Tema 4: Programación funcional avanzada en Scheme

Índice

1 Let.. 2
2 Let*.. 5
3 Letrec... 5
4 Quote...6
5 Eval.. 7
6 Datos compuestos: parejas... 8
7 Los datos compuestos pueden ser funciones..11
8 Listas.. 12
8.1 Funciones sobre listas..15
9 Referencias..17

Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

Vamos a continuar hoy viendo conceptos y construcciones nuevas del lenguaje de programación Scheme y relacionándolos con las características ya vistas de la programación funcional.

Recordemos que las formas especiales son construcciones de Scheme distintas de las funciones, con su propia semántica y forma de evaluación. Más adelante veremos que en Scheme es posible programar nuevas formas especiales mediante *macros*. Hasta ahora hemos visto las siguientes formas especiales:

- define
- if
- cond
- lambda

En este tema veremos las siguientes formas especiales

- let
- let*
- letrec
- quote
- eval

1. Let

La forma especial `let` tiene la siguiente sintaxis:

```
(let
  ((<ident-1> <exp-1>)
      ...
    (<ident-n> <exp-n>))
  <cuerpo>)
```

Esta forma especial permite definir unas variables locales, darles un valor y evaluar una expresión con ese valor definido. Veamos algunos ejemplos, y después definiremos más formalmente la semántica del let.

```
> (let ((x 4)
        (y 2))
   (* x y))
8
```

Vemos que el resultado es el esperado. Se definen las variables locales `x` e `y` con los valores 4 y 2 y se evalúa con esos valores la expresión del cuerpo, devolviéndose 8.

La utilización de `let` permite hacer los programas más legibles, ya que podemos definir

Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

algunos *conceptos previos* necesarios antes de la definición del cuerpo de la función. Hacemos esto en el siguiente ejemplo, en el que las variables `area-triangulo` y `apotema` toman como valor las funciones construidas por las expresiones lambda.

```
(define (area-hexagono lado)
   (let ((area-triangulo (lambda (base altura)
                            (/ (* base altura) 2)))
         (apotema (lambda (lado)
                     (* (sqrt 3) (/ lado 2)))))
      (* 6 (area-triangulo lado (apotema lado)))))
```

Las funciones `area-triangulo` y `apotema` son locales a la definición de `area-hexagono`.

¿Qué sucede fuera del ámbito del let? ¿Son realmente `x` e `y` variables locales que no afectan a una declaración externa? Vamos a comprobarlo.

```
> (define x 2)
> (let ((x 4)
        (y 2))
   (* x y))
8
> x
2
> y
error: reference to undefined identifier: y
```

Vemos que el valor de `x` en el exterior del let no ha cambiado y que la variable `y` sigue sin estar definida.

Veamos algún ejemplo más de qué se puede hacer y qué no se puede hacer con el let. Por ejemplo, en la siguiente expresión no se obtiene el resultado esperado, sino que se obtiene un error:

```
> (let ((x 3)
        (y (+ x 1)))
   (+ x y))
error: reference to undefined identifier: x
```

El error nos indica que en la segunda línea del let el identificador `x` no está definido. La lección que debemos aprender con este error es que en las definiciones del let no podemos referirnos a otras variables definidas en el propio let. Las definiciones de las variables locales no se hacen secuencialmente (recuerda: en la programación funcional no hay estado ni secuencia de instrucciones).

Un último ejemplo que da más pistas sobre cuál es la semántica del let:

```
> (define x 5)
> (let ((x 3)
```


Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

```
      (y (+ x 2)))
  (+ x y))
10
```

El valor de `x` en la segunda línea del let es 5 (el definido fuera del let) y no 3 (el definido en la primera línea del let).

Las siguientes reglas de evaluación del let explican todos estos ejemplos:

1. Evaluar todas las expresiones de la derecha de las variables y guardar sus valores en variables auxiliares locales.
2. Definir un ámbito local en el que se ligan las variables del let con los valores de las variables auxiliares.
3. Evaluar el cuerpo del let en el ámbito local

Parece muy difícil implementar toda esta semántica sin salirnos del paradigma funcional, pero es posible. Realmente el let es lo que se denomina *azúcar sintáctico*, una construcción meramente sintáctica que hace más sencillo utilizar el lenguaje pero que no proporciona ninguna funcionalidad nueva que no podamos implementar con lo que tenemos definido hasta ahora. Esto es, la forma especial let no proporciona ninguna funcionalidad semántica nueva. Vamos a verlo a continuación.

Implementación de la forma especial let con lambda

Para implementar la funcionalidad de la forma especial let con lambda hay que transformar la expresión general del let:

```
(let ((var1 exp1)
          ...
      (varn expn))
   <cuerpo>)
```

en la siguiente expresión:

```
((lambda (var1 ... varn)
    <cuerpo>) exp1 ... expn)
```

Si comprobamos despacio la evaluación de esta expresión, veremos que el resultado es el que pretendemos. La llamada interior a `(lambda (var1 .. varn) cuerpo)` construye un procedimiento con los argumentos `var1` ... `varn` (las variables del let) y el cuerpo del let. De esta forma conseguimos que las variables `var1` ... `varn` sean locales; declarándolas como argumentos formales de una función que construye el lambda. Los valores que toman las variables del let son l

Como ejemplo concreto, la expresión:

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


Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

```
     (* x y))
```

es equivalente a:

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

Al evaluar esta última expresión, la forma especial lambda construye un procedimiento con argumentos `x` e `y` y cuerpo `(* x y)`. El primer paréntesis provoca la evaluación de ese procedimiento pasándole como argumentos el valor 3 y el resultado de evaluar `(+ 3 2)`. Las siguientes expresiones muestran esta evaluación paso a paso:

```
((lambda (x y)
   (* x y)) 3 (+ 3 2)) ->
(#<procedure:(* x y)> 3 (+ 3 2)) ->
(#<procedure:(* x y)> 3 5) ->
(* 3 5) ->
15
```

2. Let*

La forma especial `let*` es una variante del `let` que sí permite la definición secuencial de valores a las variables locales.

Por ejemplo, la siguiente expresión devuelve un error

```
> (let ((a 0)
        (b (+ a 1))
        (c (+ c 1)))
     (list a b c))
error: reference to undefined identifier: a
```

Pero podemos realizar la definición secuencial utilizando `let*`:

```
> (let* ((a 0)
         (b (+ a 1))
         (c (+ c 1)))
     (list a b c))
(0 1 2)
```

Atención, pregunta
¿Cómo se podría implementar el let* utilizando el let?

3. Letrec

Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

Vamos a terminar esta lista de formas especiales relacionadas con let, introduciendo una última versión. Supongamos la siguiente expresión:

```
(let ((fact (lambda (x)
                (if (= x 0)
                    1
                    (* x (fact (- x 1)))))))
    (fact 3))
```

La variable local del let `fact` se liga al procedimiento que devuelve la expresión lambda, *en el que llamamos al propio procedimiento fact*. ¿Funcionará bien esta definición?

Si aplicamos la transformación definida previamente, obtenemos la siguiente expresión:

```
((lambda (fact)
   (fact 3)) (lambda (x) 1 (* x (fact (- x 1)))))
```

Si evaluamos esta expresión paso a paso obtenemos un error. Veámoslo:

```
((lambda (fact)
   (fact 3)) (lambda (x) 1 (* x (fact (- x 1))))) ->
(#<procedure:(fact 3)> (lambda (x) (* x (fact (- x 1))))) ->
(#<procedure:(fact 3)> #<procedure:(* x (fact (- x 1)))>) ->
(#<procedure:(* x (fact (- x 1)))> 3) ->
(* 3 (fact (- 3 1)))
```

El problema es que en la evaluación se pierde el nombre `fact` y en la llamada `(fact (- 3 1))` ese nombre no está ligado con ninguna función. De hecho, el mensaje de error que proporciona Scheme es:

```
error: reference to undefined identifier: fact
```

La solución está en la forma especial `letrec` que sí permite realizar esta llamada al propio nombre de variable:

```
(letrec ((fact (lambda (x)
                (if (= x 0)
                    1
                    (* x (fact (- x 1)))))))
    (fact 3))
6
```

No vamos a entrar en detalles en cómo se implementa internamente letrec, ya que es algo complicado. Lo dejamos para más adelante, cuando hablemos de macros.

4. Quote

Las dos formas especiales que veremos a continuación, quote y eval, tienen mucho que ver

Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

con el propio intérprete de Scheme. Hemos visto que la forma de interactuar con el intérprete (y la forma que éste utiliza para evaluar cualquier expresión) es la siguiente: escribimos una expresión, el intérprete la lee, la evalúa e imprime el resultado. Esto es lo que se denomina el ciclo *read-eval-print* del intérprete.

La forma especial `quote` precisamente le indica al intérprete que no debe evaluar la expresión que se pasa como argumento y que la debe devolver tal cual la recibe. Su sintaxis es:

```
(quote <expresion>)
```

Por ejemplo, veamos las siguientes expresiones y sus resultados:

```
> (quote 3)
3
> (quote hola)
hola
> (quote (1 2 3 4))
(1 2 3 4)
```

Una expresión equivalente a `quote` es la tilde ('). Las siguientes expresiones son equivalentes a las anteriores:

```
>'3
3
>'hola
hola
>'(1 2 3 4)
(1 2 3 4)
```

Realmente es incorrecto decir que el intérprete devuelve la expresión tal cual, ya que sí que la procesa (pero no la evalúa). Por ejemplo, en la siguiente expresión:

```
(define a '(1 2 3 4))
```

el intérprete debe convertir la expresión `(1 2 3 4)` en una lista de Scheme que liga a la variable `a`. Y una lista en Scheme es una estructura compleja que debe ser construida (más adelante veremos cómo).

5. Eval

La forma especial `eval` obliga al intérprete a evaluar la expresión que se pasa como parámetro y a devolver el resultado de la evaluación. La expresión que se le pasa por parámetro puede ser una expresión atómica (un símbolo o un valor) o puede ser una lista con una expresión correcta de Scheme que se desea evaluar.

Su sintaxis es:

Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

```
(eval <expresion>)
```

Veamos algunos ejemplos:

```
> (eval 3)
3
> (define x 2)
> (eval 'x)
2
> (eval '())
()
> (eval '(+ 3 4))
7
> (eval (list '+ 3 4))
7
> (eval (append '(+) '(3 4)))
7
```

Podemos comprobar en estos ejemplos otras de las características que hace de Scheme un lenguaje especial. Una expresión de Scheme (como `(+ 3 4)`) es una lista de identificadores que puede ser procesada tanto como datos (usando funciones como `append`) como código (llamando a `eval`).

Como curiosidad, terminamos presentando el siguiente código que implementa un intérprete de Scheme en Scheme utilizando el bucle read-eval-print:

```
(define (rep-loop)
   (display "mi-interprete> ")              ; imprime un prompt
   (let ((expr (read)))                     ; lee una expresión
      (cond ((eq? expr 'adios)              ; el usuario quiere parar?
             (display "saliendo del bucle read-eval-print")
             (newline))
            (else                           ; en otro caso
             (write (eval expr))            ; evaluar e imprimir
             (newline)
             (rep-loop)))))
```

6. Datos compuestos: parejas

En Scheme es posible crear datos compuestos. Como casi todo lo relacionado con Scheme, la forma de hacerlo es definiendo una construcción muy simple y usando esa construcción simple para hacer cosas más complicadas.

El tipo de dato compuesto (agregado) más simple es la pareja: una entidad formada por dos elementos. La forma de definir parejas en Scheme es con la función `cons`:

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


Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

La instrucción `cons` construye un dato compuesto a partir de otros dos datos (que llamaremos izquierdo y derecho). La expresión `(1 . 2)` que muestra el intérprete es la forma que tiene de imprimir las parejas.

Una vez definida una pareja, podemos obtener el elemento correspondiente a su parte izquierda con el operador `car` y su parte derecha con el operador `cdr`:

```
(define c (cons 1 2))
> (car c)
1
> (cdr c)
2
```

Las funciones `cons`, `car` y `cdr` quedan perfectamente definidas con las siguientes ecuaciones algebraicas:

```
(car (cons x y)) = x
(cdr (cons x y)) = y
```

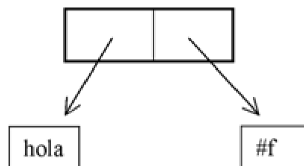
Nota:

¿De dónde vienen los nombres car y cdr? Realmente los nombres eran CAR y CDR (en mayúsculas). La historia se remonta al año 1959, en los orígenes del LISP y tiene que ver con el nombre que se les daba a ciertos registros de la memoria del IBM 709 (http://www.columbia.edu/acis/history/ibm709.html) .
Puedes encontrar una explicación completa en:
http://www.iwriteiam.nl/HaCAR_CDR.html
(http://www.iwriteiam.nl/HaCAR_CDR.html)

Es posible construir una pareja con cualquier tipo de datos, mezclando distintos tipos de datos:

```
> (define c (cons 'hola #f))
> (car c)
'hola
> (cdr c)
#f
```

Esta estructura se puede representar con el siguiente diagrama:



Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

Las flechas del diagrama indican (por ahora) contenido, no referencia. En el caso del ejemplo, el identificador "hola" y el booleano "#f" están contenidos en la parte izquierda y derecha de la pareja.

Esto quiere decir que seguimos en el paradigma funcional y que una vez creada una pareja no es posible modificar sus contenidos. Éstos quedan fijados para siempre a la pareja que se construye. Más adelante, cuando entremos en el paradigma imperativo, veremos que esto no es realmente así, ya que Scheme tiene primitivas para modificar el contenido de las parejas.

Por último, una pareja es también un tipo de datos de primer orden, que puede pasarse como argumento, devolverse como resultado de una función, e incluso utilizarse como parte de otra pareja.

Esta última propiedad es lo que se denomina *clausura* de la operación `cons`. El resultado de un `cons` puede usarse para construir otros *conses* (parejas). Por ejemplo, si realizamos las siguientes definiciones obtenemos una pareja formada a su vez por otras dos parejas:

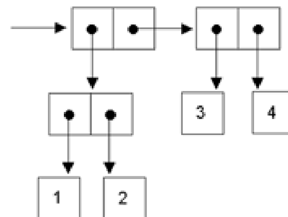
```
(define p1 (cons 1 2))
(define p2 (cons 3 4))
(define p (cons p1 p2))
```

Esto es equivalente a las siguientes instrucciones:

```
(define p (cons (cons 1 2)
          (cons 3 4)))
```

La utilización de las parejas como datos de primer orden que forman parte de otras parejas de lugar a la creación de estructuras compuestas más complicadas, pero también más útiles e interesantes.

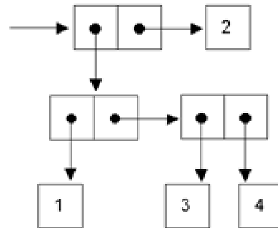
El diagrama que hemos presentado antes también puede usarse para estas estructuras. Por ejemplo, la estructura creada en el ejemplo anterior se representaría como sigue:



Estos diagramas se denominan en el Abelson & Sussman diagramas de *caja-y-puntero* (en inglés *box-and-pointer diagrams*).

Veamos otro ejemplo. ¿Qué secuencias de *conses* crearían el siguiente diagrama?

Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.



Vemos una pareja cuyo primer elemento es a su vez otra pareja formada por el número 1 y otra pareja formada por el número 3 y el número 4. La parte derecha de la pareja es el número 2:

```
(define p (cons (cons 1
                      (cons 3 4))
                2))
```

Una vez definida una estructura como la anterior, es posible que necesitemos obtener alguno de sus elementos. Para ello debemos utilizar las funciones `car` y `cdr` de forma anidada. Por ejemplo, para obtener el 3 de la estructura anterior debemos obtener su parte izquierda, después la parte derecha de ésta y por último la parte derecha:

```
> (car (cdr (car p)))
3
```

Como curiosidad, las llamadas anteriores son equivalentes a la instrucción `cadar` de Scheme.

```
>(cadar p)
3
```

Como regla mnemotécnica, el nombre de la función se obtiene concatenando a la letra "c", las letras "a" o "d" según hagamos un `car` o un `cdr` y terminando con la letra "r". Hay definidas 2^4 funciones de este tipo: `caaaar`, `caaadr`, ..., `cddddr`.

Por último, existe en Scheme una función con la que podemos consultar si un objeto es atómico o es una pareja: la función `pair?`

```
> (pair? 3)
#f
(pair? (cons 3 4))
#t
```

7. Los datos compuestos pueden ser funciones

En un tema anterior hemos comprobado que los programas en Scheme pueden también considerarse (y tratarse) como listas de identificadores, esto es, como datos.

Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

Vamos a ver ahora otra característica muy curiosa, que también distingue a Scheme del resto de lenguajes de programación. Los datos compuestos (parejas) pueden no ser una parte primitiva del lenguaje sino que pueden construirse como funciones que responden a los identificadores (*mensajes* en terminología de programación orientada a objetos) `'car` y `'cdr`.

Para ello podemos definir la función `mi-cons` como:

```
(define (mi-cons x y)
   (lambda (m)
      (cond ((equal? m 'car) x)
            ((equal? m 'cdr) y)
            (else (error "mensaje no definido: " m)) )))
```

La función anterior es equivalente a la función `cons` de Scheme. Para ello construye con la forma especial `lambda` un procedimiento que admite un argumento `m` (mensaje). Cuando a esta función construida se le pasa el mensaje `'car` devolverá el argumento `x` original de `mi-cons` y cuando se le pasa el mensaje `'cdr` devolverá el argumento `y`:

```
> (define p (mi-cons 'hola #f))
> (p 'car)
hola
> (p 'cdr)
#f
```

Hemos implementado una pareja mediante una función que se comporta igual. Lo único que nos falta es crear unas funciones equivalentes a `car` y `cdr` que *encapsulen* las llamadas a la función:

```
(define (mi-car pair)
   (pair 'car))

(define (mi-cdr pair)
   (pair 'cdr))
```

Ahora no hay ninguna diferencia entre el comportamiento de las funciones originales y el de las nuevas:

```
> (define p (mi-cons (mi-cons 1
                          (mi-cons 3 4))
            2))
> (mi-car (mi-cdr (mi-car p)))
3
```

8. Listas

Recordemos que Scheme permite manejar listas como un tipo de datos básico. Hemos visto funciones para crear, añadir y recorrer listas.

Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

Como repaso, podemos ver las siguientes expresiones:

```
(list 1 2 3 4)
'(1 2 3 4)

(define hola 1)
(define que 2)
(define tal 3)

(list hola que tal)
'(hola que tal)

(define a '(1 2 3))
(car a)
(cdr a)
(length a)
(length '())
(append '(1) '(2 3 4) '(5 6 7) '())
```

Ahora que hemos visto el tipo compuesto básico de Scheme, la pareja, podemos explicar cómo se definen las listas. Lo explicamos con las siguientes reglas recursivas:

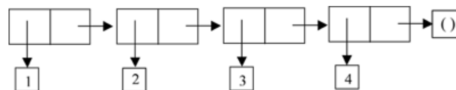
Una lista es:

- Una secuencia de parejas en las que el primer elemento es el dato y el segundo el resto de la lista.
- Un símbolo especial "'()" que denota la lista vacía

Por ejemplo, la lista '(1 2 3 4) se construye con la siguiente secuencia de parejas:

```
(cons (1
      (cons 2
            (cons 3
                  (cons 4
                        '()))))))
```

Esta estructura tiene el siguiente dibujo utilizando los diagramas caja y puntero:



Vamos a profundizar algo más. En primer lugar, veamos algunas consideraciones sobre la lista vacía. Se trata de un elemento especial que no es un símbolo (un identificador) ni una pareja, pero sí una lista. Se trata de un objeto atómico que denota una lista vacía.

```
> (symbol? '())
#f
> (pair? '())
#f
> (length '())
```


Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

```
0
```

La función `list?` nos permite comprobar si un dato es una lista:

```
> (list? '(1 2 3))
#t
> (list? '())
#t
> (list? 2)
#f
```

Para saber si un objeto es la lista vacía, podemos utilizar la función `null?`:

```
> (null? '())
#t
```

Ahora se entiende mejor por qué las funciones que permiten recorrer una lista son `car` y `cdr` como en el siguiente ejemplo:

```
(define lista (cons 1 (cons 2 (cons 3 (cons 4 '())))))
(car lista)
(cdr lista)
```

La función `car` nos devuelve el elemento izquierdo de la primera pareja, que es el primer elemento de la lista. La función `cdr`, aplicada a la primera pareja de una lista, nos devuelve su parte derecha, esto es, el resto de la lista.

También hay que hacer notar que la construcción de una lista devuelve la primera pareja de la misma. Así, tras evaluar la expresión

```
(define lista (list 1 2 3))
```

la variable `lista` tomará el valor de la primera pareja de la lista, la pareja formada por un 1 y el resto de la lista. Es lo mismo que cuando hacemos:

```
(define lista (cons 1
                    (cons 2
                          (cons 3 '()))))
```

Hay que hacer notar por lo tanto que a la función `cons` se le puede pasar un dato y otra lista (en realidad, la primera pareja de la lista) para formar una lista con un dato adicional:

```
> (define l1 '(1 2 3 4))
> (cons 'hola l1)
(hola 1 2 3 4)
```

Cuando Scheme imprime una estructura compuesta por parejas, va recorriendo la estructura intentando escribirla como una lista. Por ejemplo:

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


Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

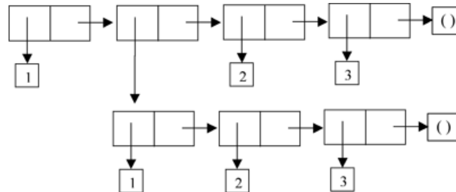
```
(1 2 3)
(cons 1 (cons 2 3))
(1 2 . 3)
```

Esta última estructura no es una lista, ya que su último elemento no es la lista vacía. En concreto, la parte derecha de la pareja "(cons 2 3)" es el número 3. Y en las listas todas las partes derechas de las parejas deben ser listas (o lista vacía).

Por último, ¿qué sucede si definimos como elemento de una lista otra lista? Podemos hacerlo de la siguiente forma:

```
(define l1 (list 1 2 3))
(define l2 (list 1 l1 2 3))
```

No habría ningún problema, ya que la estructura de parejas resultante seguiría cumpliendo la propiedad de las listas. La única diferencia con los ejemplos anteriores es que alguno de los elementos (el segundo, en concreto) no es atómico sino que es a su vez una pareja que sería el comienzo de otra lista:



La lista anterior también se puede definir con `quote`:

```
(define l2 '(1 (1 2 3) 2 3))
```

8.1. Funciones sobre listas

Una vez vista la estructura interna de una lista, es posible entender completamente el funcionamiento de funciones que construyen listas, como `append`. Recordemos la función construye una lista en la que se concatenan las listas que se pasan como argumento. Veamos como se podría implementar utilizando `cons`:

```
(define (mi-append l1 l2)
  (if (null? l1)
      l2
      (cons (car l1)
            (append (cdr l1) l2))))
```

Como ejercicio te sugerimos dibujar los diagramas caja-y-puntero de dos listas antes y después de llamar a la función `mi-append`. Verás que `append` construye la nueva lista creando tantas parejas nuevas como elementos de la primera lista y colocando en la última

Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

pareja construida la segunda lista. La segunda lista no se recorre, sino que directamente se coloca en la parte derecha de la última pareja construida.

Veamos algunas funciones más sobre listas, junto con su posible implementación

La función `length`:

```
> (length '(1 2 3 (4 5 6)))
4

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

La función `(list-ref n lista)` devuelve el elemento enésimo de una lista (empezando a contar por 0):

```
> (define lista '(1 2 3 4 5 6))
> (list-ref lista 3)
4

(define (mi-list-ref lista n)
   (cond
      ((null? lista)  (error "indice demasiado largo"))
      ((= n 0) (car lista))
      (else (mi-list-ref (cdr lista) (- n 1)))))
```

La función `(list-tail lista n)` devuelve la lista resultante de quitar n elementos de la lista original:

```
> (list-tail '(1 2 3 4 5 6 7) 2)
(3 4 5 6 7)

(define (mi-list-tail lista n)
   (cond
      ((null? lista)  (error "indice demasiado largo"))
      ((= n 0) lista)
      (else (mi-list-tail (cdr lista) (- n 1)))))
```

La función `map` aplica una función a todos los elementos de una lista y devuelve la lista resultante

```
(map abs '(-1 1 -3.4 -10))
(map (lambda (x) (* x x)) '(1 2 3 4 5))

(define (mi-map proc list)
   (if (null? list)
       '()
       (cons (proc (car list))
             (map proc (cdr list)))))
```


Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.

La función `reverse` invierte una lista

```
> (reverse '(1 2 3 4 5 6))
(6 5 4 3 2 1)

(define (mi-reverse l)
     (if (null? l) '()
      (append (reverse (cdr l)) (list (car l)))))
```

9. Referencias

Para saber más de los temas que hemos tratado en esta clase puedes consultar las siguientes referencias:

- Structure and Interpretation of Computer Programs (http://mitpress.mit.edu/sicp/full-text/book/book.html) , Abelson y Sussman, MIT Press 1996 (capítulo 2.2). Disponible biblioteca politécnica (acceso al catálogo (http://gaudi.ua.es/uhtbin/boletin/285815))
- Transparencias (http://www.cs.us.es/cursos/i1m-2002/teoria/tema-04.pdf) de Jose Antonio Alonso (http://www.cs.us.es/~jalonso/) de la asignatura Informática (http://www.cs.us.es/cursos/i1m/) del Departamento de Ciencias de Computación (http://www.cs.us.es/) de la Universidad de Sevilla.

Copyright © 2006 Depto. CCIA, Universidad de Alicante All rights reserved.