Fundamentos de lenguajes de programación

Repaso Gramáticas y Dr Racket

Agosto de 2022

Contenido

Contenido

Gramáticas

El alfabeto

Un alfabeto es un conjunto finito no vacío cuyos elementos se llaman **símbolos**.

- Sea $\Sigma = \{a, b\}$ el alfabeto que consta de los símbolos a y b. Las siguientes son cadenas sobre Σ: *aba*, *abaabaaa*, *aaaab*.
- El *alfabeto binario* $\Sigma = \{0, 1\}$ son las cadenas sobre Σ que se definen como secuencias finitas de ceros y unos.
- Las cadenas son *secuencias ordenadas* y finitas de símbolos. Por ejemplo, $w = aaab \neq w_1 = baaa$.
- Sea $\Sigma = \{a, b, c, \ldots, x, y, z\}$ el alfabeto del idioma castellano.
- El alfabeto utilizado por muchos lenguajes de programación.
- Sea $\Sigma = \{a, b, c\}$ entonces podemos formar todas las cadenas sobre Σ incluyendo la cadena vacía.

Notación de alfabetos, cadenas y lenguajes

Notación usada en la teoría de lenguajes	
Σ, Γ	denotan alfabetos.
Σ^*	denota el conjunto de todas las cadenas que se pueden formar con los símbolos del alfabeto Σ.
$a, b, c, d, e, \ldots$	denotan símbolos de un alfabeto.
$u, v, w, x, y, z, \ldots$ $\alpha, \beta, \gamma, \ldots$	denotan cadenas, es decir, sucesiones finitas de símbolos de un alfabeto.
ϵ	denota la cadena vacía, es decir, la única cadena que no tiene símbolos.
$A, B, C, \ldots, L, M, N, \ldots$	denotan lenguajes (definidos más adelante).

- Si bien un alfabeto Σ es un conjunto finito, Σ^* es siempre un conjunto infinito (enumerable).
- Hay que distinguir entre los siguientes cuatro objetos, que son diferentes entre sí: $\emptyset, \epsilon, \{\emptyset\}, \{\epsilon\}$

Alfabetos

Operaciones con alfabetos

Si Σ es un alfabeto, $\sigma \in \Sigma$ denota que σ es un símbolo de Σ, por tanto, si

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

se puede decir que $0 \in \Sigma$

Un alfabeto es simplemente un conjunto finito no vacío que cumple las siguientes propiedades, Dados Σ_1 y Σ_2 alfabetos

- Entonces $\Sigma_1 \cup \Sigma_2$ también es un alfabeto.
- $\Sigma_1 \cap \Sigma_2, \Sigma_1 - \Sigma_2$ y $\Sigma_2 - \Sigma_1$ también son alfabetos.

Lenguajes

Lenguaje

Un *lenguaje* es un conjunto de palabras o cadenas. Un lenguaje L sobre un alfabeto Σ es un subconjunto de Σ^* y si $L = \Sigma^*$ es el lenguaje de todas las cadenas sobre Σ.

- Sea $L = \emptyset$ el lenguaje vacío
- $\emptyset \subseteq L \subseteq \Sigma^*$
- $\Sigma = \{a, b, c\}$. $L = \{a, aba, aca\}$
- $\Sigma = \{a, b, c\}$. $L = \{a, aa, aaa\} = \{a^n : n \geq 1\}$
- $\Sigma = \{a, b, c\}$. $L = \{w \in \Sigma^* : w$ no contiene el símbolo $c\}$. Por ejemplo, $abbaab \in L$ pero $abbcaa \notin L$.
- Sobre $\Sigma = \{0, 1, 2\}$ el lenguaje de las cadenas que tienen igual número de ceros, unos y dos's en cualquier orden.

Lenguajes

Potencia del lenguaje

Potencia del lenguaje Dado un lenguaje A sobre Σ y ($A \subseteq \Sigma^*$) y $n \in \mathbb{N}$, se define

$$A^n = \begin{cases} \{\epsilon\}, & \text{si } n = 0 \\ A \cdot A^{n-1}, & \text{si } n \geq 1 \end{cases}$$

Ejemplo. Sea $A = \{ab\}$ sobre un alfabeto $\Sigma = \{a, b\}$, entonces:
$A^0 = \{\epsilon\}$
$A^1 = A = \{\ ab\}$
$A^2 = A \cdot A^1 = \{\ abab\}$
$A^3 = A \cdot A^2 = \{\ ababab\}$

Cerradura de Kleene

Def. formal de Cerradura de Kleene

La cerradura de Kleene de un lenguaje $A \subseteq \Sigma^*$ es la unión de las potencias: se denota por A^*

$$A^* = \bigcup_{i \geq 0} A^i = A^0 \cup A^1 \cup A^2 \cup \ldots \cup A^n$$

- `Observación:` A^* se puede describir de la siguiente manera:

  $$A^* = \{u_1 u_2 \ldots u_n : u_i \in A, n \geq 0\}$$

  Es el conjunto de todas las concatenaciones de la cadena A, incluyendo ϵ
- la cerradura positiva se denota por A^+

$$A^+ = \bigcup_{i \geq 1} A^i = A^1 \cup A^2 \cup A^3 \cup \ldots \cup A^n$$

Definición formal de expresiones regulares

Las expresiones regulares sobre un alfabeto Σ se definen recursivamente como:

- $\emptyset$, ϵ y a, $a \in \Sigma$ son expresiones regulares.
- si A y B son expresiones regulares, también lo son:

$$A \cup B \text{ (Unión)}$$
$$A \cdot B \text{ (Concatenación)}$$
$$A^* \text{(Cerradura de Kleene)}$$

- Son expresiones regulares aab^*, ab^+, $(aaba^*)^+$
- Sea el conjunto $\{\epsilon, aa, aba, ab^2a, ab^3a, ab^4a, ...\}$ entonces $\{\epsilon\} \cup ab^*a$ es una expresión regular.
- Expresión regular de todas las cadenas impares sobre $\Sigma = \{a, b\}$

$$a(aa \cup ab \cup ba \cup bb)^* \cup b(aa \cup ab \cup ba \cup bb)^*$$

Ejercicios resueltos de expresiones regulares

$\Sigma = \{a, b\}$ Lenguaje de todas las palabras que tienen un número par de símbolos (palabras de longitud par)

$$(aa \cup ab \cup ba \cup bb)^*$$

$\Sigma = \{a, b\}$ Lenguaje de todas las palabras que tienen un número impar de símbolos (palabras de longitud impar)

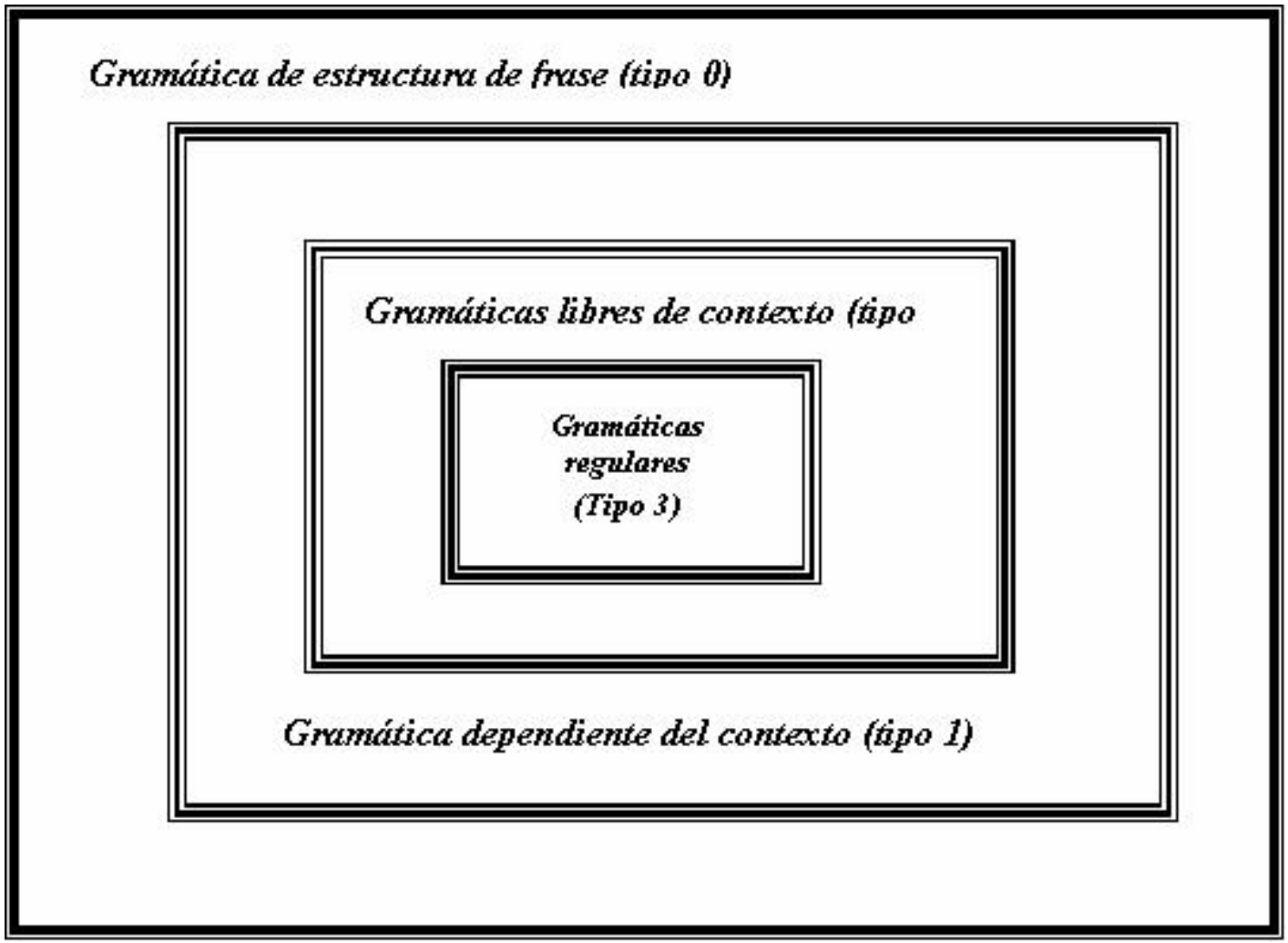
$$a(aa \cup ab \cup ba \cup bb)^* \cup b(aa \cup ab \cup ba \cup bb)^*$$

$\Sigma = \{a, b\}$ Lenguaje de todas las palabras que tienen un número par de a's.

$$b^*(ab^*a)^*b^*$$

Lenguajes y gramáticas

Según Chomsky los tipos de gramáticas se clasifican así:



Gramáticas

Gramáticas Regulares (Tipo 3)

Una gramática regular G es una 4-tupla $G = (N, \Sigma, S, P)$ que consiste de un conjunto N de no terminales, un alfabeto Σ, un símbolo inicial S y de un conjunto de producciones P. Las reglas son de la forma $A \rightarrow w$, donde $A \in N$ y w es una cadena sobre $\Sigma \cup N$ que satisface lo siguiente:

1. w contiene un no terminal como máximo.
2. Si w contiene un no terminal, entonces es el símbolo que está en el extremo derecho de w.
3. El conjunto de reglas P se define así:

$$P \subseteq N \times \Sigma^*(N \cup \epsilon) \quad o \quad P \subseteq N \times (N \cup \epsilon)\Sigma^*$$

Definición de gramática regular por la derecha

Gramáticas regulares

Sobre

$$G = (N, \Sigma, S, P)$$

Una gramática es regular por la derecha si sus producciones son de la forma:

$$\left(\left\{ \begin{array}{ll} A \longrightarrow wB, & w \in \sum^*, B \in N \\ A \longrightarrow \varepsilon & \end{array} \right\} \right)$$

Ejemplo Considere la siguiente gramática regular
$G = (N, \Sigma, S, P)$, que genera a^*, donde $\Sigma = \{a, b\}$, $N = \{S, A\}$
$P : S \to aA \mid \varepsilon$
$A \to aA \mid \varepsilon$

Definición de gramática regular por la derecha

Ejemplo. Sea la siguiente gramática regular $G = (N, \Sigma, S, P)$ que genera el lenguaje de la expresión regular $(a \cup b)^*$
$\Sigma = \{a, b\}$
$N = \{S, A\}$
$P : S \longrightarrow aS \mid bS \mid \varepsilon$

La forma de Backus-Naur

Forma de Backus-Naur

La forma de Backus-Naur se emplea para especificar reglas sintácticas de muchos lenguajes de programación y de lenguaje natural: En lugar de utilizar el símbolo $\longrightarrow$ usamos $::=$ y colocamos los símbolos no terminales entre $<>$.

La forma BNF se usa frecuentemente para especificar la sintaxis de lenguajes de programación, como Java y LISP; lenguajes de bases de datos, como SQL, y lenguajes de marcado como XML.

La forma de Backus-Naur

Sea la siguiente gramática:
$A \longrightarrow Aa \mid a \mid AB$
La forma Backus-Naur es:

BNF gramática

$< A >::=< A > a \mid a \mid < A >< B >$

La forma de Backus-Naur

La producción de enteros son signo en notación decimal. (Un **entero con signo** es un natural precedido por un signo más o un signo menos). La forma Backus-Naur para la gramática que produce los enteros con signo es:

BNF entero con signo

$$< \texttt{entero con signo} > ::= < \texttt{signo} >< \texttt{entero} >$$
$$< \texttt{signo} > ::= + \mid -$$
$$< \texttt{entero} > ::= < \texttt{dígito} > \mid < \texttt{dígito} >< \texttt{entero} >$$
$$< \texttt{dígito} > ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

La forma de Backus-Naur

Ejemplos

Una definición de listas de números

```
<lista-de-numeros> ::= ()  |  <numero> <lista-de-numeros>
```

Una definición de árbol binario:

```
<arbol-binario> ::= <numero>
<arbol-binario> ::= <simbolo> <arbol-binario>
                              <arbol-binario>
```

La forma de Backus-Naur

Ejemplo real

```
Programs

    <compilation unit> ::= <package declaration> <import declarations>
                                                    <type declarations>

Declarations

    <package declaration> ::= package <package name> ;

    <import declarations> ::= <import declaration> |
                              <import declarations> <import declaration>

    <import declaration> ::= <single type import declaration> |
                             <type import on demand declaration>

    <single type import declaration> ::= import <type name> ;
```

La forma de Backus-Naur

Ejemplo real

- JAVA `https://users-cs.au.dk/~amoeller/RegAut/JavaBNF.html`
- C++ 98 `http://www.externsoft.ch/download/cpp-iso.html#syntax`
- Python `https://docs.python.org/3/reference/grammar.html`

Contenido

Notas sobre Dr Racket

- ¡Lo puedes instalar en Windows, Linux y Mac OS !
- Se recomienda trabajar con la versión 7.9, algunas versiones como la 5.6 y 6.2 tienen problema con la librería **SLLGEN**
- Las expresiones en Dr Racket son en notación prefija, por ejemplo (+ 5 2) es equivalente 5+2 en notación infija.
- Se recomienda utilizar paréntesis para evitar problemas en la interpretación de resultados por ejemplo (+ (- 3 2) 4) es equivalente en notación infija a ((3 - 2) + 4)
- Para el curso de Fundamentos de Lenguajes de programación debe seleccionar **The Racket Lenguaje** y agregar en la primera linea:

```
#lang eopl
```

Notas sobre Dr Racket

- Se utiliza la palabra reservada **define** para la definición de variables por ejemplo

```
(define numeroA 5)
(define numeroB (* 2 numeroA))
```

- Se definen las funciones de la siguiente forma:

```
(define (nombreFuncion argumentosEntrada) <operaciones
    >)
```

Ejemplo:

```
(define (multiplique a b) (* a (* b b)))
```

¿Que debería retornar (multiplique 3 9)?.

Notas sobre Dr Racket

Así mismo, se pueden definir funciones anónimas con la expresión **lambda**. Ejemplo:

```
(lambda (x y) (if (> x y) (* x y) (- x y)))
```

Este retorna un valor función o procedimiento, estos podemos evaluarlos así:

```
( (lambda (x y) (if (> x y) (* x y) (- x y))) 3 4)
```

Observe que la función espera 2 argumentos.

Notas sobre Dr Racket

Importante

Debido a que en el paradigma funcional no hay diferencia entre funciones y valores, usaremos lambda para representar las funciones como valores.

Para el ejemplo anterior:

```
(define multiplique
  (lambda (a b)
          (* a (* b b))
   )
)
```

Notas sobre Dr Racket

Ejercicio

Escriba en el Dr Racket utilizando notación prefija:

- $2 * 2 + 3 * 5 + (\frac{1}{4})^2$ **Respuesta:** 19.0625
- $2 * (1 + 3^2 + \frac{4}{4}) + 3 * (5 - 3) + \frac{12}{4} - 3 + 4 * 5^3$ **Respuesta:** 528

Diseñe una función anónima que reciba dos valores. Si ambos valores son positivo o ambos son negativos retorna su multiplicación, en caso contrario retorna su suma.

Notas sobre Dr Racket

Ejercicio 1

$2 * 2 + 3 * 5 + (\frac{1}{4})^2$

```
(+ (* 2 2) (* 3 5) (expt (/ 1 4) 2))
```

Respuesta: 19.0625

Notas sobre Dr Racket

Ejercicio 2

$$2 * (1 + 3^2 + \frac{4}{4}) + 3 * (5 - 3) + \frac{12}{4} - 3 + 4 * 5^3$$

```
(+ (* 2 (+ 1 (expt 3 2) (/ 4 4))) (* 3 (- 5 3)) (/ 12 4) -3
    (* 4 (expt 5 3)))
```

Respuesta: 528

Notas sobre Dr Racket

Ejercicio 3

```
(lambda (x y) (if (>= (* x y) 0) (* x y) (+ x y)))
```

Probemos

```
( (lambda (x y) (if (>= (* x y) 0) (* x y) (+ x y))) 2 3)
( (lambda (x y) (if (>= (* x y) 0) (* x y) (+ x y))) 2 -3)
```

Contenido

Condicionales

En Racket tenemos el condicional que usualmente trabajamos en diferentes lenguajes

```
(if pregunta hacer_si_verdad hacer_si_falso)
```

Ejemplo:

```
(define funcionMisteriosa
  (lambda (n l)
    (if (< n l) "n es menor que l" "n es mayor o igual que l
        ")
  )
```

Condicionales

También, contamos con condicionales que permiten verificar varias condiciones sucesivamente. Sólo se da respuesta a la primera que sea verdadera, si ninguna lo es se responderá lo indicando el else.

```
(cond
        [Pregunta Respuesta]
        [Pregunta Respuesta]
        ...
        [else Respuesta]
)
```

Condicionales

Por ejemplo, una función que recibe un número y verifica si es par

```
(define verificarpar
  (lambda (n)
          (cond
                  [(even? n) "Par"]
                  [(< n 0) "Impar y menor que 0"]
                  [else "Impar y mayor o igual que 0"]
          )
  )
)
```

Condicionales

Diseñe una función que recibe un número y un símbolo

1. Si el número es menor o igual que 0 retorna el símbolo 'error
2. Si no, Si el número es mayor o igual 0 retorna:
   - Si el símbolo es 'm, retorna 'hombre
   - Si el símbolo es 'f, retorna 'mujer
   - En otro caso retorna 'error

Contenido

Definiciones locales

En algunos casos requerimientos realizar definiciones dentro de funciones, en Fundamentos de programación contábamos con la sentencia **local**, pero para este curso usaremos dos

- **let** para definiciones no recursivas. Dentro de este las funciones no se conocen a si mismas

```
(let (
        ;Zona de declaraciones
        ( .. ) ;Declaración 1
        ( .. ) ;Declaración 2
      )
      ;Zona de expresiones
)
```

- **letrec** para definiciones recursivas. En este las funciones se conocen a sí mismas. Su sintaxis es igual a la de let.

Definiciones locales

```
(define funcion
        (lambda (n)
                (let
                   (
                         (p 2)
                         (f (lambda (a b) (* a b)))
                   )
                   (f n p)
                 )
        )
)
(funcion 10)
```

Definiciones locales

```
(define funcion
        (lambda (n)
                (letrec
                   (
                         (p 2)
                         (sume
                              (lambda (a b)
                               (if (= b 0) 0 (+ a (sume a ( -b 1))))
                              )
                          )
                   )
                   (sume n p)
                 )
        )
)
(funcion 10)
```

Si utilizas let en lugar de letrec, va indicar que no conoce la función suma.

Definiciones locales

Aveces requerimos que un valor declarado conozca los anteriores y que las funciones no se conozcan a si mismas, para esto usaremos let*.

```
(define funcion
        (lambda (n)
                (let*
                   (
                         (p 2)
                         (q 3)
                         (r (+ p q n))
                   )
                   r
                 )
        )
)
(funcion 10)
```

Este código es funcional cambiando let* por letrec, la diferencia radica en las funciones recursivas.

Contenido

Listas

Las listas son una estructura recursiva, la cual consta de dos partes:

1. Un valor (cualquiera representado por scheme)
2. Una lista: La cual puede ser vacía.

```
(cons 1 (cons 2 (cons 3 empty)))
```

Dr Racket, provee la función list, la cual permite construir listas sin pensar en la recursividad. A esto se le conoce como azúcar sintáctico.

```
(list 1 2 3)
```

Listas

Para acceder a los elementos se tiene. car accede al primer elemento y cdr al resto, el cual es una lista.

```
(define l (cons 1 (cons 2 (cons 3 empty))))
(car l)
(cdr l)
```

Si observa, cdr devuelve una lista. Para acceder al segundo elemento tenemos cadr (primero del resto). Así mismo se tienen funciones similares.

Listas

A lo largo del curso vamos a utilizar una representación de listas especial:

```
'()
```

Esto es una lista vacía, podemos hacer lo siguiente:

```
'(perro gato 1 2 3)
```

Observe que perro y gato no son variables si no que se convierten en símbolos, también puede representar listas dentro de listas así:

```
'( (perro 1 2 3) (3 2 4) gato)
```

Observe que las listas sólo se declaran entre paréntesis. Esta representación permite agilizar el diseño de ciertas funciones.

Ejercicio

Enunciado

Diseñe una función que almacene los factoriales (en una lista) desde 0! hasta un valor n ingresado por el usuario.
Para esto necesitas usar locales (recursivos) ya que se requiere una función que genere una lista entre 0! y n!, la función a diseñar sólo recibe como argumento n.

Ejercicio

```
;Calcula el factorial de un numero
(define factorial
 (lambda (n)
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1)))]
    )
  )
)
;Genera una lista de factoriales desde 0! hasta n!
(define lista-factorial
  (lambda (n)
        (letrec
                (
                   (lista-factorial-aux
                     (lambda (n acc)
                        (if (= acc n)
                          (factorial n)
                                  (cons (factorial acc) (
                                       lista-factorial-aux n (+ 1 acc)))

                                  )
                          )
                     )
                )
                (lista-factorial-aux n 0)
        )
  )
)
(lista-factorial 10)
```

Contenido

Funciones como ciudadanos de primera clase

Ejemplo

¡Las funciones pueden ingresar cómo parámetros!

```
(define predicado
  (lambda (a b)
          (cond
                  [(< a b) #T]
                  [(even? a) #T]
                  [else #F]
          )
  )
)

(define filtro
  (lambda (lst num f)
          (cond
                  [(eqv? lst '()) empty]
                  [(f (car lst) num) (cons (car lst) (filtro (cdr lst) num
                        f))]
                  [else (filtro (cdr lst) num f)]
          )
  )
)

(filtro (list 1 2 3 4 5) 3 predicado)
```

¿Que hace esta función?

Funciones como ciudadanos de primera clase

Ejemplo

¡Pueden retornarse funciones!

```
(define funcionMisterio
    (lambda (a b)
        (lambda (x y) (+ x y a b))
    )
)
;Probar
(funcionMisterio 1 2)
( (funcionMisterio 1 2) 3 4)
```

Esto significa que no hay diferencia entre los valores y funciones.
Son exactamente lo mismo

Funciones como ciudadanos de primera clase

1. Diseñe una función, que reciba un número a y una función f. La función f recibe un número y retorna un booleano. Se hace el llamado **(f a)** y si el resultado es verdadero se retorna "ok", en otro caso "falso".
2. Diseñe una función que reciba dos números a y b y retorna una función t la cual espera un argumento numérico s. t evalúa si a es mayor que b si es así retorna $2 * s$ en otro caso $-2 * s$

Contenido

Asignación y secuenciación

Hasta ahora hemos trabajando en un paradigma declarativo funcional en el cual las variables están ligadas a un valor. Ahora, para trabajar en un paradigma imperativo debemos incluir la noción de estado con la instrucción set!, lo que permite a lo largo del código las variables puedan cambiar un valor. Este **estado** caracteriza la ejecución de un programa. Ejemplo:

```
(define valor 1)
(set! valor 2)
```

En este punto el estado de valor ha cambiado de 1 a 2 durante la ejecución.

Asignación y secuenciación

Otra característica de un paradigma imperativo es la secuenciación, es decir la ejecución de más de una instrucción en un paso dado, ya que el declarativo sólo permite uno. Para esto contamos con begin, este tiene la siguiente sintaxis:

```
(begin
    ;expresion
    ;expresion
    ...
    ;expresion
)
```

Este retorna la última expresión ejecutada, la ventaja es que permite ejecutar más de una expresión en un paso.

Asignación y secuenciación

Otra característica de un paradigma imperativo es la secuenciación, es decir la ejecución de más de una instrucción en un paso dado, ya que el declarativo sólo permite uno. Para esto contamos con begin, este tiene la siguiente sintaxis:

```
(define funcion
    (lambda (a b)
      (begin
        (set! a (* a b))
        (set! b (- a b))
        (+ a b)
       )
     )
)
(funcion 2 3)
```

Observe que se retorna la ultima expresión ejecutada.

Próxima sesión

- Relación entre inducción y programación (Capitulo 1 EOPL).