Lenguajes de Programación I

Iteradores y Recursión

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Iteradores

- Mecanismo para crear un lazo que procese en orden cada uno de los elementos de un conjunto bien definido.
- Generalización de los lazos controlados por Enumeración.
  - Número de iteraciones implícitamente asociado a la cardinalidad de la colección que se procesa.
  - Requiere la noción de orden total sobre la colección.
- Iteradores verdaderos – responsabilidad del lenguaje.
- Iteradores simulados – responsabilidad del programador, con "azúcar sintáctica" o API predeterminado.

Iteradores Verdaderos

- Usuales en lenguajes que permiten que *cualquier* clase contenedora provea un método para enumerarla.
- Lucen como un método cualquiera
  - Usan `yield` en lugar de `return` – permite retornar el siguiente valor.
  - Siguiente llamada al método no lo ejecuta desde el principio – *continúa* justo después del último `yield`.
  - Registro de activación se conserva después de un `yield` y se restaura en la siguiente invocación.
  - Si ejecuta un `return` explícito o alcanza el final de su ejecución por agotamiento de valores, el registro de activación se descarta.
- Registros de activación promovidos de pila a *heap*, y de regreso.
- Iteradores verdaderos rara vez en lenguajes compilados.

Iterando sobre los números de Fibonacci

Ruby tiene iteradores verdaderos

```
def fibHasta(max)
  i1, i2 = 1, 1
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end

fibHasta(1000) { |f| print f," " }
```

- Reciben un *bloque* como parámetro – son funciones de primera clase.
- Se invoca *una* vez – cada `yield` pasa control al bloque parámetro.

Iteradores con recursión

. . . como en Python

- Sea una clase `Arbol` – recorrerlo *inorder*
  - Arboles vacíos – actúan como `false`.
  - Referencia a subárboles izquierdo (`izq`) y derecho (`der`).
  - Almacenan un valor (`val`).

```
def inorder(arbol)
  if arbol :
            for s in inorder(arbol.izq)
              yield s
            yield arbol.val
            for s in inorder(arbol.der)
              yield s
```

- Cada llamada a `inorder` crea un registro de activación nuevo.
- Se genera un árbol de registros de activación suspendidos.

Iteradores Simulados

Cuando son responsabilidad del programador

- C++ moderno provee una interfaz automática `iterator`

```
vector<int> the_vector;
vector<int>::iterator the_iterator;

int total = 0;
the_iterator = the_vector.begin();
while( the_iterator != the_vector.end() ) {
  total += *the_iterator;
  ++the_iterator;
}
```

- Existen para la mayoría de los contenedores en C++ STL.
- No verifican límites de iteración.
- Cambios al contenedor causan problemas al iterador.

Iteradores Simulados

Forzando una interfaz o rol

- En Java es necesario que la clase contenedora:
  - Implante la interfaz `Iterable` con el método `iterator`.
  - Retorne objetos de la clase `Iterator`.
  - ...y sobrellevar el karma del *boxing*.
- Muy parecido a lo de C++

```
ArrayList<String> alist = new ArrayList<String>();

for (Iterator<String> it = alist.iterator(); it.hasNext(); ) {
    String s = it.next();
    System.out.println(s);
}
```

- El programador debe implantar la interfaz – más trabajo que en C++.

Iteradores en lenguajes que no los proveen

Una aproximación suficiente pero ineficiente

- Estas simulaciones logran el mismo *efecto* pero
  - Código resultante no es conciso y resulta poco claro – más aún combinado con *métodos genéricos*.
  - Generalmente es menos eficiente que con iteradores nativos.
  - Pero no hay otra alternativa en lenguajes compilados, por la complicación de registros de activación dinámicos.
- Iteradores implícitos sobre algunos tipos de datos
  - Iterador `for` para listas en Perl o Python.
  - Iterador `each` para mapas (*hashes*) en Perl, Python y Ruby.

Modelo de Ruby el más elegante – modelo de Python más flexible.

Recursión

- Hasta que comprenda la definición de **Recursión**, lea la definición de **Recursión**.
- El lenguaje debe permitir que una función se llame a sí misma, o a otra función que la pueda llamar.
  - Alcance – los nombres de las funciones pueden usarse aún cuando no se haya terminado de definir su cuerpo.
  - Ejecución – indispensable almacenamiento en pila real o simulada.
- Todo algoritmo recursivo puede escribirse con iteración y viceversa.
- La recursión es natural en lenguajes funcionales – y la única forma de expresar repeticiones.
- La iteración es natural en lenguajes imperativos – pero también se puede trabajar con recursión.

Consideraciones de Eficiencia

- "La iteración es más eficiente que la recursión"
  - Mito iniciado por los programadores inocentes.
  - Propagado por compiladores igual de inocentes.
- La "demostración" suele basarse en Fibonacci – (haciéndose los locos con el factorial)

Fibonacci Recursivo vs. Iterativo

Los competidores

- Implantación recursiva tradicional

```
unsigned long fibo (unsigned long n) {
  if ((n == 0) || (n == 1))
    return 1;
  else
    return (fibo(n-1) + fibo(n-2));
}
```

- Implantación iterativa tradicional

```
unsigned long fibo (unsigned long n) {
  unsigned long i, t, f1 = 1, f2 = 1;
  for (i = 2; i <= n; i++) {
    t = f1 + f2; f1 = f2; f2 = t;
  }
  return f2;
}
```

. . . y comparamos los tiempos.

```
$ gcc -O2 -o fib-recursivo fib-recursivo.c
$ gcc -O2 -o fib-iterativo fib-iterativo.c

$ time ./fib-recursivo 42
Fibonacci(42) = 433494437

real  0m1.992s
$ time ./fib-iterativo 42
Fibonacci(42) = 433494437

real  0m0.002s
```

"¿Viste que la recursión es terrible?"

¿Qué es lo que está mal?

- Observemos la llamada recursiva

```
return (fibo(n-1) + fibo(n-2))
```

- La llamada *actual* tiene que hacer dos llamadas y **esperar** sus resultados antes de poder retornar su resultado.
- Para eso su registro de activación tiene que mantenerse...
- ...y como son dos llamadas recursivas por cada instancia, la pila crece *exponencialmente*
- Manipular la pila requiere espacio y tiempo – eso explica la lentitud.

:'-/

Vuelve el Máximo Común Divisor

Hagamos la misma comparación

- Implantación recursiva tradicional

```
unsigned long mcd(unsigned long a, unsigned long b) {
  if (a==b) return a;
  else if (a > b) return mcd(a-b,b);
  else return mcd(a,b-a);
}
```

- Implantación iterativa tradicional

```
unsigned long mcd(unsigned long a, unsigned long b) {
  while (1) {
    if (a==b) return a;
    else if (a > b) a = a - b;
    else b = b - a;
  }
}
```

...y lo probamos con un número primo grande

Usamos 4294967291, el primo más grande de 32 bits

```
$ gcc -O2 -o mcd-recursivo mcd-recursivo.c
$ gcc -O2 -o mcd-iterativo mcd-iterativo.c

$ time ./mcd-recursivo 4294967291 2
mcd(4294967291,2) = 1

real  0m1.211s
$ time ./mcd-iterativo 4294967291 2
mcd(4294967291,2) = 1

real  0m1.215s
```

¿Dónde está tu iteración ahora? ¿Dónde?

What sorcery is this?

- Observemos las dos llamadas recursivas

```
return mcd(a-b,b)
```

y

```
return mcd(a,b-a)
```

- La llamada *actual* no necesita **esperar** nada especial – ¡su resultado sería exactamente el mismo de la llamada!
- No hace falta un registro de activación para la llamada recursiva – ¡puedo reutilizar el actual por sustitución de variables!
  - La llamada recursiva `return mcd(a-b,b)` es lo mismo que hacer `a = a - b` y repetir la ejecución.
  - La llamada recursiva `return mcd(a,b-a)` es lo mismo que hacer `b = b - a` y repetir la ejecución.

Wait, what?

Esto es así...

La versión recursiva original

```
unsigned long mcd(unsigned long a, unsigned long b) {

  if (a==b)
    return a;
  else
    if (a > b) return mcd(a-b,b);
  else
    return mcd(a,b-a);
}
```

- En la llamada `mcd(a-b,b)`, el "nuevo" `a` sería `a-b` y `b` queda igual.
- En la llamada `mcd(b,b-a)`, `a` queda igual y el "nuevo" `b` sería `b-a`.
- ...y habría que repetir todo desde el principio.

Esto es así...

La sustitución de variables

```
unsigned long mcd(unsigned long a, unsigned long b) {
principio:
  if (a==b)
    return a;
  else
    if (a > b) { a = a - b; goto principio; }
  else
    return { b = b - a; goto principio; }
}
```

- Así que hago la sustitución de expresiones y `goto`.
- C permite etiquetas, por si no lo sabían...
- ...pero este `goto` es fácil de eliminar.

Esto es así...

We meet again, at last!

```
unsigned long mcd(unsigned long a, unsigned long b) {
  while (1) {
    if (a==b)
      return a;
    else
      if (a > b) a = a - b;
    else
      return b = b - a;
}
```

Y llegamos a la versión *iterativa*...
OMG! OMG! OMG!

Recursión inteligente

...y trivialmente transformable

- Una función presenta **Recursión de Cola** cuando no hay ningún cómputo después de la llamada recursiva.
- **Recursión de Cola** (*tail recursion*) puede ser muy eficiente:
  - Cuando un compilador es capaz de optimizarla – GCC hizo *exactamente* la misma transformación.
  - Cuando un programador es capaz de inducirla – para que el compilador la optimice.

Cuando sólo hay recursión

- En los lenguajes funcionales sólo hay recursión – con compiladores que optimizan la recursión de cola.
- Y a uno le mandan a implantar Fibonacci en Haskell

```
fib n | (n == 0) || (n == 1) = 1
      | n > 2 = fib (n-2) + fib (n-1)
```

- Pero esto no tiene recursión de cola, así que será lento – y el programador de C se va a reír.

Si la función no tiene recursión de cola. . .
¡oblígala a tenerla!

No es la flecha, sino el indio

- Fibonacci, con recursión de cola

```
fib n = hack 0 1 0
        where
          hack f1 f2 i = if (i == n)
                         then f2
                         else (hack f2 (f1+f2) (i+1))
```

- La "astucia" está en `hack`
  - Primer argumento es el F_{i-1} – inicia en 0 porque $F_{-1} = 0$
  - Segundo argumento es el F_i – inicia en 1 porque $F_0 = 1$
  - Tercer argumento es *acumulador* que inicia en 0 y llega hasta `n`
  - ¡`hack` tiene recursión de cola!

¡`fib` tiene recursión de cola!

Inducir recursión de cola

- Cuando una función no tiene recursión de cola, generalmente es posible inducir recursión de cola total o parcialmente.
  - Fibonacci y Factorial permiten inducción total.
  - La inducción parcial al menos reduce la complejidad de ejecución.
- Se usan variaciones de la técnica demostrada
  - A veces se usa uno o más acumuladores – lo mostrado.
  - A veces se usa *tupling* – Programación Funcional Avanzada.
  - A veces se usan varias funciones auxiliares co-recursivas.
  - **Siempre** es un trabajo creativo.
- Aún si el lenguaje no es funcional, inducir recursión de cola asiste a los compiladores para convertirlos en ciclos cerrados.

Orden de Evaluación

- **Orden Aplicativo o Ambicioso (*Eager*)**
  - Evaluar parámetros completamente **antes** de ser pasados a una función.
  - Sólo permite crear funciones *estrictas* – fallan si alguno de sus parámetros no es evaluable.
  - Típica de lenguajes imperativos y lenguajes funcionales clásicos.
- **Orden Normal o Perezoso (*Lazy*)**
  - Parámetros se evalúan solamente cuando son *necesarios*.
  - Permite crear funciones parciales y estructuras infinitas.
  - Típica en lenguajes funcionales modernos (Haskell).
  - Dificulta el cálculo de eficiencia de programas.
  - Parcialmente simulables cuando hay funciones de primera clase.

No Determinismo

- El lenguaje permite especificar puntos (*choice points*) en el programa en los cuales la decisión de flujo queda en manos del **lenguaje**.
  - La decisión se toma a tiempo de ejecución.
  - Anidamiento de *choice points*.
- Técnica de **Backtracking** (Prolog)
  - Alternativa "falla", flujo retorna hasta la última decisión e intenta otra.
  - ¿Qué hacer con los efectos de borde luego de una falla?
- Técnica de **guardias** (*guards*)
  - Si más de una es cierta, se escoge *cualquiera* de ellas.
  - Orden de aparición, al azar, por frecuencia previa...
- Técnica de **aprendizaje reforzado** (A-LISP)
  - Recordar decisiones exitosas y aumentar su prioridad.
  - Considerar las decisión en el contexto del estado mutable.

El problema de Hamming

Faciliiiiito

- Escriba un programa que imprima los números de Hamming. Los números de Hamming se definen recursivamente:
  - El 1 es un número de Hamming.
  - Si un número es múltiplo de 2 ...
  - ... y es múltiplo de 3 ...
  - ... y es múltiplo de 5, también es número de Hamming.
  - O sea, números de la forma $2^i \times 3^j \times 5^k$ con $i, j, k \geq 0$
- La lista debe imprimirse en orden de menor a mayor...
- ... y sin repeticiones.
- En tiempo $O(n)$ para generar los primeros n elementos.
- Preferiblemente en espacio $O(1)$...

Recursión y evaluación perezosa combinadas

```
hamming =
  1 : merge (map (2*) hamming)
            (merge (map (3*) hamming)
                   (map (5*) hamming))
      where
        merge (a:x) (b:y) | a<b       = a : merge x (b:y)
                          | a>b       = b : merge (a:x) y
                          | otherwise = a : merge x y


  > take 20 hamming
  [1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36]
```

Bibliografía

- [*Scott*]
  - Secciones 6.6, 6.5.3 y 6.5.4
  - Ejercicios y Exploraciones
- Mire la librería `Stream.pm` que simula evaluación perezosa de secuencias de números en Perl aprovechando que las funciones son de primera clase. El programa `hamming.pl` lo uso para resolver el problema en Perl.