de Lisp.

- **Scheme**, como todos los dialectos de Lisp, tiene una sintaxis muy reducida y facilita la programación funcional.

→ Scheme: Dialecto de Lisp

- **Expresiones simples**
  - Literales, llamadas a procedimientos y variables
  - Definiciones, programas y ciclo lee-evalúa-imprime

    (* 2 3) → 6
  - Evaluación condicional

    (if #t 1 2) → 1

- **Tipos de datos**
  - booleanos, números, caracteres, cadenas y símbolos
  - Pares o Parejas
  - Listas
  - Vectores

Tipos de datos – Booleanos

Un booleano es un valor de verdad, que puede ser verdadero o falso.

En ***Scheme***, tenemos los símbolos **#t** y **#f** para expresar verdadero y falso respectivamente, pero en muchas operaciones se considera que cualquier valor distinto de **#f** es verdadero.

Ejemplos:

```
#t
#f
(> 3 1.5)
(= 3 3.0)
(equal? 3 3.0)
(or (< 3 1.5) #t)
(and #t #t #f)
(not #f)
(not 3)
```

Tipos de datos – Números

La cantidad de tipos numéricos que soporta ***Scheme*** es grande, incluyendo enteros de diferente precisión, números racionales, complejos e inexactos.

Ejemplos:

```
(<= 2 3 3 4 5)
(max 3 5 10 1000)
(/ 22 4) ; Devuelve una fracción
(quotient 22 4)
(remainder 22 4)
(equal? 0.5 (/ 1 2))
(= 0.5 (/ 1 2))
(abs (* 3 -2))
(sin 2.2) ; relacionados: cos, tan, asin, acos, ata
-365
1/4
23.46
1.3e27
2.7-4.5i
```

Tipos de datos – Números

Primitivas que devuelven números inexactos

(floor x) devuelve el entero más grande no mayor que x

(floor -4.3) ⇒ -5.0

(floor 3.5) ⇒ 3.0

(ceiling x) devuelve el entero más pequeño no menor que x

(ceiling -4.3) ⇒ -4.0

(ceiling 3.5) ⇒ 4.0

(truncate x) devuelve el entero más cercano a x cuyo valor absoluto no es mayor que el valor absoluto de x

(truncate -4.3) ⇒ -4.0

(truncate 3.5) ⇒ 3.0

(round x) devuelve el entero más cercano a x, redondeado

(round -4.3) ⇒ -4.0

(round 3.5) ⇒ 4.0

Tipos de datos – Números

Operaciones sobre números

```
(number? 1)
(integer? 2.3)
(integer? 4.0)
(real? 1)
(positive? -4)
(negative? -4)
(zero? 0.2)
(even? 2)
(odd? 3)
```

Tipos de datos – Caracteres

Se soportan caracteres internacionales y se codifican en UTF–8.

#\a #\A #\Space #\ñ #\á

```
(char<? #\a #\b)
(char-numeric? \#1)
(char-alphabetic? #\3)
(char-whitespace? #\space)
(char-upper-case? #\A)
(char-lower-case? #\a)
(char-upcase #\ñ)
(char->integer #\space)
(integer->char 32)
(char->integer (integer->char 5000))
```

Tipos de datos – Cadenas

Las cadenas son secuencias finitas de caracteres.

"hola“ "La palabra \"hola\" tiene 4 letras"

- **Constructores de cadenas**
  (make-string 5 #\o) ⇒ "ooooo"
  (string #\h #\o #\l #\a) ⇒ "hola"

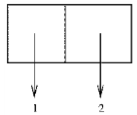
- **Operaciones con cadenas**
  (substring "Hola que tal" 2 4)
  (string? "hola")
  (string->list "hola")

- **Comparadores de cadenas**
  (string=? "Hola" "hola")
  (string=? "hola" "hola")
  (string<? "aab" "cde")

Tipos de datos – Parejas

Elemento fundamental de **Scheme**. Es un tipo compuesto formado por dos elementos (no necesariamente del mismo tipo).

```
(cons 1 2) ; cons crea una pareja
(cons 'a 3) ; elementos diferentes
(car (cons "hola" 2)) ; elemento izquierdo
(cdr (cons "bye" 5)) ; elemento derecho
```

→ 

Scheme es un lenguaje débilmente tipeado y las variables y parejas pueden contener cualquier tipo de dato. Incluso otras parejas

```
(define p1 (cons 1 2)) ; definimos una pareja formada por 1 y 2
(cons p1 3) ; definimos una pareja formada por la pareja (1 . 2) y 3
(cons (cons 1 2) 3) ; igual que la expresión anterior
```

Cuando evaluamos las expresiones muestra el resultado de construir la pareja con la sintaxis: (elemento izquierdo . elemento derecho) . Si la pareja está formada a su vez por otras parejas, el intérprete puede imprimir cosas que no son muy comprensibles a primera vista.

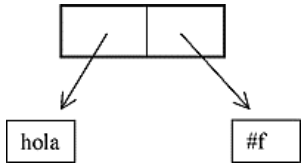
```
(cons 1 (cons 2 3)) ⇒ (1 2 . 3)
```

Tipos de datos – Parejas

Cuando evaluamos las expresiones muestra el resultado de construir la pareja con la sintaxis: *(elemento izquierdo . elemento derecho)* .

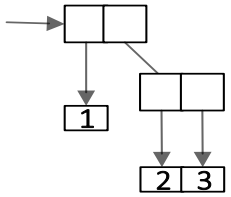
```
> (define c (cons 'hola #f))
> (car c)
'hola
> (cdr c)
#f
```

Esta estructura se puede representar con el siguiente diagrama:



Si la pareja está formada a su vez por otras parejas, el intérprete puede imprimir cosas que no son muy comprensibles a primera vista.

```
> (define p (cons 1 (cons 2 3))) ⇒ (1  2 . 3)
> (car p)
1
> (cdr p)
'(2 . 3)
```



Tipos de datos – Parejas

Ejemplo

```
(define (print-pareja pareja)
        (if (pair? pareja)
        (begin
                (display "(")
                (print-dato (car pareja))
                (display " . ")
                (print-dato (cdr pareja))
                (display ")")))))

(define (print-dato dato)
        (if (pair? dato)
           (print-pareja dato)
           (display dato)))

(define p2 (cons 1 (cons 2 3)))
(print-pareja p2) ⇒ (1. (2 . 3))
```

Tipos de datos – Listas

Es un tipo compuesto formado por un conjunto finito de elementos (no necesariamente del mismo tipo). Vamos a ver cómo definir, crear, recorrer y concatenar listas.

- (**list** 1 2 3 4) *;list crea una lista*
- '(1 2 3 4)*;otra forma de definir la lista*
- (**car** '(1 2 3 4)) *; primer elemento de la lista*
- (**cdr** '(1 2 3 4)) *; resto de la lista*
- **'()** *; lista vacía*
- (**cdr** '(1)) *; devuelve la lista vacía*
- (**null?** (cdr '(1))) *; comprueba si una lista es vacía*
- (**cons** 1 '(2 3 4 5)) *; nueva lista añadiendo un elemento a la cabeza*
- (**cons** 1 '()) *; lista con 1 elemento*
- (cons 1 (cons 2 '())) *; lista con 2 elementos*
- (**list** "hola" "que" "tal") *; lista de cadenas*
- (cons "hola" '(1 2 3 4)) *; lista de distintos tipos de datos*
- (**list** (**list** 1 2) 3 4 (**list** 5 6)) *; lista que contiene listas*
- (**cons** (**list** 1 2) '(3 4 5)) *; nueva lista añadiendo una lista*
- (**append** '(1) '(2 3 4) '(5 6)) *; nueva lista concatenando listas*
- (**list** (**cons** 1 2) (**cons** 3 4)) *; lista que contiene parejas*

Importante: las instrucciones **car** y **cdr** de las listas no son exactamente iguales que el **car** y **cdr** de las parejas. El car de las parejas devuelve el elemento izquierdo y el **car** de las listas devuelve el primer elemento de la lista. El **cdr** de las parejas devuelve el elemento derecho y el **cdr** de las listas devuelve una nueva lista sin el primer elemento.

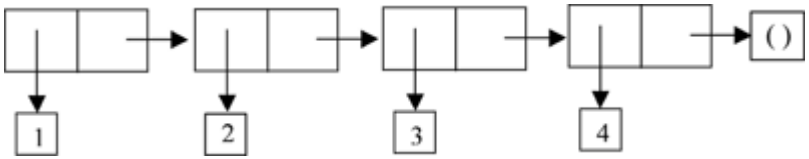
Tipos de datos – Listas

Una lista es:

- Una secuencia de parejas en las que el primer elemento es el dato y el segundo el resto de la lista.
- Un símbolo especial **'()** que denota la lista vacía

Por ejemplo, la lista se construye con la siguiente secuencia de parejas:

```
(define lista (cons 1 (cons 2 (cons 3 (cons 4 '())))))
> (car lista)
1
> (cdr lista)
'(2 3 4)
```



Esta estructura tiene el siguiente dibujo utilizando los diagramas caja y puntero

Tipos de datos – Listas

- **Funciones sobre listas**
  - La función **length**:
    > (**length** '(1 2 3 (4 5 6)))
    4
  - La función (**list-ref** *n lista*) devuelve el elemento enésimo de una lista (empezando a contar por 0):
    > (**define** lista '(1 2 3 4 5 6))
    > (**list-ref** lista 3)
    4
  - La función (**list-tail** *lista n*) devuelve la lista resultante de quitar n elementos de la lista original
    > (**list-tail** '(1 2 3 4 5 6 7) 2)
    '(3 4 5 6 7)

Tipos de datos – Listas

- **Funciones sobre listas**
  - La función **map** aplica una función a todos los elementos de una lista y devuelve la lista resultante

```
> (map abs '(-1 1 -3.4 -10))
'(1 1 3.4 10)

> (map (lambda (x) (* x x)) '(1 2 3 4 5))
'(1 4 9 16 25)
```

```
(define (mi-map proc list)
        (if (null? list)
            '()
            (cons (proc (car list))
            (map proc (cdr list)))))
```

  - La función **reverse** invierte una lista

```
>(reverse '(1 2 3 4 5 6))
(6 5 4 3 2 1)
```

```
(define (mi-reverse l)
        (if (null? l)
            '()
          (append (reverse (cdr l)) (list (car l)))))
```

→ Scheme: Dialecto de Lisp

- **Procedimientos**
  - **Expresiones lambda** (Permite crear nuevos procedimiento no necesariamente ligados a variables y probablemente anónimos)
  - **Procedimientos de primer orden** (En Lisp todos los valores son de primer-orden)
  - **Procedimientos con aridad variable** (La aridad es el número de argumentos que tiene un procedimiento)

```
(define plus (lambda x (if (null? (cdr x)) (lambda (y) (+ (car x) y))(apply + x))))
(plus 1 2) → 3
((plus 1) 2) → 3
```

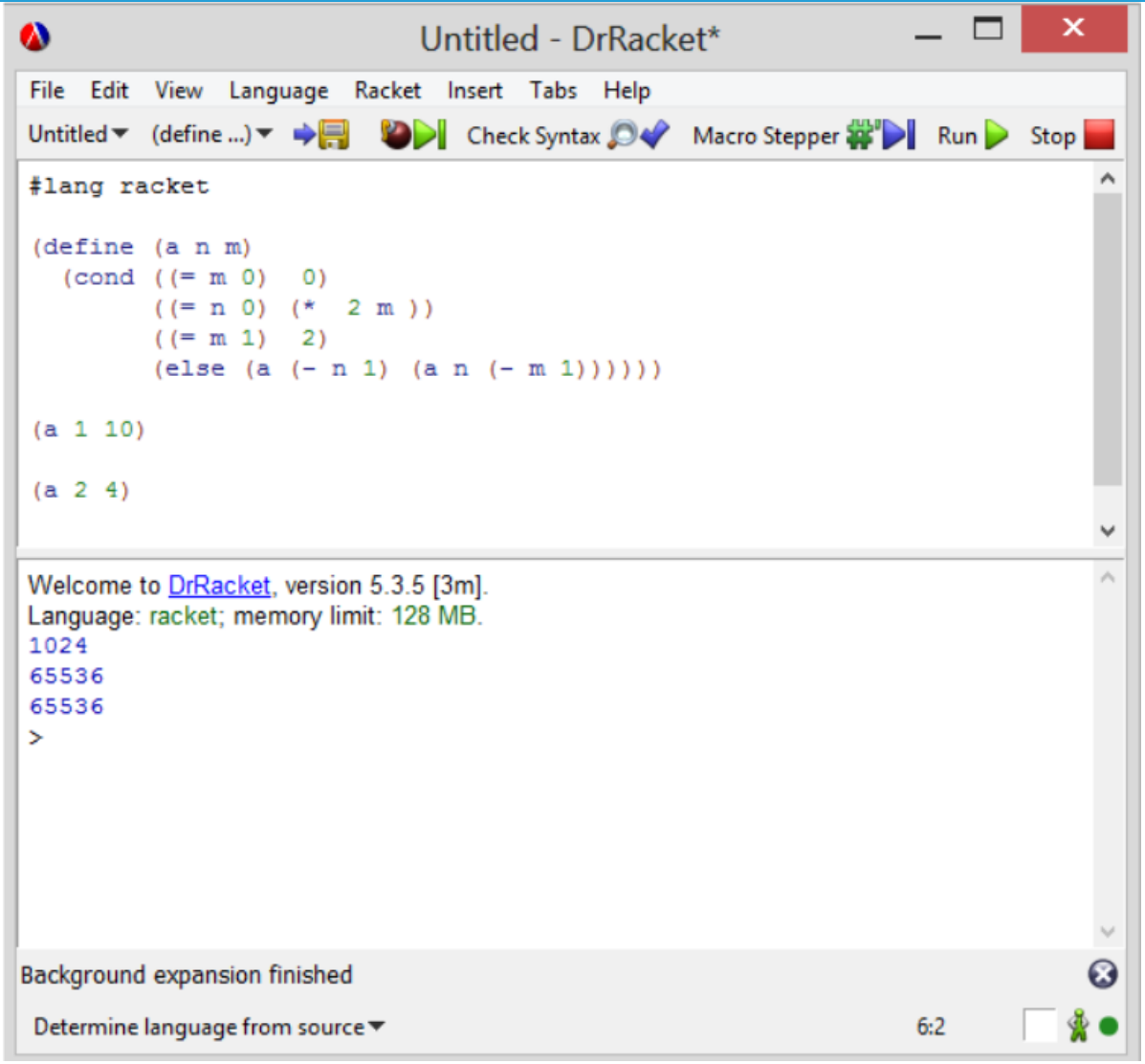
→ Scheme: Dialecto de Lisp

¿Dónde bajar Scheme?

(PLTScheme → **DrScheme**)

http://racket-lang.org/

DrRacket


Untitled - DrRacket*
File Edit View Language Racket Insert Tabs Help
Untitled (define ...) Check Syntax Macro Stepper Run Stop
#lang racket
(define (a n m)
(cond ((= m 0) 0)
((= n 0) (* 2 m))
((= m 1) 2)
(else (a (- n 1) (a n (- m 1))))))
(a 1 10)
(a 2 4)
Welcome to DrRacket, version 5.3.5 [3m].
Language: racket; memory limit: 128 MB.
1024
65536
65536
>
Background expansion finished
Determine language from source
6:2

→ DrRacket

Ejemplo:

```
> (+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
57
> (define tamaño 2)
tamaño
> tamaño
2
> (+ 3 tamaño)
5
> (* 2 tamaño)
4
```

Definición de Variables

> (define Pi 3.14159) → Pi = 3.14159

> (define radio 10) → radio = 10

> (define circunferencia (* 2 Pi radio))

> circunferencia

62.8318

Definición de Variables

```
> (define (cuadrado x) (* x x))

> (cuadrado 2)

4

> (define (circunferencia radio) (* Pi radio))

> (circunferencia 4)

12.56636
```

Sintaxis de las funciones

(define (<nombre> <parámetros formales>)
(cuerpo))

Ejemplo:

$$f(x, y) = x^2 + y^2$$

```
> (define (suma_cuadrados x y) (+ (cuadrado x) (cuadrado y)))
> (suma_cuadrados 2 3)
13
```

Composición de Funciones

$$f(a) = (a+1)^2 + (a-1)^2$$

```
> (define (f1 a)
      (suma_cuadrados (+ a 1) (- a 1)) )
>  (f1 5)
52
```

Composición de Funciones

El modelo de sustitución para evaluar funciones es:

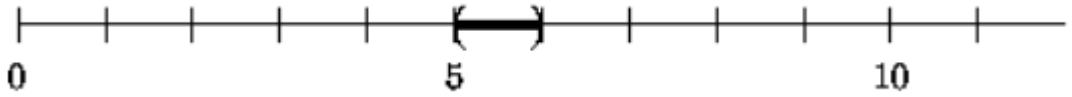
1) (f1 5)
2) (suma_cuadrados (+ 5 1) (- 5 1))
3) (+ (cuadrado (+ 5 1)) (cuadrado (- 5 1)))
4) (+ (* (+ 5 1) (+ 5 1)) (* (- 5 1) (- 5 1)))
5) (+ (* 6 6) (* 4 4))
6) (+ 36 16)
7) 52

Funciones Booleanas

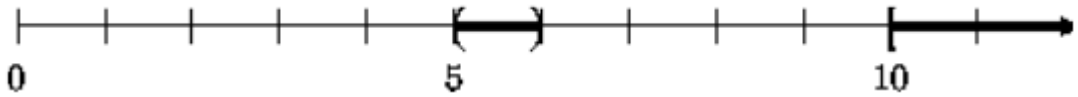
```
> (define (es-5? n)
          (= n 5))
> (es-5? 5)
#t
> (es-5? 7)
#f
```

Funciones Booleanas

```
> (define (esta-entre-5-6? n)
          (and (< 5 n) (< n 6)))
```




```
> (define (esta-entre-5-6-o-sobre-10? n)
          (or (esta-entre-5-6? n) (>= n 10)))
```



Sintaxis del *if*

(if <predicado> <consecuencia> <alternativa>)

Ejemplo

$$f(x)\begin{cases} x^2 \ si\ x \geq 2 \\ x+1\ si\ -2 < x < 2 \\ x/2 \ si\ x \leq 2 \end{cases}$$

```
> (define (f2 x)
    (if (>= x 2) (* x x)
            (if (and (< x 2) (> x -2)) (+ x 1)
                (/ x 2))))
> (f2 4)
16
```

Expresiones condicionales y predicados (1)

(**cond** (<p_1> <e_1>)
(<p_2> <e_2>)
.....
(<p_n> <e_n>)
(else <e_{n+1}>))

Ejemplo

$$f(x)\begin{cases} x^2+1 \ si\ x \geq 2 \\ 2 \ si\ x = 0 \\ x^3 \ si\ x < 0 \end{cases}$$

```
> (define (f3 x)
    (cond ((>= x 2) (+ (* x x) 1))
          ((= x 0) 2)
          (else (* x x x))))
> (f3 3)
10
```

Expresiones condicionales y predicados (2)

(cond [<p_1> <e_1>]
.....
[<p_n> <e_n>]
[**else** <e_{n+1}>])

Ejemplo

```
> (define (intereses cantidad)
    (cond [(<= cantidad 1000) 0.040]
          [(<= cantidad 5000) 0.045]
          [else 0.050]))
> (intereses 500)
0.04
> (intereses 3000)
0.045
> (intereses 10000)
0.05
```

Sentencias read y write - Evaluando un documento

```
(define (calcula-nota x y z)
        (/ (+ x y z) 3))
(define (paso? n)
        (>= n 50))
(define (nota)
        (write "Deme las notas")
        (newline)
        (calcula-nota (read) (read) (read)))
(define (resultado)
        (if (paso? (nota))
            (write "GANO")
            (write "AMPLIACION")))
```

Ejecución

```
> (resultado)
"Deme las notas"
90
80
70
"GANO“

> (resultado)
"Deme las notas"
10
20
30
"AMPLIACION"
```

Procesamiento Simbólico

- Un símbolo es una secuencia de caracteres precedida por un signo de comillas simple:

  Ejemplos: 'El 'perro 'comio 'un 'gato 'chocolate 'dos^3 'y%en%lo?

- Scheme tiene una sola operación básica de manipulación de símbolos: **symbol=?**
  - Es una operación de comparación, recibe dos símbolos y produce *true* si y sólo si los dos símbolos son idénticos: Ejemplos:
    - (symbol=? 'Hola 'Hola) => true
    - (symbol=? 'Hola 'Hello) => false
    - (symbol=? 'Hola x) => *true* si x almacena 'Hola
    - (symbol=? ‘Hola x) => *false* si x almacena 'Hello

Procesamiento Simbólico

- Los símbolos fueron introducidos por los investigadores en inteligencia artificial que querían diseñar programas que pudieran tener conversaciones con la gente.
- Por ejemplo la función **respuesta**, que responde con alguna observación a los diferentes saludos:

```
(define (respuesta s)
        (cond
                [(symbol=? s 'buenos-dias) 'Hola]
                [(symbol=? s 'como-esta?) 'bien]
                [(symbol=? s 'buenas-tardes) 'buenas]
                [(symbol=? s 'buenas-noches) 'estoy-cansado]))
```

Recursión

Un ***procedimiento recursivo*** es aquel se aplica a si mismo.

Ejemplo:

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

> (longitud '(a b c d))
4
```

Recursión y Recursión Lineal - *Ejemplo 1*

$$n! = fac(n) = \begin{cases} 1 \; si \; n = o \; \acute{o} \; n = 1 \\ n * fac(n-1) \; si \; n > 1 \end{cases}$$

Versión Recursiva

```
(define (factorial n)
    (if (or (= n 0) (= n 1))
        1
        (* n (factorial (- n 1)))))
```

Recursión y Recursión Lineal - *Ejemplo 1*

$$n! = fac(n) = \begin{cases} 1 \; si \; n = o \; ó \; n = 1 \\ n * fac(n-1) \; si \; n > 1 \end{cases}$$

*Versión Recursiva Lineal (Recursión Lineal)**

```
(define (factorial1  n)
        (fac-iter  1  1  n))
(define (fac-iter resultado i n)
        (if (> i n) resultado
        (fac-iter (* resultado i) (+ i  1) n)))
```

* En la **Recursión Lineal** cada llamada recursiva genera, como mucho, otra llamada recursiva.

Recursión y Recursión Lineal - *Ejemplo 1*

$$n! = fac(n) = \begin{cases} 1 \; si \; n = o \; \acute{o} \; n = 1 \\ n * fac(n-1) \; si \; n > 1 \end{cases}$$

Modelo de Sustitución - Versión Recursiva Lineal

1) (factorial 1 4)
2) (fac-iter 1 1 4)
3) (fac-iter 1 2 4)
4) (fac-iter 2 3 4)
5) (fac-iter 6 4 4)
6) (fac-iter 24 5 4)

Recursión y Recursión Lineal - *Ejemplo 2*

$$F_n = \begin{cases} 0 \; si \; n = 0 \\ 1 \; si \; n = 1 \\ F_{n-1} + F_{n-2} \; si \; n > 1 \end{cases}$$

Versión Recursiva

a)
```
(define (fib n)
    (cond   ((= n 0) 0)
            ((= n 1) 1)
            (else (+ (fib (- n 1)) (fib (- n 2))))))
```

b)
```
(define (fib1 n)
            (fib-iter1 0 1 0 n))
(define (fib-iter1 ant res i n)
            (if (>= i n)
                ant
                (fib-iter1 res (+ res ant) (+ i 1) n)))
```

Recursión y Recursión Lineal - *Ejemplo 2*

$$F_n = \begin{cases} 0 \; si \; n = 0 \\ 1 \; si \; n = 1 \\ F_{n-1} + F_{n-2} \; si \; n > 1 \end{cases}$$

Versión Recursiva Lineal (Recursión Lineal)

```
(define (fib3 n)
        (fib-iter3 1 0 n))

(define (fib-iter3 a b n)
        (if (= n 0)
            b
            (fib-iter3 (+ a b) a (- n 1))))
```

Funciones como Parámetro

Ejemplo 1 $\sum_{i=a}^{n} i = a + (a+1) + \ldots + n$

```
(define (serie1 a n)
        (if ( > a n)
            0
            (+ a (serie1 (+ a 1) n))))
```

Ejemplo 2 $\sum_{i=a}^{n} i^2$

```
define (serie2 a n)
        (if (> a n)
            0
            (+ (* a a) (serie2 (+ a 1) n))))
```

Funciones como Parámetro

Toda serie es de la forma:

$$\sum_{i=a}^{n} f(i)$$

```
!Programa general
(define (serie f a n)
(if (> a n)
    0
    (+ (f a) (serie f (+ a 1) n))
```

Funciones como Parámetro

Ejemplo:

```
(define (cuadrado a) (* a a))
(define (sum f a n)
      (if (> a n)
          0
          (+ (f a) (sum f (+ a 1) n))))
(define (serie2 a n)
          (sum cuadrado a n))
(define (termino i)
          (/ i (+ (cubo i) 1)))
(define (serie3 a n)
                (sum termino a n))
```

Funciones Lambda λ

Las funciones Lambda permiten definir funciones anónimas, con la siguiente sintaxis:

(lambda (<parámetros>) <cuerpo>)

Se puede abreviar como:

(define (var0 var1 … varn) e1 …)

Ejemplo:

```
(define cubo (lambda (x) (* x x x)))
>(cubo 3)
27
(define (cubo x) (* x x x))
>(cubo 3)
27
```

Funciones Lambda λ - ejemplo

```
(define (serie f a n) (tradicional)
        (if (> a n)
            0
            (+ (f a) (serie f (+ a 1) n))))

(define (serie4 a n) (función lambda)
        (serie (lambda (i) (/ i (+ (cubo i) 1))) a n))
```

Uso del let

Esta forma especial permite definir unas variables locales, darles un valor y evaluar una expresión con ese valor definido y tiene la siguiente sintaxis:

```
(let ((<ident-1> <exp-1>)
      ...
      (<ident-n> <exp-n>))
      <cuerpo>)
```

```
> (let ((x 4) (y 2)) (* x y))
8
```

La utilización de **let** permite hacer los programas más legibles, ya que podemos definir algunos conceptos previos necesarios antes de la definición del cuerpo de la función.

```
(define (area-hexagono lado)
        (let ((area-triangulo (lambda (base altura)
                                (/ (* base altura) 2)))
              (apotema (lambda (lado)
                               (* (sqrt 3) (/ lado 2))))))
        (* 6 (area-triangulo lado (apotema lado)))))
```

* Las funciones area-triangulo y apotema son locales a la definición de area-hexagono.

Uso del let*

La forma especial **let*** es una variante del **let** que *sí* permite la definición secuencial de valores a las variables locales.

Por ejemplo, la siguiente expresión devuelve un error

```
> (let ((a 0)
        (b (+ a 1))
        (c (+ c 1)))
   (list a b c))
   error: reference to undefined identifier: a
```

Pero podemos realizar la definición secuencial utilizando **let***:

```
> (let* ((a 0)
         (b (+ a 1))
         (c (+ c 1)))
> (list a b c))
 (0 1 2)
```

Uso del set!

let permite ligar un valor a una variable en su cuerpo (local), **define** permite ligar un valor a una variable de nivel superior (global). Sin embargo, **let** y **define** no permiten cambiar el ligado de una variable ya existente, como lo haría una asignación. Son constantes!

set! Permite en *Scheme* re-ligar a una variable existente un nuevo valor, como lo haría una asignación.

```
(define abcde '(a b c d e))
> abcde
'(a b c d e)

(set! abcde (cdr abcde))
> abcde
'(b c d e)
```

Uso del set!

Ejemplos

```
(define raices-cuadradas
      (lambda (a b c)
            (let ((menosb 0) (raiz 0)
                  (divisor 1) (raiz1 0) (raiz2 0))
            (set! menosb (- 0 b))
            (set! divisor (* 2 a))
            (set! raiz (sqrt (- (* b b) (* 4 (* a c)))))
            (set! raiz1 (/ (+ menosb raiz)
            divisor))
            (set! raiz2 (/ (- menosb raiz)
            divisor))
            (cons raiz1 raiz2)
            )
      )
)
>(raices-cuadradas 2 -4 -30)
 (5 . -3)
```

```
(define raices-cuadradas1
      (lambda (a b c)
            (let ( (menosb (- 0 b))
                  (raiz (sqrt (- (* b b) (* 4 (* a c)))))
                  (divisor (* 2 a))
            )
            (let ((raiz1 (/ (+ menosb raiz) divisor))
                  (raiz2 (/ (- menosb raiz) divisor)))
                  (cons raiz1 raiz2))
            )
      )
)

>(raices-cuadradas 2 -4 -30)
 (5 . -3)
```

Funciones de orden superior I

Llamamos funciones de orden superior (*higher order functions*) a las funciones que toman otras como parámetro o devuelven otra función, esto permiten generalizar soluciones con un alto grado de abstracción.

"Una función es de orden superior si toma una función como argumento o devuelve una función como resultado."

La función **map** es un ejemplo típico de función de orden superior.

(map funcion lista**)**

Devuelve una nueva lista resultante de aplicar la función a cada uno de los elementos de la lista inicial.

```
(define (suma x y)
        (+ x y))
(map suma '(1 2 3) '(4 5 6))
⇒ (5 7 9)
(map * '(1 2 3) '(4 5 6))
⇒ (4 10 18)
```

```
(define (suma-n lista n)
        (map (lambda (x) (+ x n)) lista))

(suma-n '(1 2 3 4) 10)
⇒ (11 12 13 14)
```

Funciones de orden superior II

El uso de funciones de orden superior permite realizar un código muy conciso y expresivo. Ya no es necesario escribir la recursión de forma explícita, sino que la función de orden superior es la que se encarga de abstraerla.

Por ejemplo

Supongamos que queremos definir la función (contienen-letra caracter lista) que devuelve las palabras de una lista que contienen un determinado carácter.

```
(contienen-letra #\a '("En" "un" "lugar" "de" "la" "Mancha"))
⇒ ("lugar" "la" "Mancha")
```

Funciones de orden superior III

Podemos implementar **contienen-letra** usando la función de orden superior **filter**, y pasando como función de filtrado una función auxiliar que creamos con la llamada a lambda y que comprueba si la palabra contiene el carácter:

```
(define (contienen-letra caracter lista-pal)
        (filter (lambda (pal) (letra-en-pal? caracter pal)) lista-pal))
```

Sólo nos falta implementar la función **letra-en-pal?** que comprueba si una palabra contiene un carácter.

Usando las funciones **primero** y **resto** que nos devuelven el primer carácter de una cadena y el resto de la cadena, la podemos implementar con una recursión sencilla:

```
(define (letra-en-pal? caracter palabra)
        (cond ((equal? "" palabra) #f)
              ((equal? caracter (primero palabra)) #t)
        (else (letra-en-pal? caracter (resto palabra)))))
```

Funciones que retornan funciones

Ejemplo: Escribiremos una función que calcula la derivada de una función, que será otra función (Método Newton).

$$\frac{df(x)}{dx} = f'(x) = \lim_{h \to 0} \frac{f(x-h) - f(x)}{h} \approx \frac{f(x-h) - f(x)}{h}$$

```
> (define (derivada f h)
            (lambda (x) (/ (- (f (+ x h)) (f x)) h)))

> (define cubo  (lambda (x) (* x x x)))  →  x³

> ((derivada cubo 0.0001) 5)
75.00001501625775
```

Entradas / Salidas de ficheros

- Para abrir un archivo para **lectura**, se utiliza la función
  (open-input-file <ruta> {#:mode { 'binary | 'text } }?)

  donde la opción por defecto para el parámetro #:mode es 'binary

- Para abrir un archivo para **escritura**, se usa la función
  (open-output-file <ruta> {#:mode { 'binary | 'text } }?
  {#:exists { 'error | 'append | 'update | 'can-update | 'replace | 'truncate | 'must-truncate } }?)

  donde el parámetro #:exits indica el comportamiento a seguir cuando el archivo indicado en <ruta> ya existe. Por defecto se toma 'error.

  Su signicado es:

  'error lanza una excepción cuando el archivo ya existe.
  'append si el archivo ya existe, el cursor se coloca al final del archivo.
  'update se coloca al final del archivo y genera una excepción si no existe.
  'can-update abre el archivo sin truncarlo (es decir, sin borrarlo), o lo crea si no existe.
  'replace borra el archivo, si existe, y crea uno nuevo.
  'truncate borra el contenido del archivo, si existe.
  'must-truncate borra el contenido de un archivo existente, y lanza una excepción si no existe.

Entradas / Salidas de ficheros

- Para abrir un archivo para **lectura y escritura** se usa la función

  (**open-input-output-file** <ruta> {#:mode { 'binary | 'text } }?
  {#:exists { 'error | 'append | 'update | 'replace | 'truncate } }?)

  donde el parámetro #:exits indica el comportamiento a seguir cuando el archivo indicado en <ruta> ya existe. Por defecto se toma 'error.

  Su signicado es:

  'error lanza una excepción cuando el archivo ya existe.

  'append si el archivo ya existe, el cursor se coloca al final del archivo.

  'update se coloca al final del archivo y genera una excepción si no existe.

  'replace borra el archivo, si existe, y crea uno nuevo.

  'truncate borra el contenido del archivo, si existe.

Entradas / Salidas de ficheros

```
#lang racket ; lee-lineas.rkt
(define ( mostrar-lineas nombre-archivo )
      (define ( aux flujo numero-de-linea )
            (let ([linea (read-line flujo )])
                  (if (eof-object? linea )
                        (close-input-port flujo )
                        (begin
                              (printf "~a: ~s\n" numero-de-linea linea )
                              (aux flujo ( add1 numero-de-linea ))
))))

(if (file-exists? nombre-archivo )
      (aux (open-input-file nombre-archivo ) 1)
      (error (string-append "No existe el archivo " nombre-archivo ))
))

(let ([parametros (current-command-line-arguments)])
      (if (not (= 1 (vector-length parametros )))
            (error "Se espera el nombre de un archivo como parámetro ")
            (mostrar-lineas (vector-ref parametros 0))
))
```

```
$ racket lee-lineas.rkt <nombre-archivo>
```

Entradas / Salidas de ficheros

Procedimientos que cambian los puertos de entrada o salida **with-input-to-file** y **with-output-to-file** puede asociar un fichero al puerto de salida actual.

Sintaxis (with-input-to-file f p - with-output-to-file f p)

- f : nombre de un fichero que se va a leer/crear
- p : procedimiento sin argumentos

Significado

1. Abre el fichero f y lo asocia al puerto de entrada o salida actual.
2. Llama al procedimiento p, que realiza sus operaciones de lectura o escritura en el fichero asociado al puerto de salida actual.

Entradas / Salidas de ficheros

Procedimientos que cambian los puertos de entrada o salida **with-input-to-file** y **with-output-to-file** puede asociar un fichero al puerto de salida actual.

```
(define (contar-palabras)
      (cond
            ((eof-object?(read))  0)
            (else                 (+ 1 (contar-palabras)))
      )
)
;; Llamada
(with-input-from-file "datos.txt" contar-palabras)
```

Todas las sentencias de lectura del contar-palabras van dirigidas al fichero "datos.txt"

```
(define (hanoi n a b c)
      (define (cambio a b)
            (display a)
            (display "→")
            (display b)
            (newline)
            1)
      (cond(     ( = n 1) (cambio a b))
                 (else  (    +
                             (hanoi (-n 1) a c b)
                             (cambio a b)
                             (hanoi (-n1) c b a))
)     )     )
;; Llamada a la función
(hanoi 3 "a" "b" "c")
(with-output-to-file "salida.txt" (lambda() (load "hanoi.rkt")))
```

Entradas / Salidas de ficheros

El procedimiento **with-input-to-file** y **with-output-to-file** puede asociar un fichero al puerto de salida actual

```
#lang racket   ;leer-escribir.rkt
(define file-content
    (with-input-from-file "input.txt"
        (lambda ()
            (let loop ((lst null))
                (define new (read-char))
                (if (eof-object? new)
                    (apply string lst)
                    (loop (append lst (list new)))))))))

(with-output-to-file "output.txt"
    (lambda ()
        (write file-content)))

$ racket leer-escribir.rkt
```

Esto no acaba aquí ….

Continuara!