Tema 5: Programación imperativa

Contenidos

5. Bibliografía

- Abelson y Sussman: Capítulo 3.3 (Modeling with Mutable Data, apartados 3.3.1–3.3.3)

1. Historia y características de la programación imperativa

1.1. Historia de la programación imperativa

1.1.1. Orígenes de la programación imperativa

- La programación imperativa es la forma natural de programar un computador, es el estilo de programación que se utiliza en el ensamblador, el estilo más cercano a la arquitectura del computador
- Características de la arquitectura arquitectura clásica de Von Newmann:
  - memoria donde se almacenan los datos (referenciables por su dirección de

memoria) y el programa
    - unidad de control que ejecuta las instrucciones del programa (contador del programa)
    - Los primeros lenguajes de programación (como el Fortran) son abstracciones del ensamblador y de esta arquitectura. Lenguajes más modernos como el BASIC o el C han continuado con esta idea.

1.1.2. Programación procedural

- Uso de procedimientos y subrutinas
- Los cambios de estado se localizan en estos procedimientos
- Los procedimientos especifican parámetros y valores devueltos (un primer paso hacia la abstracción y los modelos funcionales y declarativos)
- Primer lenguaje con estas ideas: ALGOL

1.1.3. Programación estructurada

- Artículo a finales de los 60 de Edsger W. Dijkstra: GOTO statement considered harmful en el que se arremete contra la sentencia GOTO de muchos lenguajes de programación de la época
- La programación estructurada mantiene la programación imperativa, pero haciendo énfasis en la necesidad de que los programas sean correctos (debe ser posible de comprobar formalmente los programas), modulares y mantenibles.
- Lenguajes: Pascal, ALGOL 68, Ada

1.1.4. Programación Orientada a Objetos

- La POO también utiliza la programación imperativa, aunque extiende los conceptos de modularidad, mantenibilidad y estado local
- Se populariza a finales de los 70 y principios de los 80

1.2. Características principales de la programación imperativa

- Idea principal de la programación imperativa: la computación se realiza cambiando el estado del programa por medio de sentencias que definen pasos de ejecución del computador
- Estado del programa modificable
- Sentencias de control que definen pasos de ejecución

Vamos a ver unos ejemplos de estas características usando el lenguaje de programación Java.

1.2.1. Modificación de datos

- Uno de los elementos de la arquitectura de Von Newmann es la existencia de celdas de memoria referenciables y modificables

```
int x = 0;
x = x + 1;
```

- Otro ejemplo típico de este concepto en los lenguajes de programación es el array: una estructura de datos que se almacena directamente en memoria y que puede ser accedido y modificado.
- En Java los arrays son tipeados, mutables y de tamaño fijo

```
String[] unoDosTres = {"uno","dos","tres"};
unoDosTres[0] = unoDosTres[2];
```

- Un ejemplo de un método que recibe un parámetro de tipo array en Java. El parámetro se pasa por referencia:

```
public void llenaCadenas(String[] cadenas, String cadena) {
    for (int i = 0; i < cadenas.length; i++) {
        cadenas[i] = cadena;
    }
}
```

- Todos los ejemplos anteriores en un programa `main` Java:

```
public class Main {

    public static void main(String[] args) {
        int x = 0;
        x = x + 1;
        System.out.println("x: " + x);
        String[] unoDosTres = {"uno","dos","tres"};
        unoDosTres[0] = unoDosTres[2];
        for (String valor : unoDosTres) {
            System.out.println(valor);
        }
        llenaCadenas(unoDosTres, "uno");
        for (String valor : unoDosTres) {
            System.out.println(valor);
        }
    }

    public static void llenaCadenas(String[] cadenas, String cadena) {
        for (int i = 0; i < cadenas.length; i++) {
            cadenas[i] = cadena;
        }
    }
}
```

1.2.2. Almacenamiento de datos en variables

- Todos los lenguajes de programación definen variables que contienen datos
- Las variables pueden mantener valores (tipos de valor o value types) o referencias (tipos de referencia o reference types)
- En C, C++ o Java, los datos primitivos como int o char son de tipo valor y los objetos y datos compuestos son de tipo referencia
- La asignación de un valor a una variable tiene implicaciones distintas si el tipo es de valor (se copia el valor) o de referencia (se copia la referencia)

Copia de valor (datos primitivos en Java):

```
int x = 10;
int y = x;
x = 20;
System.out.println(y); // Sigue siendo 10
```

Copia de referencia (objetos en Java):

```
// import java.awt.geom.Point2D;
Point2D p1 = new Point2D.Double(2.0, 3.0);
Point2D p2 = p1;
p1.setLocation(12.0, 13.0);
System.out.println("p2.x = " + p2.getX()); // 12.0
System.out.println("p2.y = " + p2.getY()); // 13.0
```

- El uso de las referencias para los objetos de clases y para los tipos compuestos está generalizado en la mayoría de lenguajes de programación
- Tiene efectos laterales pero permite obtener estructuras de datos eficientes

1.2.3. Igualdad de valor y de referencia

- Todos los lenguajes de programación imperativos que permite la distinción entre valores y referencias implementan dos tipos de igualdad entre variables
- Igualdad de valor (el contenido de los datos de las variables es el mismo)
- Igualdad de referencia (las variables tienen la misma referencia)
- Igualdad de referencia => Igualdad de valor (pero al revés no)
- En Java la igualdad de referencia se define con `==` y la de valor con el método `equals`:

```
Point2D p1 = new Point2D.Double(2.0, 3.0);
Point2D p2 = p1;
Point2D p3 = new Point2D.Double(2.0, 3.0);
```

```
System.out.println(p1==p2);          // true
System.out.println(p1==p3);          // false
System.out.println(p1.equals(p3));   // true
```

1.2.4. Sentencias de control

- También tiene su origen en la arquitectura de Von Newmann
- Sentencia que modifica el contador de programa y determina cuál será la siguiente instrucción a ejecutar
- Tipos de sentencias de control en programación estructurada:
  - Las sentencias de secuencia definen instrucciones que son ejecutados una detrás de otra de
    forma síncrona. Una instrucción no comienza hasta que la anterior ha terminado.
  - Las sentencias de selección definen una o más condiciones que determinan las instrucciones
    que se deberán ejecutar.
  - Las sentencias de iteración definen instrucciones que se ejecutan de forma repetitiva hasta
    que se cumple una determinada condición.

Bucles y variables

Ejemplo imperativo que imprime en Java una tabla con los productos de los números del 1 al 9:

```
for (int i = 1; i <= 9 ; i++) {
    System.out.println("Tabla del " + i);
    System.out.println("-----------");
    for (int j = 1; j <= 9; j++) {
        System.out.println(i + " * " + j + " = " + i * j);
    }
    System.out.println();
}
```

2. Programación imperativa en Scheme

- Al igual que LISP, Scheme tiene características imperativas
- Vamos a ver algunas de ellas
  - Pasos de ejecución
  - Asignación con la forma especial `set!`
  - Datos mutables con las formas especiales `set-car!` y `set-cdr!`

- Una nota importante: todos los ejemplos que hay a continuación están escritos en el lenguaje *Scheme R5RS*

2.1. Pasos de ejecución

- Es posible definir pasos de ejecución con la forma especial `begin`
- Todas las sentencias de la forma especial se ejecutan de forma secuencial, una tras otra
- Tanto en la forma especial `let` como en `lambda` y `define` es posible definir cuerpos de función con múltiples sentencias que se ejecutan también de forma secuencial

Ejemplo `begin`:

```
(begin
    (display "Escribe un número: ")
    (define x (read))
    (display "Escribe otro: ")
    (define y (read))
    (define maximo (max x y))
    (display (string-append "El máximo de "
                            (number->string x)
                            " y "
                            (number->string y)
                            " es "
                            (number->string maximo))))
```

Ejemplo `define`:

```
(define (display-tres-valores a b c)
   (display a)
   (newline)
   (display b)
   (newline)
   (display c)
   (newline))
```

2.2. Mutación con formas especiales *set!*

2.2.1. Forma especial `set!`

- La forma especial `set!` permite asignar un nuevo valor a una variable
- La variable debe haber sido previamente creada con `define`
- La forma especial no devuelve ningún valor, modifica el valor de la variable usada

Sintaxis:

```
(set! <variable> <nuevo-valor>)
```

Por ejemplo, la típica asignación de los lenguajes imperativos se puede realizar de esta forma

en Scheme:

```
(define a 10)
(set! a (+ a 1))
a   ;;-> 11
```

Ejemplo (usando `let`):

```
(define a '(1 2 3 4))
(define b '(hola adios))
(let ((aux a))
    (set! a b)
    (set! b aux)))
```

2.2.2. Datos mutables

- En Scheme se definen las formas especiales `set-car!` y `set-cdr!` que permite modificar (mutar) la parte izquierda o derecha de una pareja una vez creada
- Al igual que `set!`, no devuelven ningún valor

Sintaxis:

```
(set-car! <pareja> <nuevo-valor>)
(set-cdr! <pareja> <nuevo-valor>)
```

Ejemplo

```
(define p (cons 1 2))
(set-car! p 10)
(set-cdr! p 20)
p   ;;-> (10 . 20)
```

2.2.3. Efectos laterales

- La introducción de la asignación y los datos mutables hace posible que Scheme se comporte como un lenguaje imperativo en el que más de una variable apunta a un mismo valor y se producen efectos laterales
- Un efecto lateral se produce cuando el valor de una variable cambia debido a una sentencia en la que no aparece la variable

Ejemplo:

```
(define a (cons 1 2))
(define b a)
```

```
(car b)
;; 1
(set-car! a 20)
(car b)
;; 20
```

2.3. Igualdad de referencia y de valor

- La utilización de referencias, la mutación y los efectos laterales hace también necesario definir dos tipos de igualdades: igualdad de referencia e igualdad de valor.
- Igualdad de referencia: dos variables son iguales cuando apuntan al mismo valor
- Igualdad de valor: dos variables son iguales cuando contienen el mismo valor
- En Scheme la función `eq?` comprueba la igualdad de referencia y `equal?` la igualdad de valor
- Igualdad de referencia implica igualdad de valor, pero no al revés

Ejemplo:

```
(define z1 '(a b))
(define z2 '(a b))
(define z3 z1)
(equal? z1 z2) ;;--> #t
(eq? z1 z2) ;;--> #f
(equal? z2 z3) ;;--> #t
(eq? z2 z3)  ;;--> #f
```

3. Estructuras de datos mutables

- La utilización de las formas especiales `set-car!` y `set-cdr!` permite un estilo nuevo de manejo de las estructuras de datos ya vistas (listas o árboles)
- Es posible implementar funciones más eficientes que actualizan la estructura modificando directamente las referencias de unas celdas a otras
- Las operaciones no construyen estructuras nuevas, sino que modifican la ya existente

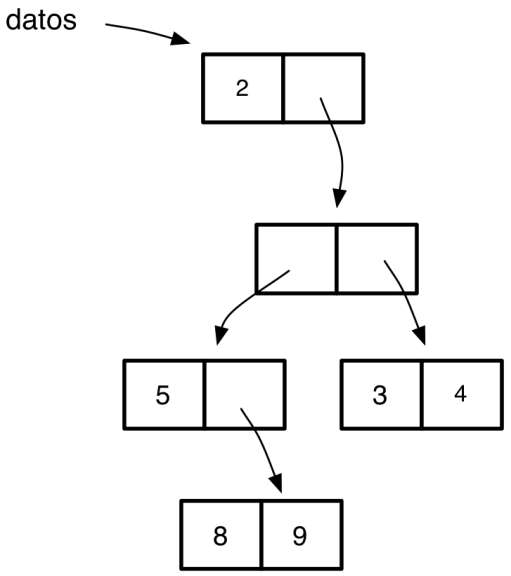
3.1. Mutación de elementos

Vamos a empezar con un ejemplo sencillo en el que vamos a mutar un elemento de una estructura de datos formada por parejas. Supongamos la siguiente estructura:

```
(define datos (cons 2
                    (cons (cons 5
                                (cons 8 9))
                          (cons 3 4))))
```

¿Cuál sería el diagrama *box-and-pointer*? Dibújalo. Fíjate que hay elementos de las parejas

que son datos atómicos y otros que son referencias a otras parejas.



Vamos ahora a *mutar* la estructura utilizando las sentencias `set-car!` y `set-cdr!`. Para mutar una pareja acceder a la pareja y modificar su parte derecha o su parte izquierda con las sentencias anteriores.

Por ejemplo, ¿cómo cambiaríamos el `8` por un `18`? Deberíamos obtener la pareja `(8 . 9)` que está al final de la estructura y modificar su parte izquierda:

```
(set-car! (cdr (car (cdr datos))) 18)
```

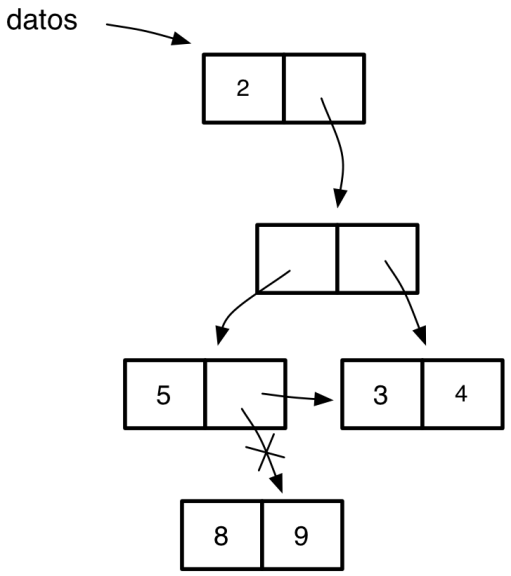
La expresión `(cdr (car (cdr datos)))` devuelve la pareja que queremos modificar y la sentencia `set-car!` modifica su parte izquierda.

Si ahora vemos qué hay en `datos` veremos que se ha modificado la estructura:

```
(print-pareja datos)
//=> (2 . ((5 . (18 . 9)) . (3 . 4)))
```

Recuerda que `print-pareja` es una función que vimos en el tema 3.2.

Hemos mutado un dato por otro. También podemos mutar las referencias a las parejas. Por ejemplo, podríamos modificar la parte derecha de la pareja que contiene el 5 para que apunte a la pareja `(3 . 4)`:



Para ello habría que obtener la pareja que contiene el 5 con la expresión `(car (cdr datos))` y mutar su parte derecha (`set-cdr!`) con la referencia a la pareja `(3 . 4)` que se obtiene con la expresión `(cdr (cdr datos))` :

```
(set-cdr! (car (cdr datos)) (cdr (cdr datos)))
```

3.2. Funciones mutadoras: `append!`

- Normalmente las funciones mutadoras no devuelven una estructura, sino que modifican la que se pasa como parámetro
- Por convenio, indicaremos que una función es mutadora terminando su nombre con un signo de admiración

Como ejemplo inicial, la siguiente función es la versión mutadora de `append` .
La llamamos `append!` :

```
(define (append! l1 l2)
  (if (null? (cdr l1))
      (set-cdr! l1 l2)
      (append! (cdr l1) l2)))

(define a '(1 2 3 4))
(define b '(5 6 7))
(append! a b)
a ;-> (1 2 3 4 5 6 7)
```

Algunas puntualizaciones:

- Al igual que `set!` , `set-car!` o `set-cdr!` , la función `append!` no devuelve ningún valor, sino que modifica directamente la lista que se pasa como primer parámetro
- Al modificarse la lista, todas las referencias que apuntan a ellas quedan también

modificadas
- La función daría un error en el caso en que la llamáramos con una lista vacía como primer argumento

3.3. Lista ordenada mutable

Vamos a presentar un tipo de dato mutable completo, una lista ordenada.

Barrera de abstracción

Constructor

```
(define (make-olist)
  (list '*olist*))
```

Hay que hacer notar que el constructor `make-olist` devuelve una pareja que hace de cabecera de la lista y contiene en su parte izquierda el símbolo `*olist*`. Este símbolo es un convenio y podríamos sustituirlo por cualquier otro.

Selectores

```
(define (empty-olist? olist)
  (null? (cdr olist)))

(define (first-olist olist)
  (cadr olist))

(define (rest-olist olist)
  (cdr olist))
```

Mutadores

Las funciones mutadoras modifican la estructura de datos.

Definimos la función mutadora `insert!` que modifica añade una nueva pareja a la lista, insertándola en la posición correcta modificando las referencias

La cabecera de la lista sirven para anclar la lista y definir una referencia inmutable a la misma, a la que las variables pueden apuntar. De esta forma podremos insertar elementos en primera posición de la lista.

La función `(add-item! item ref)` es la función clave que crea una nueva pareja con el `item` y la añade en el `cdr` de la pareja a la que apunta `ref`.

```
(define (add-item! item ref)
  (set-cdr! ref (cons item (cdr ref))))
```

```
(define (insert-olist! n ref)
  (cond
    ((null? (cdr ref)) (add-item! n ref))
    ((< n (cadr ref)) (add-item! n ref))
    ((= n (cadr ref)) #f)   ; el valor devuelto no importa
    (else (insert-olist! n (cdr ref))) ))
```

Ejemplo de uso:

```
(define c (make-olist))
(insert-olist! 5 c)
(insert-olist! 8 c)
```

3.4. Tabla hash mutable

- Veamos ahora un ejemplo más: una tabla hash definida mediante una lista de asociación formada por parejas de clave y valor

```
(define l-assoc (list (cons 'a 1) (cons 'b 2) (cons 'c 3)))
```

- La función de Scheme `assq` recorre la lista de asociación y devuelve la tupla que contiene el dato que se pasa como parámetro como clave

```
(assq 'a l-assoc) --> (a.1)
(assq 'b l-assoc) --> (b.2)
(assq 'c l-assoc) --> (c.3)
(assq 'd l-assoc) --> #f
(cdr (assq 'c l-assoc)) --> 3
```

- La función `assq` busca en la lista de asociación usando la igualdad de referencia `eq?`

```
(define h "hola")
(define l '(1 2))
(define l-assoc (list (cons h 1) (cons l 2) (cons 'c 3)))
(assq "hola" l-assoc) --> #f
(assq h l-assoc) --> ("hola".1)
```

- Para la tabla hash definimos un constructor que define una cabecera de la tabla:

```
(define (make-table)
   (list '*table*))
```

- Y las funciones `get` y `put`:

```
(define (get key table)
  (let ((record (assq key (cdr table))))
    (if (not record)
      #f
      (cdr record))))

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

Ejemplos de uso:

```
(define tab (make-table))
(put 'a 10 tab)
(get 'a tab)
(put 'b '(a b c) tab)
(get 'b tab)
(put 'a 'ardilla tab)
(get 'a tab)
```

3.5. Ejemplo de mutación con listas de asociación

Una vez introducidos distintas estructuras de datos mutables, incluyendo listas de asociación, vamos a terminar estos ejemplos con un ejemplo práctico en el que intervienen las listas y las listas de asociación. Se trata de escribir un procedimiento `regular->assoc!` que transforme una lista regular en una lista de asociación sin crear nuevas parejas. La lista regular `(k1 v1 k2 v2 k3 v3 ...)` deberá convertirse en la lista de asociación `((k1 . v1) (k2 . v2) (k3 . v3) ...)`.

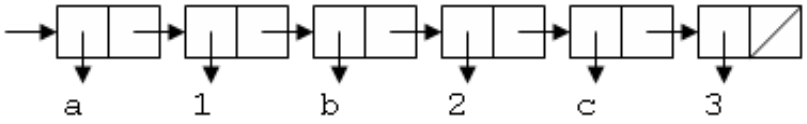
Ejemplo:

```
(define my-list (list 'a 1 'b 2 'c 3))
my-list ;--> (a 1 b 2 c 3)
(regular->assoc! my-list)
my-list ;--> ((a . 1) (b . 2) (c . 3))
```

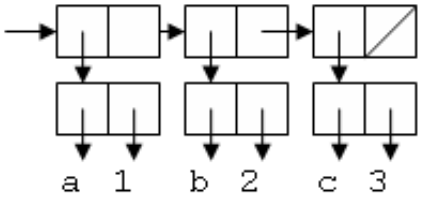
Una posible solución a este problema sería la siguiente (no es la única solución):

Cuando trabajamos con este tipo de problemas, es muy útil ayudarse con los diagramas caja y puntero. Vamos a crear los diagramas caja y puntero para el ejemplo anterior:

Antes de llamar a regular->assoc!:

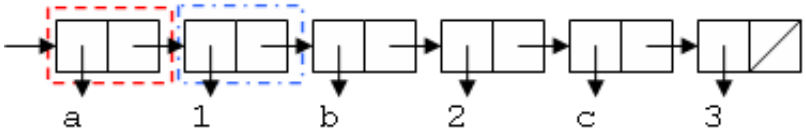


Después de llamar a regular->assoc!:

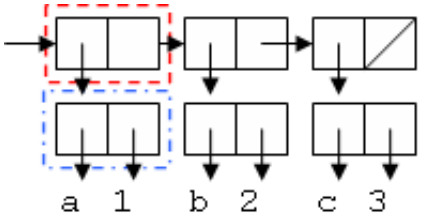


Por un lado, no podemos crear nuevas parejas, por lo que cada pareja en el primer diagrama corresponde a una pareja particular en el segundo diagrama. Por otro lado, no podemos modificar la primera pareja de la lista (porque perderíamos la ligadura de la variable `my-list`), por lo que es primera pareja tiene que permanecer en el primer lugar. Por otra parte, `a` y `1` van a formar parte del mismo par en la lista de asociación, por lo que vamos a considerar el siguiente cambio:

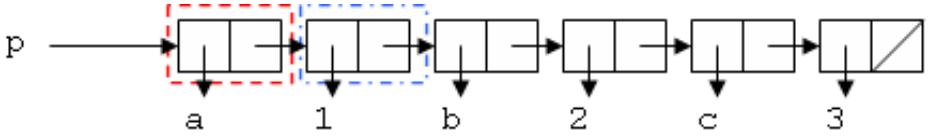
Antes de llamar a `regular->assoc!`:



Después de llamar a `regular->assoc!`:



Vamos a nombrar el par mostrado en rojo como `p`:

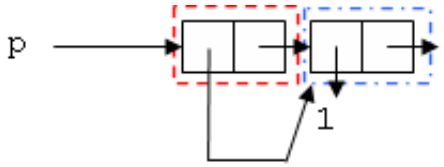



```
(define (manejar-dos-parejas! p)
    (let ((key (car p)))                    ;; 1
        (set-car! p (cdr p))                ;; 2
```

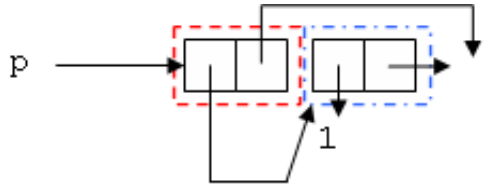
```
        (set-cdr! p (cdr (car p)))          ;; 3
        (set-cdr! (car p) (car (car p)))    ;; 4
        (set-car! (car p) key)))            ;; 5
```

Se han numerado las líneas para una mejor explicación. En la línea 1, `(let ((key (car p)))` , utilizamos un `let` para guardar el valor actual del `(car p)` , ese valor será la clave de la primera pareja de la lista de asociación; necesitamos almacenarlo para no perderlo.

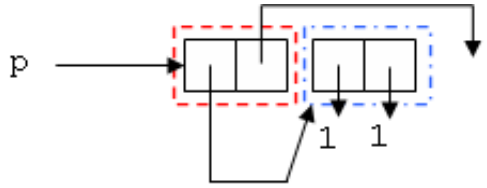
En la línea 2, `(set-car! p (cdr p))` , cambiamos el `car` de `p` para que apunte a la siguiente pareja (la azul):



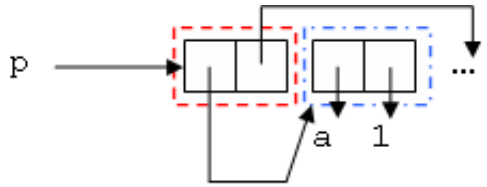
En la línea 3, `(set-cdr! p (cdr (car p)))` , cambiamos el `cdr` de `p` para que apunte al `cdr` de la pareja en azul:



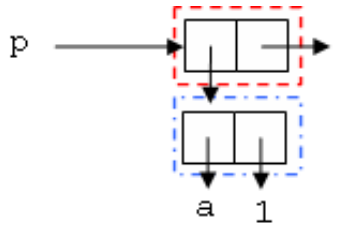
En la línea 4, `(set-cdr! (car p) (car (car p)))` , cambiamos el `cdr` de la pareja en azul al `car` de la misma pareja. Hacemos ésto porque en una lista de asociación, los valores se guardan en los `cdrs` y las claves en los `cars` :



Por último, en la línea 5 `(set-car! (car p) key)))` , completamos el problema poniendo la clave que habíamos guardado, en el `car` de la pareja azul:



Reordenamos el diagrama para verlo más claro:



Hemos definido un procedimiento que maneja un subproblema (dos parejas) del problema. Ahora sólo nos queda definir la función que maneja toda la lista:

```
(define (regular->assoc l)
  (if (null? l)
     'okay
     (begin (manejar-dos-parejas! l)
            (regular->assoc! (cdr l)))))
```

4. Clausuras con mutación = estado local mutable

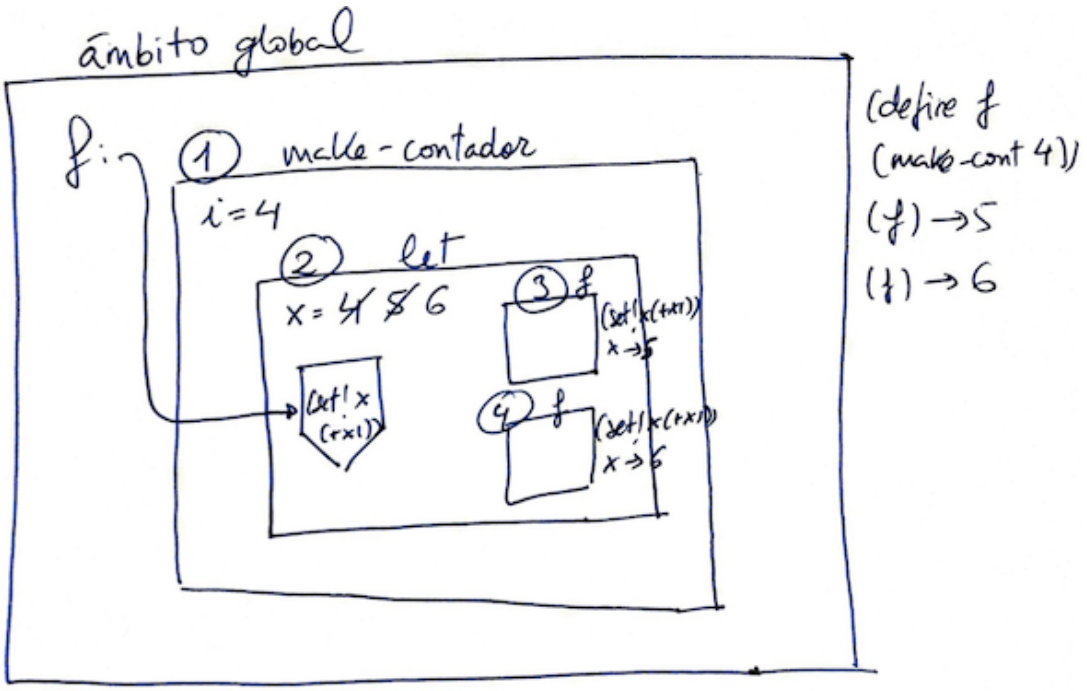
4.1. Estado local mutable

- Si combinamos una clausura con la posibilidad de mutar las variables encerradas en la clausura obtenemos estado local mutable asociado a funciones, un primer paso para la programación orientada a objetos:

```
(define (make-contador i)
  (let ((x i))
    (lambda ()
      (set! x (+ x 1))
      x)))

(define f (make-contador 10))
(f) ⇒ 11
(f) ⇒ 12
```

El diagrama de ámbitos que muestra la evaluación de las expresiones anteriores es el siguiente:



- La variable `x` es un estado local a la función que se devuelve
- El estado se mantiene y se modifica entre distintas invocaciones a `f` (diferencia con el paradigma funcional)
- Se crean tantos ámbitos locales como invocaciones a make-contador. Por ejemplo:

```
(define h (make-contador 10))
(define g (make-contador 100))
(h) ⇒ 11
(h) ⇒ 12
(g) ⇒ 101
(g) ⇒ 102
```

4.2. Paso de mensajes

Es posible acercarnos un poco más a la programación orientada objetos si hacemos que la clausura reciba un símbolo cadena (*un mensaje*) y ejecute distinto código en función del mensaje que recibe.

Vamos a cambiar el ejemplo anterior para que la clausura pueda hacer dos cosas:

- Si recibe el símbolo `'get` devuelve el valor del contador
- Si recibe la cadena `'inc` incrementa el contador y devolver su valor

```
(define (make-contador i)
  (let (( x i))
    (lambda (msg)
      (cond
        ((equal? msg 'get) x)
        ((equal? msg 'inc)
         (begin
```

```
            (set! x (+ x 1))
            x))
         ((equal? msg 'dec)
          (begin
            (set! x (- x 1))
            x))
         (else (error "Mensaje desconocido"))))))
```

Por ejemplo:

```
(define c (make-contador 100))
(c 'get) ⇒ 100
(c 'inc) ⇒ 101
(c 'dec) ⇒ 100
```
