Laboratorio de Lenguajes de Programación

Recursión - Parte I

Manuel Soto Romero

Universidad Nacional Autónoma de México
Facultad de Ciencias

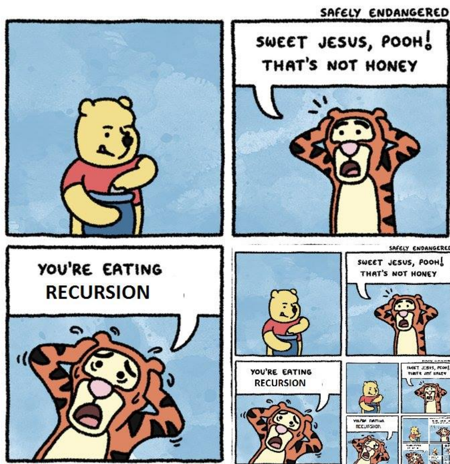
19 de octubre de 2016

Objetivos

- Recordar el concepto de recursión.
- Recordar en qué consiste recursión de cola.
- Recordar qué es una continuación y en qué consiste la técnica de *Continuation Passing Style*.

¿Qué es recursión?

Recursión es la forma en la cual se especifica un proceso basado en su propia definición.



El típico ejemplo

El factorial de un número está definido recursivamente pues el proceso para calcularlo está basado en su propia definición:

```
(define (factorial n)
  (if (zero? n))
    1
    (* n (factorial (sub1 n)))))

> (factorial 5)
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
120
```

¿Cómo se ve esto es memoria?

Pila de ejecución del programa

Cuando un programa llama a una función, la función llamada debe saber cómo regresar a la que lo llamó, por lo que la dirección de retorno que hizo la llamada se mete a la **pila de ejecución del programa**. Cuando acaba su trabajo, se saca de la pila y se continúa con la función que lo mandó llamar. Cada dato almacenado en la pila se conoce como **registro de activación**.

El registro de activación se compone de:

Resultado de la llamada
Cuerpo de la función
Parámetros con su valor
Nombre de la función

Si ocurren más llamadas a funciones de las que puedan almacenar sus registros de activación se produce un error conocido como *stack overflow* (desbordamiento de pila) pues no tenemos memoria infinita.

Ejemplo

```
> (factorial 5)
```

El primer registro de activación se genera como sigue:

(* 5 (factorial 4))
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 5
factorial

Este registro de activación deja una llamada pendiente por lo que obtenemos un nuevo registro de activación:

(* 4 (factorial 3))
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 4
factorial

Ejemplo

Nuevamente dejamos llamada pendiente:

(* 3 (factorial 2))
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 3
factorial

(* 2 (factorial 1))
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 2
factorial

(* 1 (factorial 0))
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 1
factorial

Ejemplo

1
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 0
factorial

A partir de aquí comienza una serie de retornos y se va reduciendo el tamaño de la pila...

(* 1 1)
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 1
factorial
(* 2 1)
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 2
factorial

Ejemplo

(* 3 2)
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 3
factorial

(* 4 6)
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 4
factorial

(* 5 24)
(if (zero? n) 1 (* n (factorial (sub1 n))))
n = 5
factorial

→ *120*

¿Cómo se vería la pila de ejecución del programa con (factorial 100)?

Recursión de cola

Se dice que una función utiliza **recursión de cola** cuando la llamada recursiva es la última instrucción de la función, con la peculiaridad de que dentro de dicha instrucción no debe existir ninguna otra sentencia más que la propia llamada.

Al utilizar recursión de cola se evitan desbordamientos en la pila de ejecución del programa.

Una función que utiliza recursión simple se puede convertir a otra con recursión de cola, agregando parámetros adicionales llamados **acumuladores**, los cuales serán utilizados para ir guardando un resultado parcial y de esta manera la llamada recursiva ya no tendrá operaciones pendientes.

Recursión simple a recursión de cola

Un método que sirve para la mayoría de los casos (en especial cuando sólo se tiene una llamada pendiente) es el siguiente:

1. **Reescribir la función.** Se reescribirá agregando parámetros para el acumulador, en un principio, tenemos que agregar tantos acumuladores como llamadas pendientes haya en la función original.

```
(define (factorial-tail n acc)
  ...)
```

2. **Reescribir el caso base.** Dado que el acumulador va guardando el resultado parcialmente, cuando caigamos en el caso base, tenemos que devolver el resultado del acumulador.

```
(define (factorial-tail n acc)
  (if (zero? n)
    acc
    ...))
```

3. **Reescribir la llamada recursiva.** Se reescribirá quitando la operación que ocasiona llamadas pendientes y realizándola con ayuda del acumulador.

```
(define (factorial-tail n acc)
   (if (zero? n)
      acc
      (factorial-tail (sub1 n) (* n acc))))
```

4. **Reescribir la función principal.** El usuario no tiene que interactuar con esta nueva función pues no tiene porque manipular el acumulador. Debe haber una función principal que llame a la nueva función, asignándole un valor inicial al acumulador. El valor inicial debe ser, en un principio, el valor que regresaba en el caso base la función original.

```
(define (factorial n)
   (factorial-tail n 1))
```

Recursión simple a recursión de cola

```
> (factorial 5)
> (factorial-tail 5 1)
> (factorial-tail 4 5)
> (factorial-tail 3 20)
> (factorial-tail 2 60)
> (factorial-tail 1 120)
> (factorial-tail 0 120)
120
```

¿Cuántos registros de activación ocupa factorial-tail?

Continuation passing style

La idea básica detrás de una continuación es la de considerar a la pila de ejecución de un programa como un valor el cual puede devolverse o pasarse como argumento a otra función. Esto se conoce como **materialización** de la pila y permite entre otras cosas reemplazar la pila de control actual por otra.

Con continuaciones podemos ya sea seguir con una evaluación en un estado que ya se había utilizado o bien regresar a un cómputo anterior arbitrario para poder «revivirlo» en cualquier momento.

El principal uso de las continuaciones va desde generar simples optimizaciones hasta elegantes implementaciones de compiladores entre intérpretes mediante la técnica de programación por paso de continuaciones (*Continuation Passing Style*, CPS).

En pocas palabras... una continuación es una función que representa al resto de un programa. Por ejemplo:

```
(define (foo (x y))
  (+ (* 2 x) (* 3 y) (/ 1 x) (/ 2 y))))
```

- Si detenemos la evaluación de esta función justo antes de la primera división habremos calculado `(+ (* 2 x) (* 3 y))`
- Cuando la división `(/ 1 x)` se haya completado, el proceso de evaluación continuará con la suma y otra división.
- De esta manera decimos que la continuación de `(/ 1 x)` es el cómputo después de `(/ 1 x)`, es decir, el cómputo pendiente.

```
(define (foo (x y))
  (let* ([antes (+ (* 2 x) (* 3 y))]
         [cont (lambda (div) (+ antes div (/ 2 y)))])
    (cont (/ 1 x))))
```

Recursión a CPS

Veamos ahora un método para convertir una función con recursión simple a su respectiva versión en CPS:

1. **Pasar la función a recursión de cola.** Usando el método pasado, obtenemos la función en recursión de cola.

2. **Reescribir el caso base.** Dado que ahora iremos guardando el estado en una función, no tenemos acumulador sino una función, de manera que vamos a evaluar la función con el caso base de la función original (con recursión simple).

```
(define (factorial-cps n k)
  (if (zero? n)
     (k 1)
     ...))
```

Recursión simple a CPS

3. **Reescribir la llamada recursiva.** En lugar de usar un acumulador, definimos la continuación:

```
(define (factorial-cps n k)
  (if (zero? n)
    (k 1)
    (factorial (sub1 n) (lambda (v) (k (* n v))))))
```

4. **Reescribir la función principal.** El usuario no tiene que interactuar con esta nueva función pues no tiene porque manipular las continuaciones. Llamamos a la función con la identidad.

```
(define (factorial n)
  (factorial n (lambda (x) x)))
```

```
> (factorial 5)
> (factorial-cps 5 (lambda (x) x))
> (factorial-cps 4 (lambda (v1) (k (* 5 v1))))
> (factorial-cps 3 (lambda (v2) (v1 (* 4 v2))))
> (factorial-cps 2 (lambda (v3) (v2 (* 3 v3))))
> (factorial-cps 1 (lambda (v4) (v3 (* 2 v4))))
> (factorial-cps 0 (lambda (v5) (v4 (* 1 v5))))
(v5 1)
(v4 (* 1 1))
(v3 (* 2 1))
(v2 (* 3 2))
(v1 (* 4 6))
(k (* 5 24))
120
```