Más sobre funciones con listas y árboles

Fundamentos de programación

EISC-Facultad de Ingeniería
Universidad del Valle
Ángela Villota Gómez
avillota@eisc.univalle.edu.co

Contenido

- Funciones que tienen datos complejos en la entrada.
- Procesando datos complejos de forma simultanea:
  - Caso 1 y 2.
  - Caso 3.
- Árboles

Funciones que tienen datos complejos en la entrada

Existen tres tipos de funciones que podemos enfrentar:

1. Funciones en las que podemos trabajar con las entradas como si fueran datos simples (pero no lo son)
2. Funciones con entradas complejas pero el programador tiene información sobre el contenido y estructura.
3. Funciones en las que las entradas son complejas pero no tenemos información de ellas.

Funciones que tienen datos complejos en la entrada

- Los dos primeros tipos de funciones pueden ser resueltas usando la estrategia de diseño, tal como está.
- Para el tercero se sugiere crear una tabla en la cual se revisen las posibles entradas. Ejemplo:

	alos		
		(empty? alos)	(cons? alos)
n	(= n 1)		
	(> n 1)		

Para una función con entradas: n (natural) y alos (una lista)

Procesando dos listas: caso1 y 2

- Teniendo en cuenta el siguiente contrato, propósito y encabezado :

```
;;remplazar-eol-por: lista lista -> lista
;; construir una lista reemplazando empty
  en la primera lista por la lista 2
(define (remplazar-eol-por lista1 lista2)
  ...)
```

- Según el contrato, vamos a trabajar con dos listas de las cuales no tenemos información.

Procesando dos listas: caso1 y 2

- Veamos los ejemplos:
  - Si la primera entrada es **empty** entonces la función debe retornar la segunda lista:
    - (remplazar-eol-por empty L) = L
  2. En el caso contrario:
     - (remplazar-eol-por (cons 1 empty) L)

       ;; debe retornar: (cons 1 L)
     - (remplazar-eol-por (cons 2 (cons 1 empty)) L)

       ;; debe retornar: (cons 2 (cons 1 L))
     - (remplazar-eol-por (cons 2 (cons 11 (cons 1 empty))) L)

       ;; debe retornar: (cons 2 (cons 11 (cons 1 L)))

Procesando dos listas: caso1 y 2

- Teniendo en cuenta los ejemplos y el análisis de datos de la diapositiva anterior el cuerpo de la función queda asi:

```
(define (remplazar-eol-por lista1 lista2)
  (cond
    [(empty? lista1) lista2]
    [else
   (cons (first lista1)
   (remplazar-eol-por (rest lista1) lista2))
   ]))
```

Procesando dos listas: caso 3

- Veamos la función:

  ;; elemento-n? : lista N[>= 1] -> elemento

  ;; determinar el n-esimo elemento de una lista, o un error.

  (define (elemento-n? lista n) ...)

- El siguiente paso es proponer ejemplos:
  - (elemento-n? empty 1) expected behavior: (error 'elemento-n? "...")
  - (elemento-n? (cons 'a empty) 1) expected value: 'a
  - (elemento-n? empty 3) expected behavior: (error 'elemento-n? "...“)
  - (elemento-n? (cons 'a empty) 3) expected behavior:
  - (error 'elemento-n? "...")

Procesando dos listas: caso 3

- Los ejemplos no hacen claras las condiciones, entonces usamos la tabla de entradas:

	(empty? alos)	(cons? alos)
(= n 1)	(and (= n 1) (empty? alos))	(and (= n 1) (cons? alos))
(> n 1)	(and (> n 1) (empty? alos))	(and (> n 1) (cons? alos))

- Escribimos las condiciones usando un **and** porque es necesario tener en cuenta ambas posibilidades.

Procesando listas: caso 3

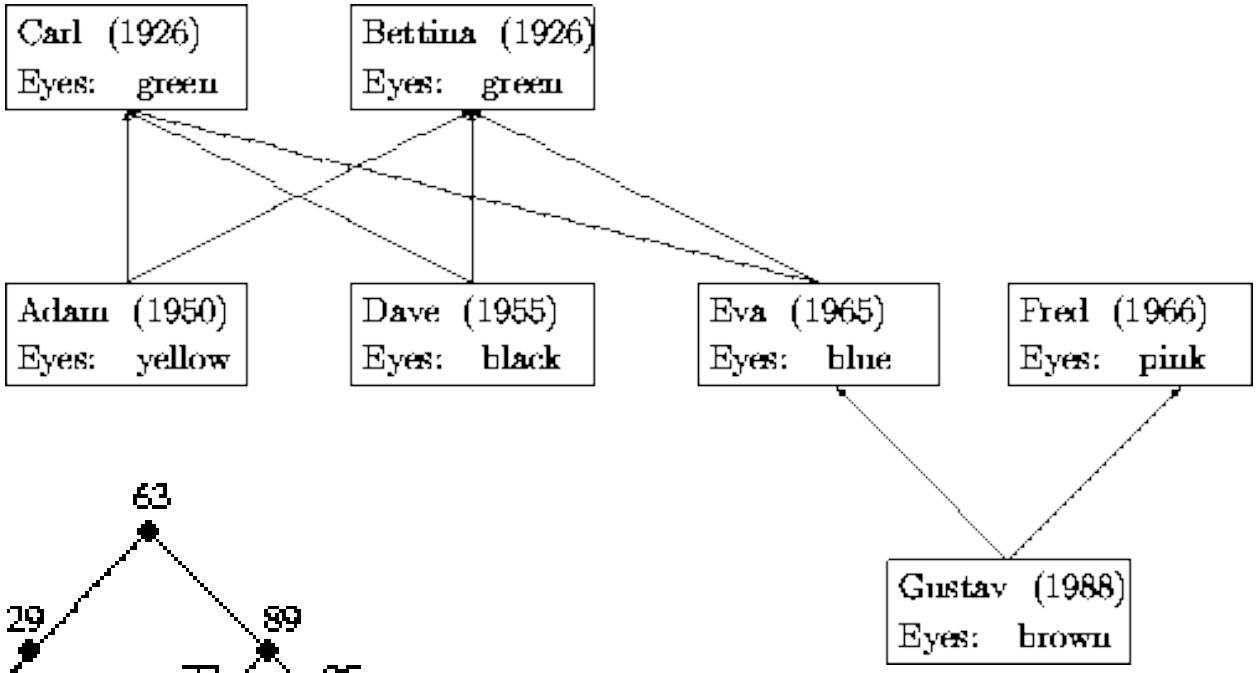
- El cuerpo del programa tiene la siguiente forma:

```
(define (elemento-n? alos n)
  (cond
    [(and (= n 1) (empty? alos)) ...]
    [(and (> n 1) (empty? alos)) ...]
    [(and (= n 1) (cons? alos)) ...]
    [(and (> n 1) (cons? alos)) ...]))
```

- Ejercicio: terminar la función.

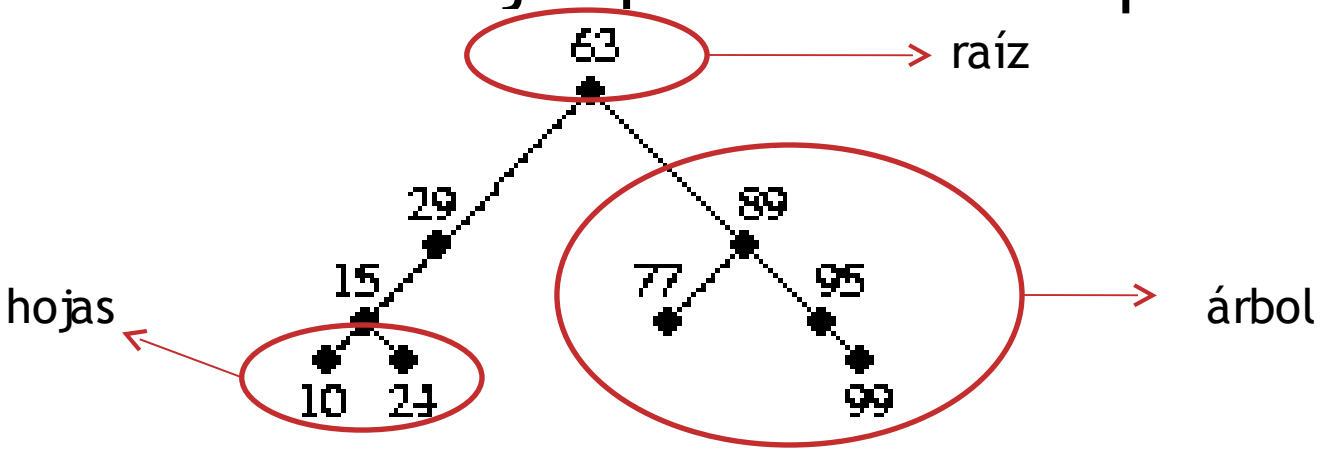
Árboles

- Un árbol es un tipo de dato recursivo, con una representación gráfica. Ejemplos:



Nodos

- Los elementos de un árbol se llaman nodos, hay dos tipos de nodos:
  1. Hojas: no tienen hijos
  2. Raíz: tienen hijos pero no tienen padres



- Los hijos de un nodo son arboles también

Representación de árboles

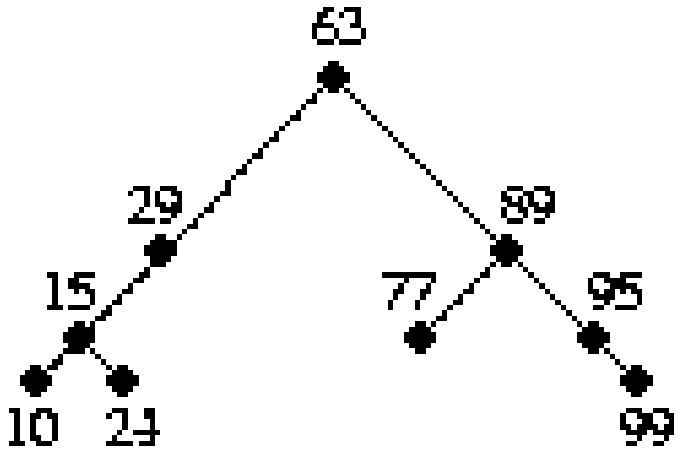
- Al representar árboles debemos primero pensar en cómo representar un nodo y qué datos almacena un nodo.
- En los ejemplos anteriores, hay un árbol que almacena números y otro que tiene símbolos.
- Los nodos de la clase de hoy los implementamos con estructuras, pero también podríamos usar listas.

Representación: Ejemplos

- Este es un árbol binario, porque cada nodo tiene solo dos hijos.
- Representación en scheme:
- Nodo:

```
define-struct node (ssn name
   left right))
```

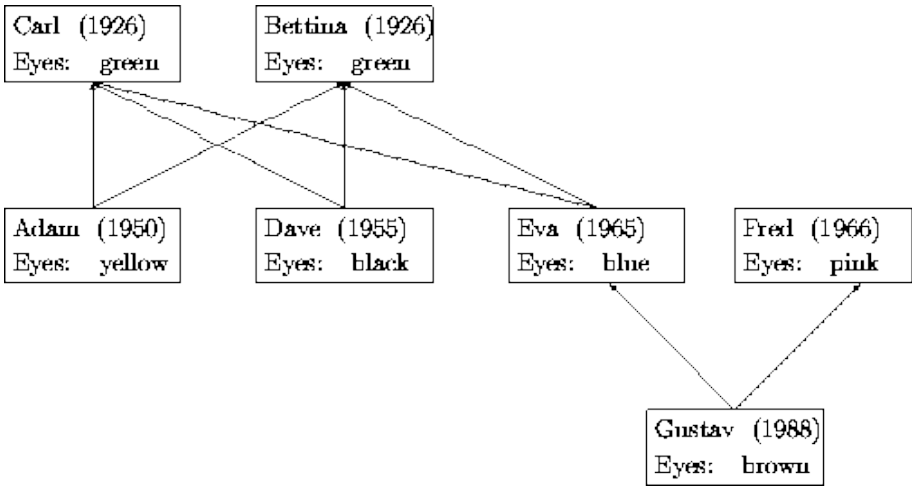
- Arbol:
  1. false; o
  2. (make-node soc lft rgt)
- soc es un número y lft y rgt son árboles



- Para definir un árbol es necesario:
  1. definir un nodo
  2. hacer la definición recursiva

Representación: Ejemplos

- Este es un árbol de ancestros.
- Nodo:
  - (define-struct child (father mother name date eyes))
- Árbol: un árbol de ancestros es:
  1. empty; o
  2. (make-child f m na da ec)
- F y m son arboles, na y ec son símbolos y da es un número



Representación: Ejemplos

```
;; Oldest Generation:
(define Carl (make-child empty empty 'Carl 1926 'green))
(define Bettina (make-child empty empty 'Bettina 1926 'green))

;; Middle Generation:
(define Adam (make-child Carl Bettina 'Adam 1950 'yellow))
(define Dave (make-child Carl Bettina 'Dave 1955 'black))
(define Eva (make-child Carl Bettina 'Eva 1965 'blue))
(define Fred (make-child empty empty 'Fred 1966 'pink))

;; Youngest Generation:
(define Gustav (make-child Fred Eva 'Gustav 1988 'brown))
```

Ejercicio

- Escribir en scheme el árbol binario de los ejemplos anteriores.
- Diseñe la función cuantos? Que tiene como entrada un árbol binario y retorna el número de nodos que tiene dicho árbol.
- Diseñe la función es-padre? Que toma como entrada dos nodos n1 y n2 y retorna true, en caso en que n1 sea el papá o la mamá de n2.
- Diseñe la función ancestros que dado un árbol de ancestros y un nodo, retorna la lista de nombres de los ancestros de dicho nodo.

Ejercicio

- Diseñe la función hermano? que tiene como entrada un par de nodos y retorna true si tienen el mismo padre, o la misma madre, false en el caso contrario.
- Diseñe la función hijos que retorna la lista de nombres de los hijos de un nodo. La entrada es un nodo.
- Con la ayuda de las dos funciones anteriores, diseñe la función primos, que dado un árbol de ancestros retorne la lista de nombres de los primos de un nodo.