novática
UPGRADE Revista de la Asociación de Técnicos de Informática

*Novática* , revista fundada en 1975 y decana de la prensa informática española, es el órgano oficial de expresión y formación continua de ATI (Asociación de Técnicos de Informática). *Novática* edita también *Upgrade* , revista digital de CEPIS (*Council of European Professional Informatics Societies*), en lengua inglesa.

**<http://www.ati.es/novatica/>**
**<http://www.upgrade-cepis.org/>**

ATI es miembro fundador de CEPIS (*Council of European Professional Informatics Societies*) y tiene un acuerdo de colaboración con ACM (*Association for Computing Machinery*). Tiene asimismo acuerdos de vinculación o colaboración con AdaSpain , AI2 y ASTIC.



**Coordinación Editorial, Redacción Central y Redacción ATI Madrid**
Padilla 66, 3º, dcha., 28006 Madrid
Tlfn.914029391; fax.913093685 <novatica@ati.es>
**Composición, Edición y Redacción ATI Valencia**
Av. del Reino de Valencia 23, 46005 Valencia
Tlfn./fax 963330392 <secreval@ati.es>
**Administración y Redacción ATI Cataluña**
Via Laietana 41, 1º, 1ª, 08003 Barcelona
Tlfn.934125235; fax 934127713 <secregen@ati.es>
**Redacción ATI Andalucía**
Isaac Newton, s/n, Ed. Sadiel,
Isla Cartuja 41092 Sevilla, Tlfn./fax 954460779 <secreand@ati.es>
**Redacción ATI Aragón**
Lagasca 9, 3-B, 50006 Zaragoza.
Tlfn./fax 976235181 <secreara@ati.es>
**Redacción ATI Asturias-Cantabria** <gp-astucant@ati.es>
**Redacción ATI Castilla-La Mancha** <gp-clmancha@ati.es>
**Redacción ATI Galicia**
Recinto Ferial s/n, 36540 Silleda (Pontevedra)
Tlfn.986581413; fax 986580162 <secregal@ati.es>
**Suscripción y Ventas**
<http://www.ati.es/novatica/interes.html>,o en ATI Cataluña o ATI Madrid
**Publicidad**
Padilla 66, 3º, dcha., 28006 Madrid
Tlnf.914029391; fax.913093685 <novatica.publicidad@ati.es>
**Imprenta**
9 Impressió S.A., Juan de Austria 66, 08005 Barcelona.
**Depósito legal:** B 15.154-1975 -- ISSN: 0211-2124; CODEN NOVAEC
**Portada:** Antonio Crespo Foix / © ATI 2004
**Diseño:** Fernando Agresta / © ATI 2004

Nº 167, enero-febrero 2004, año XXX

# sumario



**Monografía del próximo número: "UML (*Unified Modeling Language*)"**

# Reconstrucción de árboles inclinados a partir de dos de sus recorridos

Cristóbal Pareja Flores[1], Ángel Herranz Nieva[2]
*[1]Depto. Lenguajes y Sistemas Informáticos, Universidad Complutense de Madrid; [2] Facultad de Informática, Universidad Politécnica de Madrid*

<cpareja@sip.ucm.es>,<aherranz@fi.upm.es>

Solución del problema B de los planteados en el I Concurso Universitario de la Comunidad Autónoma de Madrid (CUPCAM 2003). El enunciado de este problema apareció en el número 166 de Novática (noviembre-diciembre 2003, p. 71).

## 1. Resumen del problema

El problema, brevemente, pedía encontrar el número de árboles binarios desviados a la izquierda cuyos recorridos por niveles y en 'inorden' eran dados como datos de entrada. En el problema se decía que "un árbol binario está desviado a la izquierda cuando todos sus niveles están alineados a la izquierda".

## 2. Fuerza bruta

La primera opción en la que se puede pensar es en la del uso de la tan socorrida fuerza bruta: generar árboles binarios, más o menos indiscriminadamente, con n nodos (donde n es la longitud de los recorridos) que respondan a uno de los recorridos, para comprobar luego si encajan en el otro recorrido. A priori, la ineficiencia estaría garantizada: sin nada de inteligencia en la generación de árboles, el algoritmo tendría una complejidad de orden factorial (! ).

Dada la simplicidad del algoritmo creemos que merece la pena mostrar su esquema. El lenguaje de programación utilizado es Haskell 98 [1][2]:

```
> desdeInorden : String -> [ ArbolChar]
>
> porNiveles : ArbolChar -> String
>
> estaDesviado : ArbolChar -> Bool
>
> solucion : (String, String)
>            -> [ ArbolChar]
> solucion (niv,inord) =
>   (filter ((niv==) . porNiveles)
>    . filter (estaDesviado)
>    . desdeInorden) inord
```

donde las operaciones desdeInorden, porNiveles y estaDesviado son relativamente sencillas de implementar y el tipo ArbolChar se define en la sección 4.

## 3. Inteligencia

Un análisis más cauto revela propiedades del problema que permitirían realizar búsquedas en un dominio mucho más pequeño.

### 3.1 . Sufijos viables en el recorrido por niveles

Empezamos observando lo siguiente:

En el último nivel del árbol solución, el carácter inicial es también el primero del recorrido en 'inorden' dado; y ése es el único principio posible de la última fila.

Buscando ese carácter en el recorrido por niveles, encontraremos distintas posibilidades (o posiblemente ninguna). Cada una de ellas da lugar a un "proyecto de solución" que consta de una lista de árboles (por ahora, triviales: árboles hoja), que son los del último nivel: $[t_1, t_2, ..., t_k]$. Por ejemplo, el par de recorridos (niveles, 'inorden') siguiente

```
("121121121", "112211211")
```

indica que el último nivel empieza con un "1" (primer carácter del 'inorden'). Así que el último nivel está entre los siguientes,

```
{ "1",  "121",  "1121",  "121121",
"1121121", "121121121"}
```

que son los sufijos del recorrido por niveles que empiezan por un 1. Aparentemente, tenemos seis proyectos de solución prometedores.

### 3.2. Límites en la construcción de árboles

Pero ahora podemos hacer otra observación. Una lista de k árboles nivelados no puede unirse con menos de k-1 nodos intermedios y, de ellos, al menos la mitad han de estar en el primer nivel nuevo.

En nuestro ejemplo, si el último nivel fuera "121121", debería completarse con tres caracteres ("121") lo que es materialmente imposible. Esto limita los seis segmentos candidatos del recorrido por niveles según su longitud. En nuestro caso concreto, nos quedamos con los tres primeros candidatos:

```
{"1", "121", "1121"}
```

La figura 1 muestra una búsqueda completa de árboles válidos para los recorridos estudiados como ejemplo aplicando el algoritmo esquematizado en esta sección e implementado en la siguiente.

## 4. Implementación

Representaremos los datos de entrada, los recorridos por niveles y en 'inorden', como pares de cadenas de caracteres:

```
> type Recorridos = (String, String)
```

Para representar los árboles nos apoyaremos en el siguiente tipo algebraico:

```
> data Arbol a =
>         Vacio
>       | Nodo (Arbol a) a (Arbol a)
```

Como la información de los nodos serán caracteres:

```
> type ArbolChar = Arbol Char
```

Por otra parte, con cada proyecto de solución $[t_1, t_2, ..., t_k]$, el primer carácter c del nivel nuevo ha de ser tal que, junto con los dos primeros árboles y de la lista, ha de dar lugar a un árbol cuyo recorrido en 'inorden'

```
ordenCentral (Nodo  c )
```

sea prefijo del 'inorden' global dado, donde ordenCentral se puede definir de esta forma:

```
> ordenCentral :: ArbolChar -> String
> ordenCentral Vacio =
>    ""
> ordenCentral (Nodo t1 c t2) =
>    ordenCentral  t1
>    ++ c : ordenCentral t2
```

Siguiendo con nuestro ejemplo, la elección de "1121" como último nivel daría lugar a un recorrido en 'inorden' de la forma "1?1...", lo que no encaja con el 'inorden' dado.

Esta segunda restricción nos permite filtrar aún más los proyectos de solución.

Estas dos ideas determinan los posibles pares de segmentos en el recorrido por niveles: el del siguiente nivel y el resto, para los niveles posteriores:

```
> segmentos ::
>    ([ ArbolChar] , Recorridos)
>    -> [ (Recorridos)]
> segmentos (arbs, niv, inord) =
>    let l = length arbs
>        n = length niv
>    in [  (izda, dcha)
>       | k <- [ div (l+1) 2
>                .. div (n+1) 2],
>         let dcha = drop (n-k) niv,
>         esViable
>           (prefijo arbs $ head dcha)
>           inord,
>         let izda = take (n-k) niv
>       ]
```

donde las functiones prefijo y esViable se han programado en el siguiente código:

```
> prefijo :: [ ArbolChar]  -> Char
>             -> ArbolChar
> prefijo [ t1]  c = Nodo t1 c Vacio
> prefijo (t1:t2:_) c = Nodo t1 c t2

> esViable :: ArbolChar -> String
>              -> Bool
> esViable arbol cadena =
>    and $ zipWith (==)
```

> El problema pedía encontrar el número de árboles binarios *desviados a la izquierda*

```
>              (ordenCentral arbol)
>              cadena
```

Ahora, las maneras de completar un nivel vienen dadas por los posibles caracteres viables según la función segmentos, ya descrita:

```
> completar ::
>    ([ArbolChar], Recorridos)
>   -> [ ([ArbolChar], Recorridos)]
> -  PRE: las cadenas de caracteres
> -         son no vacías
> completar (arbs, niv, inord) =
>   let pares = segmentos (arbs,
>                          niv,
>                          inord)
>   in [  (ponerFila arbs dcha,
>          izda,
>          inord)
>       |  (izda,  dcha) <- pares ]
```

donde ponerFila empareja los demás árboles, pudiendo quedar uno suelto y, también, sobresalir en ese nivel, más árboles triviales, hasta completar el substring que estamos considerando:

```
> ponerFila :: [ArbolChar] -> String
>                -> [ArbolChar]
> ponerFila [] resto =
>   [  Nodo Vacio c Vacio
>    | c <- resto ]
> ponerFila [a]  (r:resto) =
>   Nodo a r Vacio
>       : ponerFila [] resto
> ponerFila (a:b:bs) (r:resto) =
>   Nodo a r b : ponerFila bs resto
```

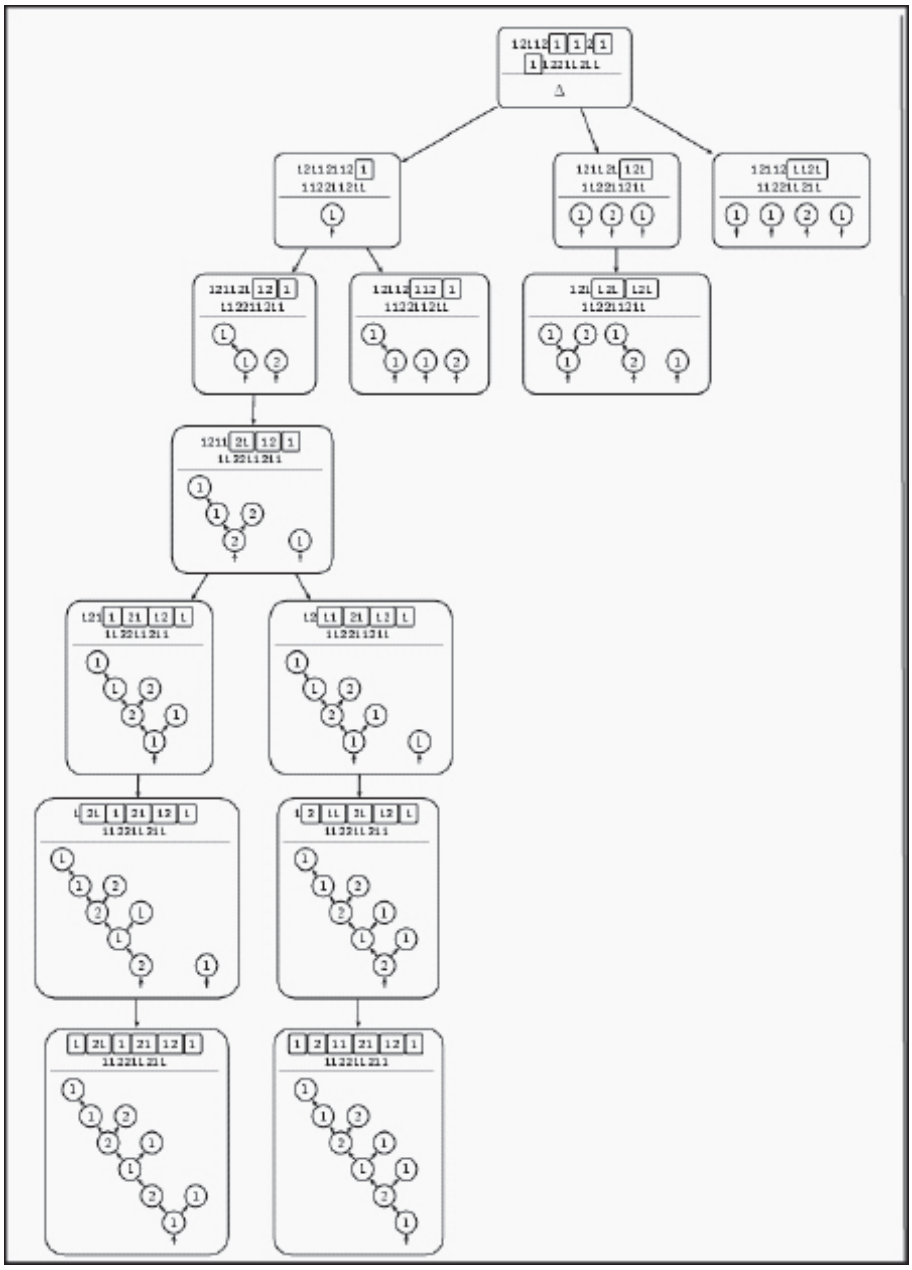

Figura 1. Búsqueda completa de árboles válidos para los recorridos.

## Notas

**[1] P. Hudak, J. Fasel, J. Peterson.** *A gentle introduction to Haskell*. Informe Técnico YALEU/DCS/RR-90, Yale University, 1996.

**[2] S. Peyton Jones, J. Hughes.** *Report on the Programming Language Haskell 98. A Non-strict Purely Functional Language*, Febrero, 1999.