2. Segunda sesión: *Recursión*

2.1. Objetivos

- ⋆ Presentar un método para diseñar funciones.
- ⋆ Dar un repaso general al concepto de recursión.
- ⋆ Presentar las estructuras de datos par y lista.
- ⋆ Dar un repaso general al concepto de inducción estructural y su relación con la técnica de caza de patrones.
- ⋆ Definir qué es una `lambda` su relación con la aplicación de funciones.

2.2. Introducción

2.2.1. Diseño de funciones

Cuando se tiene que implementar una función que cumple cierto objetivo, es importante seguir los siguientes pasos, en este orden:

1. Entender lo que la función tiene que hacer.
2. Escribir la descripción de la función.
3. Escribir su contrato, o sea su tipo (cuántos parámetros, de qué tipo, qué retorna).
4. Escribir las pruebas unitarias asociadas a esta función sobre los distintos posibles datos de entradas que pueda tener (casos significativos).
5. Finalmente, implementar el cuerpo de la función.

```
Entender -> Descripción -> Contrato -> Pruebas unitarias ->
                                    Implementar
```

El contrato y la descripción se especifican en comentarios:

```
;; Función que calcula la potencia de una base b elevada
;; a un exponente b.
;; potencia: number number -> number
```

Escribir las pruebas nos obliga a entender, cómo se va a usar la función a definir. Para las pruebas usaremos la función `test`:

```
(test (<nombre> <parámetros>+) <resultado-esperado>)
```

```
> (test (potencia 2 0) 1)
> (test (potencia 2 1) 2)
> (test (potencia 2 3) 8)
```

2.2.2. Recursión

Para programar usando recursión se necesita pensar inductivamente, tal y como lo hacemos con los números naturales. Demostramos la propiedad para el caso base (el cero) y asumiendo que se cumple para un n, demostramos que es cierto para un $n+1$ $(s\,(n))$.

$$P(0)$$
$$P(n) \Rightarrow P\,(n+1)$$

Analizando nuestra función potencia, tenemos:

$b^e = b\,(b^{e-1})\,(b^{e-2})\,...(b^0)$

Por ejemplo:

$2^3 = 2\,(2^2)\,(2^1)\,(2^0)$, recordando que $2^0 = 1$ por definición

Es decir,

$b^e = b\,(b^{e-1})$

En sintaxis de Racket:

```
> (potencia 2 3) = (* 2 (potencia 2 2))
                 = (* 2 (* 2 (potencia 2 1)))
                 = (* 2 (* 2 (* 2 (potencia 0))))
                 = (* 2 (* 2 (* 2 1))))
                 = (* 2 (* 2 2))
                 = (* 2 4)
                 = 8
```

De este análisis, tenemos que el caso base es $2^0 = 1$ y el paso recursivo es: $b^e = b\,(b^{e-1})$:

```
(define (potencia b e)
    (if (zero? e)
        1 ; caso base
        (* b (potencia b (- e 1))))) ; paso recursivo
```

2.2.3. Estructuras de datos

La estructura más básica es el par que pega dos valores juntos.

```
> (cons 1 2)
```

```
> (car (cons 1 2))
```
Devuelve el primer elemento del par. Car := Contents of the Adress part of Register.

```
> (cdr (cons 1 2))
```
Devuelve el segundo elemento del par. Cdr := Contest of the Decrement part of Register.

```
> '(1 . 2)
```
Representación usando quote.

Ejemplo.

1. Definir una función `(suma u v)` que regresa la suma de dos pares.

```
(define (suma u v)
   (cons (+ (car u) (car v)) (+ (cdr u) (cdr v))))
```

2. Definir una función `(prod k u)` que regresa el producto por escalar del par `u` y `k`.

```
(define (prod k u)
   (cons (* k (car u)) (* k (cdr u))))
```

3. Define una función `(punto u v)` que regresa el producto punto de dos pares.

```
(define (punto u v)
   (+ (* (car u) (car v)) (* (cdr u) (cdr v))))
```

•

2.2.4. Listas

Muchas estructuras pueden ser construidas a partir de pares. Tal es el caso de las listas.

- ⋆ La lista vacía es una lista y se representa por `'()`, `null` o `empty`.
- ⋆ Si `x` es un elemento cualquiera y `xs` es una lista, entonces `(cons x xs)` es una lista. `x` es la cabeza y `xs` es el resto.

```
> '()

> empty

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

Hay varias formas de construir listas

- ⋆ Con `cons`.
- ⋆ Con `list`: `(list 1 2 3)`
- ⋆ Con `quote`: `'(1 2 3)`

`quote`: ‘‘Tómalo literal, no lo interpretes’’

```
> (1 2 3)
```
Sin `quote` *se entiende que el primer elemento es una función y se intenta evaluar... ¡Pero no!*

```
> '(1 2 3)
```
Quote toma el 1 literal y no lo interpreta.

Una lista con `quote` *es distinta a una con* `list` *pues esta sí evalúa:*

```
> '(1 2 (+ 1 2))
> (list 1 2 (+ 1 2))

> (define l1 '(1 2 3))

> (define l2 '(4 5 6))

> (append l1 l2)

> (car l1)

> (cdr l1)

> (car (cdr l1))
> (cadr l1)
> (caddr l1)
```
¡Azúcar sintáctica everywhere!

```
> (first l1)

> (second l1)

> (second (list 1 (+ 2 3) 3))

> (second '(1 (+ 2 3) 3))

> (fourth l1)

> (length l1)

> (list-ref l1 0)

> (reverse l1)

> (findf even? '(1 2 3 4 5 6))
```

En los lenguajes funcionales, las funciones son valores, es decir, una función podría recibir otra función como parámetro.

```
> (map add1 '(1 2 3))

> (filter even? '(1 2 3))
```

2.2.5. Inducción estructural

Generaliza la inducción a estructuras más complejas que los números naturales. Se tiene un caso base, pero en lugar de paso inductivo se tiene un patrón recursivo.

Para lograr la instrucción sobre estructuras, suele usarse la técnica de caza de patrones (*pattern matching*). La idea es definir un caso para cada constructor de la estructura, siguiendo así la gramática.

Para usar esta técnica, nos apoyamos de `match`:

```
(match <expresión>
   [<patrón> <expresión>]+)

(match (and #t #f)
   [#t "bien"]
   [#f "mal"])

(match 1
   [2 #t]
   [3 #f])
```

Si no coincide con ningún patrón devuelve error.

```
(match 1
   [2 #t]
   [3 #f]
   [else "No hay caso para ese número :("])
```

Se puede definir un caso por defecto con `else`.

Ejemplo. Define una función que devuelva la longitud de una lista.

```
;; Función que regresa la longitud de una lista.
;; longitud : list -> number
(define (longitud l)
   (match l
      ['() 0]
      [(cons x xs) (+ 1 (longitud xs))]))

(test (longitud '()) 0)
(test (longitud '(1 2 3 4)) 4)
```

•

Ejemplo. Define una función que determine si un elemento está contenido en una lista.

```
;; Predicado que determina si un elemento está contenido en una
;; lista.
;; contiene?: list -> boolean
(define (contiene? l e)
   (match l
      ['() #f]
      [(cons x xs) (or (equal? x e) (contiene? xs e))]))

(test (contiene? '() 5) #f)
(test (contiene? '(1 2 3 4) 4) #t)
(test (contiene? '(1 2 3 4) 5) #f)
```

•

Ejemplo. Define una función que devuelva la suma de los elementos de una lista.

```
;; Función que suma los elementos de la lista.
;; suma-elems: list -> number
(define (suma-elems l)
   (match l
      ['() 0]
      [(cons x xs) (+ x (suma-elems xs))]))

(test (suma-elems '()) 0)
(test (suma-elems '(1 2 3 4)) 10)
```

•

Ejemplo. Define una función que devuelva el producto de los elementos de una lista.

```
;; Función que multiplica los elementos de la lista.
;; mult-elems: list -> number
(define (mult-elems l)
   (match l
      ['() 1]
      [(cons x xs) (* x (mult-elems xs))]))

(test (mult-elems '()) 1)
(test (mult-elems '(1 2 3 4)) 24)
```

•

Las funciones de plegado `foldr` y `foldl` encapsulan estos patrones de recursión, permitiendo definir funciones sobre listas recibiendo un operador y un valor final como argumentos.

```
> (foldr + 0 '(1 2 3 4))

> (foldl + 0 '(1 2 3 4))

> (foldr * 1 '(1 2 3 4))

> (foldl * 1 '(1 2 3 4))
```

```
(define (foldr f v l)
   (match l
      ['() v]
      [(cons x xs) (f x (foldr f v xs))]))

(foldr + 0 '(1 2 3 4)) = (+ 1 (foldr + 0 '(2 3 4)))
                       = (+ 1 (+ 2 (foldr + 0 '(3 4))))
                       = (+ 1 (+ 2 (+ 3 (foldr + 0 '(4)))))
                       = (+ 1 (+ 2 (+ 3 (+ 4 (foldr + 0 '())))))
                       = (+ 1 (+ 2 (+ 3 (+ 4 0))))
                       = (+ 1 (+ 2 (+ 3 4)))
                       = (+ 1 (+ 2 7))
                       = (+ 1 9)
                       = 10

(define (foldl f v l)
   (match l
      ['() v]
      [(cons x xs) (foldl f (f x v) xs)]))

(foldl + 0 '(1 2 3 4)) = (foldl + (+ 1 0) '(2 3 4))
                       = (foldl + (+ 2 (+ 1 0)) '(3 4))
                       = (foldl + (+ 3 (+ 2 (+ 1 0))) '(4))
                       = (foldl + (+ 4 (+ 3 (+ 2 (+ 1 0)))) '())
                       = (+ 4 (+ 3 (+ 2 (+ 1 0))))
                       = (+ 4 (+ 3 (+ 2 1)))
                       = (+ 4 (+ 3 3))
                       = (+ 4 6)
                       = 10
```

2.2.6. Funciones anónimas

En Racket se pueden definir funciones anónimas usando la forma sintáctica `lambda`. Suelen usarse cuando usamos una función para un caso muy particular y no se necesite más adelante.

```
> (lambda (x) (+ x 124))
> (map (λ (x) (+ x 124)) '(1 2 3))
```
*Insert lambda o ctrl + *

`> (define (foo x) x)` ≡ `(define foo (λ (x) x))`
¡Azúcar sintáctica!

`> (foo 10)` ≡ `(let ([f (λ (x) x)]) (f 10))`
¡Azúcar sintáctica!

Para recursión se usa `letrec`

```
(letrec ([fact (lambda (n)
                         (if (< n 2)
                             1
                             (* n (fact (sub1 n))))))])
          (fact 5))
```

2.3. Ejercicios

1. Escribe una función recursiva `(longitud n)` que devuelva la longitud de un número entero. Ejemplos:

```
>(longitud 1)
1
>(longitud 17)
2
>(longitud 1729)
4
```

2. Definir la función `(norma u)` que devuelve la norma del par `u`.

3. Dados los siguientes ingredientes para preparar una hamburguesa:

Y la función `above` que pone un ingrediente sobre otro.

Indica qué regresan las siguientes llamadas a funciones:

a) 

b) 

4. Define una función anónima que determine si un número es múltiplo de 13. Úsala en combinación de `findf` para encontrar el primer múltiplo de 13 de una lista..

5. Redefine la función anterior con una expresión `let` y evalúa con 13.

2.4. Referencias

[1] Felleisen, M., Barski C., *Realm of Racket*, No Starch Press, First Edition.

[3] Tanter, Éric., *PrePLAI: Scheme y Programación Funcional*, 2011. Disponible en: http://users.dcc.uchile.cl/∼etanter/preplai/

[4] Flatt, Matthew., *The Racket Reference*.
Disponible en: https://docs.racket-lang.org/reference/index.html