Tema 9: Asignación, estado local y modelo de entornos

Índice

1 Programación imperativa ... 2
2 Formas especiales set! ... 3
3 Entornos ... 3
4 Estado local ... 8
4.1 Ejemplo 1: contador global ... 8
4.2 Ejemplo 2: primer intento de estado local (que no funciona) ... 9
4.3 Ejemplo 3: Estado local (sí que funciona) ... 9
4.4 Ejemplo 4: Variable local como parámetro ... 11
5 El modelo de computación de Scheme basado en entornos ... 13
5.1 Ejemplos ... 15
6 Referencias ... 27

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

1. Programación imperativa

El paradigma de programación imperativa es el paradigma de programación más utilizado. Surgió con los primeros ordenadores sobre los años 1940s reflejando las características de la arquitectura Von Newman de la mayoría de ordenadores: el ordenador dispone de una memoria para almacenar las instrucciones de los programas (*the program store*) y los datos (*the data store*).

Este paradigma debe su nombre al papel dominante que desempeñan las sentencias imperativas, es decir aquellas que indican realizar una determinada operación que modifica los datos guardados en memoria. Su esencia es resolver un problema complejo mediante la ejecución repetitiva y paso a paso de operaciones y cálculos sencillos con la asignación de los valores calculados a posiciones de memoria.

La programación en este paradigma consiste en determinar qué datos son requeridos para el cálculo, asociar a esos datos una direcciones de memoria, y efectuar, paso a paso, una secuencia de transformaciones en los datos almacenados, de forma que el estado final represente el resultado correcto.

Principales características

- **Celdas de memoria**: El principal componente de la arquitectura es la memoria, compuesta por un gran número de celdas que contienen los datos. Las celdas, llamadas variables, tienen nombres que las referencian y se organizan en diferentes estructuras de datos.
- **Asignación**: Estrechamente ligado a la arquitectura de memoria se encuentra la idea de que cada valor calculado debe ser almacenado, o sea, asignado, en una variable. Estas asignaciones se realizan repetitivamente sobre la misma celda de memoria, remplazando los valores anteriores. La asignación determina el estado de una variable, que consiste en el valor que contiene en un momento en particular. Por lo tanto los resultados corresponden al estado final que asumen las variables.
- **Algoritmos**: Un programa imperativo, normalmente realiza su tarea ejecutando repetidamente una secuencia de pasos elementales, ya que en este modelo computacional la única forma de ejecutar algo complejo es repitiendo una secuencia de instrucciones. La programación requiere la construcción de "algoritmos", a modo de receta, método, técnica, procedimiento o rutina, que se definen como conjuntos finito de sentencias, ordenadas de acuerdo a sus correspondientes estructuras de control, que marcan el flujo de ejecución de operaciones para resolver un problemas específico.
- **Programación estructurada**: La programación estructurada surge como un conjunto de técnicas para facilitar el desarrollo de sistemas en lenguajes del paradigma imperativo, pero presenta ideas que también fueron tenidas en cuenta en lenguajes de otros paradigmas.

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

- **Estructuras básicas de control**: La base teórica de la programación estructurada plantea que cualquier programa, por más grande y complejo que fuera, puede representarse mediante tres tipos de estructuras de control: secuencia, selección e iteración.

2. Formas especiales set!

La forma especial `set!` cambia el valor de una variable. Sintaxis: `(set! variable expresion)`

```
(define a 3)
(set! a 5)
a -> 5
(set! b 8) -> error
```

Las formas especiales `set-car!` y `set-cdr!` cambian el valor asociado a la parte izquierda y derecha de una pareja. Sintaxis: `(set-car! pareja expresion)`

```
(define c (cons 1 2))
c -> (1 . 2)
(set-car! c 'hola)
c -> (hola . 2)
(set-cdr! c 'adios)
c -> (hola . adios)
```

En las estructuras: set-TIPO-CAMPO!

```
(define-struct persona (nombre dni))
(define p1 (make-persona 'juan '212121221))
(persona-nombre p1) ;-> juan
(set-persona-nombre! p1 'ana)
(persona-nombre p1) ;-> ana
```

Diferencias entre define y set

`define` crea una variable nueva y `set` cambia el valor de una variable ya existente

El uso de set! nos aparta de la programación funcional.

3. Entornos

Como hemos visto, la operación `set!` (asignación) nos permiten modelar objetos que tiene estado local. Cuando una asignación está presente, una variable no es sólo considerada como un nombre asociado a un valor, sino que la variable debe estar asociado a un "lugar" donde los valores se guarden. En el modelo de evaluación de Scheme, esos "lugares" se denominan

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

entornos. Los entornos permiten formalizar también el concepto de ámbito de variable.

Definimos un "entorno" como un conjunto de variables y valores asociados a ellas. El entorno principal es el que existe cuando arrancamos el intérprete de Scheme.

Cuando usamos las formas especiales set! estamos modificando el entorno.

En el nuevo modelo de computación que veremos más adelante el concepto de entorno es central.

Una expresión se evalua en un entorno.

Ejemplo 1 de entorno:

```
(define x 5)
(set! x (+ 1 x))
x -> 6
```

Dibujo de los entornos resultantes:

Imagen del entorno 1

En el entorno global la variable x queda ligada al valor 5.

Imagen del entorno 1

Al evaluar la instrucción `(set! x (+ 1 x))`, la variable x incrementa en uno su valor.

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

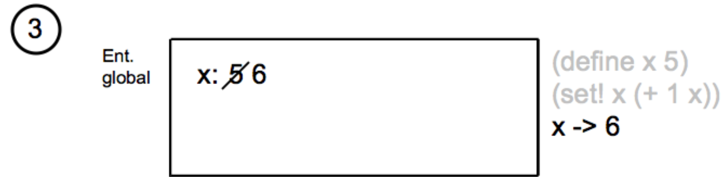


Imagen del entorno 1

Se devuelve el valor ligado a x, que es 6.

Ejemplo 2 de entorno:

¿Qué pasa si la modificación de la variable x la hacemos en una llamada a una función?

```
(define x 5)
(define (prueba)
   (set! x (+ 1 x))
   x)
(prueba) -> 6
x        -> 6
```

Dibujo de los entornos resultantes:

Imagen del entorno 2

En el entorno global la variable x queda ligada al valor 5.

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

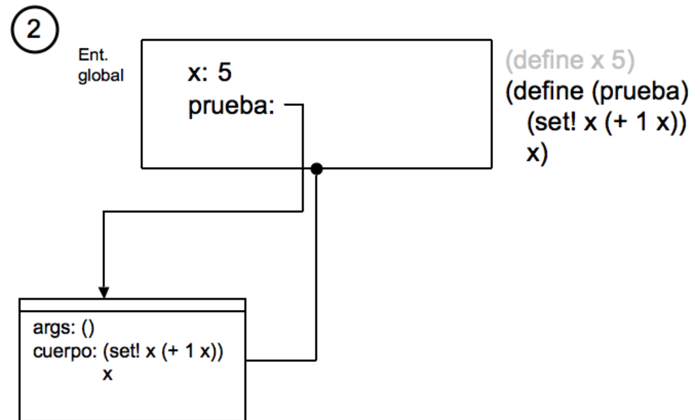


Imagen del entorno 2

En el entorno global, el identificador prueba queda ligado a un procedimiento sin argumentos y cuyo cuerpo es `(set! x (+ 1 x)) x)`. Éste a su vez está sociado al entorno global.

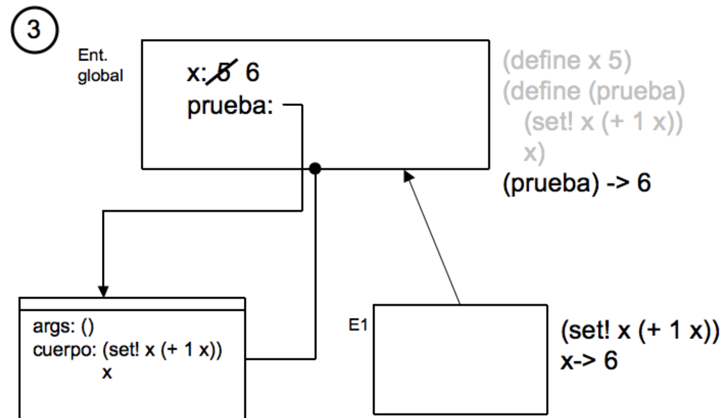


Imagen del entorno 2

Al hacer la llamada al procedimiento `(prueba)`, se extiende el entorno asociado al procedimiento. En este entorno local a la función es donde se evalúa la función. Se incrementa la variable x en 1 y se devuelve.

Ejemplo 3 de entorno:

¿Qué pasa si dentro de la llamada a "prueba" se define otra variable x?

```
(define x 5)
```


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

```
(define (prueba)
   (define x 10)
   (set! x (+ 1 x))
   x)
(prueba) -> 11
x        -> 5
```

Dibujo del entorno resultante:

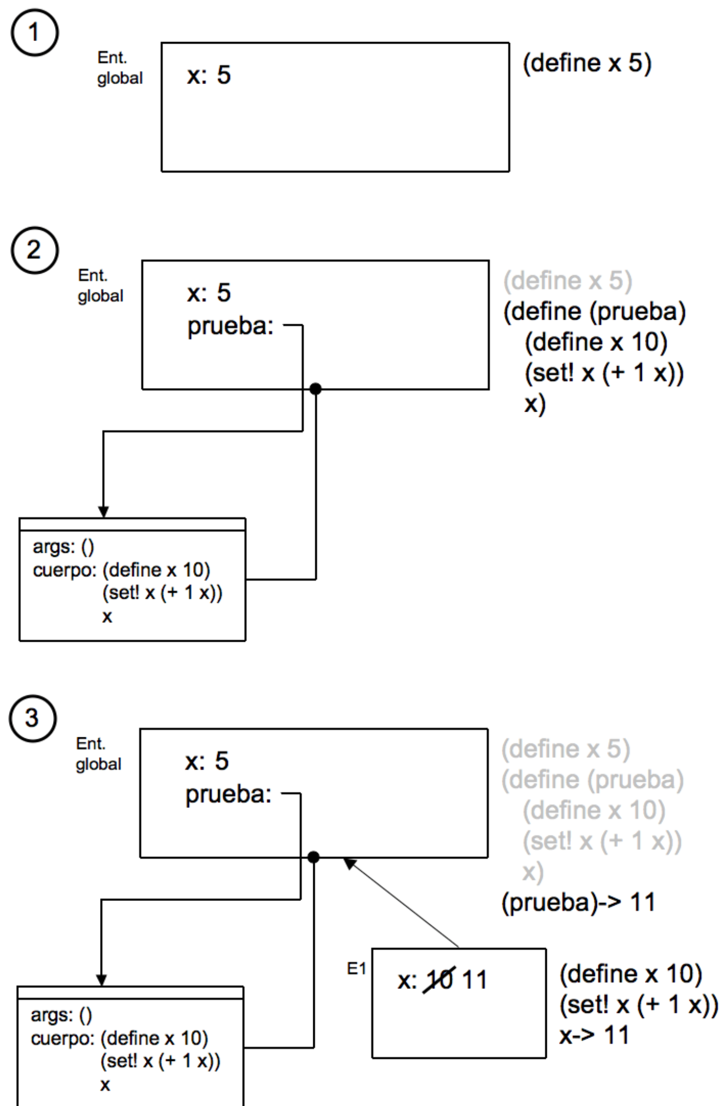


Imagen del entorno 3

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

La invocación a un procedimiento (como **prueba** en el ejemplo anterior) genera un entorno en el que se evalúa el cuerpo del procedimiento. Este entorno es algo así como la memoria local asociada a la evaluación del procedimiento. Una regla muy importante que veremos más adelante: este entorno local debe extender el entorno *en el que fue creado el procedimiento que se invoca*.

4. Estado local

Vamos a jugar con el ejemplo más sencillo posible de programación no funcional: un contador. Y vamos a utilizar este ejemplo para continuar introduciendo los elementos que posteriormente utilizaremos en el modelo de entornos.

Se trata de definir una funcion count que implemente un contador y devuelva cada vez un valor incrementado en 1.

```
(count) -> 0
(count) -> 1
(count) -> 2
...
```

4.1. Ejemplo 1: contador global

Vamos a jugar con el ejemplo más sencillo posible de programación no funcional: un contador.

Se trata de definir una funcion count que implemente un contador y devuelva cada vez un valor incrementado en 1.

```
; (count) -> 0
; (count) -> 1
; (count) -> 2
; ...

(define counter 0)

(define (count)
  (set! counter (1+ counter))
  counter)

(count) -> 1
(count) -> 2
```

Es necesario mejorar esta versión, porque la utilización de un contador global nos puede llevar a errores. Por ejemplo, alguna otra función puede cambiar el valor del contador sin que nos demos cuenta.

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

Vamos a intentar implementar en Scheme el estado local algo asi como el static de C. Buscamos poder definir un contador que vayamos incrementando y que podamos consultar a traves de una funcion, pero que no esté disponible globalmente.

```
int count(){
   static int iCount = 0;

   iCount += 1;
   return iCount;
}

count = count();  // devuelve 1
count = count();  // devuelve 2
```

4.2. Ejemplo 2: primer intento de estado local (que no funciona).

Utiliza let para definir una variable local, pero no tiene estado ya que cada llamada a count vuelve a inicializar su valor a 0.

```
(define (count)
  (let ((counter 0))
    (set! counter (+ counter 1))
    counter))
```

4.3. Ejemplo 3: Estado local (sí que funciona).

Usamos un constructor del contador (make-counter) que va a devolver el contador propiamente dicho.

```
(define (make-counter)
   (define x 0)
   (lambda ()
      (set! x (+ counter 1))
      x))

(define count (make-counter))
(count) ;-> 1
(count) ;-> 2
```

Nuevo modelo computacional: evaluacion del entorno. Cuando llamamos a lambda() para crear la función, ésta se crea en el entorno definido por la llamada a make-counter, en el que se ha creado la variable x con el valor 0. A partir de este momento la función creada por el lambda queda asociada a este entorno y usa x como una variable local. El valor de x no es visible desde el entorno global.

Los entornos resultantes son los siguiente:

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

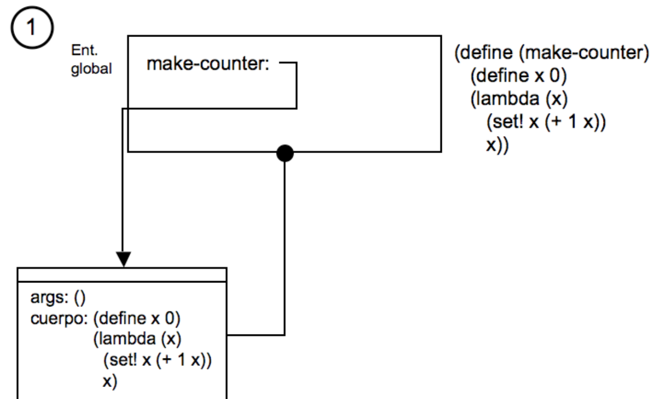


Imagen del entorno ejemplo 3

En el entorno global, el identificador make-counter queda ligado a un procedimiento sin argumentos cuyo cuerpo es:

```
(define x 0)
(lambda ()
   (set! x (+ counter 1))
   x)
```

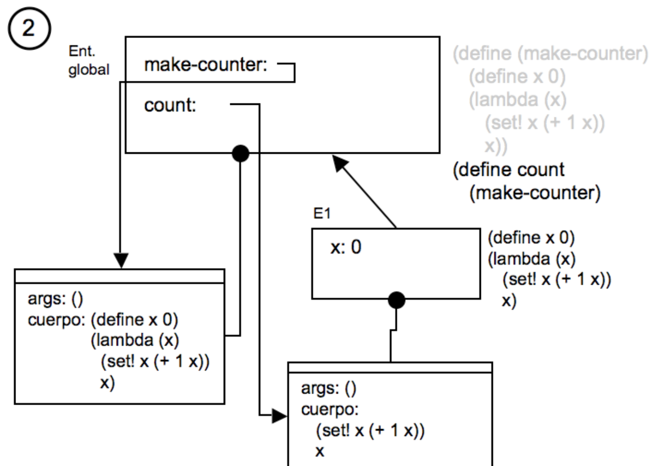


Imagen del entorno ejemplo 3

En el entorno global, el identificador count queda ligado al resultado de la llamada a make-counter, que ha obligado a evaluarlo:

- En primer lugar, se ha extendido el entorno asociado a make-counter. En este entorno

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

local E1 a la función, la variable x ha quedado ligada al valor cero.

- En segundo lugar, la evaluación de lambda(x).... ha devuelto un procedimiento, que ha quedado ligado al identificador count en el entorno global. Este procedimiento queda asociado al entorno local E1 que es donde fue creado.

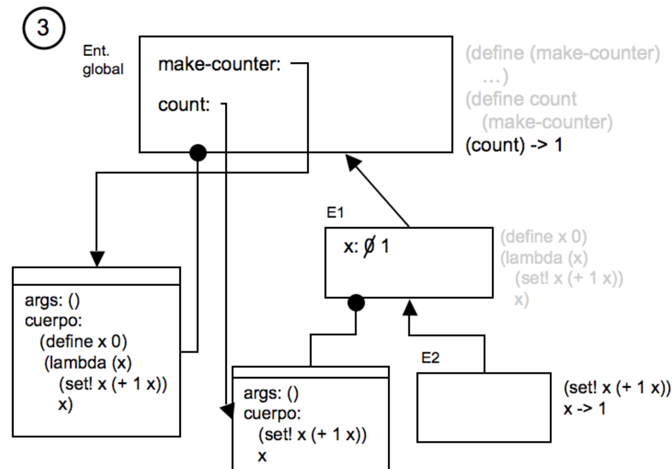


Imagen del entorno ejemplo 3

La llamada a count obliga a evaluar este procedimiento, el cual extiende su entorno asociado. En este entorno local E2 se evalúa el cuerpo de la función, modificando el valor de x incrementándolo en 1.

4.4. Ejemplo 4: Variable local como parámetro

Igual que el 3, pero usando como variable local un parámetro de la funcion make-counter que representa el valor inicial del contador.

```
(define (make-counter x)
    (lambda ()
      (set! x (+ 1 x))
      x))

(define count (make-counter 0))
(count) ;-> 1
(count) ;-> 2
```

Los entornos resultantes se muestran en la siguiente figura:

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

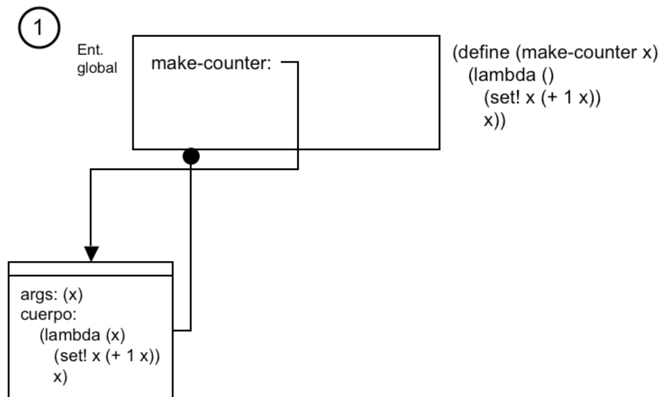


Imagen del entorno 5

En el entorno global, el identificador make-counter queda ligado a un procedimiento con un argumento x cuyo cuerpo es:

```
(lambda ()
  (set! x (+ counter 1))
  x)
```

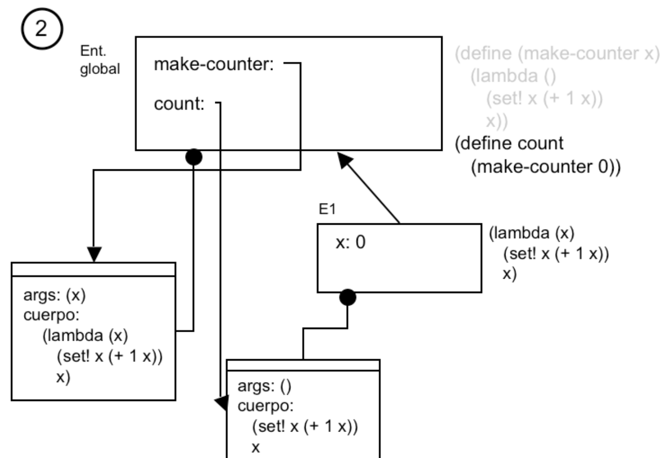


Imagen del entorno 5

En el entorno global, el identificador count queda ligado al resultado de la llamada a (make-counter 0), que ha obligado a evaluarlo:

- En primer lugar, se ha extendido el entorno asociado a make-counter. En este entorno local E1 a la función, la variable x ha quedado ligada al valor cero (valor pasado como

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

parámetro).

- En segundo lugar, la evaluación de lambda(x).... ha devuelto un procedimiento, que ha quedado ligado al identificador count en el entorno global. Este procedimiento queda asociado al entorno local E1 que es donde fue creado.

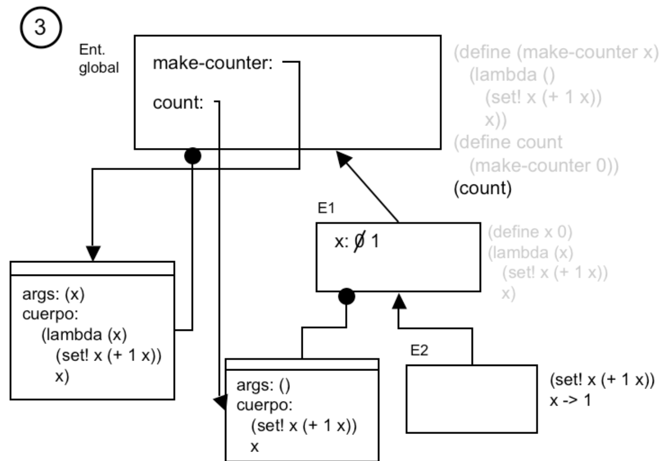


Imagen del entorno 5

La llamada a count provoca la evaluación de este procedimiento, el cual extiende su entorno asociado. En este entorno local E2 se evalúa el cuerpo de la función, modificando el valor de x incrementándolo en 1.

5. El modelo de computación de Scheme basado en entornos

El objetivo de los diagramas de entorno es modelar las reglas de Scheme para el ámbito de las variables. El término en ciencia de la computación *variable scoping* se refiere a las reglas del lenguaje sobre el lugar apropiado (*scope*) para definir, modificar o utilizar el valor de una variable.

El ámbito de las variables en Scheme funciona utilizando un estilo de *ámbito léxico*, donde un entorno, en el que se evalua el cuerpo de un procedimiento, se forma **extendiendo** el entorno que era el actual cuando ese procedimiento se creó (es decir, cuando la expresión lambda que creó el procedimiento fue evaluada). Este estilo nos permite un control muy potente sobre el ámbito, utilizando una sintaxis de programación muy sencilla.

En Java y en C++, el funcionamiento del ámbito o *scoping* es más obvio, pero la sintaxis del lenguaje es mucho más compleja en cuanto a reflejar el ámbito más explícitamente, aparte de que se pierde alguna flexibilidad. Mediante el uso de diagramas de entorno, estamos más preparados para la utilización del ámbito de variables en lenguajes como Scheme y Perl (de

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

ámbito léxico), lo que nos lleva a estar mejor preparados también para modelar las reglas del ámbito en el resto de lenguajes de programación (de ámbito no léxico o *not-lexically-scoped programming languages*).

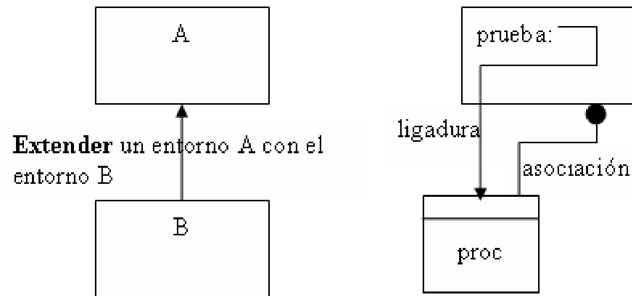
Conceptos previos

Una expresión es un átomo (expresión atómica) o una lista (expresión compuesta). En cualquier momento hay un entorno actual, inicialmente el entorno global. Un entorno es una colección de ligaduras (bindings) de símbolos a valores.

"**Extender** un entorno A con el entorno B" significa hacer que el entorno A sea el padre del B. Desde el entorno B se podrá acceder a las variables definidas en A, pero no al revés. El entorno B es el entorno local a la función (donde se evalúa la función).

Una **ligadura** representa una asociación entre un identificador a un valor (o procedimiento).

Una **asociación** entre un procedimiento y un entorno, significa que fue en ese entorno donde se creó el procedimiento.



Conceptos del modelo de entornos

Reglas para evaluar una expresión en el entorno actual

1. **Búsqueda de un identificador: (+ x 10)** En primer lugar, buscar en el entorno actual. Si no existe, seguir la asociación desde el entorno actual hasta el entorno al que se enlaza. Continuar de esta forma hasta que encontremos la declaración del identificador o hasta que agotemos la cadena de asociaciones (y lleguemos al entorno global). Si no encontramos la ligadura en el entorno global, se trata de un identificador no declarado.
2. **Define: (define var expresion)** Un define añade una ligadura en el frame actual (donde se evalua el define).
3. **Set!: (set! var expresion)** Buscar recursivamente el identificador empezando en el entorno actual. Si no se encuentra, error. Religar el identificador en el entorno donde se ha encontrado con el valor de la expresión.
4. **Lambda: (lambda (parametros) expresion)** La evaluación de una expresión lambda produce un nuevo objeto procedimiento. Estará asociado al entorno donde se evaluó el

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

lambda (donde fue creado el procedimiento) y contendrá el cuerpo del procedimiento y la lista de parámetros formales.

5. **Combinación / Aplicación de un procedimiento:** Para evaluar una expresión respecto a un entorno, primero se evalúan las subexpresiones respecto al entorno y entonces se aplica el valor de la subexpresión del operador a los valores de las subexpresiones de los operandos.
   - Dibujar un nuevo entorno.
   - Extender el nuevo entorno al entorno asociado al procedimiento (donde fue creado).
   - Ligar las variables de los parámetros formales a los argumentos del procedimiento (similar a define)
   - Evaluar el cuerpo del procedimiento respecto al entorno actual.
6. **Let: (let ((n v)... cuerpo) --> ((l (n ...) cuerpo) v...)** Eliminar el azúcar sintáctico. Se dibuja un entorno asociado al entorno actual, se ligan las variables y se evalua la expresión en ese nuevo entorno.

5.1. Ejemplos

Ejemplo 1

```
(define (cuadrado x)
   (* x x))
(cuadrado 5)
```

El define de la función "cuadrado" se transforma en

```
(define cuadrado (lambda (x) (* x x)))
```

Entornos resultantes:

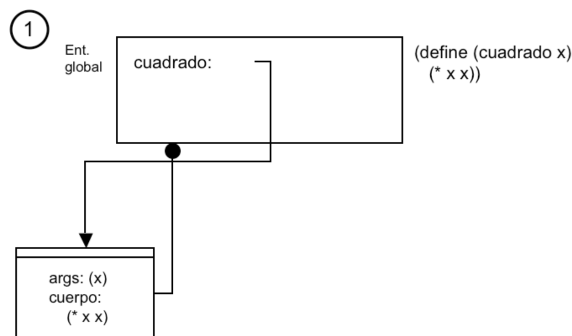


Imagen del ejemplo 1 del modelo de entornos

La variable cuadrado queda ligada al procedimiento creado por la evaluación de lambda. El

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

entorno actual (entorno global) queda asociado al procedimiento (para registrar en qué entorno se creó el procedimiento).

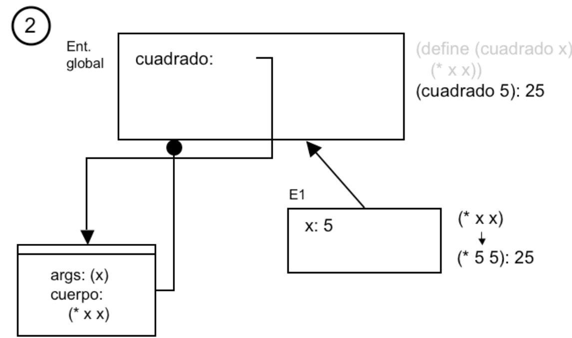


Imagen del ejemplo 1 del modelo de entornos

La llamada al procedimiento `(cuadrado 5)` extiende el entorno en el que se definió el procedimiento con un nuevo entorno llamado E1. En él se evalúa el argumento (el 5 se autoevalúa) y se evalúa el cuerpo del procedimiento `(* 5 5)`. El resultado es 25.

Ejemplo 2

```
(define (f x)
   (define (g y)
     (+ x y))
   (g 3))
(f 5)
```

Entornos resultantes:

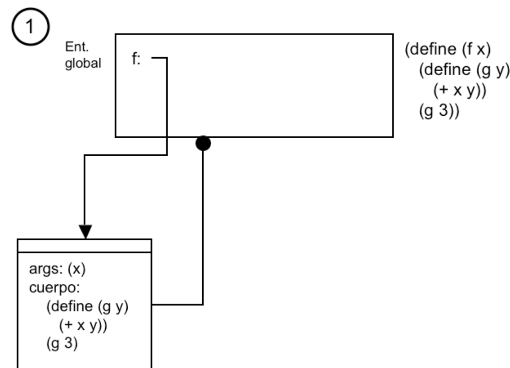


Imagen del ejemplo 2 del modelo de entornos

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

La evaluación del primer `define` provoca que la variable `f` quede ligada a un procedimiento y el entorno global quede asociado a él.

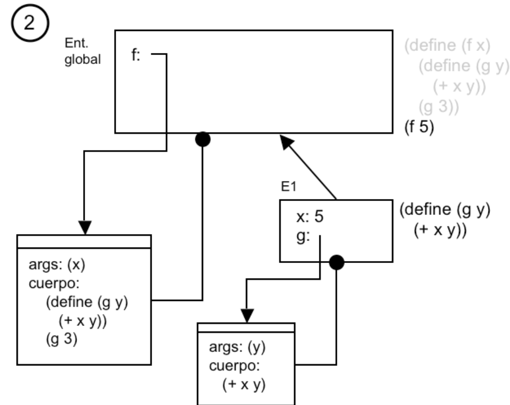


Imagen del ejemplo 2 del modelo de entornos

Al hacer la llamada `(f 5)` se evalúa el cuerpo del procedimiento `f` con argumento 5: se extiende el entorno asociado al procedimiento con el nuevo entorno llamado E1. En él se evalúa el argumento (el 5 se autoevalúa y queda ligado a la variable x) y se evalúa el cuerpo del procedimiento, de manera que la variable g queda ligada en el entorno E1 al procedimiento con parámetro `y` y cuerpo `(+ x y)`. Este procedimiento a su vez está asociado al entorno E1 que es donde se creó.

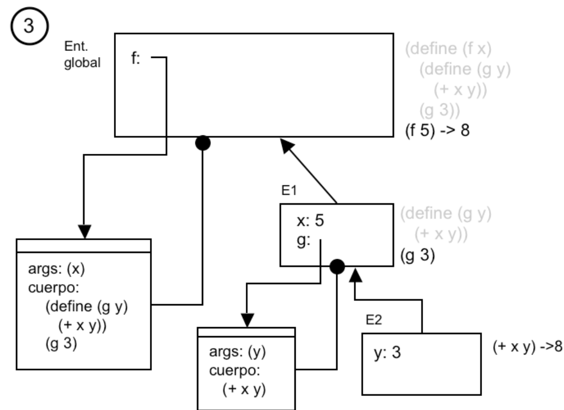


Imagen del ejemplo 2 del modelo de entornos

A continuación se evalúa la instrucción `(g 3)`, por lo que se extiende el entorno E1 asociado al procedimiento, creándose el entorno E2. En él se evalúa el argumento 3,

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

quedando ligada la variable `y` al resultado del mismo. Se evalúa el cuerpo del procedimiento `(+ x y)`. La variable `x` está ligada al valor 5 en el E1 y la variable `y` al valor 3 en el E2, `(+ 5 3)` obteniéndose el valor.

Ejemplo 3.

```
(define (suma-y y)
   (lambda (x)
      (+ x y)))
(define suma-5 (suma-y 5))
(suma-5 12)
```

Entornos resultantes:

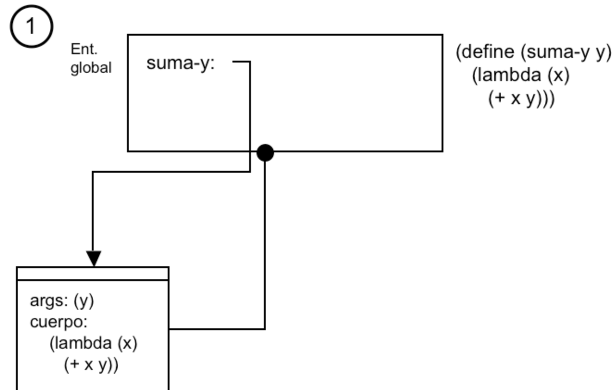


Imagen del ejemplo 3 del modelo de entornos

Se evalúa el primer define quedando ligada la variable `suma-y` a un procedimiento con un argumento `y` y cuyo cuerpo es:

```
(lambda(x) (+ x y))
```

Este procedimiento está asociado al entorno global que es donde se creó.

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

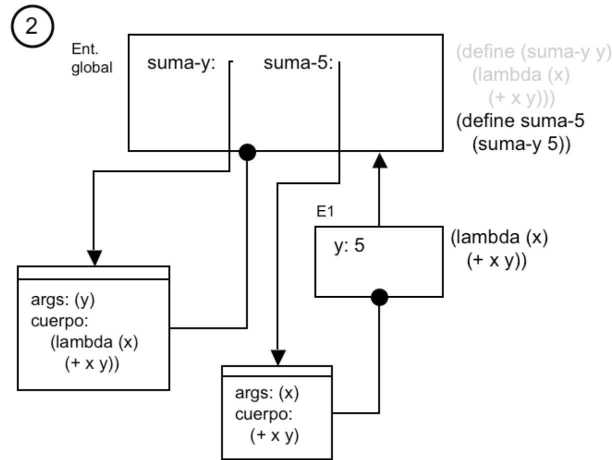


Imagen del ejemplo 3 del modelo de entornos

Con la evaluación del segundo define la variable `suma-5` queda ligada a lo que devuelve la llamada a `(suma-y 5)`, que invoca al procedimiento `suma-y`. La invocación del procedimiento provoca que se cree un entorno E1 donde se evalúa su cuerpo (donde la variable `y` está ligada al valor 5), que extiende el entorno global donde se creó el procedimiento. Al evaluarse el cuerpo del procedimiento (lambda), se devuelve un procedimiento de un argumento `x` y cuyo cuerpo es `(+ x y)`. La variable `suma-5` queda ligada a este procedimiento y este procedimiento queda asociado al entorno E1 que es donde se creó.

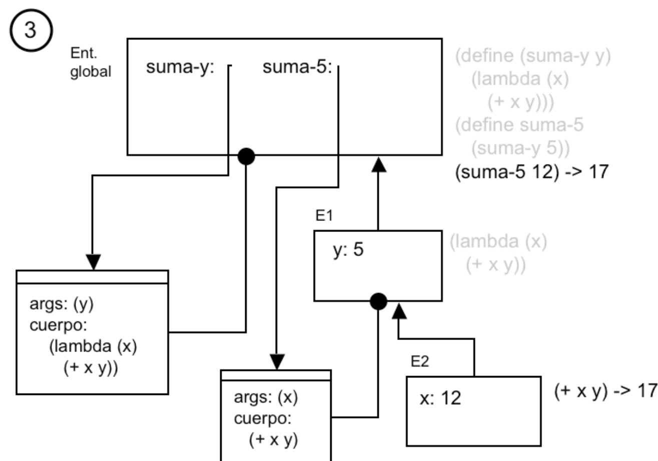


Imagen del ejemplo 3 del modelo de entornos

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

Con la evaluación de `(suma-5 12)` se extiende el entorno asociado al procedimiento y se crea un nuevo entorno E2 donde se evalúa su cuerpo. La variable `x` se liga al valor 12 y se evalúa `(+ x y)`, se sustituye cada variable por su valor y se evalúa `(+ 12 5)`, devolviendo como resultado 17.

Ejemplo 4.

```
(let ((x 1) (y 2))
   (+ x y))
```

Recordemos que la forma especial let se construye como una llamada a lambda para construir el procedimiento que implementa su cuerpo y una invocación de este procedimiento con los valores definidos por los valores de las variables del let.

Así, la expresión anterior quedaría como:

```
((lambda (x y) (+ x y)) 1 2)
```

Entornos resultantes:

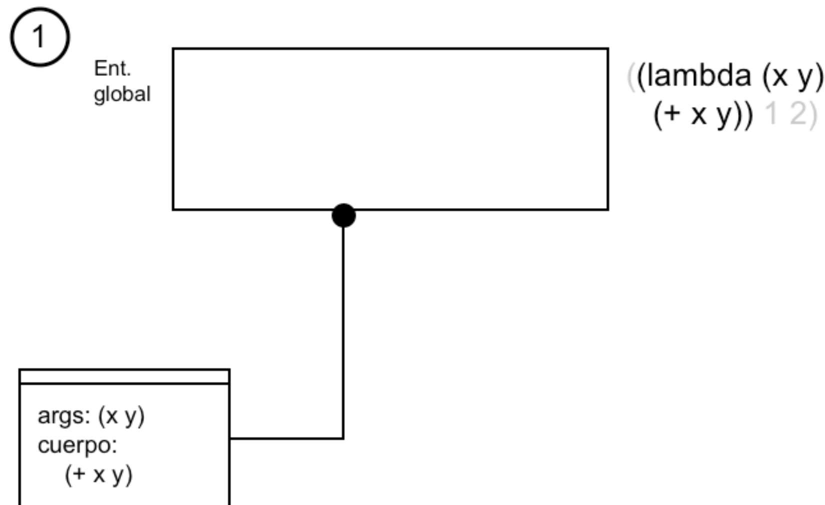


Imagen del ejemplo 4 del modelo de entornos

En primer lugar se evalúa el lambda, por lo que se crea un procedimiento asociado al entorno global, de dos argumentos `x y` y cuyo cuerpo es `(+ x y)`.

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

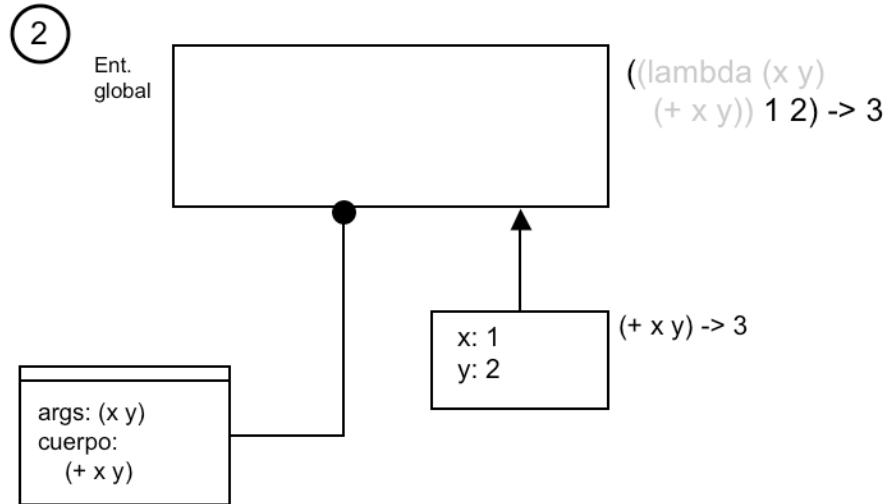


Imagen del ejemplo 4 del modelo de entornos

Posteriormente se evalúa ese procedimiento (recordemos que los paréntesis significan invocación de procedimiento) con parámetros 1 y 2. Se extiende el entorno en el que se evalúa el let que contiene las variables `x:1` y `y:2`. Se evalúa su cuerpo `(+ 1 2)` que es `3`.

A partir de ahora vamos a obviar estos pasos y **cuando haya que definir los entornos resultantes de la evaluación de un let consideraremos directamente la figura final en la que se ha creado un nuevo entorno que extiende el entorno en el que se evalua el let, que contiene las variables definidas por el let y en el que se evalua su cuerpo**. Veamos algunos ejemplos.

Ejemplo 5.

```
(define count
  (let ((x 0))
    (lambda ()
      (set! x (+ 1 x))
      x)))
```

Entornos resultantes:

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

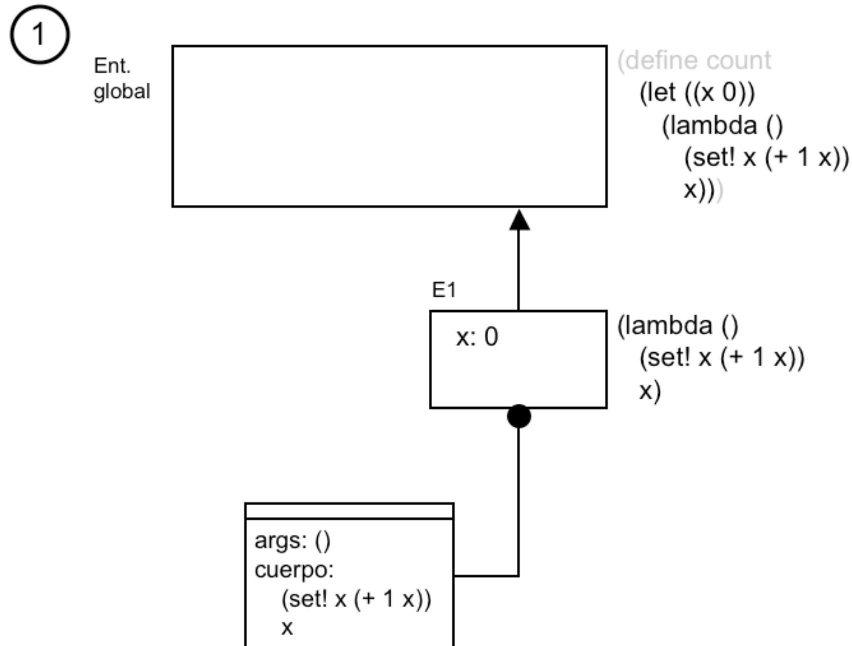


Imagen del ejemplo 5 del modelo de entornos

En primer lugar se evalúa el let (recordemos que la forma especial define primero evalúa la expresión y luego asocia el valor resultante con el símbolo). Se crea un nuevo entorno E1 que extiende el entorno en el que se evalua el let, donde la variable x definida por el let toma el valor 0 y en el que se evalua su cuerpo: su cuerpo contiene un lambda que al evaluarse devuelve un procedimiento asociado al entorno E1 que es donde se creó. Este procedimiento no tiene argumentos y su cuerpo es `(set! x (+ 1 x)) x`.

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

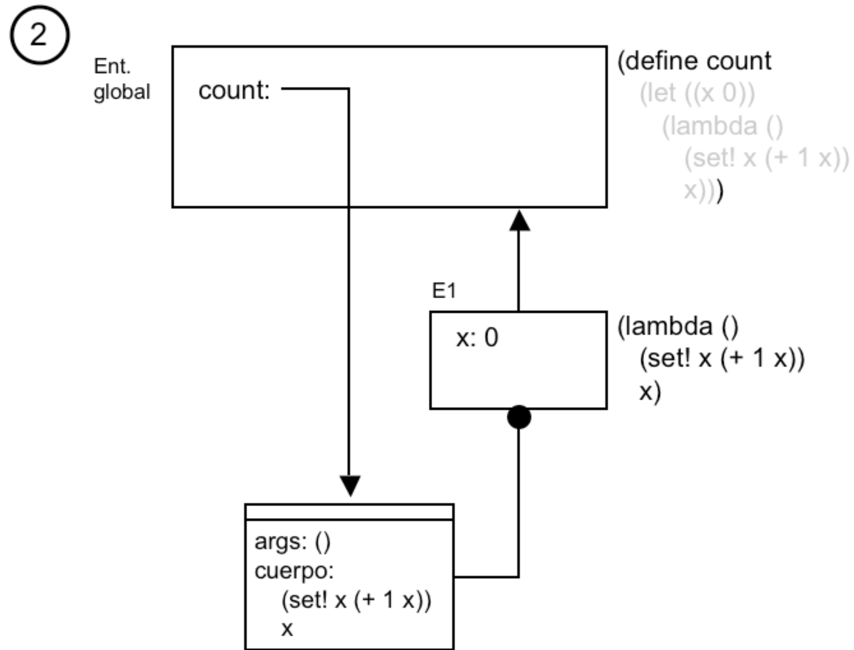


Imagen del ejemplo 5 del modelo de entornos

Ahora la variable count queda ligada al procedimiento que ha devuelto la expresión.

Ejemplo 6.

```
(define (make-count)
   (let ((x 0))
      (lambda ()
         (set! x (+ 1 x))
         x)))
(define count (make-count))
(count)
```

Entornos resultantes:

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

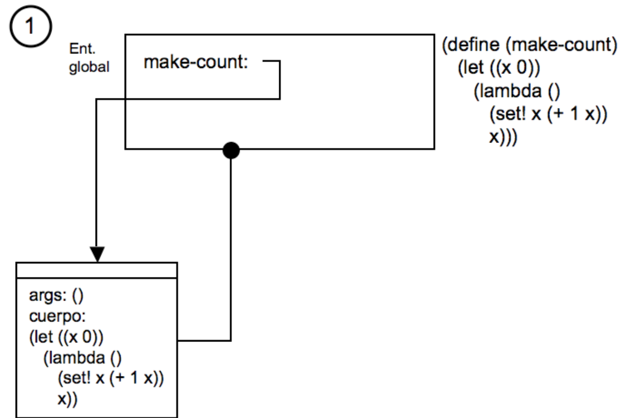


Imagen del ejemplo 6 del modelo de entornos

La variable `make-count` queda ligada a un procedimiento sin argumentos asociado al entorno global que es donde se creó. Su cuerpo es: `(let ((x 0)) (lambda() (set! x (+ 1 x)) x))`

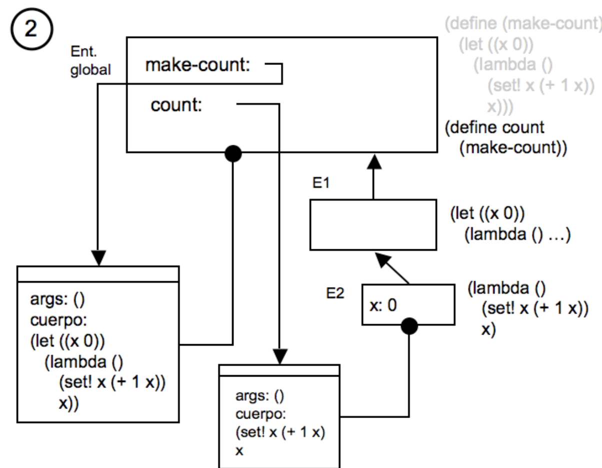


Imagen del ejemplo 6 del modelo de entornos

La expresión `(make-count)` invoca al procedimiento `make-count`: se crea un nuevo entorno E1 que extiende el entorno global donde se evalúa el cuerpo del procedimiento. En él se evalúa el let y se crea un nuevo entorno E2 que extiende el entorno E1, donde la variable x definida por el let toma el valor 0 y en el que se evalúa su cuerpo, el cual contiene un lambda que al evaluarse devuelve un procedimiento asociado al entorno E2 que es donde se creó. Este procedimiento no tiene argumentos y su cuerpo es `(set! x (+ 1 x)) x`. La variable `count` queda ligada a este procedimiento.

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

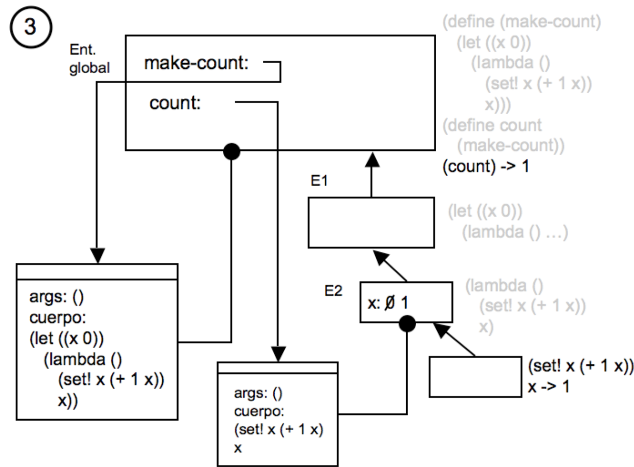


Imagen del ejemplo 6 del modelo de entornos

Al llamar al procedimiento ligado a la variable count se crea un nuevo entorno que extiende el entorno E2 (asociado al procedimiento). En él se evalúa su cuerpo modificando en valor de la variable x (definida en el E2) sumándole 1. A continuación se devuelve su valor.

Ejemplo 7.

```
(define h
   (let ((x 1) (y 2))
      (lambda (z)
         (set! z (+ z x y))
         (set! x z)
         z)))
(h 4)
(h 4)
```

Entornos resultantes:

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

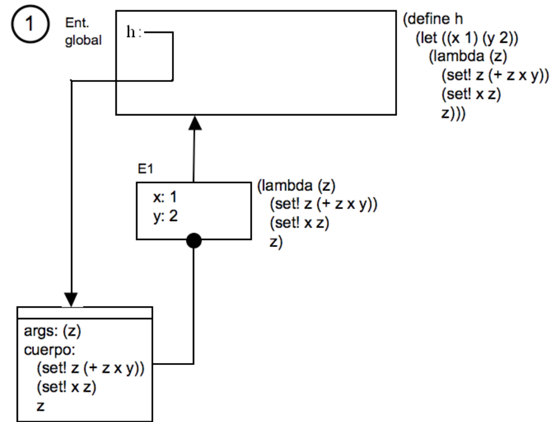


Imagen del ejemplo 7 del modelo de entornos

En primer lugar se evalúa el let. Se crea un nuevo entorno E1 que extiende el entorno en el que se evalua el let, donde la variable x definida por el let toma el valor 1 y la variable y el valor 2 y en el que se evalúa su cuerpo: su cuerpo contiene un lambda que al evaluarse devuelve un procedimiento asociado al entorno E1 que es donde se creó. Este procedimiento tiene un argumento z y su cuerpo es `(set! z (+ z x y)) (set! x z) z .`

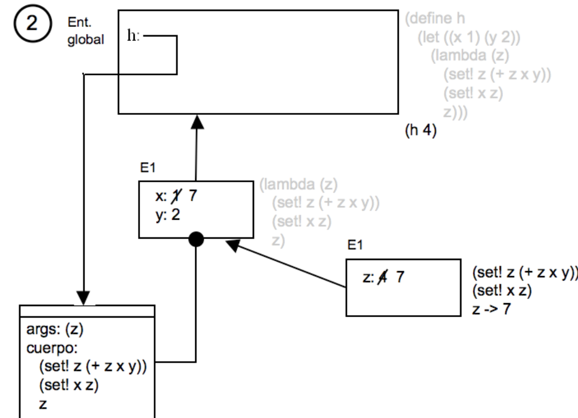


Imagen del ejemplo 7 del modelo de entornos

Con la llamada `(h 4)` se crea un nuevo entorno que extiende el entorno E1 (asociado al procedimiento) donde la variable z toma el valor 4. En él se evalúa su cuerpo modificando en valor de la variable z sumándole el valor x y el valor y (definidas en el entorno E1), con lo que z pasa a valer 7. Con la evaluación del segundo set! la variable x del E1 modifica su valor tomando a su vez el valor de z, 7. Se devuelve z.

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

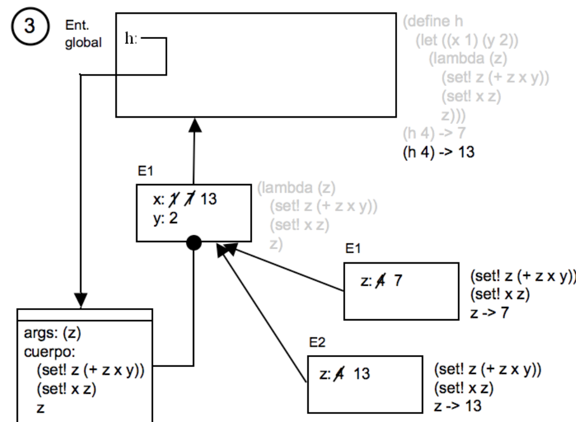


Imagen del ejemplo 7 del modelo de entornos

Con la segunda llamada `(h 4)` se crea un nuevo entorno E2 que extiende el entorno E1 (asociado al procedimiento) donde la variable z toma el valor 4. En él se evalúa su cuerpo modificando en valor de la variable z sumándole el valor x y el valor y (definidas en el entorno E1), con lo que z pasa a valer 13, ya que la variable x modificó su valor en la llamada anterior y ahora vale 7. Con la evaluación del segundo set! la variable x del E1 modifica su valor tomando a su vez el valor de z, 13. Se devuelve z.

6. 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 (pp. 217-224). Disponible biblioteca politécnica (acceso al catálogo (http://gaudi.ua.es/uhtbin/boletin/285815))
- Programming languages: principles and paradigms. Allen Tucker y Robert Noonan, McGrawHill. Capítulo 4: Imperative Programming. Disponible biblioteca politécnica (acceso al catálogo (http://gaudi.ua.es/uhtbin/boletin/300185))
- Encyclopedia of Computer Science (Wiley, 2000). Disponible en la biblioteca politécnica (POE R0/E/I/ENC/RAL). Consultar la entrada "Procedure-oriented languages", pp 1441 y ss

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