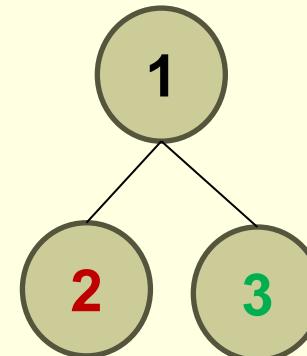


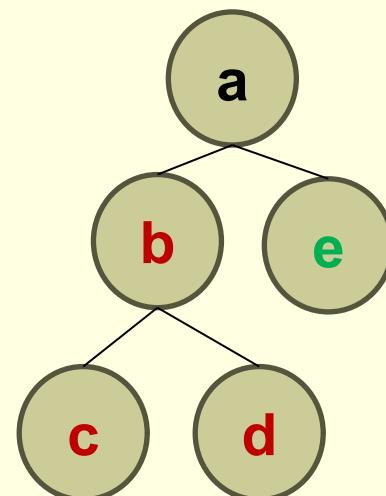
Algunos ejercicios de árboles

- Representación mediante lista (raiz,izq,der)
 - Un nodo hoja tiene como subárboles Izquierdo y derecho la lista vacía

```
(define arbol1 '(1  
    (2 () ())  
    (3 () ()))))
```



```
(define arbol2 '(a  
    (b  
        (c () ())  
        (d () ()))  
    (e () ()))  
))
```



Algunos ejercicios de árboles

1. Definir las funciones:
 - (raiz A) retorna el elemento que está en la raíz del árbol A
 - (hijo-izq A) retorna el subárbol izquierdo del árbol A
 - (hijo-der A) retorna el subárbol derecho del árbol A
2. Definir las funciones para recorrer un árbol en preorden, inorder y posorden. Cada función debe retornar una lista con los nodos que se visitan (según el tipo de recorrido).
3. Dados un árbol y un elemento cualquiera, determinar si el elemento pertenece al árbol.
4. Un árbol binario de nivel n es *completo*, si “*cada nodo de nivel n es una hoja y cada nodo de nivel menor que n no tiene subárboles izquierdo y derecho vacíos*”. Definir una función de un argumento para determinar si el árbol dado como argumento es completo.

Funciones de Orden Superior

Una **función de orden superior (FOS)** es una función que cumple al menos una de las siguientes condiciones:

1. Tiene una o más funciones como argumentos.
2. Devuelve una función como resultado.

Cálculo Lambda – sintaxis

<termino> ::= < variable> |

$\lambda <\text{variable}> . <\text{termino}>$ | Abstracciones funcionales

(<termino> <termino>) Aplicaciones funcionales

Esta sintaxis permite la definición de funciones de orden superior

Una **función de orden superior** (FOS – First Order Function) es una función que cumple al menos una de las siguientes condiciones:

1. Puede tomar una o más funciones como argumentos.
2. Puede devolver una función como resultado.

Cálculo Lambda – sintaxis

Recibe como argumento una función

$$(\lambda x.x \lambda z.z)$$
$$\lambda z.z$$

Da como resultado una función

Recibe como argumento una variable

$$(\lambda x.\lambda y.(x y) z)$$
$$\lambda y.(z y)$$

Da como resultado una función

Recibe como argumento una función

$$(\lambda x.y \lambda z.z)$$
$$y$$

Da como resultado una variable

En scheme las funciones son tratadas como datos de primera clase

Un tipo de dato se considera de primera clase si puede ser:

- El valor de un variable (esto es, puede ser nombrado)
- Componente de una estructura de datos mayor
- **Un argumento de una función**
- **El valor que devuelve una función**

DEFINICIÓN DE FUNCIONES

(lambda (x) (+ x 1)) } Definición de función sin nombre
➤#<procedure>

(define suma-1 (lambda (x) (+ x 1)))

➤suma_1

(define resta-1 (lambda (x) (- x 1)))

➤resta-1

(suma-1 3)

➤4

(suma-1 (resta-1 3))

➤3

(define prefijo (lambda (p l)
 (if (null? p) #t (if (equal? (car p) (car l))

Si no tuviera nombre debo escribir
((lambda (x)(+ x 1)) ((lambda (x) (- x 1)) 3))

suma-1

resta-1

(prefijo (cdr p) (cdr l))
 #f))))

Construcción de una Lista de Funciones

```
(define cuadrado (lambda (x)
    (* x x)))
```

```
(define cubo (lambda (x)
    (* x x x)))
```

```
> (define listaf '(cubo cuadrado))
```

```
> ((car listaf) 3)
```

. procedure application: expected procedure, given:
cubo; arguments were: 3

```
> (define listaf (list cubo cuadrado))
```

```
> ((car listaf) 3)
```

FUNCIONES DE ORDEN SUPERIOR

Funciones como argumentos

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

```
(define sumatoria (lambda (n)
  (if (= n 0) 0
      (+ n (sumatoria (- n 1))))))
```

Función

```
(define combina (lambda (f n cb)
  (if (= n 0) cb
      (f n (combina f (- n 1) cb )))))
```

Caso Base

FUNCIONES DE ORDEN SUPERIOR

Funciones como argumentos

Cómo se reescribirían factorial y sumatoria?

```
(define combina (lambda (f n cb)
  (if (= n 0) cb
      (f n (combina f (- n 1) cb )))))
```

```
(define sumatoria2 (lambda (n)
  (combina + n 0)))
```

```
(define fact2 (lambda (n)
  (combina * n 1)))
```

FUNCIONES DE ORDEN SUPERIOR

Funciones que construyen funciones

Aplicación

(*función*₁ arg₁ arg₂ arg_n)

($\lambda x. \lambda y. (x y) \lambda z. z$)

Resultado

*función*₂

$\lambda y. (\lambda z. z y)$

(define *funcion*₁

 (lambda (arg₁ arg₂ arg_n)

 (lambda (arg_i arg_j ... arg_k)

)

)

*función*₂ → Función retornada



FUNCIONES DE ORDEN SUPERIOR

Funciones que construyen funciones

```
(define aplicaSi (lambda (condic operac)
  (lambda (l)
    (if (null? l) '()
        (if (condic (car l))
            (cons (operac (car l)) ((aplicaSi condic operac) (cdr l)))
            (cons (car l) ((aplicaSi condic operac) (cdr l)))))))
  ))
```

Función retornada

```
(define mayor5 (lambda (x) (> x 5)))
(define increm (lambda (x) (+ x 1)))
```

```
> (aplicaSi mayor5 increm)
#<procedure:... 2c/PF/ejemplos.rkt:28:9>

> ((aplicaSi mayor5 increm) '(1 8 3 1 45 6))
(1 9 3 1 46 7)
```

Funciones que construyen funciones

```
(define composicion  
  (lambda (f g)  
    (lambda (x) (f (g x))))  
  )  
)  
Función retornada
```

➤(composicion suma-1 resta-1)

#<procedure>

➤((composicion suma-1 resta-1) 2)

2

Función de orden superior *map*

map es una función de orden superior, primitiva de Scheme, que tiene como argumentos una:

- *función de un argumento*
- *una lista*

retornando la lista resultante de los resultados de la aplicación de la función a todos los elementos de la lista argumento.

```
➤(map suma-1 '(1 2 3))  
'(2 3 4)
```

```
➤(map invertir '((1 2) (3 4)))  
'((2 1) (4 3))
```

Ejercicios

Definir la función **mapeo** que recibe como argumento una función de 1 argumento y una lista. Mapeo devuelve una lista que resulta de aplicar la función recibida como argumento a cada elemento de la lista

Por ejemplo:

```
> (mapeo suma-1 '(1 2 3))  
(2 3 4)
```

```
> (mapeo car '((1 2 3) ('a 'b 'c) ("hola" "chau")))  
1 'a "hola")
```

```
> (mapeo cdr '((1 2 3) ('a 'b 'c) ("hola" "chau")))  
((2 3) ('b 'c) ("chau")))
```

```
(define mapeo (lambda (f l)
  (if (null? l) l
      (cons (f (car l))
            (mapeo f (cdr l)) )
  ) ) )
```

Función de orden superior *filtro*

filtro es una función de orden superior, que tiene como argumentos una:

- **función de un argumento que devuelve #t o #f**
- **una lista**

retornando una lista cuyos elementos son los elementos de la lista original que devolvieron #t al aplicarle la función recibida como argumento

➤ **(filtro mayor2 '(1 2 3 4))**

(3 4)

➤ **(filtro list? '((1 2 3) 3 (12 3)))**

((1 2 3) (12 3))

```
(define filtro (lambda (f l)
  (if (null? l) l
      (if (f (car l))
          (cons (car l) (filtro f (cdr l)))
          (filtro f (cdr l)))
      )))
))
```

Ejercicios

Defina la función (**aplica-mapeo LFunciones Lista**) que devuelve una lista que resulta de aplicar cada una de las funciones que son miembros de LFunciones a cada elemento de Lista.

Por ejemplo:

```
> (define suma-1 (lambda (x) (+ x 1)))
```

```
> (define resta-1 (lambda (x) (- x 1)))
```

```
> (define por-2 (lambda (x) (* x 2)))
```

```
> (aplica-mapeo (list suma-1 por-2) '(1 2 3))
```

```
(4 6 8)
```

```
> (aplica-mapeo (list cdr car) '((1 2 3) ('a 'b 'c) ("hola" "chau")))
```

```
(2 'b "chau")
```

Ejercicios

Defina la función (**aplica-filtro listaFiltros**) que retorne una función cuyo argumento es una lista. En este caso, listaFiltros es una lista de funciones de un argumento que devuelven verdadero o falso. Al evaluar la función resultante con una lista específica se deben aplicar todos los filtros de la lista de funciones para obtener la lista resultante.

Por ejemplo:

```
> (aplica-filtro (list mayor2 menor4))
```

```
#<procedure>
```

```
> ((aplica-filtro (list mayor2 menor4)) '(4 2 3 5 1 6))
```

```
(3)
```

Ejercicios

Defina una función de orden superior que recibe como parámetros una función (que retorna verdadero o falso) y una lista.

La función (**take-while condicion L**) devuelve una lista con los n primeros elementos de la lista L para los que condicion retorne verdadero. Donde n es la posición del primer elemento para el que la evaluación de condicion retorne falso. Por ejemplo:

```
(define mayor2 (lambda (x) (> x 2)))
(take-while mayor2 '(5 6 7 2 8))
(5 6 7)
(take-while mayor2 '(5 6 7 9 8))
(5 6 7 9 8)
(take-while mayor2 '(1 2 3 3 2))
()
```

Ejercicios

De manera similar al punto anterior, defina la función
(drop-while condition L)

que devuelve la lista L a la que se le eliminaron los primeros elementos que satisfacen la condición (esto es aquellos elementos que al aplicarle la función condition dan verdadero). Por ejemplo:

```
(define menor4 (lambda (x) (< x 4) ) )
(drop-while menor4 '(2 3 1 5 6 ) )
(5 6)
(drop-while menor4 '(2 3 1 5 6 1) )
(5 6 1)
(drop-while menor4 '(5 6 1 2 3) )
(5 6 1 2 3)
```

Ejercicios

Dada una función de un argumento f y un entero positivo n , se le solicita que defina la función de orden superior (**repetir f n**), la cual retorna una función de un argumento que aplica n veces la función f . Por ejemplo:

```
(define cua (lambda (x) (* x x))  
((repetir cua 1) 2)  
4  
((repetir cua 3) 2)  
256
```

Ejercicios

a) Se le solicita que defina la función *Aplica* de **tres argumentos, una lista, una función f y un número entero n**, que dará como resultado la lista original pero con el elemento que se encuentra en la posición **n** reemplazado por el resultado de aplicar la función **f** al enésimo elemento. Por ejemplo:

```
(define Suma1 (lambda (x) (+ x 1)))
(Aplica Suma1 '(2 4 9) 1)
(3 4 9)
(Aplica Suma1 '(2 4 9) 3)
(2 4 10)
```

b) Se le solicita que defina una función de dos argumentos, una lista y una función f, *aplica-all*, que dará como resultado una lista de listas, en la cual el enésimo elemento es la lista original en la cual se ha reemplazado su enésimo elemento por el resultado de aplicar f al mismo. Por ejemplo:

```
(define Suma1 (lambda (x) (+ x 1)))
(aplica-all Suma1 '(2 4 9))
((3 4 9) (2 5 9) (2 4 10))
```