Capítulo 2

Procedimientos y Recursividad

2.1 Introducción

Es muy fácil definir un procedimiento en Scheme. El lenguaje proporciona una forma elegante de hacerlo, basada en el cálculo lambda de Alonzo Church [4]. Supongamos que queremos definir un procedimiento que tome como argumento un valor (ya sea un símbolo o un número) y queremos que regrese una lista que contenga dicho valor como su único elemento:

(**lambda** (*val*) (**cons** *val* '()))

Considere los siguientes ejemplos:

> ((**lambda** (val) (**cons** val '())) 'Carlos)
(Carlos)

> ((**lambda** (val) (**cons** val '())) 5)
(5)

Puesto que no queremos estar realizando esta misma operación una y otra vez, definiremos un procedimiento:

```
(define haz-lista-de-uno ; el punto y coma se puede usar para los comentarios
  (lambda (val)
    (cons val '())))
```

> (haz-lista-de-uno 7)
(7)

> (haz-lista-de-uno 'a)
(a)

Se recomienda que siempre se usen nombres apropiados para los procedimientos. Una observación importante es que el procedimiento puede ser invocado desde otro, e incluso desde sí mismo (a esto se le llama **recursividad**).

El procedimiento **list** toma cualquier número de argumentos y construye una lista que contiene dichos argumentos. Considere los siguientes ejemplos:

(**list** 'a 'b 'c 'd) $\Longrightarrow$ **(a b c d)**
(**list** '(1 2) '(3 4)) $\Longrightarrow$ **((1 2) (3 4))**
(**list**) $\Longrightarrow$ **()**

2.1.1 Problemas Resueltos

1. Defina un procedimiento **rota**, que rote una lista de tres elementos. El procedimiento **rota** regresa una lista que es un rearreglo de la lista que se le da de entrada, de manera que el primer elemento se vuelve el segundo, el segundo se vuelve el tercero, y el tercero se vuelve el primero. Pruebe su procedimiento con:

   (**rota** 'salta 'veloz 'cerca) $\Longrightarrow$ **(cerca salta veloz)**
   (**rota** 'perro 'muerde 'hombre) $\Longrightarrow$ **(hombre perro muerde)**

   Solución 1:

   (**define** rota
   (**lambda** (elem1 elem2 elem3)
   (**cons** elem3 (**cons** elem1 (**cons** elem2 '())))))

   Solución 2:

   (**define** rota
   (**lambda** (elem1 elem2 elem3)
   (**list** elem3 elem1 elem2)))

2.2 Expresiones Condicionales

La palabra clave **cond** se deriva de la palabra **condicional**, y se refiere a una expresión usada para emular la sentencia **case** (sentencia de casos) de los lenguajes imperativos. Considere el siguiente ejemplo:

```
(define tipo-de
 (lambda (elem)
  (cond
((pair? elem) 'pair)
((symbol? elem) 'symbol)
((number? elem) 'number)
((boolean? elem) 'boolean)
((null? elem) 'empty-list)
     (else 'tipo-desconocido))))
```

Advierta que cada clásula se evalúa secuencialmente. Si alguna de ellas regresa **#t**, entonces se retorna su consecuente correspondiente. De lo contrario, se evalúa la siguiente cláusula. La cláusula **else** es opcional. Si se omite y todas las condiciones son falsas, entonces Scheme no especificará el valor que retornará la expresión **cond**. Por ello es recomendable utilizar **else**, para evitar tener expresiones **cond** que regresen un valor no especificado. El formato general de **cond** es:

(**cond**
(condición$_1$ consecuente$_1$)
(condición$_2$ consecuente$_2$)
$\vdots$
(condición$_n$ consecuente$_n$)
(**else** alternativa))

También es posible usar la palabra clave **if** cuando tenemos solamente dos casos, como suele hacerse en la mayoría de los lenguajes de programación tradicionales. Consideremos el siguiente ejemplo en el que se re-escribirá el mismo procedimiento usando **cond** e **if**:

Usando cond:

```
(define car-cond-pair
   (lambda (elem)
(cond
   ((pair? elem) (car elem))
   (else elem))))
```

Usando if:

```
(define car-if-pair
   (lambda (elem)
(if (pair? elem) (car elem) elem)))
```

El formato general de **if** es:

(**if** *condición consecuente alternativa*)

or

(**if** *condición consecuente*)

También podemos usar **if**s anidados para emular expresiones **cond**, pero eso no se recomienda, porque nuestro código generalmente se volverá muy complejo y confuso.

2.3 Condiciones Compuestas

Una condición compuesta es una condición que involucra una combinación de dos o más condiciones simples. Las condiciones compuestas se construyen usando los operadores lógicos:

(**and** $expr_1$ $expr_2$ $expr_3$ ... $expr_n$)
(**or** $expr_1$ $expr_2$ $expr_3$ ... $expr_n$)
(**not** *expr*)

La expresión **and** evalúa cada subexpresión, y si cualquiera de ellas es falsa retorna **#f** y se detiene. Si todas son ciertas, regresa el valor de la última subexpresiøn.

La expresión **or** evalúa cada subexpresión en sucesión. Si cualquiera de ellas es cierta, deja de evaluar el resto de las subexpresiones y el valor de la expresión será cierta. Si todas las subexpresiones son falsas, entonces el valor de la expresión será **#f**.

El valor de la expresión **not** es **#f** cuando *expr* es cierto, y **#t** cuando *expr* es falso.

Scheme tiene la convención de tratar cualquier valor que no es falso como cierto. Por ejemplo:

$$> (\textbf{if}\ \text{'gato 'animal 'mascota}) \Longrightarrow \text{animal}$$

Puesto que la condición 'gato se evalúa como *gato*, lo cual no es falso. Aunque este mecanismo puede usarse para implementar algunos trucos con el intérprete de Scheme, su uso se recomienda únicamente cuando se manejan expresiones Boleanas en las condiciones.

2.3.1 Problemas Resueltos

1. Suponga que **a**, **b** y **c** son expresiones que pueden evaluarse a **#t** y que **e** y **f** son expresiones que se evalúan a **#f**. Decida si las siguientes expresiones son falsas o verdaderas.

   a. (**and** a (**or** b e))
   (and #t (or #t #f))
   (and #t #t) $\Longrightarrow$ **#t**

   b. (**or** e (**and** (**not** f) a c))
   (or #f (and (not #f) #t #t))
   (or #f (and #t #t #t))
   (or #f #t) $\Longrightarrow$ **#t**

   c. (**not** (**or** (**not** a) (**not** b)))
   (not (or (not #t) (not #t)))
   (not (or #f #f))
   (not #f) $\Longrightarrow$ **#t**

d. (**and** (**or** a f) (**not** (**or** b e)))
(and (or #t #f) (not (or #t #f)))
(and #t #f) $\Longrightarrow$ **#f**

2. Decida si las siguientes expresiones son ciertas o falsas si **expr** es alguna expresión Boleana.

a. (**or** (**symbol?** expr) (**not** (**symbol?** expr)))
#t

b. (**and** (**null?** expr) (**not** (**symbol?** expr)))
#f

c. (**not** (**and** (**or** expr #f) (**not** expr)))
#t

d. (**not** (**or** expr #t))
#f

2.4 Recursividad

Cuando un procedimiento se invoca a sí mismo dentro del cuerpo de la expresión lambda que lo define, decimos que el procedimiento es **recursivo**. Considere el siguiente ejemplo:

```
(define ultimo-elemento
   (lambda (ls)
(cond
   ((null? (cdr ls)) (car ls))
(else (ultimo-elemento (cdr ls))))))
```

Esta función regresa el último elemento de alto nivel de una lista:

> (ultimo-elemento '(1 2 3 4 5)) $\Longrightarrow$ **5**
> (ultimo-elemento '(a b (c d))) $\Longrightarrow$ **(c d)**
> (ultimo-elemento '(gato)) $\Longrightarrow$ **gato**
> (ultimo-elemento '((gato))) $\Longrightarrow$ **(gato)**

Ahora definiremos el procedimiento **miembro?**, que decide si su primer argumento es igual a uno de los elementos de alto nivel de la lista que se le proporciona como segundo argumento:

```
(define miembro?
   (lambda (elem ls)
(cond
   ((null? ls) #f)
(else (or (equal? (car ls) elem)
   (miembro? elem (cdr ls)))))))
```

Algunos ejemplos de su uso son los siguientes:

```
> (miembro? 'gato '(perro gallina gato puerco)) ⟹ #t
> (miembro? 'zorra '(perro gallina gato puerco)) ⟹ #f
> (miembro? 2 '(1 (2 3) 4)) ⟹ #f
> (miembro? '(2 3) '(1 (2 3) 4)) ⟹ #t
> (miembro? 'gato '()) ⟹ #f
```

Otro procedimiento útil es **remueve-primero**, el cual, como su nombre lo indica remueve la primera ocurrencia de alto nivel de un elemento en una lista dada. Su definición es la siguiente:

```
(define remueve-primero
   (lambda (elem ls)
(cond
   ((null? ls) '())
   ((equal? (car ls) elem) (cdr ls))
(else (cons (car ls) (remueve-primero elem (cdr ls)))))))
```

Algunos ejemplos de su uso son los siguientes:

```
> (remueve-primero 'zorra '()) ⟹ ()
> (remueve-primero 'zorra '(gallina zorra pollo gallo))
⟹ (gallina pollo gallo)
> (remueve-primero 'zorra '(gallina zorra pollo zorra gallo))
⟹ (gallina pollo zorra gallo)
> (remueve-primero 'zorra '(gallina (zorra pollo) gallo))
⟹ (gallina (zorra pollo) gallo)
> (remueve-primero '(1 2) '(1 2 (1 2) ((1 2))))
⟹ (1 2 ((1 2)))
```

2.4.1 Problemas Resueltos

1. Defina un procedimiento **subst-primero** que tome tres parámetros: un elemento **nuevo**, un elemento **viejo**, y una lista de elementos **ls**. El procedimiento **subst-primero** busca la primera ocurrencia de alto nivel del elemento **viejo** en **ls** y la reemplaza con el elemento **nuevo**. Pruebe su procedimiento con:

   > (subst-primero 'perro 'gato '(mi gato es inteligente))
   $\Longrightarrow$ **(mi perro es inteligente)**
   > (subst-primero 'b 'a '(c a b a c))
   $\Longrightarrow$ **(c b b a c)**
   > (subst-primero '(0) '(*) '((*) (1) (*) (2)))
   $\Longrightarrow$ **((0) (1) (*) (2))**
   > (subst-primero 'dos 'uno '()) $\Longrightarrow$ **'()**

   El procedimiento es el siguiente:

   (**define** subst-primero
   (**lambda** (nuevo viejo ls)
   (**cond**)
   ((**null?** ls) '())
   ((**equal?** (**car** ls) viejo) (**cons** nuevo (**cdr** ls)))
   (**else** (**cons** (**car** ls) (subst-primero nuevo viejo (**cdr** ls)))))))

2. Defina un procediemiento **inserta-izq-primero** que inserte un **nuevo** elemento a la izquierda de la primera ocurrencia de alto nivel del elemento **viejo** de la lista **ls** que se le indique. Pruebe su procedimiento con:

   > (inserta-izq-primero 'mucha 'pizza '(yo como pizza))
   $\Longrightarrow$ **(yo como mucha pizza)**
   > (inserta-izq-primero 'divertido 'juego '(muy divertido))
   $\Longrightarrow$ **(muy divertido)**
   > (inserta-izq-primero 'a 'b '(a b c a b c))
   $\Longrightarrow$ **(a a b c a b c)**
   > (inserta-izq-primero 'a 'b '()) $\Longrightarrow$ **()**

El procedimiento es el siguiente:

```
(define inserta-izq-primero
   (lambda (nuevo viejo ls)
     (cond
          ((null? ls) '())
          ((equal? (car ls) viejo) (cons nuevo (cons viejo (cdr ls))))
          (else (cons (car ls)
     (inserta-izq-primero nuevo viejo (cdr ls)))))))
```

3. Defina un procedimeinto **reemplaza** que reemplace cada elemento de alto nivel de una lista **ls** por un elemento dado **nuevo-elem**. Pruebe su procedimiento con:

> (reemplaza 'no '(me haces un favor))
> $\Longrightarrow$ **(no no no no)**
> (reemplaza 'si '(te gusta el helado))
> $\Longrightarrow$ **(si si si si)**
> (reemplaza 'porque '(no)) $\Longrightarrow$ **(porque)**
> (reemplaza 'quizas '()) $\Longrightarrow$ **()**

El procedimiento es el siguiente:

```
(define reemplaza
   (lambda (nuevo-elem ls)
(cond
   ((null? ls) '())
(else (cons nuevo-elem (reemplaza nuevo-elem (cdr ls)))))))
```

4. Defina un procedimiento **remueve-segundo** que elimine la segunda ocurrencia de un elemento dado **a** de una lista **ls**. Puede usar el procedimiento **remueve-primero** para definir **remueve-segundo**. Pruebe su procedimiento con:

> (remueve-segundo 'nueva '(mi nueva vecina adora la ropa nueva))
> $\Longrightarrow$ **(mi nueva vecina adora la ropa)**
> (remueve-segundo 'nueva '(mi nueva vecina adora la ropa))

$\Longrightarrow$ **(mi nueva vecina adora la ropa)**
> (remueve-segundo 'vecina '(mi vecina y tu vecina hablan sobre la nueva vecina)
$\Longrightarrow$ **(mi vecina y tu hablan sobre la nueva vecina)**
> (remueve-segundo 'gato '()) $\Longrightarrow$ **()**

El procedimiento es el siguiente:

```
(define remueve-segundo
(lambda (elem ls)
(cond
    ((null? ls) '())
    ((equal? (car ls) elem)
            (cons elem (remueve-primero elem (cdr ls))))
(else (cons (car ls) (remueve-segundo elem (cdr ls)))))))
```

5. Defina un procedimiento **sandwich-primero** que tome dos elementos, **a** y **b**, y una lista **ls** como sus argumentos. El procedimiento debe reemplazar la primera ocurrencia de dos **b**'s consecutivas en **ls** con **b a b**. Pruebe su procedimiento con:

> (sandwich-primero 'carne 'pan '(carne queso pan pan))
$\Longrightarrow$ **(pan queso pan carne pan)**
> (sandwich-primero 'carne 'pan '(pan jamon pan queso pan))
$\Longrightarrow$ **(pan jamon pan queso pan)**
> (sandwich-primero 'carne 'pan '()) $\Longrightarrow$ **()**

El procedimiento es el siguiente:

```
(define sandwich-primero
   (lambda (a b ls)
      (cond
           ((null? ls) '())
           ((null? (cdr ls)) (cons (car ls) '()))
           ((and (equal? (car ls) b) (equal? (cadr ls) b))
(cons b (cons a (cons b (cddr ls)))))
      (else (cons (car ls) (sandwich-primero a b (cdr ls)))))))
```

6. Defina un procedimiento **lista-de-simbolos?** que evalúe si los elementos de alto nivel de una lista **ls** son símbolos. Pruebe su procedimiento con:

   > (lista-de-simbolos? '(uno dos tres cuatro cinco)) $\Longrightarrow$ **#t**
   > (lista-de-simbolos? '(gato perro (gallina cerdo) vaca)) $\Longrightarrow$ **#f**
   > (lista-de-simbolos? '(a b 3 4 d)) $\Longrightarrow$ **#f**
   > (lista-de-simbolos? '()) $\Longrightarrow$ **#t**

   El procedimiento es el siguiente:

```
(define lista-de-simbolos?
(lambda (ls)
(cond
((null? ls) #t)
(else (and (symbol? (car ls))
        (lista-de-simbolos? (cdr ls)))))))
```

2.5 Ejercicios Propuestos

1. Defina un procedimiento **remueve-excepto-el-ultimo** que quite todas las ocurrencias del nivel superior de un elemento dado **elem** en una lista excepto por el último.

   Evalúe su procedimiento usando:

   (**remueve-excepto-el-ultimo** 'a '(b a n a n a s)) $\Longrightarrow$ (b n n a s)
   (**remueve-excepto-el-ultimo** 'a '(b a n a n a)) $\Longrightarrow$ (b n n a)
   (**remueve-excepto-el-ultimo** 'a '(c a r l o s)) $\Longrightarrow$ (c a r l o s)
   (**remueve-excepto-el-ultimo** 'a '(r o b e r t)) $\Longrightarrow$ (r o b e r t)
   (**remueve-excepto-el-ultimo** 'a '()) $\Longrightarrow$ ()

2. Defina un procedimiento **pluralizar** que tome una lista **ls** y cambie los elementos anteriores según las reglas siguientes:

(a) Si el último elemento del nivel superior es **f**, sustituirlo por **v** y agregar **e s** a la cola de la lista.

(b) Si los dos últimos elementos de nivel superior son **f e**, sustituir la **f** por una **v**, dejando la **e** sin alterar y agregando una **s** al final de la lista.

(c) Si las reglas 1 y 2 no se aplican, entonces no modifique la lista.

Evalúe su procedimiento usando:

(**pluralizar** '(s h e l f)) $\Longrightarrow$ (s h e l v e s)
(**pluralizar** '(w i f e)) $\Longrightarrow$ (w i v e s)
(**pluralizar** '(p e o p l e)) $\Longrightarrow$ (p e o p l e)
(**pluralizar** '(f e e l)) $\Longrightarrow$ (f e e l)
(**pluralizar** '(r e f e r e e)) $\Longrightarrow$ (r e f e r e e)
(**pluralizar** '(c o f f e e)) $\Longrightarrow$ (c o f f e e)
(**pluralizar** '(g i f t)) $\Longrightarrow$ (g i f t)
(**pluralizar** '(f e f e f e)) $\Longrightarrow$ (f e f e v e s)
(**pluralizar** '()) $\Longrightarrow$ ()

3. El siguiente procedimiento, llamado **misterio**, toma como argumentos una lista que contiene al menos dos elementos de alto nivel

```
(define misterio
(lambda (ls)
   (if (null? (cddr ls)) (cons (car ls) '())
    (cons (car ls) (misterio (cdr ls))))))
```

¿Cuál es el valor de (**misterio** '(1 2 3 4 5))? Describa el comportamiento general de **misterio**. Sugiera un nombre apropiado para este procedimiento.

4. Defina un procedimiento **lista-de-primeros** que tome como argumento una lista compuesta de listas no vacías de elementos. Este procedimiento debe devolver una lista compuesta de los primeros elementos

de alto nivel de cada una de las sublistas. Pruebe su procedimiento con:

(**lista-de-primeros** '((a) (b c d) (e f))) $\Longrightarrow$ (a b e)
(**lista-de-primeros** '((1 2 3) (4 5 6))) $\Longrightarrow$ (1 4)
(**lista-de-primeros** '((uno))) $\Longrightarrow$ (uno)
(**lista-de-primeros** '()) $\Longrightarrow$ ()

5. Defina un procedimiento **remueve-ultimo** que remueva la última ocurrencia de alto nivel de un elemento dado **elem** en una lista **ls**. Pruebe su procedimiento con:

   (**remueve-ultimo** 'a '(b a n a n a s)) $\Longrightarrow$ (b a n a n s)
   (**remueve-ultimo** 'a '(b a n a n a)) $\Longrightarrow$ (b a n a n)
   (**remueve-ultimo** 'a '()) $\Longrightarrow$ ()

6. Defina un procedimiento **todo-igual?** que tome como argumento una lista **ls** y pruebe si todos los elementos de alto nivel de **ls** son iguales. Pruebe su procedimiento con:

   (**todo-igual?** '(a a a a a)) $\Longrightarrow$ #t
   (**todo-igual?** '(a b a b a b)) $\Longrightarrow$ #f
   (**todo-igual?** '((a b) (a b) (a b))) $\Longrightarrow$ #t
   (**todo-igual?** '(a)) $\Longrightarrow$ #t
   (**todo-igual?** '()) $\Longrightarrow$ #t