Tema 4: Procedimientos y estructuras recursivas

Contenidos

Bibliografía

- Abelson y Sussman: Cap 1.2 (Procedures and the processes they generace), Introducción capítulo 2 (Building Abstraction with Data) pp. 79–89 y pp. 107–113

Objetivos

Semana 1

- Entender el diseño de funciones recursivas más avanzadas como las que dibujan el triángulo de Sierpinski o la curva de Hilbert
- Ser capaz de diseñar funciones recursivas, expresando una solución en un lenguaje declarativo e implementándolas después en Scheme
- Ser capaz de hacer un análisis de la evaluación de una función recursiva y de las distintas llamadas que se producen, entendiendo el concepto de pila de recursión y evaluación en espera
- Entender la diferencia entre un proceso recursivo e iterativo
- Ser capaz de diseñar una versión iterativa de un algoritmo recursivo
- Entender y ser capaz de utilizar la técnica de *memoization* para mejorar la eficiencia de las llamadas recursivas

1. Recursión

Ya hemos visto algunos ejemplos de funciones recursivas. Una función es recursiva cuando se llama a si misma. Una vez que uno se acostumbra a su uso, se comprueba que la recursión es una forma mucho más natural que la iteración de expresar un gran número de funciones y procedimientos.

Recordemos un ejemplo típico, el de longitud de una lista

```
(define (longitud lista)
   (if (null? lista)
      0
      (+ 1 (longitud (cdr lista)))))
```

La formulación matemática de la recursión es sencilla de entender, pero su implementación en un lenguaje de programación no lo es tanto. El primer lenguaje de programación que permitió el uso de expresiones recursivas fue el Lisp. En el momento de su creación existía ya el Fortran, que no permitía que una función se llamase a si misma.

En la clase de hoy veremos cómo diseñar procedimientos recursivos y cuál es el coste espacial y temporal de la recursión. Más adelante comprobaremos que no siempre una función recursiva tiene un comportamiento recursivo, sino que hay casos en los que genera un *proceso iterativo*.

En las siguientes clases del tema veremos que la recursión no sólo se utiliza para definir funciones y procedimientos sino que existen estructuras de datos cuya definición es recursiva, como las listas o los árboles.

1.1. Pensando recursivamente

Para diseñar procedimientos recursivos no vale intentarlo resolver por prueba y error. Hay que diseñar la solución recursiva desde el principio. Debemos fijarnos en *lo que devuelve la función* y debemos preguntarnos cómo sería posible descomponer el problema de forma que podamos lanzar la recursión sobre una versión más sencilla del mismo. Supondremos que la llamada recursiva funciona correctamente y devuelve el resultado correcto. Y después debemos transformar este resultado correcto de la versión más pequeña en el resultado de la solución completa.

Es muy importante escribir y pensar en las funciones de forma declarativa, teniendo en cuenta lo que hacen y no cómo lo hacen.

Debes **confiar en que la llamada recursiva va a hacer su trabajo y devolver el resultado correcto**, sin preocuparte de cómo lo va a hacer. Después tendrás que utilizar lo que la llamada recursiva ha devuelto para componer la solución definitiva al problema.

Para diseñar un algoritmo recursivo es útil no ponerse a programar directamente, sino reflexionar sobre la solución recursiva con algún ejemplo. El objetivo es obtener una formulación abstracta del caso general de la recursión antes de programarlo. Una vez que encontramos esta formulación, pasarlo a un lenguaje de programación es muy sencillo.

Por último, deberemos reflexionar en el caso base. Debe ser el caso más sencillo que puede recibir como parámetro la recursión. Debe devolver un valor compatible con la definición de la función. Por ejemplo, si la función debe construir una lista, el caso base debe devolver también una lista. Si la función construye una pareja, el caso base también devolverá una pareja. No debemos olvidar que el caso base es también un ejemplo de invocación de la

función.

1.1.1. Longitud de una lista

Empecemos con un sencillo ejemplo, la definición de la longitud de una lista. ¿Cómo definir la longitud de una lista en términos recursivos? Tenemos que pensar: “Si puedo calcular la longitud de una cadena más pequeña, ¿cómo puedo calcular la longitud de la cadena total?”.

Una posible definición del caso general de la recursión es esta: “Calculo la longitud de la lista sin el primer elemento con una llamada a la recursión, y le sumo 1 al número que devuelve esa llamada”.

Lo podríamos representar de la siguiente forma:

> Longitud (lista) = 1 + Longitud (resto (lista))

Y, por último, necesitamos que la función devuelva un valor concreto cuando llegue al caso base de la recursión. El caso base puede ser la lista vacía, que tiene una longitud de 0:

> Longitud (lista-vacía) = 0

La implementación en Scheme es:

```
(define (mi-length items)
    (if (null? items)
        0
        (+ 1 (length (cdr items)))))
```

1.1.2. Lista palíndroma

Veamos un ejemplo algo más complicado ¿cómo definimos una lista palíndroma de forma recursiva?. Por ejemplo, las siguientes listas son palíndromas:

```
'(1 2 3 3 2 1)
'(1 2 1)
'(1)
'()
```

Comenzamos con una definición **no recursiva**:

> Una lista es palíndroma cuando es igual a su inversa.

Esta definición no es recursiva porque no llamamos a la recursión con un caso más sencillo.

La definición **recursiva** del caso general es la siguiente:

> Una lista es palíndroma cuando su primer elemento es igual que el último y la lista resultante de quitar el primer y el último elemento también es palíndroma

En el caso base debemos buscar el caso más pequeño no contemplado por la definición anterior. En este caso, una lista de un elemento y una lista vacía también las consideraremos palíndromas.

> palindroma(lista) <=> (primer-elemento(lista) == ultimo-elemento(lista)) y palindroma(quitar-primero-ultimo(lista))
> palindroma(lista) <=> un-elemento(lista) o vacía(lista)

Vamos a escribirlo en Scheme:

```
(define (palindromo? lista)
    (or (null? lista)
        (null? (cdr lista))
        (and (equal? (car lista) (ultimo lista))
             (palindromo? (quitar-primero-ultimo lista)))))
```

La función auxiliar `quitar-primero-ultimo` la podemos definir así:

```
(define (quitar-ultimo lista)
    (if (null? (cdr lista))
        '()
        (cons (car lista)
              (quitar-ultimo (cdr lista)))))

(define (quitar-primero-ultimo lista)
    (cdr (quitar-ultimo lista)))
```

1.1.3. Gráficos de tortuga en Racket

Se pueden utilizar los gráficos de tortuga en Racket cargando la librería `graphics/turtles` : `(require graphics/turtles)` estando en el lenguaje *Muy Grande*.

```
(require graphics/turtles)
(turtles #t)
```

Los comandos más importantes:

- `(turtles #t)` : abre una ventana y coloca la tortuga en el centro, mirando hacia el eje X (derecha)
- `(clear)` : borra la ventana y coloca la tortuga en el centro
- `(draw d)` : avanza la tortuga dibujando *d* píxeles
- `(move d)` : mueve la tortuga *d* píxeles hacia adelante (sin dibujar)
- `(turn g)` : gira la tortuga *g* grados (positivos: en el sentido contrario a las agujas del reloj)

Prueba a realizar algunas figuras con los comandos de tortuga, antes de escribir el algoritmo

en Scheme del triángulo de Sierpinski.

Por ejemplo, podemos definir una función que dibuja un triángulo rectángulo con catetos de longitud `x`:

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

(define (triangulo-rectangulo x)
    (draw x)
    (turn 90)
    (draw x)
    (turn 135)
    (draw (hipot x))
    (turn 135))

(triangulo-rectangulo 100)
```

La función `(hipot x)` devuelve la longitud de la hipotenusa de un triángulo rectángulo con dos lados de longitud `x`. O sea, la expresión:

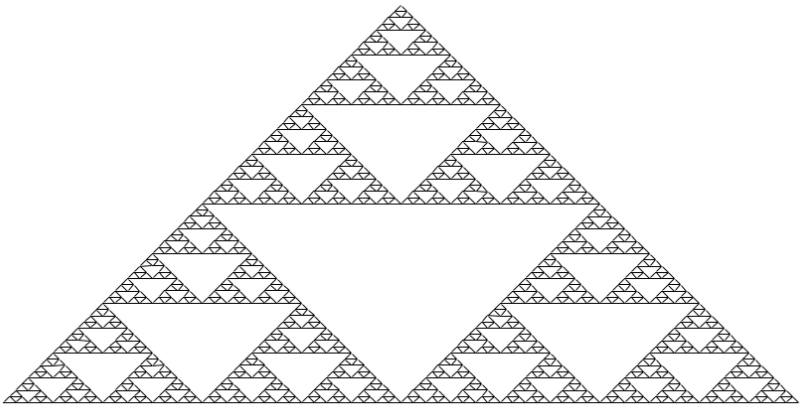
$$hipot(x) = \sqrt{x^2 + x^2} = x\sqrt{2}$$

Como puedes comprobar, el código es imperativo. Se basa en realizar una serie de pasos de ejecución que modifican el estado (posición y orientación) de la *tortuga*.

El siguiente código es una variante del anterior que dibuja un triángulo rectángulo de base `w` y lados `w/2`. Va a ser la figura base del triángulo de Sierpinski.

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

1.1.4. Triángulo de Sierpinski



Triángulo de Sierpinski

- ¿Ves alguna recursión en la figura?
- ¿Cuál podría ser el parámetro de la función que la dibujara?
- ¿Se te ocurre un algoritmo recursivo que la dibuje?

La figura es *autosimilar* (una característica de las figuras fractales). Una parte de la figura es idéntica a la figura total, pero reducida de escala. Esto nos da una pista de que es posible dibujar la figura con un algoritmo recursivo.

Para intentar encontrar una forma de enfocar el problema, vamos a pensarlo de la siguiente forma: supongamos que tenemos un triángulo de Sierpinski de anchura *h* y altura *h/2* con su esquina inferior izquierda en la posición 0,0. ¿Cómo podríamos construir **el siguiente** triángulo de Sierpinski?.

Podríamos construir un triángulo de Sierpinski más grande dibujando 3 veces el mismo triángulo, pero en distintas posiciones:

1. Triángulo 1 en la posición (0,0)
2. Triángulo 2 en la posición (h/2,h/2)
3. Triángulo 3 en la posición (h,0)

El algoritmo recursivo se basa en la misma idea, pero *hacia atrás*. Debemos intentar dibujar un triángulo de altura *h* situado en la posición *x*, *y* basándonos en 3 llamadas recursivas a triángulos más pequeños. En el caso base, cuando *h* sea menor que un umbral, dibujaremos un triángulo de lado *h* y altura *h/2*:

O sea, que para dibujar un triángulo de Sierpinski de base *h* y altura *h/2* debemos:

- Dibujar tres triángulos de Sierpinsky de la mitad del tamaño del original (*h/2*) situadas en las posiciones *(x,y)*, *(x+h/4, y+h/4)* y *(x+h/2,y)*
- En el caso base de la recursión, en el que *h* es menor que una constante, se dibuja un triángulo de base *h* y altura *h/2*.

Una versión del algoritmo en *pseudocódigo*:

```
Sierpinsky (x, y, h):
```

```
    if (h > MIN) {
        Sierpinsky (x, y, h/2)
        Sierpinsky (x+h/4, y+h/4, h/2)
        Sierpinsky (x+h/2, y, h/2)
    } else dibujaTriangulo (x, y, h)
```

1.1.5. Sierpinski en Racket

La siguiente es una versión imperativa del algoritmo que dibuja el triángulo de Sierpinski. No es funcional porque se realizan *pasos de ejecución*, usando la forma especial `begin` o múltiples instrucciones en una misma función (por ejemplo la función `triangle`).

```
(require graphics/turtles)
(turtles #t)

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

(define (triangle w)
    (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 a la posición original
        (triangle w)))
```

La llamada a

```
(sierpinski 40)
```

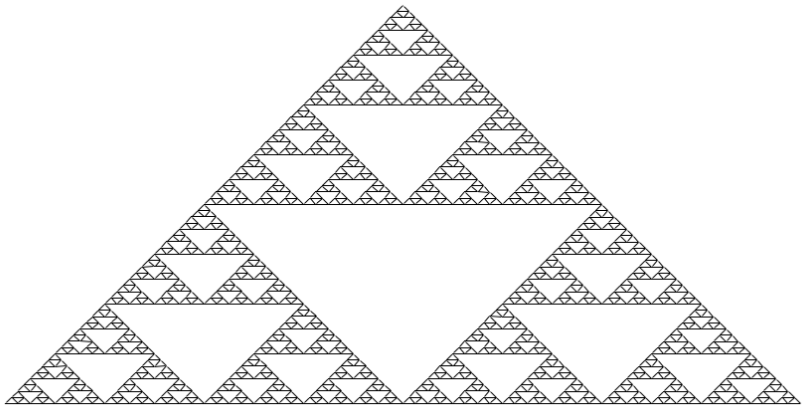
produce la siguiente figura:

(sierpinski 40)

La llamada a

```
(sierpinski 700)
```

Produce la figura que vimos al principio del apartado:



(sierpinski 700)

1.1.7. Recursión mutua

En la recursión mutua definimos una función en base a una segunda, que a su vez se define en base a la primera.

También debe haber un caso base que termine la recursión

Por ejemplo:

- x es par si x–1 es impar
- x es impar si x–1 es par
- 0 es par

Programas en Scheme:

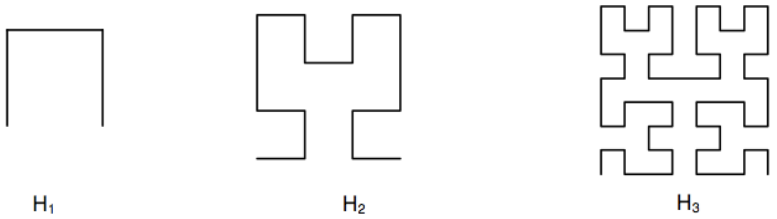
```
(define (par? x)
    (if (= 0 x)
        #t
        (impar? (- x 1))))

(define (impar? x)
    (if (= 0 x)
        #f
        (par? (- x 1))))
```

1.1.8. Ejemplo avanzado: curvas de Hilbert

La curva de Hilbert es una curva fractal que tiene la propiedad de rellenar completamente el espacio

Su dibujo tiene una formulación recursiva



Curva de Hilbert

La curva H3 se puede construir a partir de la curva H2. El algoritmo recursivo se formula dibujando la curva i-ésima a partir de la curva i–1.

Como en la curva de Sierpinsky, utilizamos la librería `graphics/turtles`, que permite usar la tortuga de Logo con los comandos de Logo `draw` y `turn`

La función `(h-der i w)` dibuja una curva de Hilbert de orden i con una longitud de trazo w a la *derecha* de la tortuga

La función `(h-izq i w)` dibuja una curva de Hilbert de orden i con una longitud de trazo w a la *izquierda* de la tortuga

Para dibujar una curva de Hilbert de orden i a la *derecha* de la tortuga:

```
1. Gira la tortuga -90
2. Dibuja una curva de orden i-1 a la izquierda
3. Avanza w dibujando
4. Gira 90
5. Dibuja una curva de orden i-1 a la derecha
6. Avanza w dibujando
7. Dibuja una curva de orden i-1 a la derecha
8. Gira 90
9. Avanza w dibujando
10. Dibuja una curva de orden i-1 a la izquierda
11. Gira -90
```

- El algoritmo para dibujar a la izquierda es simétrico

El algoritmo en Scheme:

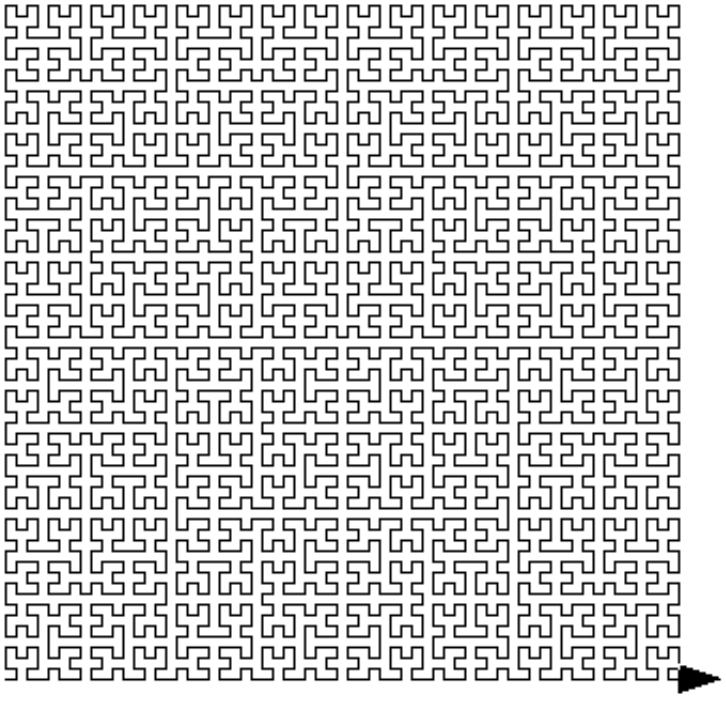
```
(require graphics/turtles)
(turtles #t)
(define (h-der i w)
    (if (> i 0)
        (begin
          (turn -90)
          (h-izq (- i 1) w)
          (draw  w)
          (turn 90)
```

```
          (h-der (- i 1) w)
          (draw w)
          (h-der (- i 1) w)
          (turn 90)
          (draw w)
          (h-izq (- i 1) w)
          (turn -90))))
(define (h-izq i w)
    (if (> i 0)
      (begin
        (turn 90)
        (h-der (- i 1) w)
        (draw  w)
        (turn -90)
        (h-izq (- i 1) w)
        (draw w)
        (h-izq (- i 1) w)
        (turn -90)
        (draw w)
        (h-der (- i 1) w)
        (turn 90))))
```

Podemos probarlo con distintos parámetros de grado de curva y longitud de trazo:

```
(clear)
(h-izq 3 20)
(h-izq 6 5)
```

El resultado de esta última llamada es:



Hilbert en Scheme

1.2. El coste de la recursión

1.2.1. La pila de la recursión

Vamos a estudiar el comportamiento del proceso generado por una llamada a un procedimiento recursivo. Supongamos la función `mi-length`:

```
(define (mi-length items)
  (if (null? items)
      0
      (+ 1 (mi-length (cdr items)))))
```

Examinamos cómo se evalúan las llamadas recursivas:

```
(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 una función en espera de ser evaluada cuando la recursión devuelva un valor (en el caso anterior el +). Esta función, junto con sus argumentos, se almacenan en la *pila de la recursión*.

Cuando la recursión devuelve un valor, los valores se recuperan de la pila, se realiza la llamada y se devuelve el valor a la anterior llamada en espera. Si la recursión está mal hecha y nunca termina se genera un stack overflow.

Es posible hacer que Racket haga una traza de la secuencia de llamadas a la recursión utilizando la librería `trace.ss`. Una vez cargada la librería puedes activar y desactivar las trazas de funciones específicas con `(trace <función>)` y `(untrace <función>)`. Debes tener activo el lenguaje *Muy Grande*.

Un pequeño problema de las trazas es que sólo se pueden tracear funciones definidas por el usuario, no se pueden tracear funciones primitivas de Scheme como `+` o `car`. Si queremos comprobar un ejemplo de traza en donde haya una llamada a una función primitiva, podemos definir una función propia que llame a la primitiva y tracear nuestra función.

Por ejemplo, vamos a tracear las llamadas recursivas a `mi-length` y a la suma:

```
(require (lib "trace.sss))
(define (suma x y) (+ x y))
(define (mi-length lista)
```

```
    (if (null? lista)
        0
        (suma 1 (mi-length (cdr lista)))))

(trace suma)
(trace mi-length)

(mi-length '(a b c d e f))

>(mi-length '(a b c d e f))
> (mi-length '(b c d e f))
> >(mi-length '(c d e f))
> > (mi-length '(d e f))
> > >(mi-length '(e f))
> > > (mi-length '(f))
> > > >(mi-length '())
< < < <0
> > > (suma 1 0)
< < < 1
> > >(suma 1 1)
< < <2
> > (suma 1 2)
< < 3
> >(suma 1 3)
< <4
> (suma 1 4)
< 5
>(suma 1 5)
<6
6
```

1.2.2. Coste espacial de la recursión

El coste espacial de un programa es una función que relaciona la memoria consumida por una llamada para resolver un problema con alguna variable que determina el tamaño del problema a resolver.

En el caso de la función mi-length el tamaño del problema viene dado por la longitud de la lista. El coste espacial de mi-lenght es O(n), siendo n la longitud de la lista.

1.2.3. El coste depende del número de llamadas a la recursión

Veamos con un ejemplo que el coste de las llamadas recursivas puede dispararse. Supongamos la famosa secuencia de Fibonacci: 0,1,1,2,3,5,8,13,…

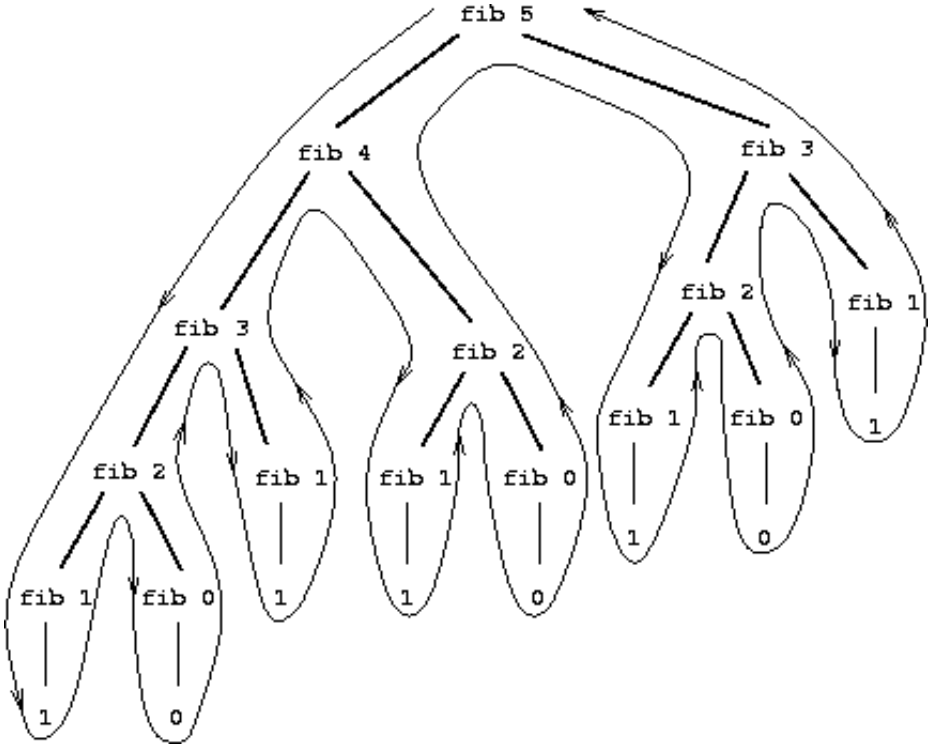
Formulación matemática de la secuencia de Fibonacci:

```
>Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
>Fibonacci(0) = 0
>Fibonacci(1) = 1
```

Formulación recursiva en Scheme:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Evaluación de una llamada a Fibonacci:



Llamada recursiva a Fibonacci

Cada llamada a la recursión produce otras dos llamadas, por lo que el número de llamadas finales es 2^n siendo n el número que se pasa a la función.

El coste espacial y temporal es exponencial, $O(2^n)$. ¿Qué pasa si intentamos evaluar (fibonaci 100)?

1.3. Soluciones al coste de la recursión: procesos iterativos

Diferenciamos entre procedimientos y procesos: un procedimiento es un algoritmo y un proceso es la ejecución de ese algoritmo.

Es posible definir procedimientos recursivos que generen procesos iterativos (como los bucles en programación imperativa) en los que no se dejen llamadas recursivas en espera ni se incremente la pila de la recursión. Para ello construimos la recursión de forma que en cada llamada se haga un cálculo parcial y en el caso base se pueda devolver directamente el resultado obtenido.

Este estilo de recursión se denomina *recursión por la cola* (tail recursion, en inglés).

Se puede realizar una implementación eficiente de la ejecución del proceso, eliminando la pila de la recursión.

1.3.1. Factorial iterativo

Es posible modificar la formulación de la recursión para se eviten las llamadas en espera:

- Definimos la función `(fact-iter-aux product n)` que es la que define el proceso iterativo
- Tiene un parámetro adicional (`product`) que es el parámetro en el que se irán guardando los cálculos intermedios
- Al final de la recursión el factorial debe estar calculado en `product` y se devuelve

```
(define (factorial-iter n)
   (fact-iter-aux n n))

(define (fact-iter-aux product n)
    (if (= n 1)
        product
        (fact-iter-aux (* product (- n 1))  (- n 1))))
```

Secuencia de llamadas:

```
(factorial-iter 4)
(factorial-iter-aux 4 4)
(factorial-iter-aux 12 3)
(factorial-iter-aux 24 2)
(factorial-iter-aux 24 1)
24
```

1.3.2. Versión iterativa de mi-length

¿Cómo sería la versión iterativa de mi-length?

Solución:

```
(define (mi-length-iter lista)
   (mi-length-iter-aux lista 0))

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

1.3.3. Procesos iterativos

- La recursión resultante es menos elegante

- Se necesita una parámetro adicional en el que se van acumulando los resultados parciales
- La última llamada a la recursión devuelve el valor acumulado
- El proceso resultante de la recursión es iterativo en el sentido de que no deja llamadas en espera ni incurre en coste espacial

1.3.4 Fibonacci iterativo

Cualquier programa recursivo se puede transformar en otro que genera un proceso iterativo.

En general, las versiones iterativas son menos intuitivas y más difíciles de entender y depurar.

Ejemplo: Fibonacci iterativo

```
(define (fib-iter n)
  (fib-iter-aux 1 0 n))

(define (fib-iter-aux a b count)
  (if (= count 0)
      b
      (fib-iter-aux (+ a b) a (- count 1))))
```

1.3.5. Triángulo de Pascal

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10   10  5   1
1   6  15  20   15  6   1
1   7  21  35   35  21  7   1
             ...
```

Formulación matemática:

> Pascal (0,0) = Pascal (1,0) = Pascal (1,1) = 1
> Pascal (fila, columna) = Pascal (fila–1,columna–1) + Pascal (fila–1, columna)

La versión recursiva pura:

```
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) (- col 1))
                 (pascal (- row 1) col) ))))
```

La versión iterativa:

```
(define (pascal-iter fila col)
   (list-ref (pascal-iter-aux '(1 1) fila) col))

(define (pascal-iter-aux fila n)
    (if (= n (length fila))
        fila
        (pascal-iter-aux (pascal-sig-fila fila) n)))

(define (pascal-sig-fila fila)
    (append '(1)
            (pascal-sig-fila-central fila)
            '(1)))

(define (pascal-sig-fila-central fila)
   (if (= 1 (length fila))
      '()
       (append (list (+ (car fila) (car (cdr fila))))
                   (pascal-sig-fila-central (cdr fila)))))
```

1.4. Soluciones al coste de la recursión: memoization

Una alternativa que mantiene la elegancia de los procesos recursivos y la eficiencia de los iterativos es la memoization. Si miramos la traza de (fibonacci 4) podemos ver que el coste está producido por la repetición de llamadas; por ejemplo (fibonacci 3) se evalúa 2 veces.

En programación funcional la llamada a (fibonacci 3) siempre va a devolver el mismo valor.

Podemos guardar el valor devuelto por la primera llamada en alguna estructura (una lista de asociación, por ejemplo) y no volver a realizar la llamada a la recursión las siguientes veces.

1.4.1. Fibonacci con memoization

Usamos los métodos procedurales `put` y `get` que implementan un diccionario *clave-valor* (para probarlos hay que utilizar el lenguaje R5RS):

```
(define lista (list '*table*))

(define (get key lista)
  (let ((record (assq key (cdr lista))))
    (if (not record)
    '()
    (cdr record))))

(define (put key value lista)
  (let ((record (assq key (cdr lista))))
    (if (not record)
    (set-cdr! lista
          (cons (cons key value)
            (cdr lista)))
```

```
    (set-cdr! record value)))
  'ok)
```

La función `(put key value lista)` asocia un valor a una clave y la guarda en la lista (con mutación).

La función `(get key lista)` devuelve el valor de la lista asociado a una clave.

Ejemplos:

```
(define mi-lista (list '*table*))
(put 1 10 mi-lista)
(get 1 mi-lista) -> 10
(get 2 mi-lista) -> '()
```

La función `fib-memo` realiza el cálculo de la serie de Fibonacci utilizando el proceso recursivo visto anteriormente y la técnica de memoización, en la que se consulta el valor de Fibonacci de la lista antes de realizar la llamada recursiva:

```
 (define (fib-memo n lista)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          ((not (null? (get n lista)))
           (get n lista))
          (else (let ((result
                       (+ (fib-memo (- n 1) lista)
                          (fib-memo (- n 2) lista))))
                  (begin
                    (put n result lista)
                    result)))))
```

Podemos comprobar la diferencia de tiempos de ejecución entre esta versión y la anterior. El coste de la función *memoizada* es O(n). Frente al coste $O(2^n)$ de la versión inicial que la hacía imposible de utilizar.

```
(define lista (list '*table*))
(fib-memo 200 lista) -> 280571172992510140037611932413038677189525
```
