Capítulo 5

Procedimientos Definidos Localmente

5.1 Distinguiendo entre definiciones globales y locales

Todas las primitivas (o procedimientos) disponibles en Scheme (p.ej., **car**, **cons**, **cdr**, etc.) se definen en lo que se denomina el **ambiente global inicial**. Un **ambiente** en general es realmente una tabla que contiene información respecto a la definición de un conjunto de procedimientos. En este caso, nuestras primitivas están pre-definidas, y el usuario normalmente no tiene acceso a tales definiciones.

Hasta ahora, hemos usados **define** para crear nuevos procedimientos. Este tipo de definición es llamada global porque permanece en efecto hasta que el usuario abandone el intérprete. También podemos usar **define** para crear variables globales. Por ejemplo:

>(**define** x 10)

Como en la mayoría de los demás lenguajes de programación, en Scheme también hay una forma de definir variables locales e incluso procedimientos locales que permanecerán en efecto sólo durante su ejecución y posteriormente serán removidos de memoria. Estas definiciones locales son muy útiles para ciertas aplicaciones, aunque tenemos que ser cuidadosos con su uso, porque la sintaxis orientada a los paréntesis de Scheme tiende a con-

fundir a los usuarios novatos acerca de las asociaciones correspondientes. Un tipo de variable local que ya hemos usado antes es el caso cuando pasamos parámetros a un procedimiento a través de una expresión lambda. Considere el siguiente ejemplo:

(**lambda** (a b) (* a b))

Las variables **a** y **b** están asociadas localmente a la expresión **(* a b)**. Ahora veamos lo que ocurre cuando asociamos valores numéricos a estas variables:

((**lambda** (a b) (* a b)) 5 2)

Esto regresa **10** porque la **a** está asociada localmente a **5** y **b** está asociada localmente a **2**.

Una variable que ocurra en una expresión lambda que no esté asociada a nada en una expresión es llamada **libre** con respecto a esa expresión. La definición de tales variables puede encontrarse en el ambiente global o en un ambiente local de otra expresión lambda. Por ejemplo:

```
> ((lambda (x)
((lambda (y) (+ x y)) 9))
  11)
```

Esto retorna **20** porque aunque **x** está libre en el cuerpo de la expresión lambda más interna, su asociación puede encontrarse en el ambiente local de la otra expresión lambda.

5.2 Entorno Léxico

Se dice que una expresión está en el **entorno** de una variable **x** si dicha expresión se encuentra en el cuerpo de una expresión lambda en la que **x** está en la lista de parámetros. Viendo a un programa en Scheme, se puede decir fácilmente si una cierta expresión dada está en el cuerpo de alguna expresión lambda y determinar, en consecuencia, si las variables en dicha expresión están en el entorno de la expresión lambda. Un lenguaje en el cual el entorno de las variables puede determinarse a partir de ver simplemente

sus programas se denomina **de entorno léxico**. Scheme es un lenguaje de este tipo.

Para asociar la variable **var** al valor de una expresión **val** en la expresión **cuerpo**, usamos una expresión **let**. Su sintaxis es:

(**let** ((*var val*)) *cuerpo*)

Para hacer varias asociaciones locales de este tipo en la expresión *cuerpo* (p.ej., para asociar var_1 con val_1, var_2 con val_2, ..., var_n con val_n), entonces escribimos

(**let** ((var_1 val_1) (var_2 val_2), ..., (var_n val_n)) *cuerpo*)

El entorno de cada una de las variables var_1, var_2, ..., var_n es sólo el *cuerpo* dentro de la expresión **let**. Por ejemplo, la expresión

```
>(let ((x 10) (y 2)) (* x y))
20
```

Aquí, **x** está asociada con 10, **y** está asociada con 2 y, por tanto, la multiplicación de ambos regresa **20**.

Las asociaciones locales siempre tienen prioridad sobre las asociaciones globales o no locales. Por ejemplo:

```
> (define x 4)          | > (let ((x 8))
> (+ x 1)               |     (begin
5                       |     (writeln (sub1 x))
> (let ((x 9))          |     (let ((x 4))
      (+ x 1))          |     (writeln (sub1 x)))
10                      |     (sub1 x)))
> (+ x 1)               | 7
5                       | 3
                        | 7
```

La expresión **let**

(**let** ((var_1 val_1) (var_2 val_2) ... (var_n val_n)) *cuerpo*)

es equivalente a la siguiente aplicación de una expresión lambda:

((**lambda** ((var_1 var_2 ... var_n) *cuerpo*) val_1 val_2 ... val_n)

Es importante saber a todo momento que ambiente usar para evaluar una expresión, o de lo contrario podríamos obtener resultados sorpresivos. Considere el siguiente ejemplo:

```
(define multx
(let ((x 50))
(lambda (y)
(* x y))))
```

Ahora, si realizamos la siguiente asociación:

```
> ((let ((x 10))
        (multx 5))
250
```

La razón por la que esta invocación retorna **250** es porque la expresión **lambda** está dentro del entorno de la expresión **let**.

Cuando se evalúa una expresión **let** que contiene varias parejas de asociaciones, tales como

(**let** ((var_1 val_1) (var_2 val_2) ... (var_n val_n)) *cuerpo*)

no hay garantía en torno al orden en que se evaluarán las parejas de asociaciones (var_k val_k). Hay, sin embargo, un constructor de asociaciones que hace que las parejas de asociaciones se evalúen de izquierda a derecha, y en la que cualquier parte *val* puede contener *var*s de parejas previas de asociaciones. Su nombre es **let***:

(**let*** ((var_1 val_1) (var_2 val_2) ... (var_n val_n)) *cuerpo*)

y es equivalente a una secuencia de expresiones **let** anidadas, cada una de las cuales contiene una de las parejas de asociaciones:

(**let** ((var_1 val_1))
  (**let** ((var_2 val_2))
    ...
    (**let** ((var_n val_n))
      cuerpo) ...))

5.3 Recursividad en Asociaciones Locales

El valor de una expresión **lambda** es un procedimiento (llamado también cierre), que consta de 3 partes:

1. La lista de parámetros.
2. El cuerpo de la expresión lambda.
3. El ambiente en el cual se asocian las variables libres en el cuerpo al momento de evaluar la expresión lambda.

Cuando se aplica el procedimiento, sus parámetros se asocian a sus argumentos y se evalúa el cuerpo, buscándose las variables libres en el ambiente almacenado en el cierre.

En una expresión **let**

(**let** ((*var val*)) *cuerpo*)

cualquier variable que ocurra en *val* y que no esté asociada en la expresión *val* misma, debe ser asociada afuera de la expresión **let** (es decir, en un ambiente no local). Esto ocurre porque al evaluar *val*, Scheme busca fuera de la expresión **let** las asociaciones posibles de cualquier variable libre que ocurra en *val*. Por lo tanto

```
> (let ((fib (lambda (n)
               (if (< n 2) n
               (+ (fib (- n 1)) (fib (- n 2)))))))
       (fib 7))
```

retornará el mensaje: **Error: undefined variable fib (package user)**.
Este mensaje se refiere a las definiciones de **fib** que ocurren en la expresión

lambda, pero que no están asociadas fuera de la expresión **let**. Por ende, si queremos usar una definición recursiva en la parte *val* de una expresión **let**, tenemos que evitar el problema de las variables no asociadas que hemos confrontado en este caso. Una forma de afrontar el problema es usar las expresiones **letrec** que hacen asociaciones locales en las que es posible realizar llamadas recursivas.

Letrec tiene la misma sintaxis que **let**:

(**letrec** ((var_1 val_1) (var_2 val_2) ... (var_n val_n)) *cuerpo*)

pero ahora cualquiera de las variables var_1, var_2, ..., var_n puede aparecer en cualquier parte de las expresiones val_1, val_2, ..., val_n de forma que pueda realizarse recursividad en la definición de estas variables. Por ejemplo:

```
> (letrec ((fib (lambda (n)
                  (if (< n 2) n
                  (+ (fib (- n 1)) (fib (- n 2)))))))
          (fib 7))
13
```

ahora retorna la respuesta correcta.

También podemos tener recursividad mutua en una expresión **letrec**, tal y como se ilustra en el siguiente ejemplo:

```
(letrec ((par? (lambda (x)
        (or (zero? x) (impar? (sub1 x)))))
        (impar? (lambda (x)
        (and (not (zero? x)) (par? (sub1 x))))))
(par? 5))
#f
```

Finalmente, podemos tambin escribir una versión iterativa de **fib** usando **letrec**:

```
(define fib
(lambda(n)
(letrec (k acc1 acc2)
```

(**if** (= k 1) acc2
(fib-it (sub1 k) acc2
(+ acc1 acc2))))))
(fib-it n 0 1))))

Ahora, si escribimos:

>(**fib** 10)

Se imprime el número **55**, lo que indica que nuestro procedimiento funciona correctamente.

5.3.1 Problemas Resueltos

1. Encuentre el valor de cada una de las expresiones siguientes, escribiendo los ambientes locales para cada una de las expresiones **let** anidadas. Dibuje flechas desde cada variable hacia el parámetro al cual está ligada en una expresión **lambda** o **let**. Dibuje también una flecha desde el parámetro hasta el valor al cual está asociado.

   (a) (**let** ((a 5))
   (**let** ((fun (**lambda** (x) (**max** x a))))
   (**let** ((a 10)
   (x 20))
   (fun 1))))

   Esta expresión retorna **5**. En la expresión lambda, **x** está asociada a **1** porque ese es el valor que se pasa a través de **(fun 1)**. Por lo tanto, **x** se asocia a **5** dentro de **(max x a)** porque esa es una definición local. Por ende, lo que realmente estamos calculando es **(max 1 5)**, lo cual retorna **5**.

   (b) (**let** ((a 1) (b 2))
   (**let** ((b 3) (c (+ a b)))
   (**let** ((b 5))
   (**cons** a (**cons** b (**cons** c '()))))))

El valor que se retorna en este caso es **'(1 5 3)**. Primero, busquemos la definición de **a**. Puesto que **a** está libre, se le asocia el **1** (ambiente previo). Posteriormente, buscamos el valor de **b**. De la expresión más interna (la que efectúa los **cons**) es claro que **b** se asocia localmente con **5**. Finalmente, buscamos el valor de **c**. Esta vez, resulta que **c** está libre en la expresión más interna, de manera que buscamos su definición en el ambiente previo. Ahí encontramos que **c** está definida en términos de la suma de **a** y **b**. Ya sabemos que **a** está asociada con **1**, así que la única pregunta que resta es ¿cuál es el valor asociado con **b**? Algunos podrían pensar que **b** se asocia con **3**, pero eso no es cierto, ya que **(b 3)** y **(c (+ a b))** son parte de la misma definición y por lo tanto no se conocen entre ellas. Por lo tanto **b** realmente está libre y por lo tanto se le asocia con **2** (ambiente previo), y **c** se asocia con **3**.

2. Encuentre el valor de cada una de las siguientes expresiones **letrec**:

(a) (**letrec**
((ciclo
(**lambda** (n k)
(**cond**
((**zero?** k) n)
((< n k) (loop k n))
(**else** (ciclo k (**remainder** n k)))))))
(ciclo 9 12))

Esta expresión retorna **3**. Primero, **n**se asocia con **9** y **k** se asocia con **12**.Puesto que **k** es diferente de cero, se le compara contra**n**. Dado que **9** < **12**, entonces losvalores se intercambian e invocamos **(ciclo 12 9)**. Nuevamente,**k** es diferente de cero, de forma que se comparanuevamente con **n**. Esta vez, la expresión retorna**#f** porque **12** > **9**. Posteriormente, invocamos **(ciclo 9 (remainder 12 9))**. El residuo(*remainder*) de dividir **12** entre **9**es **3**, así que realmente estamos invocando**(ciclo 9 3)**. Una vez más, **k** es diferente de cero,de forma que la comparamos contra **n**. **9** no esmenor que **3**, de forma que invocamos **(ciclo 3 (remainder 9 3))**. El residuo de dividir **9** entre**3** es **0**, por lo que estamos realmenteinvocando **(ciclo 3 0)**. Ahora, **k** es **0**,de forma que el valor de **n** (**3** en este caso)es lo que se retorna como resultado final del procedimiento.Esta definición calcula el mínimo común divisor de los 2números proporcionados como argumentos.

(b) (**letrec**
((**ciclo**
(**lambda** (n)
(**if** (**zero?** n)
0
(+ (**remainder** n 10) ((**quotient** n 10)))))))
(ciclo k (**remainder** n k)))))))
(loop 1234))

El valor que retorna esta expresión es **10**. Primero, **n** se asocia con **1234**, de manera que se calcula (+ (**remainder** 1234 10) (ciclo (**quotient** 1234 10))). El procedimiento **remainder** retorna **4** y el procedimiento **quotient** retorna **123**. Por lo tanto, la operación es realmente (+ 4 (ciclo 123)). Recursivamente, invocamos**ciclo** de nuevo, con **n** asociada a **123**. En este caso, tenemos que calcular (+ (**remainder** 123 10)(ciclo (**quotient** 123 10))). El procedimiento **remainder** retorna **3** y el procedimiento **quotient** retorna **12**, de forma que estamos haciendo (+3 (ciclo 12)). Recursivamente, invocamos de nuevo **ciclo**, con **n** asociada con **12**. Aquí, calculamos(+ (**remainder** 12 10) (ciclo (**quotient** 12 10))). El procedimiento **remainder** retorna **2** yel procedimiento **quotient** retorna **1**,así que estamos realmente calculando (+ 2 (ciclo 1)). Recursivamente, invocamos **ciclo** nuevamente, con **n** asociada a **1**. Aquí calculamos (+ (**remainder** 1 10) (ciclo (**quotient** 1 10))).El procedimiento **remainder** retorna **1** y el procedimiento **quotient** retorna **0**, así que realmente lo que estamos calculando es (+1 (ciclo 0)). Recursivamente, invocamos de nuevo **ciclo**, con **n** asociada con **0**. Se activa la primera decisióny se retorna **0**. Propagamos hacia atrás este valor, hasta que calculemos el resultado final, el cual resulta ser **10**. Esta definición obtiene la adición de todos los dígitos que forman el número proporcionado.

3. Encuentre el valor de cada una de las siguientes expresiones **letrec**.

(**letrec** ((misterio
(**lambda** (tupla impares pares)
(**if** (**null?** tupla)
(**append** impares pares)
(**let** ((sig-ent (**car** tupla)))

```
            (if (odd? sig-ent)
                (misterio (cdr tupla)
                          (cons sig-ent impares) evens)
                (misterio (cdr tupla)
                          impares (cons sig-ent evens))))))))
(misterio '(3 16 4 7 9 12 24) '() '()))
```

Esta expresión retorna **'(9 7 3 24 12 4 16)**. Primero,**tupla** se asocia con **'(3 16 4 7 9 12 24)** e**impares** y **pares** se asocian con la lista vacía.La operación básica de esta expresión es colocar cada número en **impares** si es impar o en **pares** si es par.Al finalizar, se unen las dos listas usando **append**, colocando primero a los números impares. El orden en que aparecen los números en el resultado final está invertido con respecto a la lista original, porque se colocaron en **impares** y **pares** haciendo **cons** del número con los elementos previos de la lista. Esto implica que se les colocó en la parte final de la lista correspondiente aunque se leyeron de la parte frontal de la lista original. Obviamente, esta expresión sólo separa números impares de números pares, y luego los mezcla colocando al frente a los números impares.

4. Re-escriba la definición del procedimiento **inserta-todo-izq** (ver Capítulo 4) usando un procedimiento definido localmente que tome a la lista **ls** como su único argumento.

   El procedimiento es el siguiente:

```
(define inserta-todo-izq
(lambda (nueva vieja ls)
(letrec
   ((inserta
(lambda (ls*)
(cond
((null? ls*) '())
((equal? (car ls*) vieja) (cons nueva
(cons vieja (inserta (cdr ls*)))))
((pair? (car ls*)) (cons (inserta (car ls*))
        (inserta (cdr ls*))))
(else (cons (car ls*) (inserta (cdr ls*))))))))
```

(inserta ls))))

5. Re-escriba la siguiente expresión **let**

```
(let ((add2 (lambda (x) (+ x 2)))
            (b (* 3 (/ 2 12))))
  (/ b (add2 b)))
```

como una expresión **lambda** que no use expresiones**let**.

La respuesta es la siguiente:

```
(lambda (add2 b)
(/ b (add2 b)))
  (lambda (x) (+ x 2)) (* 3 (/ 2 12)))
```

5.4 Problemas Propuestos

1. Defina un procedimiento **hexadecimal->decimal** que tome como argumento un número hexadecimal, en forma de lista y regrese su equivalente en decimal. Su procedimiento deberá verificar que el valor proporcionado por el usuario sea válido. Use **let** y/o **letrec** para definir entornos local (o sea, funciones auxiliares) en el procedimiento que resuelva este problema. Pruebe su procedimiento con:

(**hexadecimal->decimal** '(a b c d e f 5)) $\Longrightarrow$ 180150005
(**hexadecimal->decimal** '(z 5)) $\Longrightarrow$
Error: el argumento (z 5) no es aceptable
(**hexadecimal->decimal** '(1 g)) $\Longrightarrow$
Error: el argumento (1 g) no es aceptable
(**hexadecimal->decimal** '(1 0)) $\Longrightarrow$ 16
(**hexadecimal->decimal** '(8 f a 0)) $\Longrightarrow$ 36768
(**hexadecimal->decimal** '()) $\Longrightarrow$ 0

2. Defina un procedimiento **decimal->hexadecimal** el cual será la contraparte del procedimiento definido en el ejercicio anterior.
   Ahora el procedimiento deberá tomar un entero y regresar su equivalente en hexadecimal en forma de una lista. El procedimiento debera verificar que el número proporcionado por el usuario sea un entero mayor o igual que cero. Use **let** y/o **letrec** para definir cualquier procedimiento local (o sea, funciones auxiliares) para escribir el procedimiento que solucione este problema. Evalúe su procedimiento con:

   (**decimal->hexadecimal** 1215) $\Longrightarrow$ '(4 b f)
   (**decimal->hexadecimal** -5) $\Longrightarrow$
   Error: el argumento -5 no es aceptable
   (**decimal->hexadecimal** 4.5) $\Longrightarrow$
   Error: el argumento 4.5 no es aceptable
   (**decimal->hexadecimal** 915) $\Longrightarrow$ '(3 9 3)
   (**decimal->hexadecimal** 0) $\Longrightarrow$ '(0)

3. Escriba las dos expresiones de las partes a) y b) del Ejercicio 1 como expresiones lambda anidadas sin usar ninguna expresión **let**.

4. Considere la definición del procedimiento **mystery** que se muestra a continuación:

```
(define mystery
(lambda (n)
(letrec
   ((mystery-helper
(lambda (n s)
(cond
   ((zero? n) (list s))
(else (append
(mystery-helper (sub1 n) (cons 0 s))
   (mystery-helper (sub1 n) (cons 1 s))))))))
   (mystery-helper n '())))))
```

   ¿Qué produce la invocación (**mystery 4**)? Describa el comportamiento general de **mystery** cuando se le pasa un entero positivo cualquiera.

5. Reescriba el procedimiento **cuenta-todo-fondo** que se realizó en la tarea # 4 usando un procedimiento definido localmente que tome una lista **ls** como un su único argumento. Su respuesta se deberá escribir en un solo **define**, y deberá usar **let** y/o **letrec** para sus definiciones locales. Evalúe su procedimiento con:

(**cuenta-todo-fondo** 'a '((a) b (c a) d)) $\Longrightarrow$ 3
(**cuenta-todo-el-fondo** 'a '((((b (((a)) c)))))) $\Longrightarrow$ 2
(**cuenta-todo-el-fondo** 5 '((((((5) ((6) 5))) 5)))) $\Longrightarrow$ 1
(**cuenta-todo-el-fondo** 5 '()) $\Longrightarrow$ 0