Transformación de recursivo a iterativo

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Octubre 2008

Transformación de recursivo a iterativo

Vamos a ver cómo transformar algoritmos recursivos lineales a iterativo.

Primero vemos cómo transformar una función recursiva final.

$$
\begin{tikzcd}
\texttt{fun-rec}(\overline{x}) \arrow[r, "B_{nt}(\overline{x})"] & \texttt{fun-rec}(\texttt{s}(\overline{x})) \arrow[r, "B_{nt}(\texttt{s}(\overline{x}))"] & \cdots \arrow[r, "B_t(\overline{x'})"] & \texttt{triv}(\overline{x'}) \arrow[d, equal] \\
\overline{y} & & & \overline{y} \arrow[lll]
\end{tikzcd}
$$

fun $\texttt{f-rec}(\overline{x})$ **dev** $\overline{y}$
    casos
        $B_t(\overline{x}) \;\rightarrow\; \overline{y} := \texttt{triv}(\overline{x})$
    $\square$ $B_{nt}(\overline{x}) \;\rightarrow\; \overline{y} := \texttt{f-rec}(\texttt{s}(\overline{x}))$
    fcasos
ffun

fun $\texttt{f-it}(\overline{x})$ **dev** $\overline{y}$
var $\overline{x'}$
    $\overline{x'} := \overline{x}\,;$
    mientras $B_{nt}(\overline{x'})$ **hacer**
        $\overline{x'} := \texttt{s}(\overline{x'})$
    fmientras ;
    $\overline{y} := \texttt{triv}(\overline{x'})$
ffun

Ejemplo: suma de los elementos de un vector

fun `gfsuma-vector`$(V[0..N]$ **de** *ent*$, n :$ *nat*$, w :$ *ent*$)$ **dev** $s :$ *ent*
  casos
      $n = N \;\rightarrow\; s := w$
    $\Box\; n < N \;\rightarrow\; s :=$ `gfsuma-vector`$(V, n+1, w+V[n])$
  fcasos
ffun

fun `gfsuma-vector-it`$(V[0..N]$ **de** *ent*$, n :$ *nat*$, w :$ *ent*$)$ **dev** $s :$ *ent*
var $n' :$ *nat*$, w' :$ *ent*
  $\langle n', w' \rangle := \langle n, w \rangle$;
  mientras $n' < N$ **hacer**
    $\langle n', w' \rangle := \langle n'+1, w'+V[n'] \rangle$
  fmientras ;
  $s := w'$
ffun

Para sumar todos los elementos de V hay que hacer la llamada `gfsuma-vector-it`$(V, 0, 0)$.

Ejemplo: suma de los elementos de un vector

Ahora podemos eliminar los parámetros añadidos para conseguir una función recursiva final, inicializando la *variable local* n a 0, y utilizando s en vez de w (inicializada también a 0):

fun `suma-vector-it`$(V[0..N]$ **de** *ent*$)$ **dev** $s :$ *ent*
var $n :$ *nat*
  $\langle n, s \rangle := \langle 0, 0 \rangle$;
  mientras $n < N$ **hacer**
    $\langle n, s \rangle := \langle n+1, s+V[n] \rangle$
  fmientras
ffun

Producto escalar de dos vectores

$\{\ 1 \leq i \leq N+1\ \wedge\ pp = (\sum j : 1 \leq j < i : V[j] * W[j])\ \}$
fun `prod-esc-rec`$(V[1..N], W[1..N]$ **de** *ent*$, i :$ *nat*$, pp :$ *ent*$)$ **dev** $p :$ *ent*
  casos
    $i = N+1\ \rightarrow\ p := pp$
  $\square$ $i < N+1\ \rightarrow\ p :=$ `prod-esc-rec`$(V, W, i+1, pp + V[i] * W[i])$
  fcasos
ffun
$\{\ p = \sum j : 1 \leq j \leq N : V[j] * W[j]\ \}$

fun `prod-esc-it`$(V[1..N], W[1..N]$ **de** *ent*$, i :$ *nat*$, pp :$ *ent*$)$ **dev** $p :$ *ent*
var $i' :$ *nat*$, pp' :$ *ent*
  $\langle i', pp' \rangle := \langle i, pp \rangle$;
  mientras $i' < N+1$ **hacer**
    $\langle i', pp' \rangle := \langle i'+1, pp' + V[i'] * W[i'] \rangle$
  fmientras ;
  $p := pp'$
ffun

Producto escalar de dos vectores

Ya que los parámetros i y pp de la función recursiva se habían añadido para conseguir una función recursiva y final, pueden ser eliminados de la versión iterativa, inicializando las variables locales a los valores iniciales en la primera llamada recursiva.

fun `prod-esc-it2`$(V[1..N], W[1..N]$ **de** *ent*$)$ **dev** $p :$ *ent*
var $i :$ *nat*
  $\langle i, p \rangle := \langle 1, 0 \rangle$;
  mientras $i < N+1$ **hacer**
    $\langle i, p \rangle := \langle i+1, p + V[i] * W[i] \rangle$
  fmientras
ffun

Otro ejemplo: el máximo común divisor

fun $\texttt{mcd-rec}(a, b : ent)$ **dev** $mcd : ent$
  casos
    $a = b \;\rightarrow\; mcd := a$
    $\Box\; a > b \;\rightarrow\; mcd := \texttt{mcd-rec}(a - b, b)$
    $\Box\; a < b \;\rightarrow\; mcd := \texttt{mcd-rec}(a, b - a)$
  fcasos
ffun

fun $\texttt{mcd-it}(a, b : ent)$ **dev** $mcd : ent$
var $a', b' : ent$
  $\langle a', b' \rangle := \langle a, b \rangle$;
  mientras $a' \neq b'$ **hacer**
    casos
      $a' > b' \;\rightarrow\; a' := a' - b'$
      $\Box\; a' < b' \;\rightarrow\; b' := b' - a'$
    fcasos
  fmientras ;
  $mcd := a'$
ffun

Recursión no final a iterativo

$$
\begin{array}{ccccccc}
\texttt{fun-rec}(x_1) & \xrightarrow{B_{nt}(x_1)} & \texttt{fun-rec}(x_2) & \xrightarrow{B_{nt}(x_2)} & \cdots & \xrightarrow{B_t(x_k)} & \texttt{triv}(x_k) \\
 & & & & & & \| \\
\texttt{c}(y_2, x_1) & \longleftarrow & \cdots & \longleftarrow & \underbrace{\texttt{c}(y_k, \overbrace{\texttt{s}^{-1}(x_k)}^{x_{k-1}})}_{y_{k-1}} & \longleftarrow & y_k
\end{array}
$$

fun $\texttt{f-rec}(\overline{x})$ **dev** $\overline{y}$
  casos
    $B_t(\overline{x}) \;\rightarrow\; \overline{y} := \texttt{triv}(\overline{x})$
    $\Box\; B_{nt}(\overline{x}) \;\rightarrow$
      $\overline{y} := \texttt{c}(\texttt{f-rec}(\texttt{s}(\overline{x})), \overline{x})$
  fcasos
ffun

fun $\texttt{f-it}(\overline{x})$ **dev** $\overline{y}$
var $\overline{x'}$
  $\overline{x'} := \overline{x}$;
  mientras $B_{nt}(\overline{x'})$ **hacer**
    $\overline{x'} := \texttt{s}(\overline{x'})$
  fmientras ;
  $\overline{y} := \texttt{triv}(\overline{x'})$;
  mientras $\overline{x'} \neq \overline{x}$ **hacer**
    $\overline{x'} := \texttt{s}^{-1}(\overline{x'})$;
    $\overline{y} := \texttt{c}(\overline{y}, \overline{x'})$
  fmientras
ffun

Para que el esquema anterior se pueda aplicar tiene que existir la función inversa del sucesor, s^{-1}. Esta función no siempre existe.

s	s^{-1}
$+1$	-1
-1	$+1$
$*2$	div 2
div 2	no hay

Si no existe, se utiliza una pila para almacenar los valores que después tienen que ser recuperados.

Ejemplo: suma de los elementos de un vector

fun $\texttt{gsuma-vector}(V[0..N]$ **de** *ent*$, n$: *nat*$)$ **dev** s : *ent*
  casos
      $n = 0 \;\rightarrow\; s := 0$
    $\square\; n > 0 \;\rightarrow\; s := \texttt{gsuma-vector}(V, n-1)\,;$
                  $s := s + V[n-1]$
  fcasos
ffun
fun $\texttt{gsuma-vector-it}(V[0..N]$ **de** *ent*$, n$: *nat*$)$ **dev** s : *ent*
var n' : *nat*
  $n' := n\,;$
  mientras $n' > 0$ **hacer**
    $n' := n' - 1$
  fmientras ;
equivalente a $n' := 0$
  $s := 0\,;$
  mientras $n' \neq n$ **hacer**
    $n' := n' + 1\,;$
    $s := s + V[n' - 1]$
  fmientras
ffun

Otro ejemplo: división entera

fun `dividir-rec`$(dv, ds : \mathit{ent})$ **dev** $\langle c, r : \mathit{ent} \rangle$
  casos
    $dv < ds \;\rightarrow\; \langle c, r \rangle := \langle 0, dv \rangle$
    $\Box\; dv \geq dv \;\rightarrow\; \langle c, r \rangle :=$ `dividir-rec`$(dv - ds, ds)$
      $c := c + 1$
  fcasos
ffun

fun `dividir-it`$(dv, ds : \mathit{ent})$ **dev** $\langle c, r : \mathit{ent} \rangle$
var $dv', ds' : \mathit{ent}$
  $dv' := dv$; $ds' := ds$;
  mientras $dv' \geq ds'$ **hacer**
    $dv' := dv' - ds'$
  fmientras ;
  $\langle c, r \rangle := \langle 0, dv' \rangle$;
  mientras $dv' \neq dv$ **hacer**
    $dv' := dv' + ds'$; $c := c + 1$
  fmientras
ffun

Transformación a iterativo con pila

fun `f-rec`$(\overline{x})$ **dev** $\overline{y}$
  casos
    $B_t(\overline{x}) \;\rightarrow\; \overline{y} :=$ `triv`$(\overline{x})$
    $\Box\; B_{nt}(\overline{x}) \;\rightarrow$
      $\overline{y} :=$ `c`(`f-rec`(`s`$(\overline{x})$)$, \overline{x}$)
  fcasos
ffun

fun `f-it`$(\overline{x})$ **dev** $\overline{y}$
var $\overline{x'}, p : \mathit{pila}$
  $\overline{x'} := \overline{x}$;
  $p :=$ `pila-vacía`() ;
  mientras $B_{nt}(\overline{x'})$ **hacer**
    `apilar`$(\overline{x'}, p)$;
    $\overline{x'} :=$ `s`$(\overline{x'})$
  fmientras ;
  $\overline{y} :=$ `triv`$(\overline{x'})$
  mientras $\neg$`es-pila-vacía?`(p) **hacer**
    $\overline{x'} :=$ `cima`(p) ;
    `desapilar`(p) ;
    $\overline{y} :=$ `c`$(\overline{y}, \overline{x'})$
  fmientras
ffun

Transformación a iterativo

fun $\texttt{suma-dígitos}(n : \textit{nat})$ **dev** $s : \textit{nat}$
    casos
        $n < 10 \;\rightarrow\; s := n$
    $\square$ $n \geq 10 \;\rightarrow\; s := \texttt{suma-dígitos}(n \text{ div } 10) + (n \text{ mód } 10)$
    fcasos
ffun

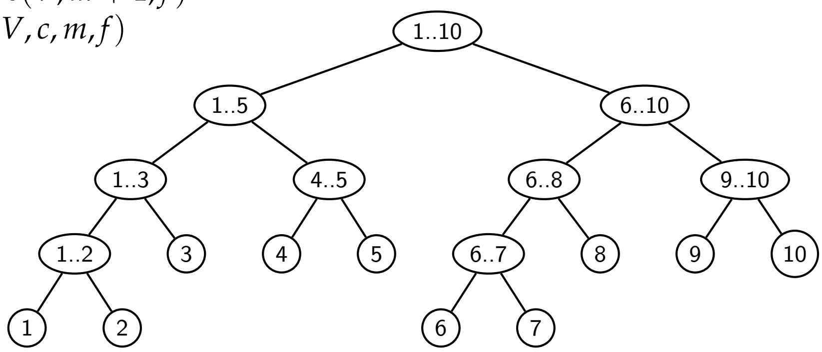
Transformación a iterativo

fun $\texttt{suma-dígitos-it}(n : \textit{nat})$ **dev** $s : \textit{nat}$
var $n' : \textit{nat}, p : \textit{pila[nat]}$
    $n' := n$;
    $p := \texttt{pila-vacía}()$;
    mientras $n' \geq 10$ **hacer**
        $\texttt{apilar}(n', p)$;
        $n' := n' \text{ div } 10$
    fmientras ;
    $s := n'$;
    mientras $\neg\texttt{es-pila-vacía?}(p)$ **hacer**
        $n' := \texttt{cima}(p)$;
        $\texttt{desapilar}(p)$;
        $s := s + (n' \text{ mód } 10)$
    fmientras
ffun

Recursión múltiple

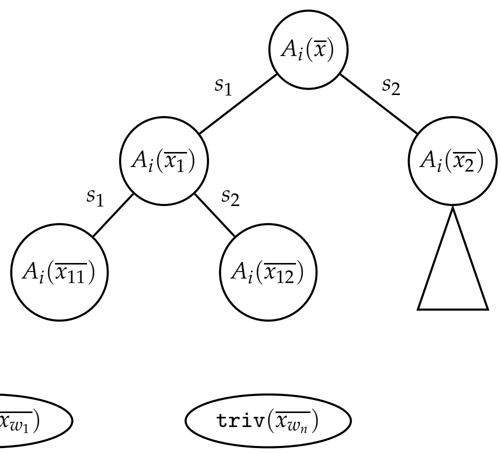
En este caso las llamadas recursivas generadas pueden representarse por medio de un árbol de activaciones.

proc $\texttt{mergesort}(V[1..n]$ **de** *elem*$, c, f$: *nat*$)$
  casos
    $c \geq f \;\rightarrow\; \{\;$ nada $\;\}$
  $\square\; c < f \;\rightarrow$
      $m := (c+f)\ \mathrm{div}\ 2$
      $\texttt{mergesort}(V, c, m)$
      $\texttt{mergesort}(V, m+1, f)$
      $\texttt{mezclar}(V, c, m, f)$
  fcasos
fproc



Esquema general simple de recursión múltiple

proc $\texttt{recursivo}(\overline{x})$
  casos
    $B_t(\overline{x}) \;\rightarrow\; \texttt{triv}(\overline{x})$
  $\square\; B_{nt}(\overline{x}) \;\rightarrow\; A_1(\overline{x})$
      $\texttt{recursivo}(s_1(\overline{x}))$
      $A_2(\overline{x})$
      $\texttt{recursivo}(s_2(\overline{x}))$
      $A_3(\overline{x})$
  fcasos
fproc



Simplificamos a cuando solo existe A_1 (preorden), A_2 (inorden) o A_3 (postorden). Obtenemos versiones iterativas de los recorridos.

Método de desplegado-plegado: Preorden

Este método consiste en generalizar las funciones recursivas que definen los recorridos estándar, para obtener funciones recursivas lineales finales, que transformaremos a iterativo.

La definición recursiva del recorrido en preorden de un árbol binario es la siguiente:

$$
\begin{array}{lcl}
\texttt{preorden}(\texttt{árbol-vacío}) & = & [] \\
\texttt{preorden}(\texttt{plantar}(iz, x, dr)) & = & [x] +\!\!+ \texttt{preorden}(iz) +\!\!+ \texttt{preorden}(dr)
\end{array}
$$

Primera generalización: añadir como parámetro adicional una lista con los nodos ya recorridos:

$$\texttt{gpre}(xs, a) = xs +\!\!+ \texttt{preorden}(a)$$

Es una generalización del preorden ya que

$$\texttt{gpre}([], a) = [] +\!\!+ \texttt{preorden}(a) = \texttt{preorden}(a)$$

Cuando el árbol es vacío tenemos un caso básico:

$$\texttt{gpre}(xs, \texttt{árbol-vacío}) = xs +\!\!+ \texttt{preorden}(\texttt{árbol-vacío}) = xs +\!\!+ [] = xs$$

Cuando el árbol no es vacío, es decir, es de la forma $\texttt{plantar}(iz, x, dr)$, tenemos

$$
\begin{array}{lcl}
\texttt{gpre}(xs, \texttt{plantar}(iz, x, dr)) & = & xs +\!\!+ \texttt{preorden}(\texttt{plantar}(iz, x, dr)) \\
& = & xs +\!\!+ ([x] +\!\!+ \texttt{preorden}(iz) +\!\!+ \texttt{preorden}(dr)) \\
& = & (xs +\!\!+ [x]) +\!\!+ \texttt{preorden}(iz) +\!\!+ \texttt{preorden}(dr) \\
& = & \texttt{gpre}(xs +\!\!+ [x], iz) +\!\!+ \texttt{preorden}(dr) \\
& = & \texttt{gpre}(\texttt{gpre}(xs +\!\!+ [x], iz), dr)
\end{array}
$$

Pero seguimos teniendo dos llamadas recursivas.

Mantenemos no solo un árbol como argumento, sino una pila de árboles.

La generalización necesaria es

$$\texttt{ggpre}(xs, p) = xs +\!\!+\, \texttt{prep}(p)$$

donde xs es una lista, p es una pila de árboles y

$$\begin{array}{lcl} \texttt{prep}(\texttt{pila-vacía}) & = & [\,] \\ \texttt{prep}(\texttt{apilar}(a, p)) & = & \texttt{preorden}(a) +\!\!+\, \texttt{prep}(p) \end{array}$$

Es una generalización del preorden:

$$\begin{array}{lcl} \texttt{ggpre}([\,], \texttt{apilar}(a, \texttt{pila-vacía})) & = & [\,] +\!\!+\, \texttt{prep}(\texttt{apilar}(a, \texttt{pila-vacía})) \\ & = & \texttt{preorden}(a) +\!\!+\, \texttt{prep}(\texttt{pila-vacía}) \\ & = & \texttt{preorden}(a) +\!\!+\, [\,] \\ & = & \texttt{preorden}(a) \end{array}$$

Definimos recursivamente ggpre desplegando y plegando. Cuando la pila es vacía tenemos un caso básico:

$$\texttt{ggpre}(xs, \texttt{pila-vacía}) = xs +\!\!+\, \texttt{prep}(\texttt{pila-vacía}) = xs +\!\!+\, [\,] = xs$$

Cuando la pila no es vacía tenemos

$$\begin{array}{lcl} \texttt{ggpre}(xs, \texttt{apilar}(a, p)) & = & xs +\!\!+\, \texttt{prep}(\texttt{apilar}(a, p)) \\ & = & xs +\!\!+\, \texttt{preorden}(a) +\!\!+\, \texttt{prep}(p) \end{array}$$

Ahora, cuando a es vacío podemos seguir así

$$\begin{array}{lcl} & = & xs +\!\!+\, [\,] +\!\!+\, \texttt{prep}(p) \\ & = & xs +\!\!+\, \texttt{prep}(p) \\ & = & \texttt{ggpre}(xs, p) \end{array}$$

y cuando a no es vacío, es decir, es de la forma $\texttt{plantar}(iz, x, dr)$, tenemos

$$\begin{array}{lcl} & = & xs +\!\!+\, ([x] +\!\!+\, \texttt{preorden}(iz) +\!\!+\, \texttt{preorden}(dr)) +\!\!+\, \texttt{prep}(p) \\ & = & xs +\!\!+\, [x] +\!\!+\, \texttt{preorden}(iz) +\!\!+\, \texttt{prep}(\texttt{apilar}(dr, p)) \\ & = & xs +\!\!+\, [x] +\!\!+\, \texttt{prep}(\texttt{apilar}(iz, \texttt{apilar}(dr, p))) \\ & = & \texttt{ggpre}(xs +\!\!+\, [x], \texttt{apilar}(iz, \texttt{apilar}(dr, p))) \end{array}$$

Los dos casos recursivos son lineales finales. El algoritmo recursivo final completo es el siguiente:

fun ggpre(xs : *lista*, p : *pila[árbol-bin]*) **dev** ys : *lista*
var a : *árbol-bin*
casos
  es-pila-vacía?(p) $\rightarrow$ $ys := xs$
  $\square$ $\neg$es-pila-vacía?(p) $\rightarrow$
    $a :=$ cima(p) ; desapilar(p)
    casos
      es-árbol-vacío?(a) $\rightarrow$ $ys :=$ ggpre(xs, p)
      $\square$ $\neg$es-árbol-vacío?(a) $\rightarrow$
        añadir-der(xs, raíz(a))
        apilar(hijo-dr(a), p)
        apilar(hijo-iz(a), p)
        $ys :=$ ggpre(xs, p)
    fcasos
fcasos
ffun

¿Termina? La lista va creciendo y la pila también?

En el primer caso recursivo decrece en 1 la altura de la pila y en el segundo caso crece en 1 la altura pero decrece en 1 el número total de nodos en los árboles apilados.

La función que vamos a tomar es la siguiente: $T(xs, p) = \texttt{alt}(p) + 2 * \texttt{ntn}(p)$ donde $\texttt{alt}(p)$ es la altura de la pila, es decir, el número de elementos apilados, y $\texttt{ntn}(p)$ es el número total de nodos en los árboles apilados en p:

$$
\begin{array}{lcl}
\texttt{alt}(\texttt{pila-vacía}) & = & 0 \\
\texttt{alt}(\texttt{apilar}(a, p)) & = & 1 + \texttt{alt}(p) \\
\texttt{ntn}(\texttt{pila-vacía}) & = & 0 \\
\texttt{ntn}(\texttt{apilar}(a, p)) & = & \texttt{nodos}(a) + \texttt{ntn}(p) \\
\texttt{nodos}(\texttt{árbol-vacío}) & = & 0 \\
\texttt{nodos}(\texttt{plantar}(iz, x, dr)) & = & \texttt{nodos}(iz) + 1 + \texttt{nodos}(dr)
\end{array}
$$

La función T decrece en los dos casos recursivos:

$$T(xs, \texttt{apilar}(\texttt{árbol-vacío}, p)) > T(xs, p)$$
$$T(xs, \texttt{apilar}(\texttt{plantar}(iz, x, dr), p)) > T(xs \mathbin{+\!\!+} [x], \texttt{apilar}(iz, \texttt{apilar}(dr, p)))$$

La versión iterativa del preorden, transformando el algoritmo recursivo final ggpre y dando los valores iniciales a los parámetros acumuladores es:

```
fun preorden-it(a : árbol-bin) dev xs : lista
var p : pila[árbol-bin]
    xs := [] ; p := pila-vacía() ; apilar(a,p)
    mientras ¬es-pila-vacía?(p) hacer
        a := cima(p) ; desapilar(p)
        si ¬es-árbol-vacío?(a) entonces
            añadir-der(xs,raíz(a))
            apilar(hijo-dr(a),p)
            apilar(hijo-iz(a),p)
        fsi
    fmientras
ffun
```


Versión iterativa de quicksort

La versión recursiva del algoritmo de ordenación quicksort es la siguiente:

```
proc quicksort(V[1..n] de elem, c, f : nat)
    casos
          c ≥ f  →  { nada }
        □ c < f  →
            partición(V, c, f, p)
            quicksort(V, c, p − 1)
            quicksort(V, p + 1, f)
    fcasos
fproc
```

La llamada inicial para ordenar todo el vector V es quicksort$(V, 1, n)$.

Para obtener la versión iterativa de `quicksort` modificamos el algoritmo `preorden-it` pensando

- cómo representamos los nodos del árbol,
- cuándo un árbol es vacío,
- cómo pasar a los hijos izquierdo y derecho de un árbol, y
- qué trabajo hay que realizar en cada nodo.

Representamos las llamadas recursivas (los nodos del árbol de activaciones) como parejas $\langle c,f \rangle$, donde c y f nos indican el segmento del vector V que vamos a ordenar.

- La raíz del árbol es $\langle 1,n \rangle$.
- $\langle c,f \rangle$ representa el árbol vacío cuando $c \geq f$.
- Visitar el nodo $\langle c,f \rangle$ consiste en hacer una partición del vector V entre los límites c y f. El procedimiento `partición` nos devuelve el índice p donde se ha colocado el pivote.
- El hijo izquierdo de $\langle c,f \rangle$ será $\langle c,p-1 \rangle$ y el hijo derecho $\langle p+1,f \rangle$.

Con estas ideas, la versión iterativa de quicksort es la siguiente:

```
proc quicksort-it(V[1..n] de elem)
var p : pila[pareja]
   p := pila-vacía() ; apilar(⟨1,n⟩,p)
   mientras ¬es-pila-vacía?(p) hacer
      ⟨c,f⟩ := cima(p) ; desapilar(p)
      si c < f entonces
         partición(V,c,f,piv)
         apilar(⟨piv+1,f⟩,p)
         apilar(⟨c,piv−1⟩,p)
      fsi
   fmientras
fproc
```

Aquí no se produce una lista, sino que el trabajo se va realizando sobre el parámetro de entrada/salida V.

Método *primero-último-sucesor*

Método general para obtener versiones iterativas de cualquier recorrido.

A la hora de recorrer un árbol, empezamos por un nodo (el primero en el recorrido), hacemos algo con ese nodo, y pasamos al siguiente en el recorrido. Así hasta llegar al último nodo del recorrido.

Por tanto, para definir un recorrido de un árbol, es suficiente definir las siguientes funciones:

- `primero`: devuelve el primer nodo en el recorrido,
- `sucesor`: devuelve el siguiente nodo del recorrido,
- `último`: devuelve el último nodo en el recorrido, y
- `visitar`: realiza una acción sobre un nodo.

Entendemos las funciones `primero`, `último` y `sucesor` como funciones que dado un árbol nos devuelven otro árbol cuyo nodo raíz es el que nos interesa.

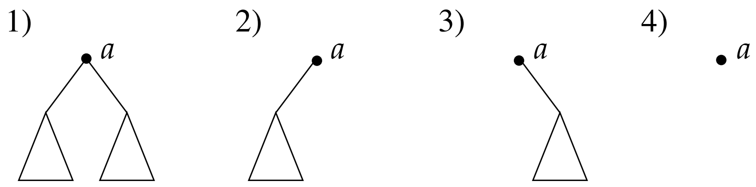
Utilizando estas funciones, la versión iterativa del recorrido de un árbol no vacío a será:

p := primero(a)
u := último(a)
visitar(raíz(p))
mientras $p \neq u$ **hacer**
    p := sucesor(p)
    visitar(raíz(p))
fmientras

Lo que falta es obtener versiones iterativas de las funciones `primero`, `último` y `sucesor`.

Método *primero-último-sucesor*: Inorden

Definimos las operaciones `primero`, `último` y `sucesor` distinguiendo los siguientes casos:



- Definimos `primero`. En los dos primeros casos, el árbol a tiene hijo izquierdo, y el primer nodo a visitar en un recorrido en inorden estará en ese hijo, será el primero del hijo izquierdo:

  $\texttt{primero}(a) = \texttt{primero}(\texttt{hijo-iz}(a)) \qquad \textbf{si } \texttt{tiene-iz?}(a)$

  En los dos últimos casos no existe hijo izquierdo y, por tanto, el primer nodo visitado en el recorrido en inorden será la raíz:

  $\texttt{primero}(a) = a \qquad \textbf{si } \neg\texttt{tiene-iz?}(a)$

La implementación de `primero` como algoritmo recursivo (final) es la siguiente:

fun $\texttt{primero}(a : \textit{árbol-bin})$ **dev** $p : \textit{árbol-bin}$
    casos
        $\neg\texttt{tiene-iz?}(a) \;\rightarrow\; p := a$
      $\square\; \texttt{tiene-iz?}(a) \;\rightarrow\; p := \texttt{primero}(\texttt{hijo-iz}(a))$
    fcasos
ffun

y la versión iterativa

fun $\texttt{primero-it}(a : \textit{árbol-bin})$ **dev** $p : \textit{árbol-bin}$
    $p := a$
    mientras $\texttt{tiene-iz?}(p)$ **hacer**
        $p := \texttt{hijo-iz}(p)$
    fmientras
ffun

- Definimos `último`. En el primer y tercer caso existe hijo derecho, y el último nodo se encuentra en ese hijo, será el último del hijo derecho:

  $$\texttt{último}(a) = \texttt{último}(\texttt{hijo-dr}(a)) \qquad \textbf{si } \texttt{tiene-dr?}(a)$$

  En el segundo y cuarto caso no existe hijo derecho, y por tanto el último nodo visitado será la raíz de a:

  $$\texttt{último}(a) = a \qquad \textbf{si } \neg\texttt{tiene-dr?}(a)$$

La versión iterativa de `último` es:

fun `último-it`(a : *árbol-bin*) **dev** u : *árbol-bin*
  $u := a$
  mientras `tiene-dr?`(u) **hacer**
    $u :=$ `hijo-dr`(u)
  fmientras
ffun

- Definimos `sucesor`. En el primer y tercer caso existe hijo derecho, y por tanto el siguiente nodo visitado será el primero del hijo derecho:

  $$\texttt{sucesor}(a) = \texttt{primero}(\texttt{hijo-dr}(a)) \qquad \textbf{si } \texttt{tiene-dr?}(a)$$

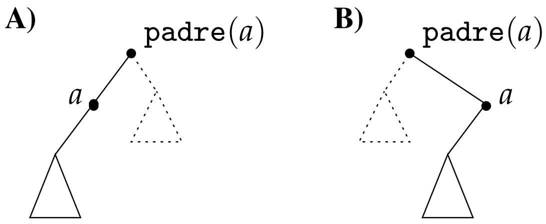
  En el segundo y cuarto caso, al no haber hijo derecho, hemos terminado de recorrer el árbol a y tenemos que pasar al árbol más grande que contiene a a. Este siempre existe, ya que el sucesor solamente se calcula cuando no se ha llegado al último.

  Necesitamos saber cuál es el árbol que contiene a a, por lo que añadimos un parámetro que represente el árbol total b que estamos recorriendo (y que contiene a todos los subárboles). Así

  $$\begin{array}{lcll} \texttt{sucesor}(a,b) & = & \texttt{primero}(\texttt{hijo-dr}(a)) & \textbf{si } \texttt{tiene-dr?}(a) \\ \texttt{sucesor}(a,b) & = & \texttt{remonta}(a,b) & \textbf{si } \neg\texttt{tiene-dr?}(a) \end{array}$$

  y ahora definimos `remonta`.

Hay que distinguir casos según a sea un hijo izquierdo o un hijo derecho:



Cuando a es un hijo izquierdo remontar consiste en pasar al padre,

$$\texttt{remonta}(a,b) = \texttt{padre}(a,b) \qquad \textbf{si } \texttt{es-izq?}(a,b)$$

y cuando a es un hijo derecho, también hemos acabado con su padre, y hay que seguir remontando *recursivamente*:

$$\texttt{remonta}(a,b) = \texttt{remonta}(\texttt{padre}(a,b),b) \qquad \textbf{si } \texttt{es-der?}(a,b)$$

Por completitud definimos las funciones es-der? y padre aunque no vamos a implementarlas directamente.

$$\begin{array}{lcl}
\texttt{es-der?}(a,\texttt{árbol-vacío}) & = & \texttt{falso} \\
\texttt{es-der?}(a,\texttt{plantar}(a',x,a)) & = & \texttt{cierto} \\
\texttt{es-der?}(a,\texttt{plantar}(a',x,a'')) & = & \texttt{es-der?}(a,a') \lor \texttt{es-der?}(a,a'') \quad \textbf{si } a \neq a''
\end{array}$$

$$\begin{array}{lcl}
\texttt{padre}(a,b) & = & b \quad \textbf{si } a = \texttt{hijo-iz}(b) \lor a = \texttt{hijo-dr}(b) \\
\texttt{padre}(a,b) & = & \texttt{padre}(a,\texttt{hijo-iz}(b)) \quad \textbf{si } \texttt{antecesor}(a,\texttt{hijo-iz}(b)) \\
\texttt{padre}(a,b) & = & \texttt{padre}(a,\texttt{hijo-dr}(b)) \quad \textbf{si } \texttt{antecesor}(a,\texttt{hijo-dr}(b))
\end{array}$$

$$\begin{array}{lcl}
\texttt{antecesor}(a,\texttt{árbol-vacío}) & = & \texttt{falso} \\
\texttt{antecesor}(a,\texttt{plantar}(a,x,a')) & = & \texttt{cierto} \\
\texttt{antecesor}(a,\texttt{plantar}(a',x,a)) & = & \texttt{cierto} \\
\texttt{antecesor}(a,\texttt{plantar}(a',x,a'')) & = & \texttt{antecesor}(a,a') \lor \texttt{antecesor}(a,a'') \quad \textbf{si } a \neq a' \land a \neq a''
\end{array}$$

Dejando por ahora pendiente la definición de las funciones para calcular el padre de un árbol y si es hijo izquierdo o derecho, la versión iterativa de `remonta` es

```
fun remonta-it(a, b : árbol-bin) dev s : árbol-bin
    s := a
    mientras es-der?(s, b) hacer
        s := padre(s, b)
    fmientras
    s := padre(s, b)
ffun
```


Inorden iterativo

La versión completa del `inorden` es

```
proc inorden-it(a : árbol-bin)
var p, u : árbol-bin
    { cálculo del primero de a }
    p := a
    mientras tiene-iz?(p) hacer
        p := hijo-iz(p)
    fmientras
    { cálculo del último de a }
    u := a
    mientras tiene-dr?(u) hacer
        u := hijo-dr(u)
    fmientras
    visitar(raíz(p))    { visita el primer nodo }
```

```
    mientras p ≠ u hacer
        { cálculo del sucesor de p }
        casos
              tiene-dr?(p)  →  { pasar al primero del hijo derecho }
                p := hijo-dr(p)
                mientras tiene-iz?(p) hacer
                    p := hijo-iz(p)
                fmientras
            □ ¬tiene-dr?(p)  →  { remontar }
                mientras [es-der?(p,a)] hacer
                    p := [padre(p,a)]
                fmientras
                p := [padre(p,a)]
        fcasos
        visitar(raíz(p))
    fmientras
fproc
```


Inorden iterativo con pila

Para conocer el padre del nodo actual p vamos a utilizar una pila de árboles. En todo momento la pila contendrá en su cima el padre de p. Así

$$\texttt{es-der?}(p) \equiv (p = \texttt{hijo-dr}(\texttt{padre}(p)))$$

Para mantener en la pila siempre el padre de p tendremos que apilar al padre cuando pasemos de él a p. En el cálculo del último nodo visitado no es necesario guardar el padre, ya que al llegar al último no vamos a seguir.

El algoritmo completo con pila es el siguiente:

```
proc inorden-it-pila(a : árbol-bin)
var p, u : árbol-bin, pila : pila[árbol-bin]
    pila := pila-vacía()
    { cálculo del primero de a }
    p := a
    mientras tiene-iz?(p) hacer
        apilar(p, pila) ;  p := hijo-iz(p)
    fmientras
```

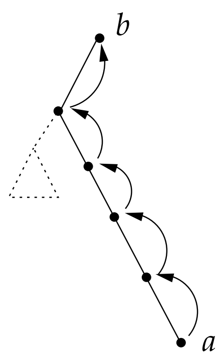
```
{ cálculo del último de a }
u := a
mientras tiene-dr?(u) hacer
    u := hijo-dr(u)
fmientras
visitar(raíz(p))    { visita el primer nodo }
mientras p ≠ u hacer
    { cálculo del sucesor de p }
    casos
          tiene-dr?(p)  →  { pasar al primero del hijo derecho }
           apilar(p,pila) ; p := hijo-dr(p)
           mientras tiene-iz?(p) hacer
               apilar(p,pila) ; p := hijo-iz(p)
           fmientras
        □ ¬tiene-dr?(p)  →  { remontar }
           mientras p = hijo-dr(cima(pila)) hacer
               p := cima(pila) ; desapilar(pila)
           fmientras
           p := cima(pila) ; desapilar(pila)
    fcasos
    visitar(raíz(p))
fmientras
fproc
```


Inorden iterativo con pila simplificado

¿Cuándo es realmente necesario guardar el padre de un nodo?

Cuando pasamos a un hijo izquierdo, sí es necesario, porque será visitado cuando acabemos con su hijo izquierdo.

En cambio, al pasar a un hijo derecho, el nodo en el que estamos ya ha sido visitado, por lo que no es necesario volver a pasar por él.



En la cima tendremos el siguiente nodo que hay que visitar cuando es necesario remontar.

Podemos cambiar la condición del bucle, para evitar el cálculo del último nodo, haciendo que el bucle termine cuando la pila sea vacía.

```
proc inorden-it-simp(a : árbol-bin)
var p : árbol-bin, pila : pila[árbol-bin]
    pila := pila-vacía()
    p := a
    mientras tiene-iz?(p) hacer
        apilar(p, pila) ; p := hijo-iz(p)
    fmientras
    pila := apilar(p, pila)
    mientras ¬es-pila-vacía?(pila) hacer
        p := cima(pila) ; desapilar(pila)
        visitar(raíz(p))
        si tiene-dr?(p) entonces
            p := hijo-dr(p)
            mientras tiene-iz?(p) hacer
                apilar(p, pila) ; p := hijo-iz(p)
            fmientras
            apilar(p, pila)
        fsi
    fmientras
fproc
```


Versión iterativa de las torres de Hanoi

El algoritmo recursivo que resuelve el problema de las torres de Hanoi es el siguiente:

```
proc Hanoi(k : nat, origen, auxiliar, destino : torre)
    casos
          k = 1 → mover(origen, destino)
        □ k > 1 →
                Hanoi(k − 1, origen, destino, auxiliar)
                mover(origen, destino)
                Hanoi(k − 1, auxiliar, origen, destino)
    fcasos
fproc
```

La llamada inicial para mover N discos utilizando las torres A, B y C es $\texttt{Hanoi}(N, A, B, C)$.

Podemos representar los nodos del árbol de activaciones como tuplas $\langle k, a, b, c \rangle$.

- Un árbol $\langle k, a, b, c \rangle$ tiene hijo izquierdo (igualmente hijo derecho) si $k > 1$.
- El hijo izquierdo del árbol $\langle k, a, b, c \rangle$ es $\langle k-1, a, c, b \rangle$ y el hijo derecho $\langle k-1, b, a, c \rangle$.
- Visitar el nodo $\langle k, a, b, c \rangle$ consiste en mover de la torre a a la torre c (y no importa que $\langle k, a, b, c \rangle$ sea una hoja o un nodo interno, ya que lo que se hace en la función en el caso básico y el caso recursivo es lo mismo).

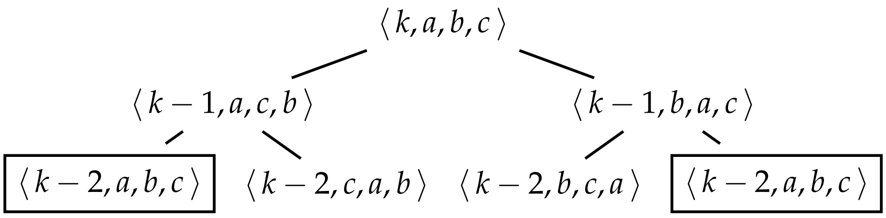
Con estos datos podemos modificar el algoritmo `inorden-it-simp` para obtener una versión iterativa.

Hanoi iterativo con pila

```
proc Hanoi-it(N : nat)
var pila : pila-tuplas
    pila := pila-vacía()
    ⟨k,a,b,c⟩ := ⟨N,A,B,C⟩
    mientras k > 1 hacer
        apilar(⟨k,a,b,c⟩,pila)
        ⟨k,a,b,c⟩ := ⟨k − 1,a,c,b⟩
    fmientras
    apilar(⟨k,a,b,c⟩,pila)
    mientras ¬es-pila-vacía?(pila) hacer
        ⟨k,a,b,c⟩ := cima(pila) ; desapilar(pila)
        mover(a,c)
        si k > 1 entonces
            ⟨k,a,b,c⟩ := ⟨k − 1,b,a,c⟩
            mientras k > 1 hacer
                apilar(⟨k,a,b,c⟩,pila)
                ⟨k,a,b,c⟩ := ⟨k − 1,a,c,b⟩
            fmientras
            apilar(⟨k,a,b,c⟩,pila)
        fsi
    fmientras
fproc
```

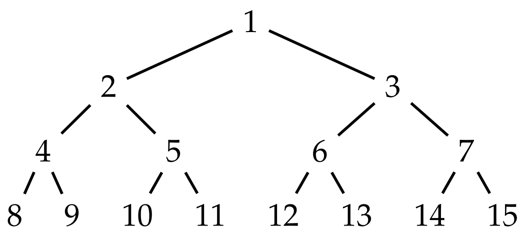
Numeración de los nodos de un árbol

En los algoritmos anteriores necesitabamos una pila auxiliar porque, en general, no podemos calcular el padre de un nodo a partir de sí mismo. En algunos casos, añadiendo cierta información a cada nodo si es necesario, podemos a partir de un nodo calcular su padre, y qué número de hijo es (para un árbol binario, si es hijo izquierdo o derecho).



Vamos a ver cómo añadir un número distinto a cada nodo de forma que podamos saber si el nodo es hijo izquierdo o derecho.

Supongamos que el árbol es binario completo o semicompleto. Comenzamos con el número 1 en la raíz, y luego vamos numerando por niveles:



La función que numera los nodos es:

$$\begin{array}{lcl}
\texttt{núm}(\textit{árbol original}) & = & 1 \\
\texttt{núm}(\texttt{hijo-iz}(a)) & = & 2 * \texttt{núm}(a) \\
\texttt{núm}(\texttt{hijo-dr}(a)) & = & 2 * \texttt{núm}(a) + 1
\end{array}$$

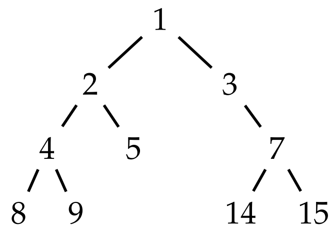
Podemos saber si un nodo es hijo izquierdo o derecho viendo si su número es par o impar, y podemos calcular el número del padre dividiendo por 2:

$$\begin{array}{lclcl}
\texttt{es-izq?}(a) & \equiv & \texttt{par?}(\texttt{núm}(a)) & \equiv & \texttt{núm}(a) \bmod 2 = 0 \\
\texttt{es-der?}(a) & \equiv & \texttt{impar?}(\texttt{núm}(a)) & \equiv & \texttt{núm}(a) \bmod 2 = 1 \\
\texttt{núm}(\texttt{padre}(a)) & = & \texttt{núm}(a) \operatorname{div} 2 & &
\end{array}$$

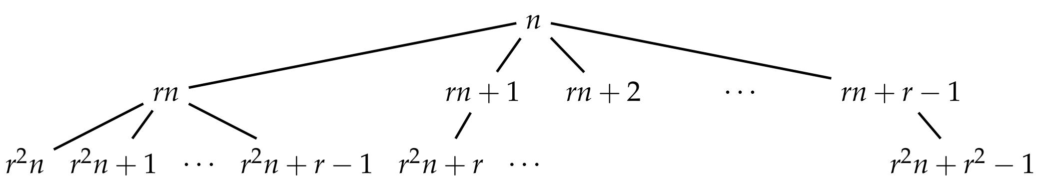
Cuando el árbol es binario pero no completo, todas las funciones anteriores siguen siendo válidas, pero restringiendo a que existan los correspondientes hijos:

$$\begin{array}{lcll} \texttt{núm}(\texttt{hijo-iz}(a)) & = & 2 * \texttt{núm}(a) & \textbf{si} \;\; \texttt{tiene-iz?}(a) \\ \texttt{núm}(\texttt{hijo-dr}(a)) & = & 2 * \texttt{núm}(a) + 1 & \textbf{si} \;\; \texttt{tiene-dr?}(a) \end{array}$$

Puede ocurrir que no utilicemos todos los números, pero eso no es importante, lo importante es que los números no se repitan.



Para numerar un árbol general utilizamos el *grado* del árbol. Para pasar al primer hijo multiplicamos el número del padre por el grado r, y utilizamos números consecutivos para numerar al resto de los hijos:



Así, la función que calcula el número de cierto hijo y el número del padre, y el predicado para comprobar el tipo de hijo se definen de la siguiente manera:

$$\begin{array}{lcll} \texttt{núm}(\texttt{hijo}(a,i)) & = & r * \texttt{núm}(a) + i - 1 & \text{con } 1 \leq i \leq n \\ \texttt{núm}(\texttt{padre}(a)) & = & \texttt{núm}(a) \text{ div } r & \\ \texttt{ser-hijo?}(a,i) & \equiv & (\texttt{núm}(a) \text{ mód } r = i - 1) & \end{array}$$

Hanoi iterativo con numeración

Añadimos un número más a cada nodo de la forma que hemos visto:

$$\langle k,a,b,c,n\rangle$$

$$\langle k-1,a,c,b,2n\rangle \qquad \langle k-1,b,a,c,2n+1\rangle$$

$$\langle k-2,a,b,c,4n\rangle \quad \langle k-2,c,a,b,4n+1\rangle \quad \langle k-2,b,c,a,4n+2\rangle \quad \langle k-2,a,b,c,4n+3\rangle$$

Ahora podemos calcular todas las funciones sin necesidad de una pila:

$$
\begin{array}{l}
\textit{árbol original} = \langle N,A,B,C,1\rangle \\
\texttt{hijo-iz}(\langle h,a,b,c,n\rangle) = \langle h-1,a,c,b,2n\rangle \\
\texttt{hijo-dr}(\langle h,a,b,c,n\rangle) = \langle h-1,b,a,c,2n+1\rangle \\
\texttt{tiene-iz?}(\langle h,a,b,c,n\rangle) = (h>1) \\
\texttt{tiene-dr?}(\langle h,a,b,c,n\rangle) = (h>1) \\
\texttt{es-izq?}(\langle h,a,b,c,n\rangle) = \texttt{par?}(n) \\
\texttt{es-der?}(\langle h,a,b,c,n\rangle) = \texttt{impar?}(n) \\
\texttt{padre}(\langle h,a,b,c,n\rangle) = \begin{cases} \langle h+1,a,c,b,n \text{ div } 2\rangle & \textbf{si } \texttt{es-izq?}(\langle h,a,b,c,n\rangle) \\ \langle h+1,b,a,c,n \text{ div } 2\rangle & \textbf{si } \texttt{es-der?}(\langle h,a,b,c,n\rangle) \end{cases} \\
\texttt{visitar}(\langle h,a,b,c,n\rangle) = \texttt{mover}(a,c)
\end{array}
$$

Hanoi iterativo con numeración

proc $\texttt{hanoi-it}(N : \textit{nat})$
    { cálculo del primero }
    $\langle k,a,b,c,n\rangle := \langle N,A,B,C,1\rangle$
    mientras $k>1$ **hacer**
        $\langle k,a,b,c,n\rangle := \langle k-1,a,c,b,2*n\rangle$
    fmientras
    { cálculo del último }
    $\langle ku,au,bu,cu,nu\rangle := \langle k,A,B,C,1\rangle$
    mientras $ku>1$ **hacer**
        $\langle ku,au,bu,cu,nu\rangle := \langle ku-1,bu,au,cu,2*nu+1\rangle$
    fmientras
    $\texttt{mover}(a,c)$ { visita raíz }
    mientras $\langle k,a,b,c,n\rangle \neq \langle ku,au,bu,cu,nu\rangle$ **hacer**
        { cálculo del sucesor }
        si $k>1$ **entonces**
            { pasar al primero del hijo derecho }
            $\langle k,a,b,c,n\rangle := \langle k-1,b,a,c,2*n+1\rangle$
            mientras $k>1$ **hacer**
                $\langle k,a,b,c,n\rangle := \langle k-1,a,c,b,2*n\rangle$
            fmientras

si no
  { remontar }
  mientras `impar?`(n) **hacer**
    $\langle k,a,b,c,n \rangle := \langle k+1,b,a,c,n \text{ div } 2 \rangle$ { padre de un hijo derecho }
  fmientras
  $\langle k,a,b,c,n \rangle := \langle k+1,a,c,b,n \text{ div } 2 \rangle$ { padre de un hijo izquierdo }
fsi
`mover`(a,c)
fmientras
fproc

Analizando el algoritmo que hemos obtenido, vemos que podemos realizar las siguientes simplificaciones:

- Ya que cada nodo viene identificado por un número distinto, para calcular el último es suficiente su número.
- Las asignaciones que asignan a una variable su valor pueden omitirse.
- El cuerpo del primer bucle intercambia los valores de las variables b y c. Por tanto, si el bucle da un número par de vueltas vuelve al estado original, y si da un número impar de vueltas, es como si intercambiara una sola vez. Ya que k termina valiendo 1 y se le resta 1 en cada vuelta, el bucle da $k-1$ vueltas. Si antes del bucle, k es par hay que intercambiar los valores de b y c.
- Lo mismo ocurre con el cuarto bucle.
- En el quinto bucle ocurre algo similar. En este caso k empieza valiendo 1 y se incrementa en 1 en cada vuelta. Si después del bucle k es par se han dado un número impar de vueltas, y hay que intercambiar los valores de las variables a y b.
- El primer y segundo bucle dan el mismo número de vueltas, por lo que se pueden unir.

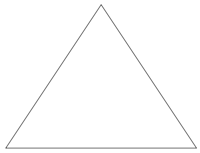
Hanoi iterativo con numeración simplificado

proc hanoi-it$(N : nat)$
    $\langle k, a, b, c, n \rangle := \langle N, A, B, C, 1 \rangle$
    $nu := 1$
    si par?(k) **entonces** $\langle b, c \rangle := \langle c, b \rangle$ **fsi**
    mientras $k > 1$ **hacer**
        $k := k - 1$
        $n := 2 * n$
        $nu := 2 * nu + 1$
    fmientras
    mover(a, c)
    mientras $n \neq nu$ **hacer**
        si $k > 1$ **entonces**
            $\langle k, a, b, n \rangle := \langle k - 1, b, a, 2 * n + 1 \rangle$
            si par?(k) **entonces** $\langle b, c \rangle := \langle c, b \rangle$ **fsi**
            mientras $k > 1$ **hacer**
                $k := k - 1$
                $n := 2 * n$
            fmientras

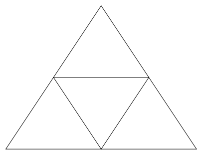
        si no
            mientras impar?(n) **hacer**
                $k := k + 1$
                $n := n \text{ div } 2$
            fmientras
            si par?(k) **entonces** $\langle a, b \rangle := \langle b, a \rangle$ **fsi**
            $\langle k, b, c, n \rangle := \langle k + 1, c, b, n \text{ div } 2 \rangle$
        fsi
        mover(a, c)
    fmientras
fproc

Recorrido no estándar: triángulos anidados

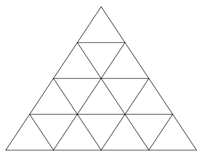
Para $n = 0$ la figura es un triángulo:



Para $n = 1$ dividimos el triángulo anterior en cuatro triángulos iguales:



Para $n = 2$ volvemos a dividir cada uno de los triángulos en cuatro:

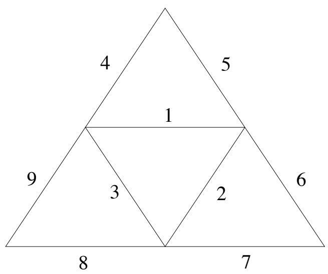


Y así sucesivamente...

El dibujo consiste en *anidar* triángulos hasta un cierto nivel dado.

Se requiere que se dibuje la figura con un solo trazo (sin levantar el lápiz del papel), sin pasar dos veces por el mismo segmento.

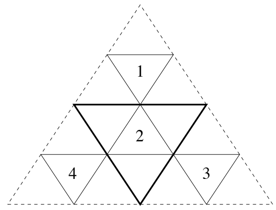
Para $n = 0$ no hay problema, pasamos una vez por cada segmento. Para $n = 1$ podemos recorrer el triángulo interno y luego el externo:



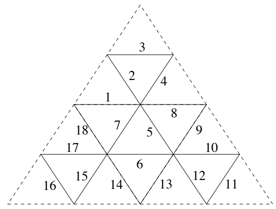
Si al pasar a $n = 2$ pintaramos cada uno de los cuatro triángulos como hemos pintado el del caso $n = 1$ habría segmentos que pintaríamos dos veces.

La solución consiste en pintar solo los triángulos interiores de forma recursiva, y pintar por separado el exterior del triángulo mayor.

Así, para $n = 2$ pintamos de forma recursiva el interior de los cuatro triángulos y además pintamos los segmentos de separación:

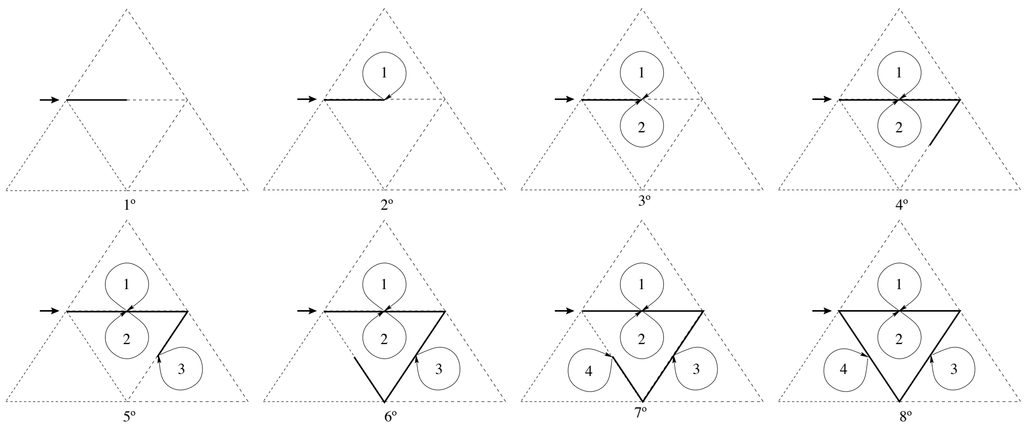


El orden en el que se pintan todos los trazos para $n = 2$ es el siguiente:



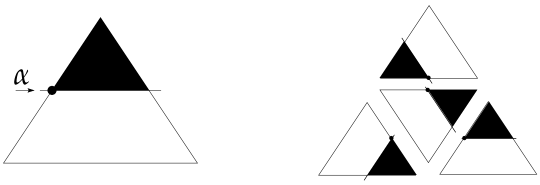
Nótese que hemos empezado y terminado en el mismo punto.

La siguiente figura muestra cómo pintamos el interior del triángulo en general, para cierto nivel n. Los triángulos de un nivel menor se pintan de forma recursiva.



Los parámetros de las llamadas recursivas serán:

- el nivel n por el que vamos, que se reducirá en 1 al hacer la llamada recursiva,
- la longitud l de los separadores que estamos dibujando, que se dividirá por 2 al hacer la llamada, y
- la orientación α del triángulo con respecto al original y teniendo en cuenta el punto por el que empezamos a dibujar el triángulo.



Si α es el ángulo con el que llegamos al punto origen del triángulo de la izquierda, el ángulo se modifica así:

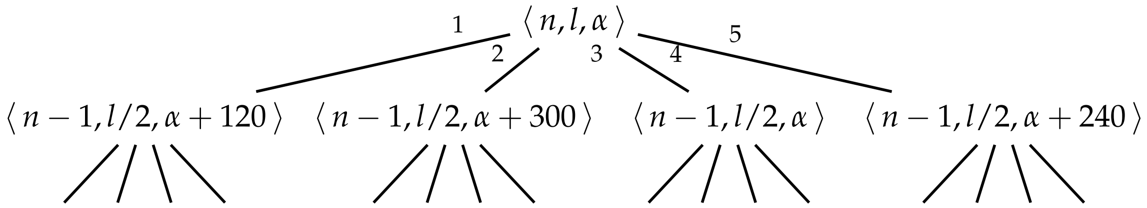
$$\alpha \longrightarrow \begin{cases} \alpha + 120 & \text{en el triángulo 1} \\ \alpha + 300 & \text{en el triángulo 2} \\ \alpha & \text{en el triángulo 3} \\ \alpha + 240 & \text{en el triángulo 4} \end{cases}$$

Algoritmo recursivo

proc `dibuja-interior`$(n : \mathit{nat}, l, \alpha : \mathit{real})$
  casos
    $n = 0 \rightarrow$ { nada }
  □ $n > 0 \rightarrow$
      `dib-sep1`(n, l, α)
      `dibuja-interior`$(n-1, l/2, \alpha + 120)$
      `dib-sep2`(n, l, α) { vacío }
      `dibuja-interior`$(n-1, l/2, \alpha + 300)$
      `dib-sep3`(n, l, α)
      `dibuja-interior`$(n-1, l/2, \alpha)$
      `dib-sep4`(n, l, α)
      `dibuja-interior`$(n-1, l/2, \alpha + 240)$
      `dib-sep5`(n, l, α)
  fcasos
fproc

Transformación a iterativo

Planteamos el árbol de activaciones. Ya que hay cuatro llamadas el árbol es de grado 4.



Se trata de un recorrido no estándar ya que la raíz se visita (se dibuja un separador) antes, después y entre cada visita a cada uno de los hijos.

Las funciones sucesor tienen inversa:

$$
\begin{array}{lcl}
\texttt{suc1}^{-1}(\langle n, l, \alpha \rangle) & = & \langle n+1, l*2, \alpha-120 \rangle \\
\texttt{suc2}^{-1}(\langle n, l, \alpha \rangle) & = & \langle n+1, l*2, \alpha-300 \rangle \\
\texttt{suc3}^{-1}(\langle n, l, \alpha \rangle) & = & \langle n+1, l*2, \alpha \rangle \\
\texttt{suc4}^{-1}(\langle n, l, \alpha \rangle) & = & \langle n+1, l*2, \alpha-240 \rangle
\end{array}
$$

Cada vez que se vuelve de un hijo a la raíz hay que dibujar un separador, pero el separador depende del número de visita por el que vayamos, por lo que vamos a añadir a cada nodo un parámetro más que indique el estado e de la visita, e puede variar de 1 a 5.

Además para poder pasar de un hijo al padre, y aprovechando que las funciones sucesor tienen inversa, tenemos que saber en qué número de hijo nos escontramos. Para ello vamos a numerar los nodos del árbol, por lo que añadimos un parámetro más.

El primer nodo es la raíz en el estado 1 $\langle N, L, 0, 1, 1 \rangle$, suponiendo que N es el nivel de anidamiento que queremos alcanzar y L el tamaño del segmento.

El último nodo es también la raíz, pero en el estado 5 $\langle N, L, 0, 5, 1 \rangle$.

Para calcular el sucesor de $\langle n, l, \alpha, e, m \rangle$ necesitamos distinguir el estado e en el que nos encontramos y si hay hijos, por lo que también hay que distinguir por n.

proc `sucesor`$(n : \mathit{nat}, l, \alpha : \mathit{real}, e, m : \mathit{nat})$
  casos

$$
\begin{array}{ll}
& n > 0 \ \wedge \ e = 1 \ \rightarrow \ \langle n, l, \alpha, e, m \rangle := \langle n-1, l/2, \alpha + 120, 1, 4 * m \rangle \\
\square & n > 0 \ \wedge \ e = 2 \ \rightarrow \ \langle n, l, \alpha, e, m \rangle := \langle n-1, l/2, \alpha + 300, 1, 4 * m + 1 \rangle \\
\square & n > 0 \ \wedge \ e = 3 \ \rightarrow \ \langle n, l, \alpha, e, m \rangle := \langle n-1, l/2, \alpha, 1, 4 * m + 2 \rangle \\
\square & n > 0 \ \wedge \ e = 4 \ \rightarrow \ \langle n, l, \alpha, e, m \rangle := \langle n-1, l/2, \alpha + 240, 1, 4 * m + 3 \rangle \\
\square & n = 0 \ \vee \ e = 5 \ \rightarrow \ \{ \text{ hay que remontar (solo un paso) } \}
\end{array}
$$

    casos

$$
\begin{array}{ll}
& m \bmod 4 = 0 \ \rightarrow \ \langle n, l, \alpha, e, m \rangle := \langle n+1, l * 2, \alpha - 120, 2, m \operatorname{div} 4 \rangle \\
\square & m \bmod 4 = 1 \ \rightarrow \ \langle n, l, \alpha, e, m \rangle := \langle n+1, l * 2, \alpha - 300, 3, m \operatorname{div} 4 \rangle \\
\square & m \bmod 4 = 2 \ \rightarrow \ \langle n, l, \alpha, e, m \rangle := \langle n+1, l * 2, \alpha, 4, m \operatorname{div} 4 \rangle \\
\square & m \bmod 4 = 3 \ \rightarrow \ \langle n, l, \alpha, e, m \rangle := \langle n+1, l * 2, \alpha - 240, 5, m \operatorname{div} 4 \rangle
\end{array}
$$

    fcasos
  fcasos
fproc

Para definir visitar hay que distinguir entre el caso básico y el caso recursivo, y este según el estado:

proc `visitar`$(n : \mathit{nat}, l, \alpha : \mathit{real}, e, m : \mathit{nat})$
  casos

$$
\begin{array}{lll}
& n > 0 \ \wedge \ e = 1 \ \rightarrow \ \texttt{dib-sep1}(n, l, \alpha) & \\
\square & n > 0 \ \wedge \ e = 2 \ \rightarrow \ \texttt{dib-sep2}(n, l, \alpha) & \{ \text{ no hace nada } \} \\
\square & n > 0 \ \wedge \ e = 3 \ \rightarrow \ \texttt{dib-sep3}(n, l, \alpha) & \\
\square & n > 0 \ \wedge \ e = 4 \ \rightarrow \ \texttt{dib-sep4}(n, l, \alpha) & \\
\square & n > 0 \ \wedge \ e = 5 \ \rightarrow \ \texttt{dib-sep5}(n, l, \alpha) & \\
\square & n = 0 \ \rightarrow \ \{ \text{ nada } \} &
\end{array}
$$

  fcasos
fproc

El algoritmo iterativo completo es el siguiente:

$\{ N > 0 \}$
proc `dibuja-interior-it`$(N : \textit{nat}, L : \textit{real})$
    $\langle n, l, \alpha, e, m \rangle := \langle N, L, 0, 1, 1 \rangle$
    `dib-sep1`(n, l, α)
    mientras $\neg(e = 5 \wedge m = 1)$ **entonces**
        `sucesor`(n, l, α, e, m)
        `visitar`(n, l, α, e, m)
    fmientras
fproc