Trees

Jan van Eijck

CWI, Amsterdam and Uil-OTS, Utrecht

jve@cwi.nl

June 2, 2009

Abstract

We give formal definitions of (syntax) trees, and define some important tree relations.

Module Declaration

```
module Trees

where
import List
import Char
```

Trees

Trees can be defined in a number of ways.

The simplest definition is as a generalization of lists. A lists of things of type a is either the empty list, or a thing of type a put in front of a list of type a.

Similarly for trees, with labels of type a and leaf information of type b: such a tree either is a leaf with a label and some info, or a branching node dominating a list of daughter trees.

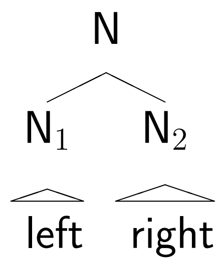
Special cases are binary trees (with label and leaf information) and binary trees with information just at the leaf nodes (without label information).

We start with the simplest case: binary trees without label information.

Binary Trees Without Node Information

Binary trees are trees where every node either is a leaf, or a node dominating a left and a right binary tree.

The top node of a (binary) tree is called the root. In a tree of the form



the nodes N_1 and N_2 are called the daughters of the root node.

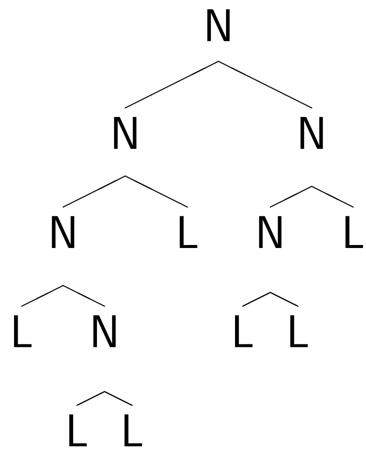
Here is the definition of the Haskell data type for this. This is our first example of a self-defined data type. The Haskell keyword for this is `data`.

```
data BinTree = L
             | N BinTree BinTree
     deriving Show
```

The `deriving Show` part of the definition is to ensure that we can display trees on the screen.

```
example1 :: BinTree
example1 = (N (N (N L (N L L)) L) (N (N L L) L))
```

More familiar representation:



Counting Nodes

Define a function `count :: BinTree -> Int` for counting the number of nodes of a binary tree.

Solution

```
count :: BinTree -> Int
count L = 1
count (N t1 t2) = 1 + count t1 + count t2
```

This gives:

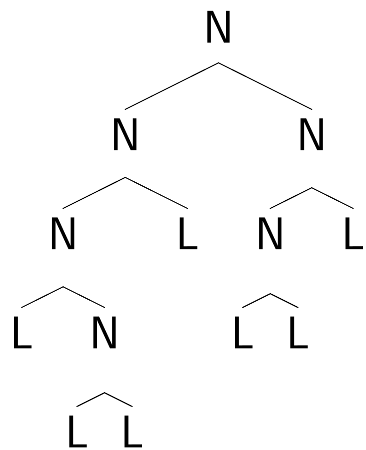
```
Trees> count example1
13
```

Depth of a Tree

A path in a tree is a list of tree nodes such that each node in the list is followed by a daughter node.

The depth of the tree is the length of its longest path.

What is the depth of



Define a function `depth :: BinTree -> Int` that computes the depth of a tree.

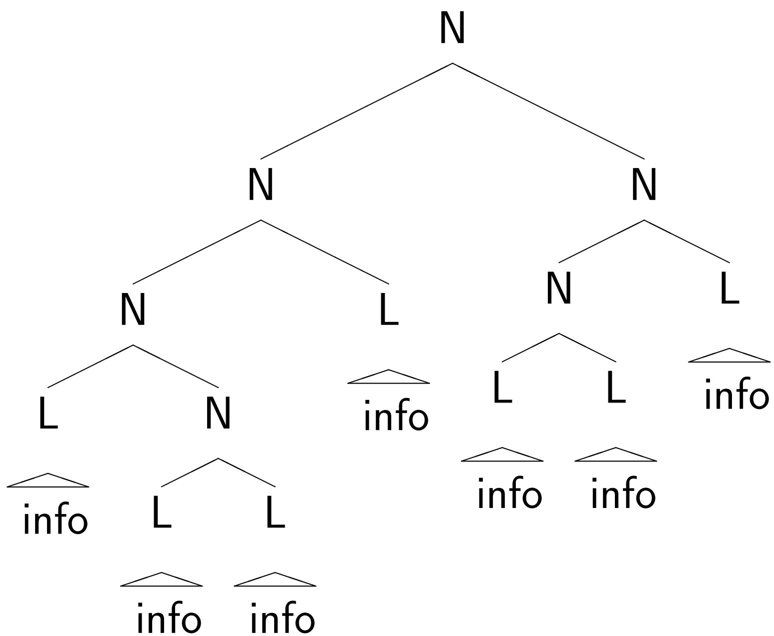
Solution

```
depth :: BinTree -> Int
depth L = 0
depth (N t1 t2) = 1 + max (depth t1) (depth t2)
```

```
Trees> depth example1
4
```

Binary Leaf Trees

Binary leaf trees are trees where every node either is a leaf with some leaf information attached, or a node dominating a left and a right binary leaf tree.

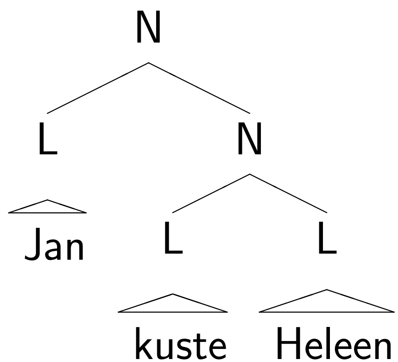

N
N
N
N
L
N
L
L
L
L
L
L
info
info
info
info
info
info
info

In the Haskell definition, the type of the leaf info is indicated by a type variable a:

```
data Bleaftree a
  = Leaf a
  | Branch (Bleaftree a) (Bleaftree a)
  deriving (Eq,Ord,Show)
```

The `deriving (Eq,Ord,Show)` expresses that equality, ordering and show properties of the type a carry over to binary leaf trees with a information.

```
example2 = Branch
             (Leaf "Jan")
             (Branch (Leaf "kuste")
                     (Leaf "Heleen"))
```



Node Counting, Depth

```
count2 :: Bleaftree a -> Int
count2 (Leaf _) = 1
count2 (Branch t1 t2) = 1 + count2 t1 + count2 t2
```

```
depth2 :: Bleaftree a -> Int
depth2 (Leaf _) = 0
depth2 (Branch t1 t2) =
     1 + max (depth2 t1) (depth2 t2)
```

The _ indicates that the value of the variable does not matters. Such _ variables are called anonymous.

Collecting the Leaf Information

```
collect :: Bleaftree a -> [a]
collect (Leaf x) = [x]
collect (Branch t1 t2) = collect t1 ++ collect t2
```

```
Trees> collect example2
["Jan","kuste","Heleen"]
```

A Mapping Function for Trees

The type of the mapping function for lists is

`map :: (a -> b) -> [a] -> [b]`.

A mapping function for leaf trees would have to change the leaf information in the same way as the list map changes the information at the elements of a list.

What is the type, what is the definition?

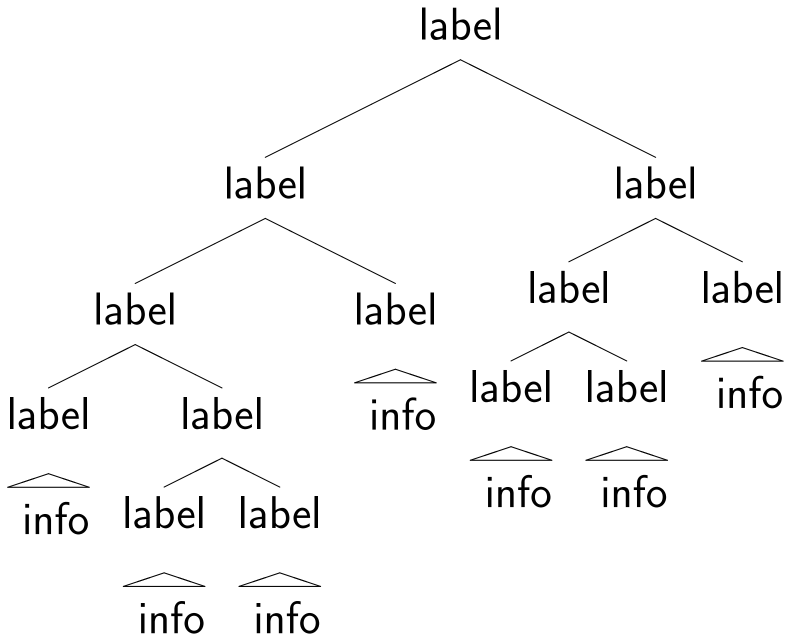
Solution

```
mapT :: (a -> b) -> Bleaftree a -> Bleaftree b
mapT f (Leaf x) = Leaf (f x)
mapT f (Branch t1 t2) =
   Branch (mapT f t1) (mapT f t2)
```

```
Trees> mapT (map toLower) example2
Branch (Leaf "jan") (Branch (Leaf "kuste") (Leaf "heleen"))
```

Binary Trees with Labels

Now let us add label information at the internal nodes. We will not assume that the labels are of the same types as the leaf decorations.


label
label
label
label
label
label
label
label
label
label
label
label
label
info
info
info
info
info
info
info
info

```
data Btree a b = Lf a b
               | Br a (Btree a b) (Btree a b)
  deriving (Eq,Ord,Show)
```

Example Binary Leaf Tree with Labels

```
example3 = Br "S"
             (Lf "NP" "Jan")
             (Br "VP" (Lf "V" "kuste")
                      (Lf "NP" "Heleen"))
```

Write your own versions of `count`, `depth`, `collect` for this.

Now there are two possible tree maps: one for mapping the label info, and one for mapping the leaf info.

What are the types?

Write these tree map functions.

Rose Trees (Trees with Arbitrary Branching)

```
data Rose a = Bud a
            | RBr [Rose a]
     deriving (Eq,Ord,Show)
```

Example Rose Tree

```
rose = RBr [Bud 1,
            RBr [Bud 2,
                 Bud 3,
                 RBr [Bud 4, Bud 5, Bud 6]]]
```

To Do: Positions in a Rose Tree

Abstract Syntax: Grammars as Tree Definitions

A context free grammar corresponds to a tree definition, as follows.

Consider the following example grammar for palindromes over $\{a, b\}$:

$$S \longrightarrow \epsilon \mid a \mid b \mid aSa \mid bSb.$$

This corresponds to the following data type:

```
data Pal = Empty | A | B | PA Pal | PB Pal

instance Show Pal where
 show Empty = ""
 show A     = "a"
 show B     = "b"
 show (PA pal) = "a" ++ show pal ++ "a"
 show (PB pal) = "b" ++ show pal ++ "b"
```

```
Trees> show (PA (PA (PB A)))
"aababaa"
Trees> show (PA (PA (PB (PB (PB Empty)))))
"aabbbbbbaa"
```