Tema 3: Programación funcional

Contenidos

Bibliografía

- SICP: Cap. 1.1.1–1.1.6, 1.3, 2.2 (2.2.1) 2.3.1
- PLP: Cap. 10
- Concepts in Programming Languages Cap. 4.4

Objetivos

Semana 2

- Ser capaz de construir y representar gráficamente estructuras de datos compuestas formadas por parejas
- Dada una estructura de parejas ser capaz de escribir expresiones en Scheme que obtengan sus elementos
- Entender la utilización de parejas para implementar las listas en Scheme
- Dada una estructura de parejas, identificar si se trata o no de una lista
- Entender e implementar funciones recursivas sobre listas
- Entender e implementar funciones que contengan un número variable de argumentos

3. Tipos de datos compuestos en Scheme

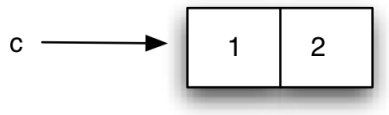
3.1. El tipo de dato pareja

Función de construcción de parejas `cons`

En Scheme el tipo de dato compuesto más simple es la pareja: una entidad formada por dos elementos. Se utiliza la función `cons` para construirla:

```
(cons 1 2) -> (1 . 2)
(define c (cons 1 2))
```

Dibujamos una pareja de la siguiente forma:



Tipo compuesto pareja

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

Funciones de acceso `car` y `cdr`

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

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

Definición declarativa

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

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

¿De dónde vienen los nombres `car` y `cdr`

Inicialmente 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.

Podemos leer la explicación completa en [The origin of CAR and CDR in LISP].

Las parejas pueden contener cualquier tipo de dato

Ya hemos comprobado que Scheme es un lenguaje *débilmente tipeado*. Las funciones pueden devolver y recibir distintos tipos de datos.

Por ejemplo, podríamos definir la siguiente función `suma` que sume tanto números como cadenas:

```
(define (suma x y)
  (cond
    ((and (number? x) (number? y)) (+ x y))
    ((and (string? x) (string? y)) (string-append x y))
    (else 'error)))
```

En la función anterior los parámetros `x` e `y` pueden ser números o cadenas (o incluso de cualquier otro tipo). Y el valor devuelto por la función será un número, una cadena o el símbolo `'error`.

Sucede lo mismo con el contenido de las parejas. Es posible guardar en las parejas cualquier tipo de dato y combinar distintos tipos. Por ejemplo:

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

Función pair?

La función `pair?` nos dice si un objeto es atómico o es una pareja:

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

Las parejas son objetos inmutables

Recordemos que en los paradigmas de programación declarativa y funcional no existe el *estado mutable*. Una vez declarado un valor, no se puede modificar. Esto debe suceder también con las parejas: una vez creada una pareja no se puede modificar su contenido.

En Lisp y Scheme estándar (R5RS) las parejas sí que pueden ser mutadas. Pero durante toda esta primera parte de la asignatura no lo contemplaremos, para no salirnos del paradigma funcional.

En Scala y otros lenguajes de programación es posible definir **estructuras de datos inmutables** que no pueden ser modificadas una vez creadas. Lo veremos también más adelante.

3.2. Las parejas son objetos de primera clase

En un lenguaje de programación un elemento es de primera clase cuando puede:

- Asignarse a variables
- Pasarse como argumento

- Devolverse por una función
- Guardarse en una estructura de datos mayor

Las parejas son objetos de primera clase.

Una pareja puede asignarse a una variable:

```
(define p1 (cons 1 2))
(define p2 (cons #f "hola"))
```

Una pareja puede pasarse como argumento y devolverse en una función:

```
(define (suma-parejas p1 p2)
    (cons (+ (car p1) (car p2))
          (+ (cdr p1) (cdr p2))))

(suma-parejas '(1 . 5) '(4 . 12))
⇒ (5 . 17)
```

Y, por último, las parejas *pueden formar parte de otras parejas*. Es lo que se denomina la propiedad de clausura de la función `cons`:

> El resultado de un `cons` puede usarse como parámetro de nuevas llamadas a `cons`.

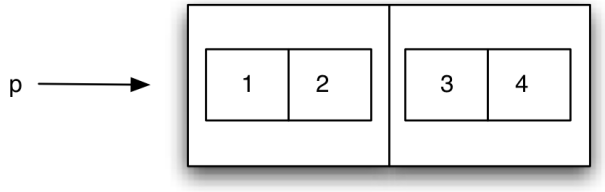
Ejemplo:

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

Expresión equivalente:

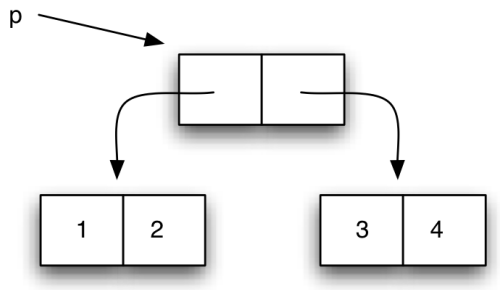
```
(define p (cons (cons 1 2)
                (cons 3 4)))
```

Podríamos representar esta estructura así:



Propiedad de clausura: las parejas pueden contener parejas

Pero se haría muy complicado representar muchos niveles de anidamiento. Por eso utilizamos la siguiente representación, entendiendo las flechas como *contiene* y no como referencia.



Las flechas denotan contenido

Llamamos a estos diagramas *diagramas caja-y-puntero* (*box-and-pointer* en inglés).

3.3. Diagramas *caja-y-puntero*

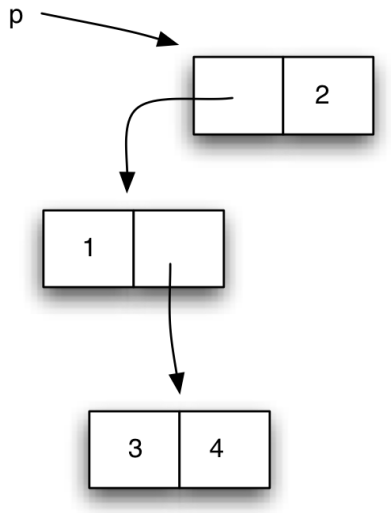
Al escribir expresiones complicadas con `cons` anidados es conveniente para mejorar su legibilidad utilizar el siguiente formato:

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

Para entender la construcción de estas estructuras es importante recordar que las expresiones se evalúan *de dentro a afuera*.

¿Qué figura representaría la estructura anterior?

Solución:



Es conveniente que pruebes a crear distintas estructuras de parejas con parejas y a dibujar su diagrama caja y puntero. Y también a recuperar un determinado dato (pareja o dato atómico) una vez creada la estructura.

La función `print-pareja` que vimos en el seminario de Scheme puede ser útil:

```
(define (print-pareja pareja)
    (if (pair? pareja)
            (begin
                    (display "(")
            (print-dato (car pareja))
            (display " . ")
            (print-dato (cdr pareja))
            (display ")"))))

(define (print-dato dato)
    (if (pair? dato)
        (print-pareja dato)
        (display dato)))
```

Repetimos lo mismo que dijimos entonces: **la función de arriba contiene sentencias como `begin` o `display` propias de la programación imperativa; no usarlas en programación funcional**.

Funciones c????r

Al trabajar con estructuras de parejas anidades es muy habitual realizar llamadas del tipo:

```
(cdr (cdr (car p)))
⇒ 4
```

Es equivalente a la función `cadar` de Scheme:

```
(cddar p)
⇒ 4
```

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`.

4. Listas en Scheme

4.1. Implementación de listas en Scheme

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

Como repaso, podemos ver las siguientes expresiones. Fijaros que las funciones `car`, `cdr` y `cons` son exactamente las mismas funciones que las vistas anteriormente.

¿Por qué? ¿Qué relación hay entre las parejas y las listas?

```
(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 '())
(cons 1 '(1 2 3 4))
(append '(1) '(2 3 4) '(5 6 7) '())
```

Ya debes haber descubierto la relación: en Scheme las listas se implementan con parejas.

Definición de listas con parejas

Una lista es (definición recursiva):

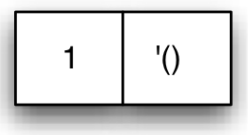
- Una pareja que contiene en su parte izquierda el primer elemento de la lista y en su parte derecha el resto de la lista
- Un símbolo especial "'()" que denota la lista vacía

Por ejemplo, una lista muy sencilla con un solo elemento, '(1), se define con la siguiente pareja:

```
(cons 1 '())
```

La pareja cumple las condiciones anteriores:

- La parte izquierda de la pareja es el primer elemento de la lista (el número 1)
- La parte derecha es el resto de la lista (la lista vacía)



La lista '(1)

El objeto es al mismo tiempo una pareja y una lista. La función `list?` permite comprobar si un objeto es una lista:

```
(define l (cons 1 '()))
```

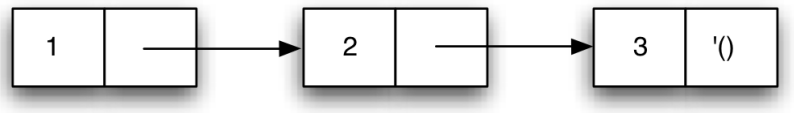
```
(pair? l)
(list? l)
```

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

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

La primera pareja cumple las condiciones de ser una lista:

- Su primer elemento es el 1
- Su parte derecha es la lista '(2 3 4)



Parejas formando una lista

Al comprobar la implementación de las listas en Scheme, entendemos por qué las funciones `car` y `cdr` nos devuelven el primer elemento y el resto de la lista.

Lista vacía

La lista vacía es una lista:

```
(list? '())
⇒ #t
```

Y no es un símbolo ni una pareja:

```
(symbol? '())
⇒ #f
(pair? '())
⇒ #f
```

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

```
(null? '())
⇒ #t
```

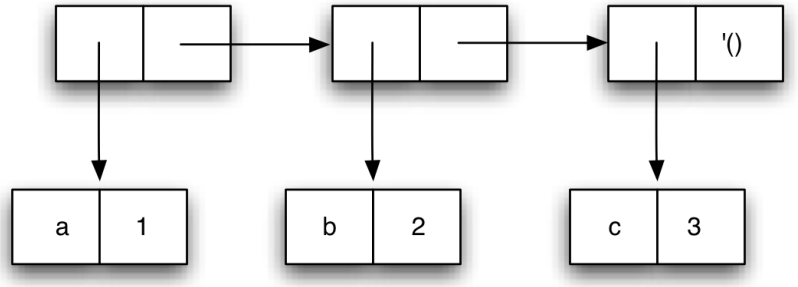
4.2. Listas con elementos compuestos

Las listas pueden contener cualquier tipo de elementos, incluyendo otras parejas.

La siguiente estructura se denomina *lista de asociación*. Son listas cuyos elementos son parejas (*clave*, *valor*):

```
(list (cons 'a 1)
      (cons 'b 2)
      (cons 'c 3))
⇒ ((a.1)(b.2)(c.2))
```

¿Cuál sería el diagrama *box and pointer* de la estructura anterior?



La expresión equivalente utilizando conses es:

```
(cons (cons 'a 1)
      (cons (cons 'b 2)
            (cons (cons 'c 3)
                  '())))
```

Listas de listas

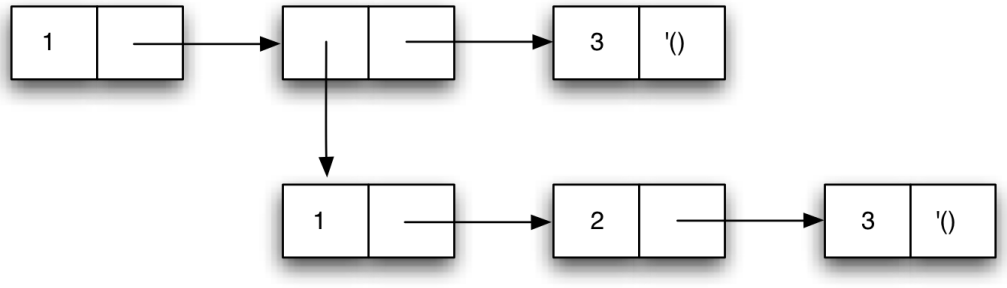
Si la pareja que guardamos como elemento de la lista es la cabeza de otra lista tenemos una lista que contiene a otra lista:

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

La lista anterior también se puede definir con quote:

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

El diagrama *box and pointer* de la lista es:



Lista que contiene otra lista como segundo elemento

Distintos niveles de abstracción

Es muy importante utilizar el nivel de abstracción correcto a la hora de trabajar con listas que contienen otros elementos compuestos, como otras parejas u otras listas.

Sólo hace falta *bajar* al nivel de caja y puntero cuando estemos definiendo funciones de bajo nivel que tratan la estructura de datos para obtener elementos concretos de las parejas.

Si, por el contrario, estamos recorriendo la lista principal y queremos tratar sus elementos, debemos *verla* como una lista normal y recorrerla con las funciones `car` para obtener su primer elemento y `cdr` para obtener el resto. O `list-ref` para obtener un elemento determinado (que puede ser atómico o compuesto).

Por ejemplo, la lista anterior `(1 (1 2 3) 3)` es una lista de 3 elementos. Si queremos obtener su segundo elemento (la lista `(1 2 3)`) bastaría con:

```
(define lista '(1 (1 2 3) 3))
(car (cdr lista))
(list-ref lista 1)
```

4.3. Funciones recursivas 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)
              (mi-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 `mi-append` construye la nueva lista creando tantas parejas nuevas como elementos de la primera lista y colocando en la última pareja construida una referencia a la segunda lista. La segunda lista no se recorre, sino que directamente se coloca en la parte derecha de la última pareja construida.

Veamos cómo implementar algunas funciones más sobre listas

`length`

La función `length`:

```
(length '(1 2 3 (4 5 6)))
⇒ 4
```

Una implementación:

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

`list-ref`

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
```

Una implementación:

```
(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)))))
```

`list-tail`

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)
```

Una implementación:

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

`reverse`

La función `reverse` invierte una lista

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

Una implementación:

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

4.4. Funciones con número variable de argumentos

Hemos visto algunas funciones primitivas de Scheme, como `+` o `max` que admiten un número variable de argumentos. ¿Podemos hacerlo también en las funciones definidas por nosotros?

La respuesta es sí, utilizando lo que se denomina notación *dotted-tail* (punto-cola) para definir los parámetros de la función. En esta notación se coloca un punto antes del último parámetro. Los parámetros antes del punto (si existen) tendrán como valores los argumentos usados en la llamada y el resto de argumentos se pasarán en forma de lista en el último parámetro.

Por ejemplo, si tenemos la definición

```
(define (f x y . z) <cuerpo>)
```

podemos llamar al procedimiento `f` con dos o más argumentos:

```
(f 1 2 3 4 5 6)
```

En la llamada, los parámetros `x` e `y` tomarán los valores 1 y 2. El parámetro `z` tomará como valor la lista `(3 4 5 6)`.

También es posible permitir que todos los argumentos sean opcionales:

```
(define (g . w) <cuerpo>)
```

Si hacemos la llamada

```
(g 1 2 3 4 5 6)
```

el parámetro `w` tomará como valor la lista `(1 2 3 4 5 6)`.

Lenguajes y Paradigmas de Programación, curso 2014–15
© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante

Domingo Gallardo, Cristina Pomares