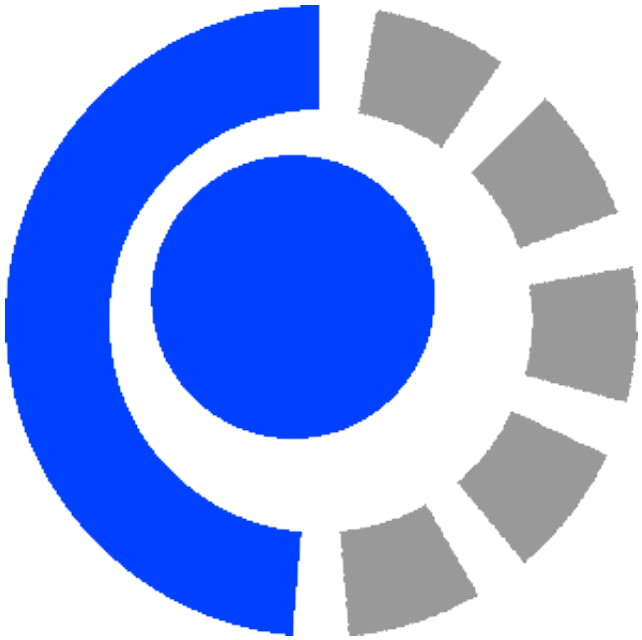
Lenguajes Declarativos: expansiones sintácticas

Ignacio Alejandro Navarro Oliva

Mayo de 2024



Scheme suele ser conocido como "el lenguaje de programación orientado a lenguajes", debido a su capacidad para establecer nuevas reglas sintácticas para crear nuevas funciones del lenguaje. Uno podría preguntarse: ¿acaso no es lo mismo que los procedimientos programados en Java por ejemplo? En el caso de Scheme, los llamados *macros* transforman una pieza de código en otra pieza de código completamente nueva, lo antes posible (en tiempo de compilación). La creación de macros posibilita emplear argumentos sin evaluar (argumentos "crudos"), de manera tal que el macro elija cuándo evaluarlos o que directamente no evalúe nada y traduzca el código según las reglas sintácticas agregadas. De este modo se pueden crear nuevos lenguajes a partir de un núcleo primitivo del lenguaje.

La forma más simple de escribir macros tiene la siguiente sintaxis:

```
(define-macro nombre transformador)
```

donde *nombre* es el identificador que recibe la expansión sintáctica, y el *transformador* es un procedimiento de un único argumento el cual toma el macrocódigo y retorna su respectiva macroexpansión.

Luego el programa reemplaza el código que desconoce, empleando las reglas sintácticas establecidas por los macros, por el macrocódigo que retorna el transformador.

También existe la notación **define-syntax-rule** y **define-syntax**, las cuales son más utilizadas. Tienen la siguiente sintáxis:

```
(define-syntax-rule pattern template)
```

En este caso *pattern* (o patrón) consiste en el nombre del macro, seguido por sus argumentos. Por ejemplo, un patrón podría ser (swap a b), donde el primer elemento swap es el nombre del macro y a y b son los argumentos o macro-pattern-variables. Luego el template es lo mismo que el transformador en el caso de **define-macro**, donde los

argumentos del patrón se emparejan con las variables que tenga template. La notación **define-syntax** admite más de un patrón:

```
(define-syntax id
    (syntax-rules (literal-id ...)
    [pattern template]
    ...))
```

Supongamos que disponemos de una versión antigua de Scheme que no posee las repetitivas while ni repeat, de modo tal que debemos implementar procedimientos o ir copiando piezas de código que repliquen su funcionamiento. El programador desea mostrar en pantalla números del 1 al 5 de forma descendiente utilizando una repetitiva while, y logra implementarlo. El problema es que deberá copiar esa pieza de código que replica el funcionamiento de while cada vez que quiera utlizar la repetitiva y tampoco podrá implementarlo como un procedimiento ya que se evaluarían los argumentos, de modo que como condición se enviaría una constante con valor falso o verdadero en lugar de una condición cambiante. Lo más conveniente es definir un macro while que pueda solucionar estos problemas y ampliar el lenguaje para nuestra conveniencia.

Se puede implementar el macro de la siguiente manera:

```
(define-syntax-rule (while condition body ...)
    (let loop ()
        (when condition
            body ...
            (loop)))))
```

Esta implementación admite muchas líneas de código para el cuerpo a través de la notación de elipsis (...). Utilizamos la sintaxis **define-syntax-rule** ya que la repetitiva solo necesita un patrón para fun-

cionar. Entonces, para utilizar este macro según intención original del programador, el código luciría de la siguiente manera:

```
(define x 5)
(while (> x 0) (displayln x) (set! x (- x
  1))))
```

Luego la regla sintáctica reemplaza los argumentos condición y cuerpo sin evaluar en su template, y produce la salida correspondiente.

Se puede emplear algo similar para crear un macro con el funcionamiento repeat. La diferencia entre repeat y while es que repeat primero ejecuta las líneas del cuerpo hasta que la condición evalúe como falsa.

```
(define-syntax-rule (repeat body ...
  condition)
    (let loop ()
        body ...
        (when (not condition)
            (loop)))))
```

Para lograr la misma salida que con el while deberíamos cambiar la manera en la que llamamos al repeat:

```
(define x 6)
(repeat (set! x (- x 1)) (displayln x) (<= x
  1))
```