COMP90048 Declarative Programming
Semester 1, 2018
Peter J. Stuckey

QUESTION 1

Write a Haskell implementation of the old Animals guessing game. Start by prompting "Think of an animal. Hit return when ready." Wait for the user to hit return, then ask: "Is it a penguin?" and wait for a Y or N answer. If the answer is yes, print out that you guessed it with 0 questions. If no, then ask them what their animal was, ask them to enter a yes or no question that would distinguish their animal from a penguin, and whether the answer is yes or no for their animal.

Then start over. This time start by asking them the question they just entered, and depending on their answer, and the answer they said to expect for their previous animal, ask them if their (new) animal is their previous animal, or ask if it is a penguin. If that is correct, print out that you guessed it with 1 question. If no, then ask them what their animal was, ask them to enter a yes or no question that would distinguish their animal from the one you just guessed, and whether the answer is yes or no for their animal.

The game proceeds like this indefinitely. You should build up a decision tree with questions at the nodes, and animals at the leaves. For each animal they think of, you traverse the tree from the root asking questions and following the branch selected by their answer until you reach a leaf, then guess the animal at the leaf, and get them to give you a question to extend the tree if you get it wrong.

ANSWER

```
> import Data.Char

> main :: IO ()
> main = do
>       mainloop $ Leaf "penguin"
>       bye

> mainloop :: DTree -> IO ()
> mainloop dtree = do
>    putStrLn ""
>    putStrLn "Think of an animal."
>    putStr "Hit return when you are ready. "
>    getLine
>    dtree' <- animalRound dtree 0
>    again <- yesOrNo "Play again? "
>    if again then mainloop dtree' else return ()

> bye :: IO ()
> bye = putStrLn "Thanks for playing!"

> yesOrNo :: String -> IO Bool
> yesOrNo prompt = do
>    response <- promptedRead  $ prompt ++ " (y/n) "
>    case dropWhile isSpace response of
>       []         -> yesOrNoAgain prompt
>       (char:_) -> case toLower char of
>                      'y' -> return True
>                      'n' -> return False
>                      (_) -> yesOrNoAgain prompt

> yesOrNoAgain :: String -> IO Bool
> yesOrNoAgain prompt = do
>       putStrLn "Please answer yes or no."
>       yesOrNo prompt

> promptedRead :: String -> IO String
> promptedRead prompt = do
>       putStr prompt
>       response <- getLine
>       let response' = dropWhile isSpace response
>       if response == [] then do
>           putStrLn "Please answer the question."
>           promptedRead prompt
>         else return response'

> data DTree = Choice String DTree DTree
>                 | Leaf String

> animalRound :: DTree -> Int -> IO DTree
> animalRound (Leaf animal) depth = do
>       answer <- yesOrNo $ "Is it a " ++ animal ++ "?"
>       if answer then do
>               putStrLn $ "I guessed it with " ++ show depth ++ "
questions."
>           return $ Leaf animal
>         else do
>              animal’ <- promptedRead ″OK, I give up.  What is your
animal? ″
>            putStrLn $ ″Please type a question that would distinguish
a ″
>                   ++ animal’ ++ ″ from a ″ ++ animal ++ ″.″
>           question’ <- promptedRead ″Question: ″
>           answer’ <- yesOrNo $
>                  ″What is the answer to this question for a ″
>                  ++ animal’ ++ ″?″
>           return $ Choice question’
>                 (Leaf $ if answer’ then animal’ else animal)
>                 (Leaf $ if answer’ then animal else animal’)
> animalRound (Choice question yesTree noTree) depth = do
>       answer <- yesOrNo question
>         dtree <- animalRound (if answer then yesTree else noTree)
(depth+1)
>       return $ Choice question
>               (if answer then dtree else yesTree)
>               (if answer then noTree else dtree)
```

Terminal input-output is a complicated business, involving
buffering, line-editing, and echoing, and depends on terminal
settings. The default settings can vary (and need to vary) for
different environments. Without the correct settings you might
find that basic line-editing doesn't work, or that prompts get
out of synch with input, or you might encounter other weird
behavior.

On the eng servers:

This code works as expected if run under runhaskell (via .lhs
symlink or file copy). This is probably the most pragmatic way
to go.

It works as expected in ghci if you enter

```
         import System.IO
         hSetBuffering stdin LineBuffering
```

before invoking main.

To run compiled by ghc (also via .lhs symlink), one solution is to put

```
        import System.IO
```

at top level, and

```
        hSetBuffering stdout NoBuffering
```

at the beginning of main's do-block, before the call to
mainloop.

A better solution is *not* to set NoBuffering for
stdout, leaving stdout as it is (which I think is still
LineBuffering by default), and instead put

```
        hFlush stdout
```

just after

```
        putStr prompt
```

in function promptedRead, and also after the putStr in
mainloop. This keeps buffering mostly intact, and just forces
flushing after partial-line prompts.