A simple interactive tree editor

Mark P. Jones

September 19, 1996

1 Forests and trees

We use standard datatypes to represent forests of so-called general trees:

```
type Forest a  =  [Node a]
data Node a    =  Node a (Forest a)
```

From these definitions, we can see that a forest is a list of nodes, each of which has a value and some children. Here is a simple example:

```
myForest  ::  Forest String
myForest  =  [Node "1"
                [Node "1.1"
                   [Node "1.1.1" []],
                 Node "1.2" []],
              Node "2" []]
```

Next we define two simple, general purpose operations for working with forests. First, *forestElems*, which enumerates the values in a forest in depth-first order:

```
forestElems  ::  Forest a → [a]
forestElems  =  concat . map nodeElems
  where nodeElems (Node x cs)  =  x : forestElems cs
```

The second function is *depthMap*, which traverses a forest and creates a new one of the same shape by applying a function that takes an extra parameter

supplying the depth of the tree at that point:

```
depthMap        ::  (Int → a → b) → Int → Forest a → Forest b
depthMap f d    =   map depthNode
  where depthNode (Node x cs)
                =   Node (f d x) (depthMap f (d + 1) cs)
```

These functions can be used to help display the structure of a forest:

```
showForest   ::  Forest String → String
showForest   =   unlines . forestElems . depthMap indent 1
  where indent d x  =  replicate (2 ∗ d) '\SP' ++ x
```

2 Navigation

For the purposes of navigation, we need to have a way of describing positions within a forest. For any position, we need to capture:

- The nodes to the left of the current position (which we will keep in a list with rightmost element first, that is, in reverse order).
- The nodes to the right of the current position, also in a list.
- A sequence of levels up the tree, from the current position to the root. We need to know the position within each level, which we represent by a triple (left, x, right) where left and right are the siblings on either side, and x is the value of the dominating node.

This leads naturally to the following datatype definition:

```
data Position a  =  Pos {left :: [Node a],
                         up :: [Level a],
                         right :: [Node a]}
type Level a     =  ([Node a], a, [Node a])
```

It is easy to convert between forests and positions (although position information is lost, because there can be many different positions within a given forest):

```
rootPosition                 ::  Forest a → Position a
rootPosition f               =   Pos [] [] f

reconstruct                  ::  Position a → Forest a
reconstruct (Pos ls us rs)   =   foldl recon (reverse ls ++ rs) us
  where recon fs (ls, x, rs) =     reverse ls ++ [Node x fs] ++ rs
```

The following function finds the value (if any) associated with the node on the immediate right of current position:

```
rightValue                       ::  Position a → Maybe a
rightValue (Pos _ _ (Node x _ : _)) =  Just x
rightValue _                     =   Nothing
```

There are four functions for moving around in a forest, either to the left, to the right, up, or down. In the last case, there are two possibilities: down the tree on the immediate left of the current position, or down the tree on the immediate right. For simplicity, we will only consider the latter. All of these functions could fail if the requested move is not possible, so the resulting position is returned in a *Maybe* type, as shown in Figure 1.

Each of these functions works by inspecting a list and taking some action if it is non-empty – which signals that a move is possible. We capture this general pattern in the following repositioning function:

```
repos           ::  [b] → (b → [b] → Position a) → Maybe (Position a)
repos [] f      =   Nothing
repos (x : xs) f =  Just (f x xs)
```

We will also want simple methods for inserting and deleting nodes to the right of the current position (we won't bother with the obvious duals for insertion or deletion on the left).

```
insertNode                  ::  a → Position a → Position a
insertNode x (Pos ls us rs) =   Pos ls us (Node x [] : rs)

deleteNode                  ::  Position a → Maybe (Position a)
deleteNode (Pos ls us rs)   =   repos rs (\_ ns → Pos ls us ns)
```

```
moveUp, moveDown, moveLeft, moveRight
    :: Position a → Maybe (Position a)

moveLeft (Pos ls us rs)
    = repos ls (\n ns → Pos ns us (n : rs))

moveRight (Pos ls us rs)
    = repos rs (\n ns → Pos (n : ls) us ns)

moveDown (Pos ls us rs)
    = repos rs (\(Node x cs) ns → Pos [] ((ls, x, ns) : us) cs)

moveUp (Pos ls us rs)
    = repos us (\(as, x, bs) vs → Pos as vs (make x : bs))
      where make x = Node x (reverse ls ++ rs)
```

Figure 1: Movement functions

As a mildly amusing little extension, we can define a *reflect* operator:

```
reflect                 :: Position a → Position a
reflect (Pos ls us rs)  =  Pos rs us ls
```

This could have been used to define *moveLeft* in terms of *moveRight* (or vice versa).

3 User interface

In this section, we present an interactive program that uses the functions described above to implement an interactive tree editor. This program allows a user to navigate around a forest (entering a child node (e), moving to the next child (n), moving back a child (b), or moving up to the parent(p)), and provides commands to insert a new node (i), to delete an existing node (d), and to show the current forest (s). The output produced by the show command includes a marker to indicate the current position in the forest.

The main interactive loop is defined by the code in Figure 2.

```
loop     ::  Pos → IO ()
loop p   =   do ch ← getChar
                putChar '\n'
                case ch of
                     -- whitepace
                  '\n'  →  loop p
                  '\t'  →  loop p
                  ' '   →  loop p

                     -- basic movement
                  'c'   →  tryTo p moveDown noNode loop
                  'n'   →  tryTo p moveRight noNode loop
                  'b'   →  tryTo p moveLeft noPrev loop
                  'p'   →  tryTo p moveUp noPar loop

                     -- delete and insert
                  'd'   →  tryTo p deleteNode noNode loop

                  'i'   →  do putStrLn "Enter new key: "
                              key ← getLine
                              loop (insertNode key p)

                     -- display commands
                  'k'   →  tryTo p rightValue noNode $ \ x →
                           putStrLn x >> loop p

                  's'   →  do putStr
                                 (showForest
                                    (reconstruct
                                       (insertNode "<*>" p)))
                              loop p


                     -- a reflection
                  'r'   →  loop (reflect p)

                     -- quit command
                  'q'   →  return ()

                  _     →  do putStrLn "Error: bad command"
                              loop p
```

Figure 2: User Interface

This definition has been simplified by abstracting out a common pattern for dealing with the results of Maybe types:

```
tryTo           ::  Pos → (Pos → Maybe a) → String → (a → IO ()) → IO ()
tryTo p f e c   =   case f p of
                      Just x    →   c x
                      Nothing   →   do putStrLn e
                                       loop p
```

For convenience, we use the following type synonym to describe the particular instance of the *Position* type that is used in this program:

```
type Pos   =   Position String
```

We have also abstracted out the strings produced by some of the error messages:

```
noNode   =   "Error: not at node"
noPrev   =   "Error: no previous sibling"
noPar    =   "Error: node has no parent"
```

A simple program that starts up the tree editor at the root of myForest can now be defined as follows:

```
main   =   loop (rootPosition myForest)
```