Metodología y tecnología de la programación

Ingeniería Informática (UCM)

Hoja de ejercicios 2 — Curso 2007/2008

Ejercicios de transformación de recursivo a iterativo

Ejercicio 1 Transformar a iterativo las siguientes versiones recursivas para calcular la raíz cuadrada entera de un entero n, obtenidas mediante inmersión de parámetros, teniendo en cuenta que las llamadas iniciales se realizan con $a = 1$ y $a = 0$ respectivamente:

```
fun raíz1(e n, a : ent) dev r : ent
    casos
        a^2 > n → r := 0
      [] a^2 ≤ n →
            r' := raíz1(n, 2 * a)
            casos
                n < (r' + a)^2 → r := r'
              [] n ≥ (r' + a)^2 → r := r' + a
            fcasos
    fcasos
ffun

fun raíz2(e n, a : ent) dev r : ent
    casos
        (a + 1)^2 > n → r := a
      [] (a + 1)^2 ≤ n → r := raíz2(n, a + 1)
    fcasos
ffun
```

Ejercicio 2 Explicar cómo transformar un algoritmo recursivo lineal no final a iterativo cuando la función que calcula el sucesor no tiene inversa, utilizando una pila. Aplicarlo al siguiente algoritmo:

```
fun potencia(a : num, n : nat) dev p : num
    si n = 0 entonces p := 1
        si no
            p := potencia(a, n div 2)
            p := p * p
            si impar?(n) entonces p := p * a fsi
    fsi
ffun
```

Ejercicio 3 Desarrollar un algoritmo iterativo para recorrer un árbol binario en *inorden*, siguiendo el método de *desplegado-plegado* y utilizando las generalizaciones definidas mediante las siguientes ecuaciones, donde xs es una lista, a es un árbol y p es una pila de árboles.

$$
\begin{array}{lcl}
\text{ggin}(xs, a, p) = xs +\!+ \text{inorden}(a) +\!+ \text{inpila}(p) & & \\
\text{inpila}(\text{pila-vacía}) & = & [\,] \\
\text{inpila}(\text{apilar}(\text{plantar}(iz, x, dr), p)) & = & [x] +\!+ \text{inorden}(dr) +\!+ \text{inpila}(p) \\
\text{inpila}(\text{apilar}(\text{árbol-vacío}, p)) & = & \text{error}
\end{array}
$$

Ejercicio 4

1. Desarrollar un algoritmo iterativo para recorrer un árbol binario en *preorden*, siguiendo el método del *primero-último-sucesor* y utilizando si es necesario las funciones padre, es-izq? y es-der?.
2. Introducir una pila en el algoritmo del apartado anterior para eliminar las llamadas a las funciones padre, es-izq? y es-der?.
3. Transformar a iterativo el algoritmo de ordenación rápida *quicksort* utilizando el recorrido del árbol de activaciones y el algoritmo del apartado anterior.

Ejercicio 5 Desarrollar un algoritmo iterativo para recorrer un árbol binario en *postorden*, siguiendo el método del *primero-último-sucesor* y utilizando una pila para calcular el padre. Aplicarlo al algoritmo de ordenación *mergesort*.

Ejercicio 6 Demostrar que el problema de las Torres de Hanoi con n discos no puede resolverse con menos de $2^n - 1$ movimientos (por lo tanto, el algoritmo recursivo usual es óptimo).

Ejercicio 7 Escribir un algoritmo recursivo para resolver el problema de las Torres de Hanoi de forma que *no* se permitan movimientos entre las torres *origen* y *destino*, es decir, todos los movimientos tienen que ser desde o hasta la torre *auxiliar*. (En este enunciado, las palabras *origen*, *auxiliar* y *destino* se refieren a las correspondientes torres en el estado inicial y no a los diferentes papeles que juegan las torres en cada llamada recursiva.) ¿Cuántos movimientos se realizan?

Ejercicio 8 Transformar a iterativo el algoritmo obtenido en el Ejercicio 7 siguiendo el método del *primero-último-sucesor* y utilizando una pila para calcular el padre.

Ejercicio 9 Transformar a iterativo el algoritmo obtenido en el Ejercicio 7 siguiendo el método del *primero-último-sucesor* y numerando los nodos del árbol de activaciones para calcular el padre.

Ejercicio 10 Transformar a iterativo el siguiente algoritmo recursivo, siguiendo el método del *primero-último-sucesor* y utilizando una pila para calcular el padre. No es suficiente con dar la versión iterativa de algoritmo, sino que hay que pasar por la versión iterativa del recorrido de árboles correspondiente.

```
proc algoritmo(V[1..n] de elemento, e c, f : nat)
    casos
        f − c + 1 ≤ 10 → básico(V, c, f) { modifica V }
      [] f − c + 1 > 10 → m := (c + f) div 2
            procesar(V, c, f) { modifica V }
            algoritmo(V, c, m)
            algoritmo(V, m + 1, f)
    fcasos
fproc
```

La llamada inicial es algoritmo$(V, 1, n)$, con $n > 10$, y se suponen disponibles los algoritmos iterativos básico y procesar.