Homework 2: Trees / Propositional Logic

CIS 352: Programming Languages

2018-01-26

Administrivia

- Part I of the homework are a few problems on binary and multi-way trees.[1] Part II of the homework builds on the propositional logic package developed in class.
- ***No teams of size larger than two!***
- If you pick up an idea from someone outside of your team or an internet site or a book, *document it* in your coversheet file.
- In your Blackboard submission, include: *(i)* your coversheet, *(ii)* your modified version of `trees.hs`, *(iii)* your modified version of `prop.hs`, *(iv)* the transcripts of test runs.

Grading Criteria

- The homework is out of 100 points.
- Each problem is, roughly, 70% correctness and 30% testing.
- Omitting your name(s) in the source code looses you 5 points.

[1] Mucking about with trees is a large part of this course. Also, these sorts of questions tend to show up in job interviews.

Binary and Multiway Trees

I looked up my family tree and found out I was the sap. — R. Dangerfield

Background

Binary trees For binary trees we use a type definition similar to the one in LYAH:

Chapter 8 of LYAH has an extended example on general binary search trees that you should read over.

```
data BTree = Empty | Branch Char BTree BTree
             deriving (Eq,Show)
```

`Empty` is an empty binary tree and `(Branch c tl tr)` constructs an `BTree` node with label `c`, left subtree `tl`, and right subtree `tr`. Thus, `(Branch c Empty Empty)` is a leaf.

Example: `t1` in below represents the tree in Figure 1.

```
t1 = Branch 'x'
       (Branch 't'
         (Branch 'a' Empty Empty)
         Empty)
       (Branch 'w'
         (Branch 'm' Empty Empty)
         (Branch 'q' Empty Empty))
```

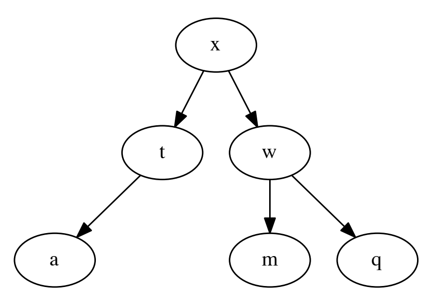

Figure 1: Tree `t1`

To count the number of `Branch`-nodes in a `Tree` we can do something like the following.

```
bcount :: BTree -> Int
bcount Empty           = 0
bcount (Branch _ tl tr) = 1+(bcount tl)+(bcount tr)
```

Multiway trees For multiway trees we use the type definition:

```
data MTree = Node Char [MTree]
             deriving (Eq,Show)
```

(`Node c [`t_1`, `t_2`,..., `t_k`]`) constructs an `MTree` node with label `c` and with subtrees $t_1, t_2, \ldots, t_k$. Thus, `(Node c [])` is a leaf. Note that there are no empty `MTree`s.

Example: `t2` below represents the tree in Figure 2.

```
t2 = Node 'u'
          [Node 'c' [],
           Node 'q' [],
           Node 'n'
                [Node 'm' [],
                 Node 'g' [],
                 Node 'j' []],
           Node 'y'
                [Node 'z' []]]
```

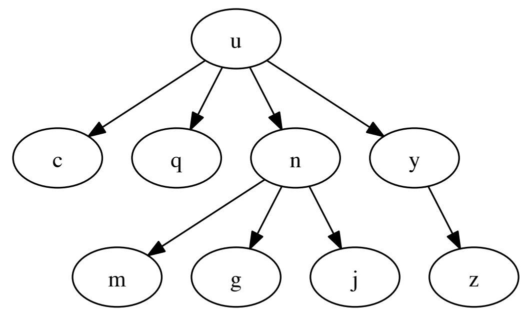

Figure 2: Tree `t2`

To count the number of `Node`'s in an `MTree` we can do something like the following:

```
mcount1 :: MTree -> Int
mcount1 (Node _ ts) = 1 + sumCounts ts

sumCounts :: [MTree] -> Int
sumCounts []     = 0
sumCounts (t:ts) = mcount1 t + sumCounts ts
```

or better yet:[2]

```
mcount2 (Node _ ts) = 1 + sum (map mcount2 ts)
```

[2] Recall:
`(map f [])` $\rightsquigarrow$ []
`(sum [])` $\rightsquigarrow$ 0

or equivalently:

```
mcount3 (Node _ ts) = 1 + sum [mcount3 t | t <- ts]
```

```
   concat :: [[a]] -> [a]
concatMap :: (a -> [b]) -> [a] -> [b]
      map :: (a -> b) -> [a] -> [b]
```

Figure 3: Some handy library functions for Part I

Testing

- For `bmaxDepth`, `mmaxDepth`, `blevel`, and `mlevel` you need to come up with your own tests.
- For `bleaves`, run: (`quickCheck bleaves_prop`).
- For `mleaves`, run: (`quickCheck mleaves_prop`).

These test functions are in the `trees.hs` file.

Problems for Part I

For the problems of this part, add your code to a copy of:
http://www.cis.syr.edu/courses/cis352/code/trees.hs.

> DEFINITION. The *depth* of a node in a (rooted) tree is the length of the path from the root to the node. Thus the root has depth 0. By convention, empty trees have depth -1.

❖ Problem 1 (8 points): Maximum depth of a BTree ❖

Define a function

```
bmaxDepth :: BTree -> Int
```

that, given a `BTree` t returns the maximum depth of any `Branch`-node in t.

Examples. For `t1` as in Figure 1:
`(bmaxDepth Empty)` $\leadsto -1$
`(bmaxDepth (Branch 'x' Empty Empty))` $\leadsto 0$
`(bmaxDepth t1)` $\leadsto 2$

❖ Problem 2 (8 points): Maximum depth of a MTree ❖

Define a function

```
mmaxDepth :: MTree -> Int
```

that, given a `MTree` t returns the maximum depth of any `Node` in t.

Examples. For `t2` is as Figure 2:
`(mmaxDepth (Node 'x' []))` $\leadsto 0$
`(mmaxDepth t2)` $\leadsto 2$

❖ Problem 3 (8 points): Collecting BTree leaves ❖

Define a function

```
bleaves :: BTree -> String
```

such that `(bleaves t)` returns the list of labels of the leaves of `BTree t`.[3]

Examples.
`(bleaves Empty)` $\leadsto$ `""`
`(bleaves t1)` $\leadsto$ `"amq"`

[3] Recall: `String` = `[Char]`.

❖ Problem 4 (8 points): Collecting MTree leaves ❖

Define a function

```
mleaves :: MTree -> String
```

such that `(mleaves t)` returns the list of labels of the leaves of `MTree t`.

Example. `(bleaves t2)` $\leadsto$ `"cqmgjz"`

> DEFINITION. A node of a (rooted) tree is at *level* n when the path from the root to the node has length $n-1$. Thus, the root node is at level 1.

❖ Problem 5 (8 points): BTree levels ❖

Define a function

```
blevel :: Int -> BTree -> String
```

Examples.
`(blevel 0 t1)` $\leadsto$ `""`
`(blevel 1 t1)` $\leadsto$ `"x"`
`(blevel 2 t1)` $\leadsto$ `"tw"`
`(blevel 3 t1)` $\leadsto$ `"amq"`
`(blevel 4 t1)` $\leadsto$ `""`

such that `(blevel k t)` returns the list of all the `Branch`-labels of nodes at level `k` in `BTree t`.

❖ ***Problem 6 (8 points): MTree levels*** ❖

Define a function

```
mlevel :: Int -> MTree -> String
```

such that `(mlevel k t)` returns the list of all the `Node`-labels at level `k` in `MTree t`.

Examples.

```
(mlevel 0 t2) ⤳ ""
(mlevel 1 t2) ⤳ "u"
(mlevel 2 t2) ⤳ "cqny"
(mlevel 3 t2) ⤳ "mgjz"
(mlevel 4 t2) ⤳ ""
```

Part II: More on Propositional Logic

This part builds on the propositional logic example discussed at the end of the slides:[4]

`http://www.cis.syr.edu/courses/cis352/slides/04algTypes.pdf`

For the problems of this part, add your code to a copy of:
`http://www.cis.syr.edu/courses/cis352/code/prop.hs`.

DEFINITION. The *nand* operator (written # in this homework) is equivalent to the negation of a conjunction, i.e., $p \# q \iff \sim(p \& q)$ or see Figure 4.

P	Q	$P \# Q$
T	T	F
T	F	T
F	T	T
F	F	T

Figure 4: The truth table for $P \# Q$

❖ ***Problem 7 (8 points)*** ❖

Extend the `Prop` datatype and associated functions in `prop.hs` to handle the nand connective as follows.

- Find the declaration of the `Prop` datatype in `prop.hs` and extend it with the infix constructor `:#:` for nand.[5]
- Find the printer (`showProp`), evaluator (`eval`), name-extractor (`names`), and subformula-extractor (`subformulas`) functions and extend their definitions to cover, `:#:`, the new nand constructor.[6]

Testing this work. Run `(fullTable (p :#: q))` which should result in the truth-table of Figure 4.

❖ ***Problem 8 (16 points)*** ❖

Background: An important property of nand is that in classical logic[7] all the other propositional connectives can be expressed in terms of it.[8] Thus, every propositional formula can be translated to an equivalent (if long-winded) formula that uses # as its only logical connective. For example:

$$\begin{aligned} \sim P &\equiv (P \# P) \\ P \& (\sim Q) &\equiv ((P \# (Q \# Q)) \# (P \# (Q \# Q))) \\ P \leftrightarrow Q &\equiv (((P \# (Q \# Q)) \# (Q \# (P \# P))) \# ((P \# (Q \# Q)) \# (Q \# (P \# P)))) \end{aligned}$$

[4] Which in turn is based on a problem set by Willem Heijltjes and Phil Wadler.

[5] *Obvious hint:* What you do for `:#:` will be pretty similar to what is done for `:&:`, `:|:`, etc.

[6] See footnote 1. A tiny bit of thought is required for `eval`.

[7] E.g., what we are working in.

[8] See the *Introduction, elimination, and equivalencies* section of `https://en.wikipedia.org/wiki/Sheffer_stroke`. **N.B.** That article uses ↑ in place of # for nand.

Your Problem: Write a function

```
nandify :: Prop -> Prop
```

that *recursively*[9] translates a proposition to an equivalent proposition that uses # as its only logical connective. Make use of the equivalences from https://en.wikipedia.org/wiki/Sheffer_stroke.

[9] E.g., to translate $\sim p$, you first translate p to p' and then translate $\sim p'$ to $p' \# p'$.

Testing `nandify`: Run `(quickCheck nand1_prop)` and `(quickCheck nand2_prop)`.

> DEFINITION. A propositional formula is in *negation normal form* when:
> - & and $\vee$ are the only binary connectives it uses and
> - the only place a $\sim$ appears is before a variable.

Examples of N.N.F. terms:
✔ $(P \vee \sim Q)\&R$
✔ $((P \vee Q)\&((\sim P)\&(\sim Q)))$
Examples of terms that aren't N.N.F.:
✘ $P \to Q$
✘ $\sim (\sim R)$
✘ $\sim (P\&Q)$
✘ $\sim \mathrm{T}$
✘ $\sim \mathrm{F}$
The last two aren't N.N.F. because T and F are constants, not variables.

❖ ***Problem 9 (30 points)*** ❖

(a) (10 points) Write a function

```
isNNF :: Prop -> Bool
```

that tests whether a proposition is in negation normal form.

(b) (18 points) Write a function

```
toNNF :: Prop -> Prop
```

that *recursively* translates a proposition to an equivalent proposition in negation normal form. In doing this translation, the following equivalences will be useful:

$$\begin{array}{rcl@{\qquad}rcl} \sim(P\&Q) & \leftrightarrow & (\sim P) \vee (\sim Q) & \sim(P \vee Q) & \leftrightarrow & (\sim P)\&(\sim Q) \\ P \to Q & \leftrightarrow & (\sim P) \vee Q & P \leftrightarrow Q & \leftrightarrow & (P\&Q) \vee ((\sim P)\&(\sim Q)) \\ \sim(\sim P) & \leftrightarrow & P & P \# Q & \leftrightarrow & \sim(P\&Q) \end{array}$$

Your answer will have *many* cases, including lots of the form

`toNNF (Not` *stuff*`) =` *other-stuff*

Testing `isNNF` and `toNNF`: Run (and pass)
✔ `runTestTT testsNNF`
✔ `quickCheck nnf1_prop`
✔ `quickCheck nnf2_prop`

Debugging note: When quickCheck finds an error and the problem expression is longish, try running quickCheck a few more times until it comes up with a shorter problem expression. Then check out what `toNNF` is doing with expressions of roughly that form.