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

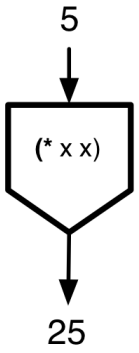
Semana 3

- Conocer las condiciones para definir un tipo de dato como de primera clase
- Entender y saber utilizar la forma especial `lambda` para crear funciones sin darles un nombre
- Entender el funcionamiento de la forma especial `define` para construir una función
- Entender y diseñar funciones *de orden superior*
- Entender y saber utilizar la forma especial `let`
- Entender la semántica de la forma especial `let` y cómo se puede construir usando `lambda`
- Entender la forma especial `let*`
- Entender los ámbitos de variables en Scheme, saber representar gráficamente los ámbitos creados por expresiones `let` y `lambda`
- Entender el concepto y los ejemplos de *closures* y saber crear este tipo de funciones
- Diferenciar el funcionamiento del *deep* y el *shallow* binding

5. Funciones como tipos de datos de primera clase

El elemento principal de la programación funcional es la función. Hemos aprendido a definir funciones que no producen efectos laterales, que no tienen estado, que toman unos objetos como entrada y producen otros objetos como salida.

Para simbolizar el hecho de que las funciones toman parámetros de entrada y devuelven una única salida, vamos a representar las funciones como un símbolo especial, una pequeña casa invertida con unas flechas en la parte superior que representan las entradas y una única flecha que representa la salida. Por ejemplo, la función que eleva al cuadrado un número, multiplicándolo por si mismo, la representaremos de la siguiente forma:



Una de las características fundamentales de la programación funcional es considerar a las funciones como *objetos de primera clase* (entendemos la palabra *objeto* no como un objeto de un lenguaje OO sino en su acepción más genérica de elemento de un lenguaje de programación o de un programa en ejecución). Recordemos que un tipo de primera clase es aquel que:

- Puede ser asignado a una variable
- Puede ser pasado como argumento a una función
- Puede ser devuelto como resultado de una invocación a una función
- Puede ser parte de un tipo mayor

Por ejemplo, en Scheme, al igual que en la mayoría de lenguajes de programación, los valores de booleanos, enteros, caracteres, cadenas, etc. cumplen las condiciones anteriores y son objetos o tipos de primera clase.

Vamos a ver que las funciones son también objetos de primera clase: vamos a poder crear funciones sin nombre y asignarlas a variables, pasarlas como parámetro de otras funciones y guardarlas en tipos de datos compuestos como listas.

La posibilidad de usar funciones como objetos de primera clase es una característica fundamental de los lenguajes funcionales. Es una característica de muchos lenguajes multi-paradigma con características funcionales como JavaScript, Python, Swift o incluso en la última versión de Java, Java 8, (donde se denominan *expresiones lambda*).

5.1. Un primer ejemplo

Vamos a empezar con un ejemplo introductorio: la función `map`. La función `map` es una

función de Scheme que recibe una lista y *otra función* y devuelve el resultado de *mapear* (aplicar) esa otra función a todos y cada unos de los elementos de la lista.

La función `map` es lo que se denomina una función *de orden superior*. Se llama así porque toma como uno de sus parámetros **otra función**.

Veamos un ejemplo de cómo funciona. Primero definimos varias funciones que vamos a utilizar de ejemplo. Todas van a tener un único número como parámetro y van a devolver una transformación de ese número:

```
(define (doble x)
   (* x 2))

(define (cuadrado x)
   (* x x))

(define (suma-1 x)
   (+ x 1))
```

Y ahora podemos comprobar cómo funciona `map`, aplicando estas funciones a todos los elementos de una lista:

```
(map doble '(1 2 3 4 5)) ⇒ (2 4 6 8 10)
(map cuadrado '(1 2 3 4 5)) ⇒ (1 4 9 16 25)
(map suma-1 '(1 2 3 4 5)) ⇒ (2 3 4 5 6)
```

Es interesante notar que el parámetro `doble`, `cuadrado` o `suma-1` es una función. Podemos incluso pasar como parámetro una función a la que no le hemos dado nombre, usando lo que se denomina una *expresión lambda*:

```
(map (lambda (x) (+ x 3)) '(1 2 3 4 5)) ⇒ (4 5 6 7 8)
```

La expresión lambda define una función anónima que recibe un parámetro (`x`) y devuelve `(+ x 3)`. En otros lenguajes de programación (como Scala o Java 8) la sintaxis de esta expresión lambda es quizás algo más fácil de leer:

```
(x) -> {x + 3}
```

5.2. Forma especial `lambda`

Todos los objetos de primera clase se pueden crear en tiempo de ejecución:

```
(define mi-cadena "hola")
(+ 2 3)
```

La cadena `"hola"` o los números `2` y `3` se crean en tiempo de ejecución y después se

asignan a una variable o se pasan como parámetros de una función. Cuando en el intérprete escribimos `"hola"` estamos creando una cadena anónima que después vamos a utilizar (darle nombre con un símbolo o pasarla como parámetro). Cualquier tipo de primera clase debe poder ser creado de esta forma.

¿Es posible hacer lo mismo con funciones en Scheme?

Sí, la forma especial `lambda` es la forma en Scheme (y Lisp) de construir funciones en tiempo de ejecución sin darles un nombre. Por ejemplo, la siguiente expresión crea una función sin nombre con un argumento *x* que eleva al cuadrado su argumento

```
(lambda (x) (* x x))
⇒ #<procedure>
```

Sintaxis de la forma especial `lambda`

La sintaxis de la forma especial `lambda` es:

```
(lambda (<arg1> ... <argn>)
    <cuerpo>)
```

El cuerpo del lambda define un *bloque de código* y sus argumentos son los parámetros necesarios para ejecutar ese bloque de código.

Con la función creada por `lambda` podemos

Es importante darse cuenta que el código creado por `lambda` se crea *en tiempo de ejecución*. La invocación a `lambda` devuelve un procedimiento y hay que hacer algo con él.

Podemos darle un nombre a la función recién creada. Por ejemplo, en la siguiente expresión, primero se evalúa la *expresión lambda* y el resultado (el procedimiento de multiplicar a un número consigo mismo) se asocia al identificador `cuadrado`. De esta forma, cuando escribimos `cuadrado` Scheme devuelve el procedimiento asociado a esta variable:

```
> (define cuadrado (lambda (x) (* x x)))
> cuadrado
#<procedure:cuadrado>
```

Si escribimos

```
(cuadrado 3) ⇒ 9
```

Scheme evalúa el símbolo `cuadrado` y devuelve la función que después invoca pasándole el argumento 3.

También podemos invocar la función sin darle un nombre:

```
((lambda (x) (* x x)) 3) ⇒ 9
```

El paréntesis a la izquierda de `lambda` invoca a la forma especial y construye la función anónima y el primer paréntesis de la expresión invoca a la función creada por el `lambda` con el argumento 3.

Y también podemos pasarla como parámetro de otra función, como en el siguiente ejemplo, que suma todos los números de una lista después de aplicarles a cada uno la función que se pasamos como parámetro:

```
(define (suma-lista-f f lista)
    (if (null? lista)
        0
        (+ (f (car lista))
           (suma-lista-f f (cdr lista)))))

(suma-lista-f (lambda (x) (* x x)) '(1 2 3 4 5))
⇒ 55
```

Es importante remarcar que con `lambda` estamos creando una función en *tiempo de ejecución*. Es código que creamos para su posterior invocación.

No hay que dejar que la sintaxis o la palabra `lambda` confunda. Lo único que estamos haciendo es crear *un bloque de código* y definir sus argumentos. Quizás ayuda ver cómo se realiza la definición anterior en distintos lenguajes.

Java 8

```
Integer x -> {x*x}
```

Scala

```
(x:Int) => {x*x}
```

Objective C

```
^int (int x)
{
   x*x
};
```

¿Qué hay en el nombre de una función?

Tras conocer `lambda` ya podemos explicarnos por qué cuando escribimos en el intérprete de Scheme el nombre de una función, se evalúa a un *procedure*:

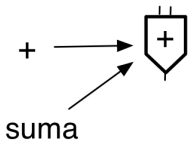
```
> +
```

```
⇒ <procedure:+>
```

El identificador se evalúa y devuelve el *objeto función* al que está ligado. En Scheme los nombres de las funciones son realmente símbolos a los que están ligados *objetos de tipo función*. Gracias a esto podemos asignar funciones ya existentes a nuevos identificadores usando `define`, como en el ejemplo siguiente:

```
> +
⇒ <procedure:+>
> (define suma +)
> (suma 1 2 3 4)
⇒ 10
```

Es muy importante darse cuenta que la expresión `(define suma +)` se evalúa de forma idéntica a `(define y x)`. Primero se evalúa el identificador `+`, que devuelve el *objeto función* suma, que se asigna a la variable `suma`. El resultado final es que tanto `+` como `suma` tienen como valor el mismo procedimiento:



Azucar sintáctico

La forma especial `define` para definir una función no es más que *azucar sintáctico*. La forma especial

```
(define (<nombre> <args>)
    <cuerpo>)
```

siempre se convierte internamente en:

```
(define <nombre>
    (lambda (<args>)
        <cuerpo>))
```

Por ejemplo

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

es equivalente a:

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

Predicado `procedure?`

Podemos comprobar si algo es una función utilizando el predicado de Scheme `procedure?`.

Por ejemplo:

```
(procedure? (lambda (x) (* x x)))
⇒ #t
(define suma +)
(procedure? suma)
⇒ #t
(procedure? '+)
⇒ #f
```

5.3. Las funciones son objetos de primera clase en Scheme

Hemos visto que las funciones pueden asignarse a variables. También cumplen las otras condiciones necesarias para ser consideradas objetos de primera clase.

Una función puede ser el argumento de otra función

Por ejemplo, podemos definir la función `aplica` que toma una función `f` y dos valores `x` e `y` y devuelve el resultado de invocar a la función `f` con `x` e `y`:

```
(define (aplica f x y)
   (f x y))

(aplica + 2 3) ⇒ 5
(aplica * 4 5) ⇒ 10
(aplica string-append "hola" "adios") ⇒ "holaadios"
(aplica (lambda (x y) (sqrt (+ (* x x) (* y y)))) 3 4) ⇒ 5
```

Otro ejemplo, la función `aplica-2` que toma dos funciones `f` y `g` y un argumento `x` y devuelve el resultado de aplicar `f` a lo que devuelve la invocación de `g` con `x`:

```
(define (aplica-2 f g x)
   (f (g x)))

(define (suma-5 x)
   (+ x 5))
(define (doble x)
   (+ x x))
(aplica-2 suma-5 doble 3)
⇒ 11
```

Una función puede ser el valor devuelto por otra

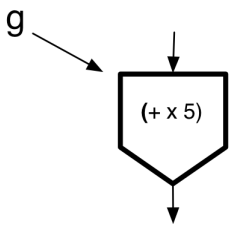
La siguiente función `hacer-sumador-5` que crea y devuelve una función que suma 5 a su

entrada.

```
(define (hacer-sumador-5)
   (lambda (x) (+ x 5)))

(define g (hacer-sumador-5))
```

Cuando llamamos a `(define g (hacer-sumador-5))` se crea en tiempo de ejecución una función que se guarda en la variable `g`. Es una función de un argumento que suma 5 a su entrada.



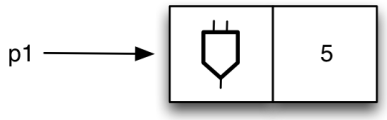
Podemos invocarla como cualquier otra función:

```
(g 10)
⇒ 15
```

Una función pueden formar parte de otras estructuras de datos

La última característica de los tipos de primera clase es que pueden formar parte de tipos de datos compuestos, como una pareja. Por ejemplo, podemos crear una pareja que contenga en su parte izquierda una función y en su parte derecha un dato:

```
(define p1 (cons (lambda (x) (+ x 3))
                 5))
```



¿Cómo podríamos llamar a la función de la parte izquierda pasándole el valor de la derecha como argumento?:

```
((car p1) (cdr p1)) ⇒ 8
```

Podemos hacer más legible la expresión anterior encapsulándola en una función `procesa-pareja`:

```
(define (procesa-pareja p)
    ((car p) (cdr p)))
```

De forma que podríamos llamar a `procesa-pareja` para evaluar el contenido de una pareja con estas características:

```
(define p1 (cons (lambda (x) (+ x 3))
                 5))
(define p2 (cons cuadrado 10))

(procesa-pareja p1)
⇒ 8
(procesa-pareja p2)
⇒ 100
```

Por último, veamos un ejemplo en el que guardamos funciones en listas.

Definimos una función `(aplica-funcs lista-funcs x)` que aplica de forma sucesiva todas las funciones de lista-funcs al número `x`.

Por ejemplo, si en lista-funcs tenemos:

```
(cuadrado cubo suma-1)
```

La llamada a `(aplica-funcs lista-funcs 5)` debería devolver:

```
(cuadrado (cubo (suma-1 5))
46656
```

El caso general de la recursión de la función `aplica-funcs` se define como:

> Para aplicar una lista de funciones a un número debemos aplicar la primera función al resultado de aplicar el resto de funciones al número

El caso base sería en el que la lista de funciones tiene sólo una función.

La implementación en Scheme es:

```
(define (aplica-funcs lista-funcs x)
    (if (null? (cdr lista-funcs))
        ((car lista-funcs) x)
        ((car lista-funcs)
            (aplica-funcs (cdr lista-funcs) x))))
```

Hay que notar los paréntesis en la expresión `((car lista-funcs) x)`. Se obtiene la primera función de la lista y la aplica al argumento `x`.

Un ejemplo de uso de la función:

```
(define lista-funcs (list (lambda (x) (* x x))
```

```
                          (lambda (x) (* x x x))
                          (lambda (x) (+ x 1))))
(aplica-funcs lista-funcs 5)
⇒ 46656
```

Función `apply`

La función `apply` es un ejemplo de una función definida en Scheme que toma como parámetro otra función. En concreto toma dos parámetros: una función y una lista de elementos:

```
(apply f lista)
```

Devuelve el resultado de llamar a la función `f` con la lista de argumentos como parámetros. Por ejemplo:

```
(apply + '(1 2 3 4))
⇒ 10
(apply cuadrado '(3))
⇒ 9
```

Cuidado, los siguientes ejemplos no funcionan. Piensa por qué:

```
(apply 'cuadrado '(3))
(apply cuadrado 3)
```

5.4. Funciones de orden superior

Generalización

La posibilidad de pasar funciones como parámetros de otras es una poderosa herramienta de abstracción. Por ejemplo, supongamos que queremos calcular el sumatorio de `a` hasta `b`:

```
(define (sum-x a b)
    (if (> a b)
        0
        (+ a (sum-x (+ a 1) b))))

(sum-x 1 10)
⇒ 55
```

Supongamos ahora que queremos calcular el sumatorio de `a` hasta `b` sumando los números al cuadrado:

```
(define (sum-cuadrado-x a b)
    (if (> a b)
        0
```

```
        (+ (* a a) (sum-cuadrado-x (+ a 1) b))))

(sum-cuadrado-x 1 10)
⇒ 385
```

Y el sumatorio de `a` hasta `b` sumando los cubos:

```
(define (sum-cubo-x a b)
    (if (> a b)
        0
        (+ (* a a a) (sum-cubo-x (+ a 1) b))))

(sum-cubo-x 1 10)
⇒ 3025
```

Vemos que el código de las tres funciones anteriores es muy similar, cada función la podemos obtener haciendo un *copy-paste* de otra previa. Lo único que cambia es la función a aplicar a cada número de la serie.

Siempre que hagamos *copy-paste* al programar tenemos que empezar a sospechar que no estamos generalizando suficientemente el código. Un *copy-paste* arrastra también *bugs* y obliga a realizar múltiples modificaciones del código cuando en el futuro tengamos que cambiar cosas.

La posibilidad de pasar una función como parámetro es una herramienta poderosa a la hora de generalizar código. En este caso, lo único que cambia en las tres funciones anteriores es la función a aplicar a los números de la serie. Podemos tomar esa función como un parámetro adicional y definir una función genérica `sum-f-x` que generaliza las tres funciones anteriores. Tendríamos el sumatorio desde `a` hasta `b` de `f(x)`:

```
(define (sum-f-x f a b)
    (if (> a b)
        0
        (+ (f a) (sum-f-x f (+ a 1) b))))
```

Las funciones anteriores son casos particulares de esta función que las generaliza. Por ejemplo, para calcular el sumatorio desde 1 hasta 10 de `x` al cubo:

```
(define (cubo x)
    (* x x x))

(sum-f-x cubo 1 10)
⇒ 3025
```

Funciones de orden superior

Llamamos funciones de orden superior (*higher order functions* en inglés) a las funciones que

toman otras como parámetro o devuelven otra función. Permiten generalizar soluciones con un alto grado de abstracción.

Los lenguajes de programación funcional como Scheme, Scala o Java 8 tienen ya predefinidas algunas funciones de orden superior que permiten tratar listas o *streams* de una forma muy concisa y compacta.

Vamos a comenzar viendo la función `map` de Scheme, como ejemplo típico de función de orden superior. Después definiremos nosotros las funciones `filter` y `fold`. Y terminaremos viendo cómo la utilización de funciones de orden superior es una excelente herramienta de la programación funcional que permite hacer código muy conciso y expresivo.

Función `map`

La combinación de funciones de nivel superior con listas es una de las características más potentes de la programación funcional.

Veamos la función de Scheme `(map funcion lista)` como un primer ejemplo. La función `map` toma como parámetros una función y una lista. Devuelve una nueva lista resultante de aplicar la función a cada uno de los elementos de la lista inicial.

```
(map cuadrado '(1 2 3 4 5))
⇒ (1 4 9 25)
```

También es posible que la función a mapear tenga más de un parámetro. En ese caso hay que pasarle como argumentos tantas listas como parámetros tenga la función. Los elementos de las listas se irán cogiendo de forma correlativa como parámetros de la función que se mapea:

```
(define (suma x y)
    (+ x y))
(map suma '(1 2 3) '(4 5 6))
⇒ (5 7 9)
(map * '(1 2 3) '(4 5 6))
⇒ (4 10 18)
```

Otro ejemplo, en el que `map` toma como parámetro una función anónima creada con `lambda` para aplicarla a una lista:

```
(map (lambda (x) (+ x 5)) '(1 2 3)')
⇒ (6 7 8)
```

Podríamos definir una función en la que definimos como parámetro el número a sumar:

```
(define (suma-n lista n)
    (map (lambda (x) (+ x n)) lista))
```

```
(suma-n '(1 2 3 4) 10)
⇒ (11 12 13 14)
```

Implementación de `map`

Un buen ejercicio es intentar implementar la función `map` nosotros, de forma recursiva. Llamamos a la función `mi-map` para no confundirla con la función `map` de Scheme:

```
(define (mi-map f lista)
    (if (null? lista)
        '()
        (cons (f (car lista))
              (mi-map f (cdr lista)))))
```

Función `filter`

Vamos a definir una nueva función de orden superior que trabaja sobre listas.

La función `(filter predicado lista)` toma como parámetro un predicado y una lista y devuelve como resultado los elementos de la lista que cumplen el predicado.

Se implementa de la siguiente forma:

```
(define (filter pred lista)
  (cond
    ((null? lista) '())
    ((pred (car lista)) (cons (car lista)
                              (filter pred (cdr lista))))
    (else (filter pred (cdr lista)))))
```

Un ejemplo de uso:

```
(filter even? '(1 2 3 4 5 6 7 8))
⇒ (2 4 6 8)
```

Función `fold`

Por último vamos a definir una potente función de orden superior, que permite recorrer una lista aplicando una función binaria de forma acumulativa a sus elementos.

Se trata de la función `(fold func base lista)` (función *plegado*) que toma como parámetros una función binaria `func`, un caso base a aplicar con el último elemento de la lista, y una lista de elementos.

La definición recursiva es:

> Para hacer el *fold* de una función `f` de dos argumentos y de una lista, debemos llamar a `f` con el primer elemento de la lista y el resultado de hacer el *fold* del resto de la

> lista. En el caso en que la lista no tenga elementos, se devolverá el caso base.

Por ejemplo:

```
(fold + 0 '(1 2 3))
⇒ (+ 1 (fold + 0 '(2 3)))
⇒ (+ 1 (+ 2 (fold + 0 '(3))))
⇒ (+ 1 (+ 2 (+ 3 (fold + 0 '()))))
⇒ (+ 1 (+ 2 (+ 3 0)))
⇒ 6
```

La implementación recursiva en Scheme es la siguiente:

```
(define (fold func base lista)
  (if (null? lista)
      base
      (func (car lista) (fold func base (cdr lista)))))
```

Podemos comprobar la potencia de la función `fold` con los siguientes ejemplos:

```
(fold string-append "" '("hola" "que" "tal"))
⇒ "holaquetal"
(fold (lambda (x y) (* x y)) 1 '(1 2 3 4 5 6 7 8))
⇒ 40320
(fold cons '() '(1 2 3 4))
⇒ (1 2 3 4)
```

Uso de funciones de orden superior

El uso de funciones de orden superior permite realizar un código muy conciso y expresivo. Ya no es necesario escribir la recursión de forma explícita, sino que la función de orden superior es la que se encarga de abstraerla.

Por ejemplo, supongamos que queremos definir la función `(contienen-letra caracter lista-pal)` que devuelve las palabras de una lista que contienen un determinado carácter.

Por ejemplo:

```
(contienen-letra #\a '("En" "un" "lugar" "de" "la" "Mancha")) ⇒ ("lugar" "la" "Manc
```

Podemos implementar `contienen-letra` usando la función de orden superior `filter`, y pasando como función de filtrado una función auxiliar que creamos con la llamada a `lambda` y que comprueba si la palabra contiene el carácter:

```
(define (contienen-letra caracter lista-pal)
   (filter (lambda (pal)
```

```
              (letra-en-pal? caracter pal)) lista-pal))
```

Sólo nos falta implementar la función `letra-en-pal?` que comprueba si una palabra contiene un carácter. Usando las funciones `primero` u `resto` definidas anteriormente que nos devuelven el primer carácter de una cadena y el resto de la cadena, la podemos implementar con una recursión sencilla:

```
(define (letra-en-pal? caracter palabra)
   (cond
      ((equal? "" palabra) #f)
      ((equal? caracter (primero palabra)) #t)
      (else (letra-en-pal? caracter (resto palabra)))))
```

6. Ambito de variables, let y closures

6.1. Ámbitos de variables

Ámbito global de variables

Ahora que hemos introducido la forma especial `lambda` y la idea de que una función es un objeto de primera clase, podemos revisar el concepto de ámbito de variables e introducir un importante concepto: clausura (*closure* en inglés).

Vamos a definir un nuevo modelo de evaluación de expresiones, que contemple la posibilidad de definir variables locales y variables globales.

Definimos el concepto de *ámbito* como un conjunto de asociaciones entre variables y valores creados en tiempo de ejecución. Representamos gráficamente un ámbito como una caja con las variables definidas y sus valores. En los ámbitos se pueden evaluar expresiones y las variables toman el valor de la variable en ese ámbito.

En Scheme existe un ámbito global de las variables en el que se les da valor utilizando la forma especial define.

```
(define a "hola")
(define b (string-append "adios" a))
(define cuadrado (lambda (x) (* x x)))
```

Todas estas variables se definen en el *ámbito global*.

Ámbito local

Además del ámbito global definimos *ámbitos locales*.

Cuando se invoca a una función se crea un ámbito local en el que los argumentos de la función toman los valores de los parámetros usados en la llamada a la función y en el que se evalúa el cuerpo.

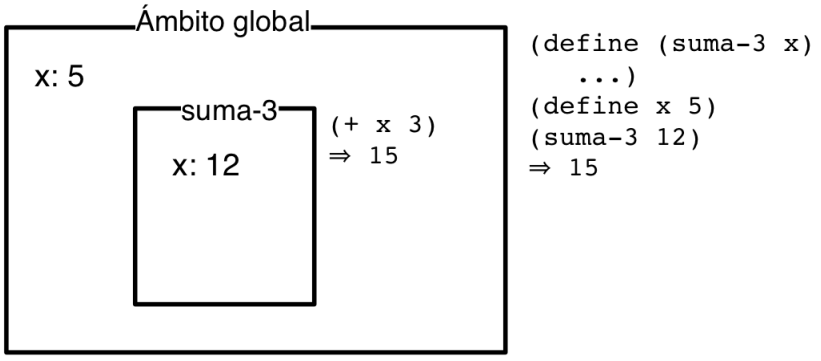
Por ejemplo, supongamos las expresiones:

```
(define x 5)
(define (suma-3 x)
   (+ x 3))

(suma-3 12)
⇒ 15
```

La invocación a `(suma-3 12)` crea un ámbito local en el que la variable `x` (el argumento de `suma-3`) toma el valor 12. Dentro de ese ámbito se evalúa el cuerpo de la función, la expresión `(+ x 3)`, devolviéndose el valor 15. Escribimos a la derecha del ámbito las expresiones que se evalúan en él.

La siguiente figura representa los ámbitos creados:

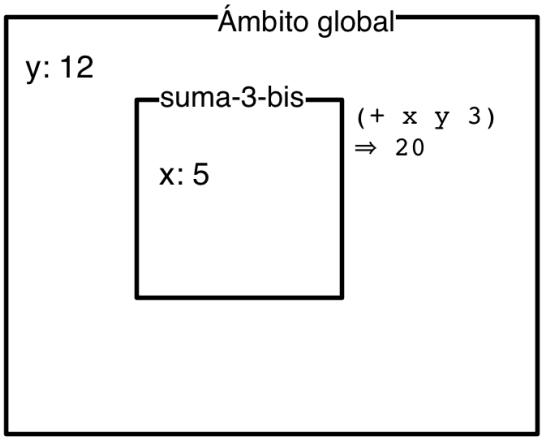


En el ámbito local también se pueden utilizar variables definidas en el ámbito superior. Por ejemplo:

```
(define y 12)
(define (suma-3-bis x)
   (+ x y 3))

(suma-3-bis 5)
⇒ 20
```

La expresión `(+ x 3 y)` se evalúa en el ámbito local en el que `x` vale 5, pero `y` no está definida. Se utiliza la definición de `y` en el ámbito superior:




```
(define (suma-3-bis x)
   ...)
(define y 12)
(suma-3-bis 5)
⇒ 20
```

6.2. Clausuras

¿Qué sucede cuando creamos una función dentro un ámbito local?

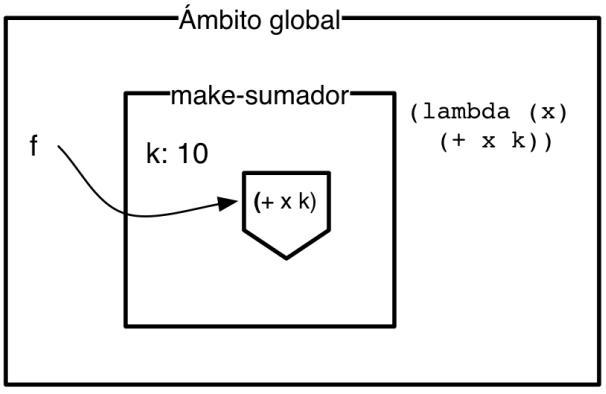
Supongamos que definimos la siguiente función `(make-sumador k)`:

```
(define (make-sumador k)
    (lambda (x) (+ x k)))

(define f (make-sumador 10))
```

La llamada a `(make-sumador 10)` crea un ámbito local en el que `k` vale 10. Dentro de ese ámbito se realiza la llamada a `(lambda (x) (+ x k))` que crea una función en tiempo de ejecución, la devuelve y queda ligada a la variable `f`.

Podríamos representar el resultado con el siguiente diagrama de ámbitos:



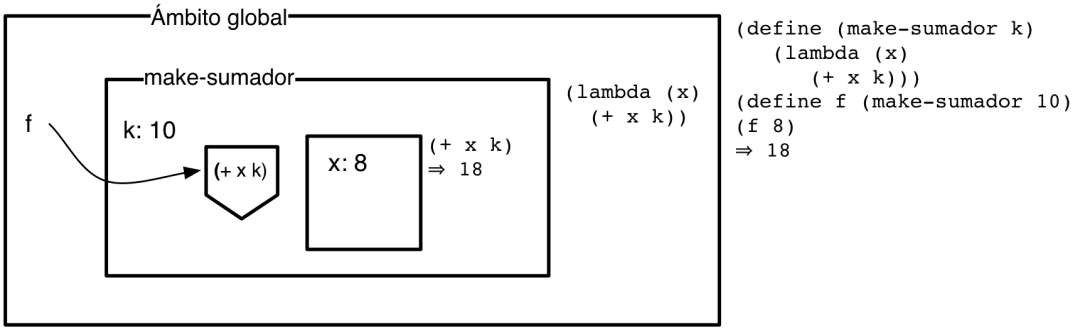

```
(define (make-sumador k)
   (lambda (x)
      (+ x k)))
(define f (make-sumador 10)
```

Supongamos que ahora invocamos a `(f 8)`. El parámetro de `f`, `x`, toma el valor 8. ¿Qué valor va a devolver esta expresión? ¿En qué ámbito se va a evaluar el cuerpo `(+ x k)`? ¿Cuánto valdrá `k`?

```
(f 8)
⇒ 18
```

Lo que ha sucedido es que la función a la que apunta `f` es una *clausura* y su invocación se evalúa en el mismo ámbito local en el que la función se ha creado. La variable local `k` queda *encerrada* en el ámbito junto con la función devuelta por `lambda`.

El dibujo de ámbitos sería el siguiente:



Closures

Lo que hemos visto se define con el nombre de *closure*: una función creada en tiempo de ejecución junto con el entorno en el que ésta se ha creado.

Mediante el uso de clausuras es posible asociar una función con una o más variables libres a un ámbito local en el que se definen estas variables.

En muchos lenguajes de programación modernos existe el concepto de *closure*, aunque pueden recibir nombre variados.

Por ejemplo, en Objective-C recibe el nombre de *block*:

```
#include <stdio.h>
#include <Block.h>
typedef int (^IntBlock)();

IntBlock MakeCounter(int start, int increment) {
        __block int i = start;

            return Block_copy( ^ {
                int ret = i;
                i += increment;
                return ret;
        });
}

int main(void) {
        IntBlock mycounter = MakeCounter(5, 2);
        printf("First call: %d\n", mycounter());
        printf("Second call: %d\n", mycounter());
        printf("Third call: %d\n", mycounter());

        /* because it was copied, it must also be released */
```

```
        Block_release(mycounter);

        return 0;
}
/* Output:
        First call: 5
        Second call: 7
        Third call: 9
*/
```

Funciones anónimas

Muchos lenguajes de programación permiten utilizar código como objeto de primera clase y pasarlo como parámetro a otras funciones. Este tipo de funciones también recibe el nombre de *funciones anónimas* (ver un interesante artículo en la Wikipedia con ejemplos en muchos lenguajes de programación).

Es muy importante tener en cuenta de que no todas las funciones anónimas usan el mismo tipo de semántica. En lenguajes dinámicos como Scheme, las funciones son creadas en tiempo de ejecución, quedando encerradas en el ámbito en que han sido creadas y ejecutándose precisamente en ese ámbito. También sucede en otros lenguajes como Objective-C. Es lo que se llama *deep binding* y cuyos efectos hemos visto en los ejemplos anteriores.

En otros lenguajes las funciones anónimas se implementan mediante código *pre-compilado* que se evalúa en el ámbito en el que finalmente se ejecuta. Esto no daría lugar a una clausura.

6.3 Forma especial let

Forma especial let

En Scheme se define la forma especial let que permite crear un ámbito local en el que se da valor a variables.

Sintaxis:

```
    (let ((<var1> <exp-1>)
          ...
          (<varn> <exp-n>))
    <exp>)
```

Las variables var1, … varn toman los valores devueltos por las expresiones exp1, … expn y la exp se evalúa con esos valores.

Por ejemplo:

```
(let ((x 1)
```

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

Let mejora la legibilidad de los programas

El uso de `let` permite crear variables locales en las funciones, aumentando la legibilidad de los programas.

Por ejemplo:

```
(define (distancia x1 y1 x2 y2)
   (let ((dx (- x2 x1))
         (dy (- y2 y1)))
      (sqrt (+ (cuadrado dx)
               (cuadrado dy)))))
```

Otro ejemplo:

```
(define (intersecta-intervalo a1 a2 b1 b2)
    (let ((dentro-b1 (and (>= b1 a1)
                          (<= b1 a2)))
          (dentro-b2 (and (>= b2 a1)
                          (<= b2 a2))))
       (or dentro-b1 dentro-b2)))
```

El ámbito de las variables definidas en el let es local

Las variables definidas en el `let` sólo tienen valores en el ámbito de la forma especial. El funcionamiento es idéntico al que hemos visto anteriormente de una llamada a una función.

```
(define x 5)
(let ((x 1)
      (y 2))
   (+ x y))
⇒ 3
x ⇒ 5
y ⇒ error, no definida
```

Cuando ha terminado la evaluación del `let` el ámbito local desaparece y quedan los valores definidos en el ámbito superior.

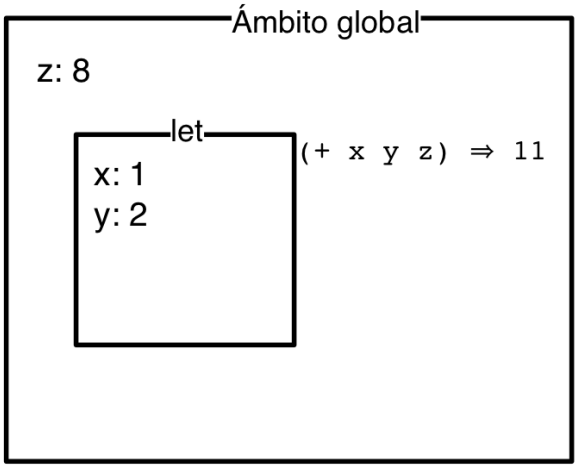
Let permite usar variables definidas en un ámbito superior

Al igual que en la invocación a funciones, desde el ámbito definido por el `let` se puede usar el ámbito superior. Por ejemplo, en el siguiente código se usa la variable z definida en el ámbito superior.

```
(define z 8)
```

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

El diagrama de ámbitos que representa gráficamente lo que sucede con las expresiones anteriores es el siguiente:




```
(define z 8)
(let ((x 1)
      (y 2))
   (+ x y z)) ⇒ 11
```

Variables en las asignaciones del let

Las expresiones del let se evalúan todas antes de asociar ningún valor con las variables. No se realiza una asignación secuencial:

```
(define x 1)
    (let ((w (+ x 3))
          (z (+ w 2))) ;; Error
        (+ w z))
```

Semántica del let

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

Let se define utilizando lambda

La semántica anterior queda clara cuando comprobamos que let se puede definir en función de lambda. En general, la expresión:

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

se puede implementar con la siguiente llamada a lambda:

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

Ejemplo:

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

Equivale a:

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

Se puede hacer una asignación secuencial con let*

La forma especial `let*` permite una asignación secuencial de las variables y las expresiones:

```
(let* ((x (+ 1 2))
       (y (+ x 3))
       (z (+ y x)))
   (* x y z))
```

¿Cómo se puede implementar con let?

```
(let ((x (+ 1 2)))
   (let ((y (+ x 3)))
      (let ((z (+ y x)))
         (* x y z))))
```

El primer let crear un ámbito local en el que se evalúa el segundo let, que a su vez crea otro ámbito local en el que se evalúa el tercer let. Se crean tres ámbitos locales anidados, y en el último se evalúa la expresión `(* x y z)`.
