Tema 7: Macros

Índice

1. Introducción

El primer párrafo del Revised5 Report on the Algorithmic Language Scheme (http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html) dice así:

Los lenguajes de programación no se deberían diseñar amontonando unas características sobre otras, sino eliminando las debilidades y restricciones que hacen necesaria la aparición de características adicionales.

Scheme tiene un conjunto mínimo de características (es un lenguaje minimalista) pero lo incrementa con un sistema muy potente de macros, que permite la creación de primitivas de más alto nivel.

En este tema se presenta una nueva característica de Scheme que lo distingue del resto de lenguajes de programación: se trata de la posibilidad de extender el propio lenguaje mediante la definición de macros. Una característica fundamental para implementar esta posibilidad es el uso de un lenguaje de patrones con el que procesar las expresiones de texto.

Las macros o extensiones sintácticas se utilizan para simplificar y regularizar patrones repetidos en un programa. Con ellas se introducen formas sintácticas con nuevas reglas de evaluación y se ejecutan transformaciones que ayudan a que los programas sean más eficientes.

El uso de patrones para el procesamiento de cadenas de texto en lenguajes de programación se remonta al año 1977 con la creación en los laboratorios Bell y AT&T (los mismos en los que se originaron el `C` y el `UNIX`) del lenguaje de programación **awk**. Posteriormente el uso de patrones se incorporó a distintos lenguajes de shell del UNIX y, sobre todo, a **Perl**, un importante lenguaje de script creado en 1987 por Larry Wall. Una buena introducción a Perl se encuentra en http://perldoc.perl.org/perlintro.html. En http://perldoc.perl.org/perlrequick.html se puede encontrar una buena introducción al uso de expresiones regulares.

Para definir una macro se han de especificar sus reglas de sintaxis (*syntax rules*). Cada regla consiste en un patrón que muestra una posible estructura para la expresión y una plantilla (*template*) que determina cómo los componentes de esa estructura se pueden reorganizar para formar una expresión aceptada por Scheme. Cuando se define una macro, el procesador de Scheme memoriza esas reglas de sintaxis. Y cuando encuentra una expresión que encaja con uno de los patrones, automáticamente reorganiza los componentes como dicta el correspondiente template, y evalúa el resultado.

2. Conceptos previos

Funciones con número variable de argumentos

Hasta ahora siempre hemos definido funciones con un número fijo de argumentos, ya sea utilizando la notación `(define (<nombre-fun> <args>) <cuerpo>)` como la notación `(lambda (<args>) <cuerpo>)`. Sin embargo, hemos visto que en Scheme podemos usar funciones con un número variable de argumentos. Por ejemplo, las funciones matemáticas `+`, `*` o `max`. ¿Cómo podemos definir funciones así? ¿Cómo se trabaja con un número variable de argumentos en el cuerpo de una función? Vamos a responder a estas preguntas.

En Scheme se pueden crear procedimiento que tomen un número variable de argumentos. Una manera de escribir procedimientos de aridad variable consistiría en usar la denominada notación de *listas impropias*, en la que el penúltimo elemento de la lista es un punto (".") y el último elemento denota todo el resto de la lista. Ésto le indica a Scheme que los argumentos del procedimiento se empaquetan en una lista cuando el procedimiento es invocado y el nombre de argumento apunta a la lista de los valores de los argumentos.

Por ejemplo, un procedimiento que toma un número cualquiera de argumentos y muestra la lista de los argumentos pasados al procedimiento.

```
(define (muestra-todo . args)
   (display args))
```

El argumento variable args recibe la lista de todos los argumentos y usamos display para mostrar esta lista. Si llamamos al procedimiento:

```
(muestra-todo 'foo 3 'bar) --> (foo 3 bar)
```

El argumento variable args se inicializa con la lista (foo 3 bar).

También funciona en expresiones lambda:

```
(define muestra-todo
       (lambda args
           (display args)))
```

En la versión lambda utilizamos un identificador, args y no (args), donde Scheme empaqueta todos los argumentos como una lista. Es un poco diferente de la versión del define, pero es la misma idea, se utiliza la variable args para almacenar la lista de los argumentos.

Se pueden escribir argumentos que reciben un número determinado de argumentos (requeridos), pero pueden tomar más. Cuando le pasamos a un procedimiento más argumentos de los que son requeridos, Scheme empaqueta los argumentos extra en una lista.

Para expresar esta idea en Scheme se usa la notación:

Notacion: arg1 . arg2

Los argumentos antes del "." (arg1) son obligatorios. El resto de argumentos se guarda en una lista y se pasa como arg2. Si no se pasan argumentos extra, la lista arg2 estará vacía.

Por ejemplo, si queremos que nuestro procedimiento muestra-todo acepte al menos un argumento:

```
(define (muestra-todo primero . resto) (display primero) (display
resto))
```

Más ejemplos:

```
(define (prueba a1 . a2)
   (print a1)
   (newline)
   (print a2))

(prueba 1 2 3 4)
(prueba 1 2)
(prueba 1)
(prueba)

(define (prueba a1 a2 . a3)
   (print a1)
   (newline)
   (print a2)
   (newline)
   (print a3))
```

Estamos definiendo la función `prueba` como una función que se debe llamar con dos parámetros obligatorios (los correspondientes a `arg1` y `arg2`) y un número de 0 o más parámetros después. Todo el resto de parámetros se guarda en una lista que se asocia con el parámetro formal `args`.

```
(prueba 1 2 3 4)
(prueba 1 2)
(prueba 1) --> # procedure prueba: expects at least 2 arguments, given
1: 1
```

En el siguiente ejemplo definimos la misma fución con un número variable de argumentos, sin argumentos obligatorios.

```
(define (prueba . a1)
```

```
   (print a1))

(prueba 1 2 3 4)
(prueba 1 2)
(prueba 1)
(prueba)
```

Con lambda también podríamos hacer lo siguiente:

```
(define prueba (lambda (a1 . a2)
                  (print a1)
                  (newline)
                  (print a2)))
```

Sin embargo, para definir una función sin argumentos obligatorios con `lambda` la notación hemos visto que cambia, ya que la siguiente definición da un error:

```
(lambda (. args)
   (print args))
```

El problema es que la lista impropia "(. args)" no es correcta, ya que cualquier lista impropia debe tener al menos un elemento propio. La definición correcta es:

```
(lambda args
   (print args))
```

Ejemplo real de uso:

```
(make-checked f predicado)
```

Función que construye una versión chequeada de otra función. La versión chequeada realiza un chequeo de los argumentos con predicado y sólo llama a f cuando todos los argumentos cumplen el predicado.

La función construida debe tener un número variable de argumentos, porque no sabemos cuántos argumentos tiene f.

Ejemplo de funcionamiento:

```
(define checked-suma (make-checked suma number?)
(checked-suma 1 2 3 4) ; -> 10
(checked-suma 1 2 'hola) ;-> #f
```

Vamos a construirla paso a paso

```
(define (mi-f f)
   (lambda args (apply f args)))
```

Ejemplo de llamada

```
((mi-f +) 1 2 3 4) ; -> 10
```

Función auxiliar chequear:

```
(define (chequear lista pred)
   (cond
      ((null? lista) #t)
      ((not (pred (car lista))) #f)
      (else (chequear (cdr lista) pred))))
```

Función hacer-chequeo:

```
(define (hacer-chequeo f pred)
   (lambda args
      (if (chequear args pred)
         (apply f args)
         #f)))
```

Sólo hay que hacer notar la función de Scheme `apply` que aplica una función a una lista de argumentos. Por ejemplo:

```
(apply + '(1 2 3 4)) --> 10
(apply append '((la frase del dia es) (confia en la recursion))) -->
(la frase del dia es confia en la recursion)>
```

Quasiquotation

La forma especial quasiquote se parece a quote en el sentido de que permite escribir expresiones que se devuelven sin evaluar, pero quasiquote es mucho más potente que quote, porque permite escribir expresiones que son la mayor parte literales, dejando "huecos" que se rellenarán con los valores computados en tiempo de ejecución.

Por ejemplo, el valor de la expresión `(quote (foo bar baz))` es la lista (foo bar baz).

Y el valor de la expresión `(quasiquote (foo bar baz))` es la lista (foo bar baz).

A simple vista parece lo mismo, pero hay una gran diferencia, y es que quote construye la lista en tiempo de compilación mientras que quasiquote la construye en tiempo de ejecución. Ésto permite a Scheme "personalizar" estructuras de datos, de forma que se puedan obtener diferentes estructuras de datos cada vez que se ejecute la misma forma quasiquote. Se puede usar además el operador unquote para especificar qué partes se quieren personalizar.

Por ejemplo, si queremos crear una lista de tres elementos cuyo primer y último elemento sean los literales foo y baz, pero el segundo elemento sea el valor de la variable bar:

```
(define bar 2)
(quasiquote (foo (unquote bar) baz))  --> (foo 2 baz)
```

Para hacer más fáciles este tipo de expresiones, Scheme proporciona azúcar sintáctico: backquote reemplaza a quasiquote y el carácter coma "," reemplaza a unquote:

```
`(foo ,bar baz) --> (foo 2 baz)
```

Más ejemplos:

```
(define a 2)
(define b 'hola)

'(1 a b)
(quasiquote (1 ,a ,b))
`(1 ,a ,b)
`(1 ,a ,b ,c)
`(1 ,+ ,-)
```

unquote-splicing

Scheme proporciona una variante de unquote para fusionar una lista "unquotada" en una lista de literales.

Por ejemplo

```
(define frase-del-dia '(debes confiar en la recursion))
`(recuerda que ,frase-del-dia) --> (recuerda que (debes confiar en la
recursion))
```

Si en lugar de usar unquote usamos unquote-splicing o su equivalente "azucar sintáctico" ,@ tendríamos:

```
`(recuerda que (unquote-splicing frase-del-dia)) --> (recuerda que
debes confiar en la recursion)
`(recuerda que ,@frase-del-dia) --> (recuerda que debes confiar en la
recursion)
```

3. Macros en Scheme

Aunque hablamos de macros, el nombre estándar que reciben las macros en Scheme son

extensiones sintácticas. Una extensión sintáctica permite añadir una nueva forma especial al lenguaje de programación. Se definen mediante las construcciones `define-syntax` y `syntax-rules`.

Las construcciones `define-syntax` y `syntax-rules` se usan con la siguiente sintaxis:

```
(define-syntax <keyword>
   (syntax-rules (<literales>)
      ((<patron-1> <plantilla-1>)
       ...
       (<patron-n> <plantilla-n>))))
```

Con `define-syntax` declaramos el identificador asociado a la macro y con `syntax-rules` las reglas de expansión de la macro. Estas reglas se definen mediante un conjunto de patrones de texto y de plantillas asociadas. Los literales que se definen después de `syntax-rules` son identificadores que deben aparecer tal cual en la llamada a la macro y que se usan en los patrones. Los identificadores que aparecen en los patrones y que no se declaran como literales son variables libres que emparejan con las expresiones correspondientes en la llamada a la macro. Las plantillas definen la expansión de la macro, una vez que la llamada a la misma ha emparejado con algún patrón. En las plantillas se usan literales y variables libres. Las variables libres toman el valor resultante del emparejamiento de la llamada a la macro con el patrón.

Los patrones y las plantillas definen unas reglas de transformación de expresiones de texto, al estilo de lenguajes como Perl o awk. En la expansión de la macro se aplican estas reglas y se transforma la llamada a la macro en una expresión resultante de su aplicación. Una vez realizada la transformación, Scheme evalua la expresión resultante.

Veamos un ejemplo en el que todo esto quedará más claro.

En este primer ejemplo definimos la macro `mi-or` que implementa una o lógica de todas las expresiones que le siguen. Al igual que la `or` de Scheme, queremos que devuelva `#t` en cuanto encuentre una expresión cierta, sin seguir evaluando el resto de expresiones. Esta característica hace imposible implementarla como una función.

```
(define-syntax mi-or
  (syntax-rules ()
    ((mi-or) #t)
    ((mi-or e) e)
    ((mi-or e1 e2 e3 ...)
     (if e1 #t (mi-or e2 e3 ...)))))
```

En esta macro no definimos literales. Los patrones que definimos son:

- Patrón 1: `(mi-or)`
- Patrón 2: `(mi-or e)`

- Patrón 3: `(mi-or e1 e2 e3 ...)`

Y las plantillas asociadas a esos patrones son:

- Plantilla 1: `#t`
- Plantilla 2: `e`
- Plantilla 3: `(if e1 #t (mi-or e2 e3 ...))`

Los patrones definen posibles formas de llamar a la macro `mi-or`. En concreto, estamos definiendo tres. En el primer patrón, se define una llamada a `mi-or` sin argumentos, en el segundo una llamada con un argumento (la expresión `e`) y en el tercero una llamada con dos o más argumentos (las expresiones `e1 e2 e3 ...`). Las variables `e`, `e1`, `e2`, `e3` son variables libres que emparejarán con alguna expresión de la llamada a la macro. Por último, los puntos suspensivos (`...`) indican una repetición de 0 o más veces del patrón anterior (en este caso, `e3`).

Las plantillas definen la expansión de la macro. La primera plantilla indica que hay que sustituir la llamada a la macro por `#t`. La segunda plantilla indica que hay que escribir la expresión `e` que se usa como argumento de la llamada a `mi-or`. Por último, la última plantilla es la que se usa cuando se hace una llamada a `mi-or` con dos o más argumentos e indica que hay que sustituir esta llamada por `(if e1 #t (mi-or e2 e3 ...))`, siendo `e1`, `e2`, `e3` los dos primeros argumentos y el resto de la llamada a la macro.

Por ejemplo, si llamamos a la macro de la siguiente forma:

```
(mi-or (equal? x 2) #f #t (equal? y 3))
```

Esta expresión emparejará con el patrón 3, definiendo los siguientes emparejamientos:

```
Expresión: (mi-or (equal? x 2) #f #t (equal? y 3))
Patron:    (mi-or e1 e2 e3 ...)
Emparejamientos:   e1 <-> (equal? x 2)
                   e2 <-> #f
                   e3 ... <-> #t (equal? y 3)
```

Al expandir la macro con la plantilla `(if e1 #t (mi-or e2 e3 ...))` queda la siguiente expresión:

```
(if (equal? x 2) #t (mi-or #f #t (equal? y 3)))
```

Para terminar este primer ejemplo, es interesante comentar que en muchos textos de Scheme la definición de la macro anterior aparecería como sigue:

```
(define-syntax mi-or
  (syntax-rules ()
```

```
    ((_) #t)
    ((_ e) e)
    ((_ e1 e2 e3 ...)
     (if e1 #t (mi-or e2 e3 ...)))))
```

El símbolo _ es una variable normal que se empareja con el símbolo que hay al comienzo de la expresión, que siempre será el símbolo `mi-or`. De hecho, el código anterior sería también equivalente al siguiente, en donde `op` hace el mismo papel de _.

```
(define-syntax mi-or
  (syntax-rules ()
    ((op) #t)
    ((op e) e)
    ((op e1 e2 e3 ...)
     (if e1 #t (mi-or e2 e3 ...)))))
```

Veamos ahora unas reglas que sirven de resumen de la semántica de la evaluación de una macro. Para evaluar una llamada a una macro (*op exp_1* `...` *exp_n*) debemos seguir las siguientes reglas:

- **1. Buscar la definición de la macro**. Buscar la forma especial `define-syntax` en la que aparece *op* como clave.
- **2. Emparejar**. Buscar en la definición de la macro la regla sintáctica con la que es posible emparejar la expresión (*op exp_1* `...` *exp_n*) que estamos evaluando. Si hay más de una regla con la que se puede emparejar la expresión, escogemos la primera de ellas.
- **3. Transformar**. Aplicar la regla para transformar la expresión. En el caso en que la expresión resultante contenga una llamada a una macro volver al paso 2 para expandir de nuevo la expresión. Terminaremos de expandir cuando la expresión resultante no contenga llamadas a macros.
- **4. Evaluar**. Evaluar la expresión resultante.

4. Más ejemplos de macros

- La macro `make-procedure` es una forma de hacer más legible la forma especial `lambda`. Tiene la misma sintaxis que `lambda` y se transforma en una llamada a esa forma especial.

```
(define-syntax make-procedure
    (syntax-rules ()
      ((make-procedure (x ...) expr ...)
       (lambda (x ...) expr ...))))
```

  Por ejemplo

```
(make-procedure (x) (* x x))
```

se transforma en

```
(lambda (x) (* x x))
```

- La macro `mi-let` explica cómo se implementa el `let` en Scheme, transformándose en una llamada a `lambda` para construir una función que tiene a las variables del `let` como argumentos y una posterior llamada a esta función con los valores como parámetros.

```
(define-syntax mi-let
  (syntax-rules ()
    ((mi-let ((x v) ...) e ...)
     ((lambda (x ...) e ...) v ...))))
```

Por ejemplo:

```
(mi-let ((x 1)
         (y (+ 2 1))
         (z (lambda (x y) (+ x y))))
  (z x y))
```

se transforma en

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

- En el siguiente ejemplo se define la macro `mi-cond` que se comporta igual que la forma especial `cond` de Scheme, aunque variando ligeramente la sintaxis. En la macro definimos los identificadores `=>` y `else` como literales.

```
(define-syntax mi-cond
   (syntax-rules (=> else)
     ((mi-cond (else => expr))
      expr)
     ((mi-cond (test1 => expr1))
      (if test1 expr1))
     ((mi-cond (test1 => expr1) (test2 => expr2) ...)
      (if test1 expr1 (mi-cond (test2 => expr2) ...)))))
```

Algunos ejemplos de llamada a esta macro son los siguientes:

```
(mi-cond (#t => 1))
(mi-cond (else => 1))
(mi-cond ((equal? a 1) => 1)
          ((equal? a 2) => 2)
          (else => 3))
```

Los siguientes ejemplos contienen la forma especial `begin` que hace que se evaluen de forma secuencial un conjunto de expresiones de Scheme. Veremos más veces esta forma especial en el tema 8.

- ```
  (define-syntax multi-print
    (syntax-rules ()
      ((multi-print arg1 arg2 ...)
       (begin (print arg1)
              (newline)
              (multi-print arg2 ...)))
      ((multi-print) #t)))
  ```

- ```
  (define-syntax when
      (syntax-rules ()
        ((when condition expr1 expr2 ...)
         (if condition (begin expr1 expr2 ...) #f))))
  ```

Para depurar las macros, muchas veces es conveniente comprobar el resultado de su expansión, sin dejar que se evaluen. Una forma de hacerlo es añadir un `quote` al comienzo de las plantillas. El `quote` hará que la expresión resultante no se evalue, sino que se devuelva como una lista o un símbolo.

Por ejemplo, si hacemos esto con la primera macro que hemos visto en este tema tendremos la siguiente macro.

```
(define-syntax mi-or
  (syntax-rules ()
    ((mi-or) '#t)
    ((mi-or e) 'e)
    ((mi-or e1 e2 e3 ...)
     '(if e1 #t (mi-or e2 e3 ...)))))
```

Las llamadas a esta macro siempre devolverán expresiones sin evaluar resultantes de la expansión de la macro:

```
>(mi-or (equal? a 1))
(equal? a 1)
>(mi-or (equal? a 1)
         (equal? a 2)
         #t)
```

```
(if (equal? a 1) #t (mi-or (equal? a 2) #t))
```

5. Referencias

- Programming languages: Application and Interpretation, Shriram Krishnamurthi. Brown University (http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/PDF/plai-2006-01-15.pdf) . Capítulo 37 (macros and their impact on language design).
- An introduction to Scheme and its implementation" (http://www.federated.com/~jim/schintro-v14/schintro_toc.html) . Secciones "Quasiquotation and Macros" y "Variable arity".
- An introduction to Scheme and its implementation" (http://www.federated.com/~jim/schintro-v14/schintro_129.html) . Sección "Quote and quasiquote".
- The Scheme Programming Language, 3rd ed. R. Ken Dybvig. Prentice Hall (http://www.scheme.com/tspl3/) . Apartados 3 y 8.
- Chez Scheme User's Guide. R. Ken Dybig (http://www.scheme.com/csug7/) . Apartado 9.
- Plt Scheme (macros) (http://download.plt-scheme.org/doc/301/html/r5rs/r5rs-Z-H-7.html#%25_sec_4.3)