Programación Funcional

Pedro O. Pérez M., PhD.

12-2025

Implementación de métodos computacionales
Tecnológico de Monterrey
pperezm@tec.mx

Contenido i

Introducción

Lenguajes de programación funcional

En la práctica, los lenguajes funcionales...

- Todo lo modelan con funciones: definiciones y llamadas, la secuencia es una composición de funciones.
- No manejan variables, sólo parámetros.
- No manejan asignación de valores.
- No utilizan iteraciones, sólo recursividad.
- Almacenan todo en listas encadenadas y con asociaciones dinámicas.
- Tratan a las funciones como a los datos: son argumentos, resultados, estructuras.

Conceptualmente, los lenguajes funcionales...

- Son **declarativos**:
  - Expresan qué resolver y no cómo resolverlo.
- Son de **muy alto nivel de abstracción**:
  - Alejados del modelo de Von Nuemann y apegados al pensamiento "natural" humano.
  - Requieren mayor esfuerzo de traducción y pueden consumir muchos recursos en la ejecución.
- Tienen **transparencia referencial**:
  - No hay efectos laterales en memoria que alteren el significado de un programa.
- Son **minimalistas**:
  - Fácil lectura, mantenimiento, **paralelización** y comprobación.

Los lenguajes funcionales...

- Existen desde **1958** con la creación de **LisP** (**Lis**t **P**rocessing).
- Tienen diferentes grados de hibridez al combinarse con diversos paradigmas. **Haskell** es uno de los consideramos más puro de todos los lenguajes funcionales.
- Han adquirido mayor importancia y popularidad por sus ventajas en el desarrollo de aplicaciones de **Inteligencia Artificial** y **Ciencia de datos**.

Estilo de Programación Funcional

El estilo de la programación funcional se puede usar en los lenguajes imperativos.

```
int factorial (int n)
{ int fact, j;
  fact = 1;
  for (j = 2; j<=n; j++)
     fact = fact * j;
  return fact;
};
```

```
int factorial (int n)
{
   if (n == 0)
     return 1;
   else
     return (n*factorial(n-1));
};
```

Estilo imperativo:

- Uso de variables y asignaciones.
- Uso de ciclos (iteraciones).
- Posible efecto lateral si usáramos variables globales.

Estilo funcional:

- No hay variables ni asignaciones.
- Uso de la recursividad.
- No hay efectos laterales al no usar variables globales.

Scheme

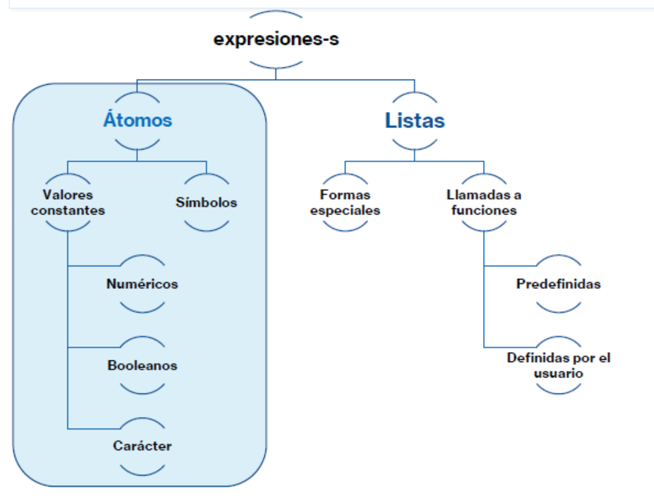
Scheme

- **Scheme** es un lenguaje funcional sencillo, minimalista, para aprender ágilmente el nuevo paradigma.
- Es un dialecto de **Lisp** creado en 1975 por Steele y Sussman (MIT).

Iniciando con Scheme...

- Toda su sintaxis se reduce al formato de una **lista** que es una **expresión**-s.
  - Las listas se delimitan con **paréntesis** y puede tener cero o más elementos.
  - Una lista es una expresión-s cuyos elementos son a su vez una expresión-s.
- Tiene pocas reglas **semánticas**, pues es un lenguaje tipado débil (no hay declaraciones de tipos de datos).

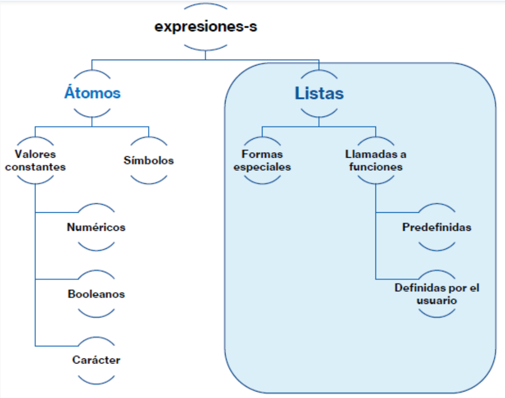
expresiones-s en Scheme



Átomos:

- La evaluación de una constante genera como resultado el valor de la propia constante.
- La evaluación de un símbolo genera como resultado el valor asociado a ese símbolo.
- Un símbolo es un identificador que se construye con cualquier carácter, excepto:

  `( ) [ ] { } ; , ‘ “ # \`



Sintaxis de las listas

<Lista> ::= (<Elemento>)

<Elemento> ::= átomo <Elemento>

<Elemento> ::= <Lista> <Elemento>

<Elemento> ::= ϵ

El **primer elemento** en la lista es el **símbolo** que identifica a la **función** o la **forma especial** que se desea evaluar, y los siguientes elementos o datos necesarios para la evaluación.

Funciones predefinidas

- Aritméticas: +, -, *, /, remainder, quotient, sqrt, etc.
- Relacionales: <, <=, >, >=, =.
- Lógicas: and, or, not.
- Predicados: positive?, zero?, even?, null?, char?, etc.
- Manejo de listas: car, cdr, cons, list.
- Manejo de funciones: map, apply.

- No hay operadores, sólo **funciones multiparámetro** que se aplican sobre los argumentos. **Importante: el formato es prefijo**.
- Ejemplos:
  - `(+)`
  - `(- 8)`
  - `(+ 3 4)`
  - `(+ 2 3 4 5 6 7 8)`
  - `(/ (+ 2 3) 5)`
  - `(+ (/ 2 3) 5)`
  - `(/ (- 7 3) (* 2 5))`

La forma especial `define`

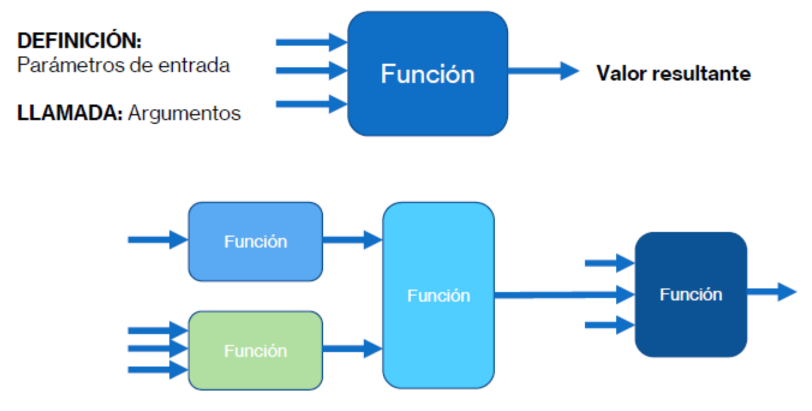
- Sintaxis: `(define símbolo expresión)`.
- Evalúa la expresión y el valor resultante lo asocia con el símbolo, de tal forma que el símbolo queda definido en el ambiente de trabajo.

Ejemplos

Vamos a desarrollar las siguientes funciones:

1. La función `sum` recibe como entrada dos números, a y b. La función regresa la suma de ambos.
2. La función `area-of-triangle` recibe como entada la base y altura de un triángulo. La función regresa el área del triángulo.

En el paradigma funciona, todo es una función...



Ejemplos

Vamos a desarrollar las siguientes funciones:

1. La función `area-of-ring` recibe como entrada el radio interno y externo del anillo. La función regresa el área del anillo.
2. La función `volume-of-cylinder` recibe como entra el radio u la altura de un cilindro. La función regresa el volumen del mismo.

Actividad de programación 2

Revisa la actividad en Canvas.

La forma especial `quote`

- Sintaxis: `(quote símbolo)`.
- Sirve para NO evaluar el símbolo, generando como resultado el propio símbolo. Útil para manejar a los símbolos como datos.
- Puede abreviarse con una comilla y sin necesidad de utilizar a la lista.
- `(quote abc)` es equivalente a `'abc`.

La forma especial `if`

- Sintaxis: `(if predicado if-true if-false)`.
- Se evalúa el predicado, si su valor es verdadero se evalúa la expresión consecuente; si es falso, se evalúa la expresión alternativa.
- Un **predicado** es una expresión que genera un valor booleano al evaluarse.
- A este tipo de evaluación se le conoce como **evaluación floja (lazy evaluation)**, pues no se evalúa lo que no es necesario.

Ejemplos

Vamos a desarrollar las siguientes funciones:

1. La función `max2` recibe como entrada dos números, a y b. La función regresa el mayor de ambos números.
2. La función `max3` recibe como entrada tres números, a, b y c. La función regresa el mayor de los tres números.

La forma especial `cond`

- Sintaxis: `(cond (`*predicado*$_1$`, `*expresion*$_1$`), (`*predicado*$_2$`, `*expresion*$_2$`), (else `*expresion*$_n$`))`.
- Forma de evaluación: evalúa la primera secuencia de expresiones cuyo predicado sea verdadero; si ningún predicado se cumple, evalúa la expresión del `else`

Ejemplos

Vamos a desarrollar las siguientes funciones:

1. La función **interest** recibe como entrada el saldo de una cuenta bancaria de un banco. El banco paga un 4 % fijo para saldos de hasta \$1000, un 4.5 % fijo anual para saldos de hasta \$5000 y un 5 % fijo para saldos de más de \$5000.
2. La función `how-many` recibe como entrada los coeficientes, a, b y c, de una ecuación cuadrática. La función determina cuántas soluciones tiene la ecuación. Asumiendo que a no es 0, la ecuación tiene:
   - 2 soluciones, si $b^2 > 4ac$.
   - 1 solución, si $b^2 = 4ac$.
   - 0 soluciones, si $b^2 < 4ac$.

Actividad de programación 3

Revisa la actividad en Canvas.

Recursividad en Scheme

- Ya conocemos las herramientas necesarias del lenguaje:
  - Definición y llamadas de funciones.
  - Decisiones con las formas especiales **if** y **cond**.
- Lo importante es **desarrollar el pensamiento recursivo** para la solución de problemas.
- Esto va más allá del uso de lenguaje, es un cambio de paradigma mental.

Para pensar recursivamente:

- Paso 1.
  - Analizar cuál es el **caso más simple o pequeño** del problema que se quiere resolver.
  - Este caso debe de tener una **solución clara y directa, no recursiva**.
  - Este caso se considera el **caso base** de la recursividad, y determina una **condición de salida** de la repetición implícita que se dá en la recursividad.
- Paso 2.
  - Analizar cómo se resuelve el **problema general**, **suponiendo** que ya se tiene "algo" que resuelve el **siguiente caso más pequeño del problema**.
  - Este caso plantea la **solución recursiva** del problema.
  - La solución al siguiente caso más pequeño, se programa con la **llamada recursiva**, que se integra a la solución general del caso.

Ejemplos

Vamos a desarrollar las siguientes funciones:

1. La función que obtiene la sumatoria desde 0 hasta n.
2. La función que despliega n veces el letrero "hola".
3. La función que despliega la secuencia desde n hasta 1.
4. La función que cuenta la cantidad de dígitos de un número entero.

Actividad de programación 4

Revisa la actividad en Canvas.

Listas

La herramienta “universal” para representar y trabajar con estructuras de datos.

Sintaxis de las listas

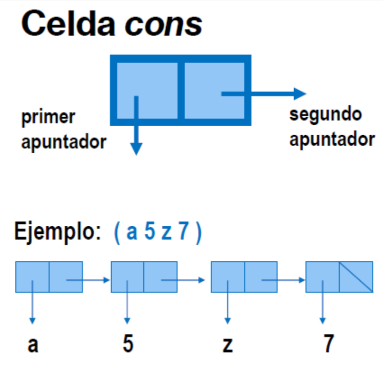
<Lista> ::= (<Elemento>)

<Elemento> ::= átomo <Elemento>

<Elemento> ::= <Lista> <Elemento>

<Elemento> ::= ϵ

Representación interna

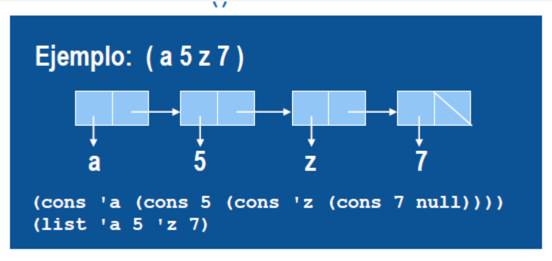


- Las listas están almacenadas internamente en memoria dinámica, utlizando nodos encadenados llamados "**celdas cons**".
- Una "**celda cons**" consiste de dos apuntadores; ambos pueden apuntar a átomos o a otra "celda cons".
- `null` o `'()` es un valor atómico que representa a la lista vacía.

Construcción de una lista

Visualmente,

- Utilizar la forma especial **quote** con la lista.
- Ejemplos: '(1 2 3 4), '(((a 1) (B 2) (+ 4 X))), '().



Funcionalmente,

- Función primitiva de bajo nivel: **cons**.
  - Formato: `(cons arg1 arg2)`.
  - Crea una **"celda cons"** en memoria, en donde el apuntador apunta a **arg1** y el segundo argumento a **arg2**.
- Función primitiva de alto nivel: **list**.
  - Formato: `(list arg1 arg2 arg3 ...)`.
  - Crea una lista propia en la que cada argumento se convierte en un elemento de la lista (en el orden correspondiente).

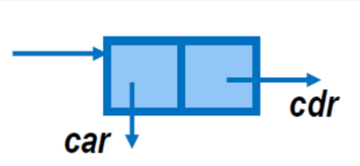
Tipos de listas

- Lista propia. Es aquella cuyo último nodo de la lista apunta a la lista vacía. Visualmente utiliza el formato normal. Ejemplo: `(1 2 3 4)`.
- Lista impropia. Es aquella donde el último nodo de la lista apunta a un átomo diferente a la lista vacía. Sólo se puede construir con la función **cons**. Visualmente termina con un punto antes del último dato. Ejemplo: `(a . 1)`. *Útil cuando se requiere una estructura para almacenar un par de datos relacionados.*
- Lista plana. Es aquella que únicamente contiene átomos en todos los primeros apuntadores de sus **"celdas cons"**. Visualmente no tiene lista anidadas. Ejemplos: `(1 2 3 4)`, `(a . 1)`.
- Lista imbricada (o profunda). Es aquella cuyos elementos no son solo átomos, sino que sus primero apuntadores de sus **"celdas cons"** apuntan a otras listas de cualquier tipo. Visualmente es una lista con listas anidadas. Ejemplo: `((a 1) (b 2) ((1) (z d)))`. *Permite representar cualquier estructura de datos no lineales y/o complejos.*

Acceso a una lista

Función **car**

- Formato: `(car lista)`.
- Genera como resultado el primero elemento de la lista, es decir, el valor apuntado por el primer apuntador de la primera "**celda cons**" de la lista.



Función **cdr**

- Formato: `(cdr lista)`.
- Genera como resultado el "**resto**" **de la lista**, es decir, el valor apuntado por el segundo apuntadora de la primera "**celda cons**" de la lista.

```
(car (cons X Y))  -->  X
(cdr (cons X Y))  -->  Y
(cons (car X) (cdr X))  -->  X
```

¿Porqué **car** y **cdr**?

- **Lisp** fue implementado original en la computadora **IBM 704** en 1958.
- Esta computadora tuvo un soporte especial para dividir una palabra de máquina de 36 bits en cuatro partes, una "address part" y "decrement part" de 15 bits cada una y una "prefix part" y "tag part" de tres bits cada una.
- El lenguaje Lisp se definió asociando esta parte técnica:
  - **car**: Abreviación de "**C**ontents of the **A**ddress part of **R**egister number".
  - **cdr**: Abreviación de "**C**ontents of the **D**ecrement part of **R**egister number".
- Otros lenguajes han sustituido estos nombres por: **first**/**rest** o **head**/**tail**.

Lenguajes como Scheme permiten simplificar las composiciones entre ambas funciones, combinando hasta cuatro a's o d's entre la c y la r

Ejemplo: `(cadddr lista)`
es equivalente a `(car (cdr (cdr (cdr lista))))`

Ejercicios

- ¿Cómo se accede el segundo elemento de una lista?
- ¿Cómo acceder al dato "b" en la lista : (a (b c)?
- ¿Qué resultado se obtiene de: (cddr '(a (b c)))?
- ¿Cómo accede al dato "c" en la lista : (a (b c))?
- ¿Qué resultado se obtiene de : (cons (cadr '(a (b c) d)) (list 'e))?

Unión de 2 listas

Función primitiva de alto nivel: **append**

- Formato: `(append lista1 lista2)`
- Genera una lista que es el resultado de encadenar la `lista1` con la `lista2`, es decir, la última "**celda cons**" de la primera lista apuntará con su segundo apuntador a la primera "**celda cons**" de la segunda lista.
- Sólo se puede utilizar con listas propias.
- Ejemplo: `(append '(1 2 3) (cons 'a (list 'b 'c)))` genera como resultado `(1 2 3 a b c)`.

- **Caso Base**. Normalmente, está basado en la solución al caso de la lista vacía.
- **Caso General**. Normalmente, supone que se tiene la solución al caso del **"resto" (cdr) de la lista** (que es una lista más pequeña), y utiliza ese resultado para calcular el resultado general.

Predicados relacionados con listas

- `null?`: Verifica si su argumento es la lista vacía o no.
- `list?`: Verifica si su argumento es una lista propia o no.
- `pair?`: Verifica si su argumento es una "**celda cons**" o no.
- `equal?`: Verifica si sus dos argumentos son idénticos en contenido.
- `eq?`: Verifica si sus dos argumentos son idénticos en contenido y en su representación física (mismo espacio de memoria).

Tipos de problemas con listas

Por el tipo de entrada y resultado

1. Problemas que reciben listas y generan un resultado atómico.
2. Problemas que reciben un valor atómico y generan una lista.
3. Problemas que reciben listas y generan una lista.

Por el tipo de lista

a. Problemas que trabajan con listas planas, o sólo con los elementos de su primer nivel:
RECURSIVIDAD PLANA

b. Problemas que trabajan con listas imbricadas (listas anidadas) y se desea llegar hasta los átomos:
RECURSIVIDAD PROFUNDA

Problemas que reciben listas y generan un valor atómico

- **Caso Base**: **Valor atómico** que soluciona el problema cuando **la lista está vacía o tiene un elemento**.
- **Caso General**: **Suponer** que ya se tiene la **solución con el resto de la lista**, y construir la **expresión** que resuelve el problema general.

Ejercicios

1. Contar los elementos de una lista.
2. Sumar los elementos de una lista.
3. Contar la cantidad de números pares de una lista.
4. Obtener el número mayor de una lista.

Actividad de programación 5

Revisa la actividad en Canvas.

Problemas que reciben un valor atómico y generan una lista

- **Caso Base**: **Lista** que soluciona el problema cuando el valor atómico de entrada es el más pequeño (**cero o uno**).
- **Caso General**: **Suponer** que ya se tiene la **lista** que es solución para el **siguiente valor más pequeño** del problema, y utilizarla para **construir la lista** (`cons`, `list`, `append`) que resuelve el problema general.

Ejercicios

1. Generar una lista con n ceros.
2. Generar una lista con los valores de n a 1.
3. Generar una lista con los valores de 1 a n.
4. Generar una lista con los valores de la serie de Fibonacci hasta el n-ésimo elemento de la serie.

Problemas que reciben una lista y generan una lista

- **Caso Base**: **Lista** que soluciona el problema cuando la **lista de entrada está vacía o tiene un elemento**.
- **Caso General**: **Suponer** que ya se tiene la **lista** que es solución para el **resto de la lista de entrada**, y utilizarla para **construir la lista** (`cons`, `list`, `append`) que resuelve el problema general.

Ejercicios

1. Incrementar los números de una lista.
2. Unir 2 listas en una sola (implementación del `append`).
3. Invertir los elementos de una lista (implementación del `reverse`).

Actividad de programación 6

Revisa la actividad en Canvas.

Recursividad Profunda

- Útil principalmente en problemas que reciben como entrada una lista profunda y se desea llegar a los átomos de la lista.
- La estrategia del pensamiento recursivo se mantiene, pero en la solución al caso general hay que agregar la **validación** de si el elemento a analizar (típicamente el car) **es una lista o no**.
  - Si **no es una lista** se aplica la solución típica de la **recursividad plana**.
  - Si **es una lista**, se aplica la **recursividad sobre ese elemento**, y **sobre el resto de la lista** para obtener la solución general.

Ejercicios

- Contar los elementos de una lista profunda.
- Sumar los elementos de una lista profunda.
- Contar la cantidad de números pares de una lista profunda.
- Obtener el número mayor de una lista profunda.

Revisa la actividad en Canvas.

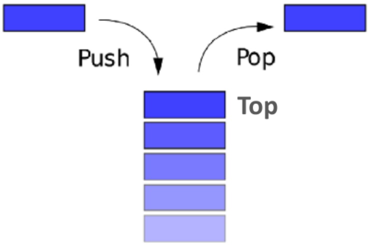
Listas como Registros y Base de datos

La herramienta "universal" para representar y trabajar con estructuras de datos.

- **Modelación**. La lista tiene la capacidad de representar a cualquier estructura de datos, sin importar si es lineal o no, al permitir **anidamientos**.
- **Inmutabilidad**. Las estructuras de datos implícitamente utilizan **memoria dinámica**, pero no son estructuras cuyo estado se modifique en memoria... Por lo tanto, se implementarán funciones que transforman las estructuras de datos de entrada en nuevas estructuras de datos modificadas.

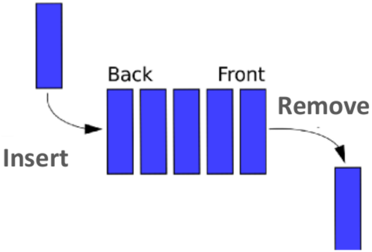
- Dado que una lista en Scheme acepta elementos de diferentes tipos, una lista de campos es la relación directa para almacenar un **registro**.
- Ejemplo, registro con datos de alumnos: matricula, nombre, calificaciones: `(176432 Juan Pérez 78 95 87)`, `(176432 (Juan José Pérez Pérez) (78 95 87))`.
- Una lista de registros conforma una base de datos. Ejemplo, `((`$registro_1$`)` `(`$registro_2$`)` `(`$registro_3$`)` $\ldots$`(`$registro_n$`))`

¿Cómo implementar una fila o una pila




```
(define (push pila dato)
    (cons dato pila))

(define (pop pila)
    (cdr pila))
```




```
(define (insert fila dato)
    (append fila (list dato)))

(define (remove fila)
    (cdr fila))
```

Matrices utilizando listas

- Se representa con una lista de renglones (o columnas), donde cada reglón (o columna), es a su vez, una lista con los datos correspondientes.

```
((1 2 3)
 (4 5 6)
 (7 8 9))
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \text{ o } \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

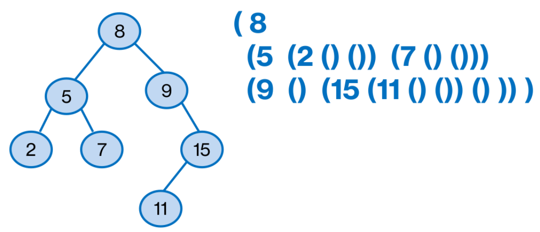
- Ejercicio: Implementar la suma de dos matrices.

Representación de un árbol binario en listas

- Formato de representación:

  `( raíz (subárbol izquierdo) (subárbol derecho) )`

- Ejemplo:



- Ejercicio:
  - Implementar la búsqueda en un ABB.
  - Implementar recorrido en PREORDEN.
  - Implementar la inserción en un ABB.

Grafos en listas

- Un grafo tiene múltiples formas de almacenarse en memoria:
  - Matriz de adyacencias.
  - Matriz de transiciones.
  - Lista de adyacencias.
  - Lista de arcos.
- Al representarlo en listas de Scheme, se tiene la ventaja que se puede definir el estándar de la modelación.

Cálculo Lambda

¿Qué significa que las funciones son de primer orden?

¡Se tratan como a los datos!

- Se pueden enviar como **argumentos** a los parámetros de otra función.
- Se pueden generad como **resultado** de otra función.
- Pueden ser parte de una **estructura de datos**.

Ejemplos

Funciones como argumentos

```
(define (aplica proc a b)
   (proc a b))

>(aplica + 3 5)
8
>(aplica * 3 5)
15
>(aplica < 3 5)
#T
```

El orden aplicativo evalúa a los símbolos que identifican a las funciones y envía como argumento el código asociado a la función.

Funciones como resultado

```
(define (tipo n)
   (cond ((= n 1) +)
         ((= n 2) -)
         ((= n 3) *)
         ((= n 4) /)))

>((tipo 2) 3 5)
-2
>((tipo 3) 3 5)
15
```

El resultado de una llamada a una función que genera una función, es aplicable con argumentos.

Funciones en estructuras

```
(define LF (list + - * /))

>((car LF) 3 5)
8
>((cadddr LF) 3 5)
0.6
```

El orden aplicativo evalúa a los símbolos que identifican a las funciones generando el código asociado a cada función, y formando una lista de códigos.

- Programar con funciones de primer orden representa una característica distintiva del paradigma funcional, que le da flexibilidad a los lenguajes, y potencializa las estrategias de solución de problemas en forma eficiente.

Ejemplo

```scheme
(define (inserta elem lista tipo)
  (cond ((null? lista) (list elem))
        ((tipo elem (car lista)) (cons elem lista))
        (else (cons (car lista)
                    (inserta elem (cdr lista) tipo)))))
```

```scheme
(define (ordena lista tipo)
  (cond ((null? lista) ())
        (else (inserta (car lista)
                       (ordena (cdr lista) tipo) tipo))))
```

Funciones map y apply

- Formato: `(map función lista)`.
- Genera una lista con el resultado de aplicar la función a cada elemento de la lista de entrada.
- Ejemplo: `(map sqrt '(4 16 25 81) => '(2 4 5 9)`

Implementación

```
(define (map fn lista)
   (cond ((null? lista) '())
         (else (cons (fn (car lista))
                     (map fn (cdr lista))))
))
```

- Formato: `(apply función lista)`.
- Genera el resultado de la aplicación sucesiva de la función con los datos de la lista de entrada.
- Ejemplo: `(apply + '(3 5 7 1) => 16`

Implementación

```
(define (apply fn lista)
    (cond ((null? lista) (fn))
          (else (fn (car lista)
                    (apply fn (cdr lista))))
))
```

Ejemplos

Dada una matriz almacenada por renglones:

- Obtener los datos de la primera columna.

```
(map car matriz)
```

- Obtener la sumatoria del primer renglón.

```
(apply + (car matriz)
```

Dada una base de datos en donde se almacenan datos de empleados en el siguiente formato de registro: `(nómina (nombre) depto sueldo)`.

- Obtener una lista con los nombres de los empleados.

```
(map cadr datos)
```

- Obtener la sumatoria de los sueldos.

```
(apply + (map cadddr datos))
```

- Implementar una función que sirva para obtener la sumatoria de todos los elementos de una matriz.
- Implementar una función que sirva para obtener la matriz transpuesta de una matriz

Otras funciones valiosas

- **map** se puede aplicar sobre varias listas, en cuyo caso, el procedimiento a mapear deberá considerar un parámetro para cada lista.
- **filter** sirve para filtrar los datos que cumplen la condición del predicado enviado como parámetro.
- **foldl** y **foldr** para realizar compresiones de listas según el procedimiento indicado.

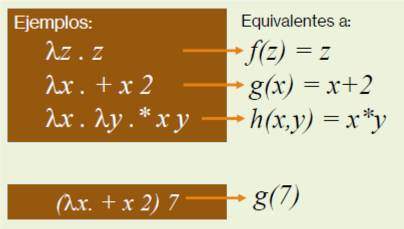
¿Qué hay detrás de todo?

Detrás de todo esto está el Cálculo Lambda.

- Propuesto por **Alonzo Church** en los años 30's, para formalizar nociones intuitivas acerca de funciones.
- En los 50's, **John McCarthy**, creador de Lisp , lo toma como base matemática, teórica y formal de los lenguajes funcionales.
- Es una **notación** especial para definir **funciones sin nombre**.
- Conocer un poco del cálculo lambda, permite entender mejor el concepto de las **funciones como elementos de primer orden** en los lenguajes funcionales.

Definición y aplicación de funciones

- Formato general de una λ-**expresión**: λ `variable . cuerpo` donde el cuerpo es a su vez una λ-**expresión**.
- Una λ-**expresión** es:
  - Un valor constante.
  - Una variable.
  - Una expresión aritmética en prefijo.
  - Una λ-expresión.
- Formato para aplicar (evaluar) una función: (función argumento)



La forma especial lambda

- Formato: `(lambda (parámetros) cuerpo)`
- Sirve para definir una **función sin nombre**, equivalente a las λ-**expresiones** del cálculo lambda de Alonzo Church.
- El cuerpo se compone de las **expresiones-s** que conforman el código de la función. Ejemplo: `(lambda (x) (+ x 2))`.
- Útil para construir funciones "en línea" sin necesidad de definirlas con nombre y enviarlas como argumento, y también para generar funciones como resultado. Ejemplo: `(map (lambda (x) (+ x 2)) '( 1 2 3))`.

Para darle nombre a una función definida con **lambda** usaremos la forma especial **define** en su concepto original.

Ejemplo:

```
(define g
    (lambda (x)
        (+ x 2)))
```

En vez de:

```
(define (g x) (+ x 2))
```

Ejercicio

- Utilizando a la forma especial `lambda`, **implementa una solución sin recursividad** equivalente a la siguiente función que suma todos los datos de una matriz

```
(define (sumatoria matriz)
  (cond ((null? matriz) 0)
        (else (+ (apply + (car matriz))
                 (sumatoria (cdr matriz))))))
```

Funciones generadoras de funciones

- Una función genera como resultado una función cuando su código de implementación es una lambda.
- Ejemplo: función que genera una función que sirve para incrementar un valor en tantas unidades como se indique en el parámetro.

```
(define genera-fn-incremento
   (lambda (inc)
     (lambda (valor)
        (+ valor inc))))
```

```
> ((genera-fn-incremento 5) 8)
13
> (define ++ (genera-fn-incremento 1))
> (++ 8)
9
> (define lista-fn-inc
          (map genera-fn-incremento '(1 2 3))
> ((cadr lista-fn-inc) 5)
7
```

Currying

- La función sirve para sumar dos datos.
- La función genera una función que sirve para incrementar un valor en tantas unidades como se indique en el parámetro

Ejercicio

- Implementar una función que sirva para generar funciones que apliquen dos veces sobre un valor la función recibida como parámetro.
  - ¿Cuál es el parámetro del procedimiento generador?
  - ¿Cuál es el parámetro del procedimiento a generar?

Actividad de programación 8

Revisa la actividad en Canvas.

Evaluación en el Cálculo Lambda

- La evaluación de funciones en cálculo lambda, sigue formalmente un proceso de transformaciones para normalizar la λ-expresión a su mínima expresión
- Tipos de transformaciones:
  - β-**reducción**: Reemplazo del parámetro por el argumento en el cuerpo de la función, eliminando el encabezado λ.
  - δ-**reducción**: Aplicación de una función primitiva operaciones aritméticas.
  - α-**reducción**: Substitución de nombre de variable.
  - η-**reducción**, θ-**reducción**, etc.

Ejemplo

$(\lambda x\ .\ +\ x\ 3)\ 5$	$(\lambda g.\lambda x.\lambda y.\ g\ -\ x\ y)\ (\lambda f.\lambda y.\lambda x.\ f\ x\ y)\ 4\ 5$
β: $+\ \ 5\ \ 3$	βg: $(\lambda x.\lambda y.\ (\lambda f.\lambda y.\lambda x.\ f\ x\ y)\ -\ x\ y)\ 4\ 5$
δ: 8	βx: $(\lambda y.\ (\lambda f.\lambda y.\lambda x.\ f\ x\ y)\ -\ 4\ y)\ 5$
	βy: $(\lambda f.\lambda y.\lambda x.\ f\ x\ y)\ -\ 4\ 5$
	βf: $(\lambda y.\lambda x.\ -\ x\ y)\ 4\ 5$
	βy: $(\lambda x.\ -\ x\ 4)\ 5$
	βx: $-\ 5\ 4$
	δ : 1

Orden de evaluación

Normal

- Comienza reduciendo la λ-expresión de más a la izquierda.
- Equivale a una sustitución tipo **parámetro por nombre**.
- Asegura llegar a la forma mínima posible.
- Ejemplo:

(λx . * x x) (+ 2 3)
βx: (* (+ 2 3) (+ 2 3))
δ (* 5 (+ 2 3))
δ (* 5 (+ 2 3))
δ (* 5 5) δ 25

Aplicativo

- Comienza reduciendo la λ-expresión más interna y/o sus argumentos.
- Equivale a una sustitución tipo **parámetro por valor**.
- Es más eficiente.
- Es la utilizada por los intérpretes.
- Ejemplo:

(λx . * x x) (+ 2 3)
δ: (λx . * x x) 5 βx: (* 5 5)
δ 25

Equivalencia con el Cálculo Lambda

- La forma especial lambda es equivalente a las λ-expresiones.
- Por ejemplo, (λx . λy . x y) (λx . * x x) 5 es equivalente a `(((lambda (x) (lambda (y) (x y))) (lambda (x) (* x x))) 5)`.
- Y el intérprete evalúa haciendo las transformaciones usando el orden aplicativo.

`(((lambda (x) (lambda (y) (x y))) (lambda (x) (* x x))) 5)`

βx: `((lambda (y) ((lambda (x) (* x x)) y)) 5)` <- Función como argumento

βx: `((lambda (x) (* x x)) 5)` <- Función como resultado

βx: `(* 5 5)`

δ: 25

Recursividad Terminal

- **Estilo de programación recursiva** en donde la **solución al problema** se obtiene al **llegar al caso base** (la "cola" del proceso).
- Se logra cuando **la solución recursiva al problema no contiene operaciones adicionales a la llamada recursiva**.
- No siempre corresponde al pensamiento natural recursivo y requerirá una **conversión** si se desea obtener sus ventajas.

Ventajas

- En muchos casos es una manera de hacer **más eficiente** la ejecución de la recursividad.
- En algunos casos, el **orden de complejidad de tiempo** se mejora significativamente.
- Algunos intérpretes son capaces de identificarla y **hacer más eficiente la ejecución**, descartando el uso de la pila de llamadas y la ejecución del regreso de todas las llamadas recursivas pendientes.

Revisaremos la función factorial

$(n!)$

¿Cómo convertir a recursividad terminal?

1. La función original se convierte en una interfase que solo sirve para llamar a una **función auxiliar** que estará implementada con la **recursividad terminal**.
   - La función auxiliar contendrá un **parámetro extra** que será el **acumulador del resultado**.
   - La **llamada** a la función auxiliar enviará al **parámetro extra** el valor de la **solución del caso base del problema**.
2. La **función auxiliar**, contendrá el código para resolver el problema recursivamente, pero cuidando de hacer la **llamada recursiva en forma terminal**.
   - El **resultado del caso base** será directamente el **valor del parámetro extra** (acumulador del resultado).
   - La llamada recursiva enviará como **argumento** al **parámetro extra** el **resultado de la expresión que se realizaba en forma no terminal**, pero ahora **utilizando el parámetro extra** (acumulador).

Actividad de programación 9

Revisa la actividad en Canvas.