Tema 3: Procedimientos recursivos

El coste de la recursión

La pila de la recursión

Supongamos la función mi-length. Las llamadas recursivas para (mi-lenth '(a b c d)) son:

```
(mi-length '(a b c d))
(+ 1 (mi-length '(b c d)))
(+ 1 (+ 1 (mi-length '(c d))))
(+ 1 (+ 1 (+ 1 (mi-length '(d)))))
(+ 1 (+ 1 (+ 1 (+ 1 (mi-length '())))))
(+ 1 (+ 1 (+ 1 (+ 1 0))))
(+ 1 (+ 1 (+ 1 1)))
(+ 1 (+ 1 2))
(+ 1 3)
4
```

Cada llamada a la recursión, deja **función en espera de ser evaluada** cuando la recursión devuelva un valor (suma). Estas llamadas en espera,se **almacenan en la pila de la recursión**.

Coste espacial de la recursión => Memoria consumida x fón para resolver problema. EJ: O(n)

Coste depende de nº llamadas a la recursión EJ: n

Soluciones al coste de la recursión:

Procesos iterativos (Uso de recursión por la cola)

- La **recursión** es **menos elegante**
- Necesita **parámetro adicional** donde **acumular resultados parciales**
- **No** dejan llamadas recursivas **en espera** ni se **incrementa la pila** de la recursión.
- En **cada llamada se hace un cálculo parcial** y en el **caso base** se devuelve **resultado**.

```
(define (mi-length lista)
        (mi-length-iter lista 0))
(define (mi-length-iter lista result)
        (if (null? lista) result
                (mi-length-iter (cdr lista) (+ result 1))))
```

Memoization

- Elegancia de los procesos recursivos y elegancia de los iterativos.
- **Guardar el valor** devuelto por la cada **llamada en alguna estructura** y **no volver a realizar** la llamada a la recursión las siguientes veces.
- Usamos **(get key dic**) para devolver el valor del dicc asociado a una clave.
- Usamos **(put key value dic)** para asociar un valor a una clave, la guarda en el dicc (*mutación*) y devuelve el valor.

```
(define (fib-memo n dic)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        ((not (equal? (get n dic) #f))
         (get n dic))
        (else (put n (+ (fib-memo (- n 1) dic)
                        (fib-memo (- n 2) dic)) dic))))
```

*** Para poder usar memoización, copiar los métodos implementados en el Tema 3 de teoría***

Recursión y gráficos de tortuga

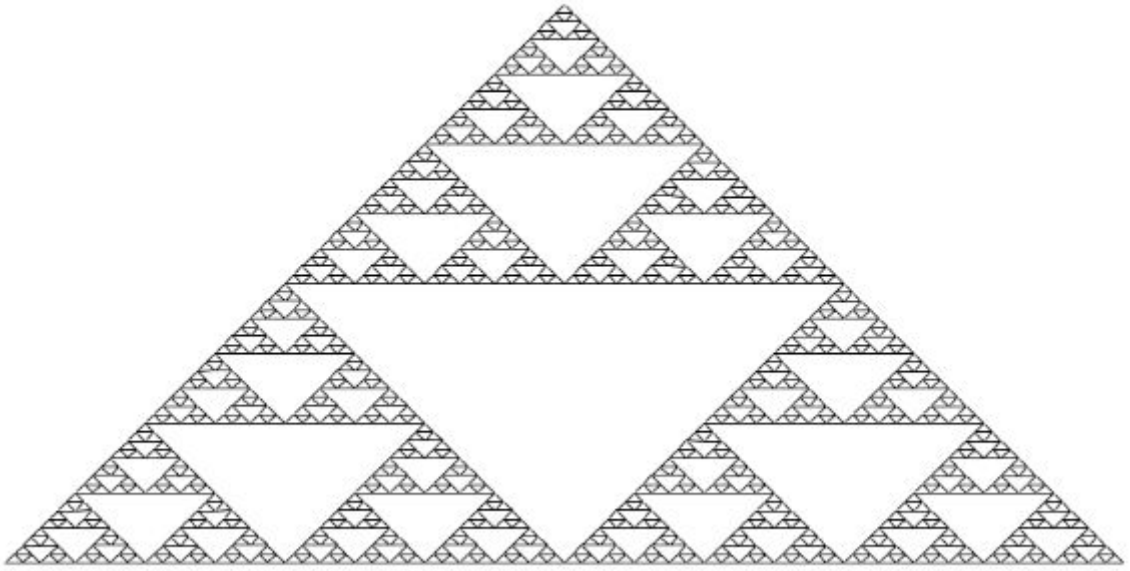
- Dibujar figuras fractales. Saliendo del paradigma funcional, dibujando los trazos de las figuras con pasos de ejecución secuenciales (**begin**).

*** Necesario cargar la librería:
#lang racket
(require graphics/turtles) ***

Comandos

- (turtles #t) => nueva ventana, coloca tortuga en centro mirando ejex (der.)
- (clear) => borra ventana y coloca tortuga en centro
- (draw d) => avanza la tortuga dibujando d píxeles
- (move d) => avanza la tortuga d píxeles
- (turn g) => gira la tortug a g grados (+ -> sentido antihorario/ - -> horario)

Ej: Sierpinski



```
#lang racket
(require graphics/turtles)

(turtles #t)

(define (hipot x)
    (* x (sqrt 2)))

(define (triangle w)
   (begin
      (draw w)
      (turn 135)
      (draw (hipot (/ w 2)))
      (turn 90)
      (draw (hipot (/ w 2)))
      (turn 135)))

(define (sierpinski w)
   (if (> w 20)
      (begin
         (sierpinski (/ w 2))
         (move (/ w 4)) (turn 90) (move (/ w 4)) (turn -90)
         (sierpinski (/ w 2))
         (turn -90) (move (/ w 4)) (turn 90) (move (/ w 4))
         (sierpinski (/ w 2))
         (turn 180) (move (/ w 2)) (turn -180)) ;; volvemos
      (triangle w)))
```

*** triangle figura base del triángulo Sierpinski. Tres triángulos recursivos de w/2 base ***

Recursión mutua

- Definición de **una función en base a una 2a**, que **a su vez** se define en **base a la 1a**.
- Debe haber un **caso base en ambas** que termine recursión.

Ej: Par funciones par? impar? que usen recursión mutua

Caso general?
-> x par si x-1 impar
-> x impar si x-1 par
Caso base?
-> 0 es par

Programas en Scheme:

```
1  (define (par? x)
2     (if (= 0 x)
3        #t
4        (impar? (- x 1))))
5
6  (define (impar? x)
7     (if (= 0 x)
8        #f
9        (par? (- x 1))))
```

Bibliografía

Los enunciados de los ejercicios resueltos, y los resúmenes, se han elaborado a partir del material publicado en https://domingogallardo.github.io/ , material del que es propietario el Departamento de Ciencia de la Computación e Inteligencia Artificial de la Universidad de Alicante, Domingo Gallardo, Cristina Pomares, Antonio Botía y Francisco Martínez.