Ayudantía 9
Programación Funcional: Scheme

Profesores: José Luis Martí Lara, Roberto Díaz Urra
Ayudantes:
Hugo Sepúlveda Arriaza
Gabriela Acuña Benito
Lucio Fondón Rebolledo
lucio.fondon@sansano.usm.cl

Universidad Técnica Federico Santa María
Departamento de Informática

Programación Funcional

Definición del Paradigma

- Paradigma diferente a los imperativos, que se aleja de la máquina de von Neumann.
- Basado en funciones matemáticas (notación funcional lambda de Church).
  - **Matemáticas**:
    - Definición de una función: `cubo(x)` $=$ `x·x·x`
    - Aplicación de la función: `cubo(2.0)` $= 8$
  - **Notación Lambda Church**: Separa la función de su nombre (*función anónima*)
    - Definición de una función: $\lambda(x) = x \cdot x \cdot x$
    - Aplicación de la función: $(\lambda(x)\ x \cdot x \cdot x)(2.0) \implies 8.0$
- La programación funcional puede:
  - **Realizar una Composición de funciones**: $h = f\ o\ g \implies h(x) = f(g(x))$
  - **Aplicación a todo: Se le aplica una misma función a una lista**: $(f, (x, y, z)) \implies (f(x), f(y), f(z))$

Programación Funcional

Definición del Paradigma

- Paradigma diferente a los imperativos, que se aleja de la máquina de von Neumann.
- Basado en funciones matemáticas (notación funcional lambda de Church).
  - **Matemáticas**:
    - Definición de una función: `cubo(x) = x·x·x`
    - Aplicación de la función: `cubo(2.0) = 8`
  - **Notación Lambda Church**: Separa la función de su nombre (*función anónima*)
    - Definición de una función: $\lambda(x) = x \cdot x \cdot x$
    - Aplicación de la función: $(\lambda(x)\ x \cdot x \cdot x)(2.0) \implies 8.0$
- La programación funcional puede:
  - **Realizar una Composición de funciones**: $h = f\ o\ g \implies h(x) = f(g(x))$
  - **Aplicación a todo: Se le aplica una misma función a una lista**: $(f, (x, y, z)) \implies (f(x), f(y), f(z))$

Función `factorial()` en Scheme

```
(define (factorial numero)
  (let rec ((num numero) (cant 1))
    (if (eq? num 0)
        cant
        (rec (- num 1) (* cant num))
        )
    )
  )
```

Función `factorial()` en C

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

Conceptos Básicos

- **Sintaxis**: Todas las evaluaciones en `Scheme` se realizan en *pre-orden* y todo entre paréntesis
- En `Scheme`, literalmente, **TODO** es evaluado (Ambiente *REPL*)
- **Tipos de Datos**:
  - Números: `3 5 2.0 -3.2` $\frac{1}{2}$ `1.2e17`
  - Números Complejos: `2.7-4.5i`
  - Carácter: `#\a`
  - String: `"Hola mundo"`
  - Booleanos: `#t #f`
- **Operaciones Aritméticas Básicas**: `+,-,*,/`
- **Expresiones Condicionales y Predicados**:
  - `>, <, >=, <=`
  - `and or not`
  - `null? eq? eqv? equal? list? integer?`

¿Cómo evalúa Scheme?

```
;; Con ;; se pueden realizar comentarios por linea
;; Scheme evalua todo preorden y con parentesis
;; (funcion par1 par2 ... par n)

(+ 2 3) ;; operador suma, suma los valores 2 + 3
(- 4 3) ;; operador resta, resta el primero con el segundo. 4 - 3
(+ (* 3 2) (/ 6 3)) ;; se pueden componer funciones. (3*2) + (6/3)

;; Presentamos el operador ' (o quote)

(quote (a b c d)) ;; (a b c d)
'(a b c d) ;; (a b c d)
(a b c d) ;; Error

;; Con define podemos declarar variables globales

(define a 2) ;; se le asigna el valor 2 a la variable a
a            ;; << 2

;; Tambien podemos realizar nuestras propias funciones
;; (define (nombre-funcion par1 par2 ... par n)
;;     (body))

(define (square x) ;; definimos la funcion
    (* x x))

(square 7) ;; 49
(+ 10 (square 2)) ;; 14
```

Pares y Listas

Pares

- En Scheme, uno puede juntar dos valores cualquiera y *pegarlos* para que formen una sola estructura llamada **pares** (pairs).
- Se crean a partir de la función `cons`
- `> (cons 1 2)` $\Longrightarrow$ `'(1.2)`

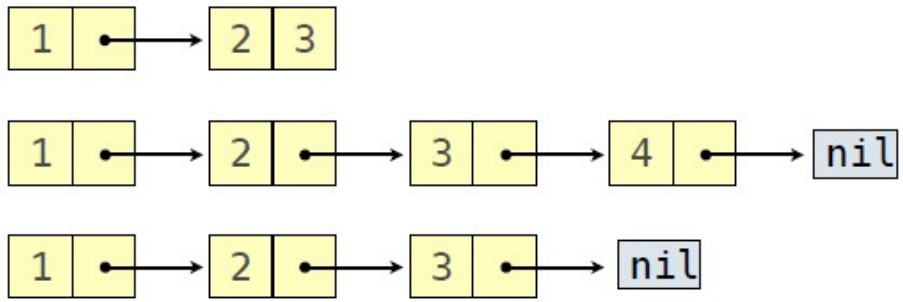
El punto indica que básicamente 1 y 2 son un solo par. También son llamados **listas impropias**

Listas

- Generalmente, uno no utiliza `cons` para agrupar valores, sino que es preferible y menos engorroso utilizar **listas**
- Una **lista** en Scheme no es más que una secuencia de pares
- Hay muchas formas de crear una lista en Scheme:
  - `> '(1 2 3)` $\Longrightarrow$ `'(1 2 3)`
  - `> (list 1 2 3)` $\Longrightarrow$ `'(1 2 3)`
  - `> (append '(1) '(2 3))` $\Longrightarrow$ `'(1 2 3)`
  - `> (cons 1 '(2 3))` $\Longrightarrow$ `'(1 2 3)`
  - `> (cons 1 (cons 2 (cons 3 '())))` $\Longrightarrow$ `'(1 2 3)`
- **Ojo**: `'()` denota una lista vacía. Es equivalente al valor nulo en Scheme (`null`)

Pares y Listas

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
```



Operaciones sobre Listas

Operaciones sobre Listas

- `car`: Función que retorna el primer elemento de la lista
- `cdr`: Función que retorna el resto de la lista
- `> (car '(1 2 3))` $\Longrightarrow$ `1`
- `> (cdr '(1 2 3))` $\Longrightarrow$ `'(2 3)`
- `> (car(cdr '(1 2 3)))` $\Longrightarrow$ `2`
- `first` y `rest` son análogos a `car` y `cdr` respectivamente

Lambda y Let

Lambda

- Scheme nos permite un mecanismo para poder crear funciones anónimas de manera rápida
- `> ((lambda ( var1 ... varm ) exp1 ... expn) val1 ... valm)`
- Se aplican de igual forma que las funciones con nombre
- Nos permitirán, por ejemplo, pasar definiciones de funciones como parámetro a otras funciones

Let

- `let` nos permite definir variables que se ligan a un valor en la evaluación de expresiones
- `> (let (( var1 val1 ) ( var2 val2 ) ... ) exp1 exp2 ... )`
- Ojo, las variables que se definen en el let son solo visibles dentro de éste
- Por lo general, es útil para combinarlo con define

Ejemplo de Lamda y Let

Estos dos procedimientos realizan lo mismo:

```
(let ((x 3) (y 4))
     (+ (* x x) (* 2 (* x y))))
```

```
((lambda (x y)
     (+ (* x x) (* 2 (* x y))))
     3 4)
```

- Notar que `let` es nada más que una abreviación de `lambda`
- Con `define` ocurre algo similar; es una abrevación también de `lambda`

```
(define square (lambda (x) (* x x))))
```

- Lo anterior equivale a:

```
(define (square x) (* x x))
```

Condicionales: Sentencia if

Se escribe de la forma:

`(if condicion consecuencia alternativa)`

- **Ejemplo**:

```
(define (foo num)
    (if (> num 0)
        #t
        #f
    )
)
```

En palabras simples, ¿qué hace la función `foo`?

Condicionales: Sentencia if

Se escribe de la forma:

`(if condicion consecuencia alternativa)`

- **Ejemplo**:

```
(define (foo num)
    (if (> num 0)
        #t
        #f
    )
)
```

En palabras simples, ¿qué hace la función `foo`? Ve si un número es positivo

Condicionales: Sentencia cond

Se escribe de la forma:

```
(cond clausula1 clausula2 ...
(condicion expresion1 expresion2 ...)
...
(else expresion1 ...))
```

- **Ejemplo**:

```
(define (bar num)
    (cond
        ((eq? (modulo num 2) 0) #t)
        ((eq? (modulo num 3) 0) #t)
        ((eq? (modulo num 5) 0) #t)
    (else #f)
    )
)
```

Ejercicios

Ejercicio 1

Evalúe las siguientes expresiones:

1. `(first(car(rest(car(rest(car '((1(2 (3 4) (5 (6 7))))))))))`
2. `(list(cons 1 (list 2 3 4)) 6 (list 7 (cons (list 9) '())) 10)`

Ejercicio 1

Evalúe las siguientes expresiones:

1. `(first(car(rest(car(rest(car '((1(2 (3 4) (5 (6 7)))))))))))`
2. `(list(cons 1 (list 2 3 4)) 6 (list 7 (cons (list 9) '())) 10)`

Respuestas:

1. `3`
2. `((1 2 3 4) 6 (7 ((9))) 10)`

Ejercicio 2

Se debe implementar una función que realice lo siguiente: La función deberá aplicar dos funciones sobre un valor, si el valor es impar se evaluará dos veces la primera función con el valor y al resultado se le sumará lo obtenido al evaluar la segunda función una vez con el valor. Si el valor es par, se evaluará la segunda función dos veces sobre el valor y al resultado se le sumará lo obtenido al evaluar la primera función una vez con el valor.

```
>(aplicar (lambda(x) (* x x)) (lambda(x) (* 2 x)) 3)
87
```

```
>(aplicar (lambda(x) (* x x)) (lambda(x) (* 2 x)) 2)
12
```

Ejercicio 3

Realice una función que tome como parámetro un número natural `n` y que retorne el enésimo número de la *Sucesión de Lucas*

$$L_n = \begin{cases} 2 & n = 0 \\ 1 & n = 1 \\ L_{n-1} + L_{n-2} & n > 1 \end{cases}$$

```
>(lucas 5)
11

>(lucas 15)
1364
```