CSCI 365 Module 2

due Wednesday, 2 February 2022

Specification

- To receive **Level 1** credit for this module:
  - Complete Exercises 1–5, 7, and 9.
  - Your code must adhere to the style guide linked from the course web page.
  - You must complete Quiz 2 for credit, which will test you on your ability to work with recursive algebraic data types in Haskell.
- To receive **Level 2** credit for this module:
  - Complete everything required for Level 1.
  - Complete exercises 6, 8, and 10–11.

Trees

For the purposes of this problem set, a *binary tree* containing values of type `a` is defined as being either

- empty; or
- a node containing a value of type `a` and (recursively) two binary trees, referred to as the “left” and “right” subtrees. See the illustration in Figure 1, and an example binary tree in Figure 2.

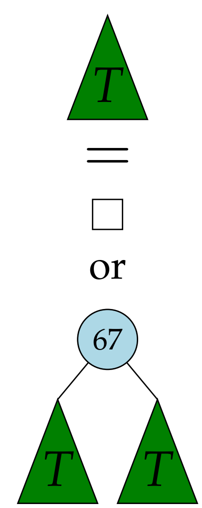

Figure 1: Definition of a binary tree T

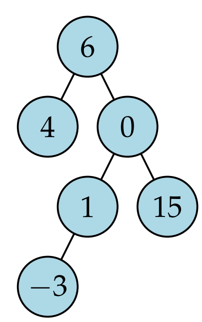

Figure 2: An example binary tree

Exercise 1 Define a recursive, polymorphic algebraic data type `Tree` which corresponds to the above definition.

Exercise 2 Define a function

```
incrementTree :: Tree Integer -> Tree Integer
```

which adds one to every `Integer` contained in a tree.

Exercise 3 Define a function

```
treeSize :: Tree a -> Integer
```

which computes the *size* of a tree, defined as the number of nodes. For example, the tree in Figure 2 has size 6.

A *binary search tree* (BST) is a binary tree of `Integer`s in which the `Integer` value stored in each node is larger than all the `Integer` values in its left subtree, and smaller than all the values in its right subtree. (For the purposes of this problem set, assume that all the values in a binary search tree must be distinct.) For example, the binary tree shown in Figure 2 is not a BST, but the one in Figure 3 is.

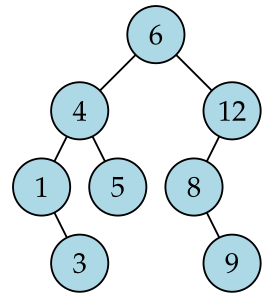

Figure 3: An example binary search tree

The following problems ask you to implement some basic binary search tree algorithms. If you don't remember how they work, you can ask me, or consult a reference such as Cormen et al. [2001, Chapter 13].

Exercise 4 Implement a function

```
bstInsert :: Integer -> Tree Integer -> Tree Integer.
```

Given an integer `i` and a valid BST, `bstInsert` should produce another valid BST which contains `i`. If the input BST already contains `i`, it should be returned unchanged.[1]

[1] It does not matter what `bstInsert` does when given an input `Tree` which is not a valid BST. Later in the course we will talk about ways to use the type system to help enforce invariants such as this.

Exercise 5 Write a function

```
isBST :: Tree Integer -> Bool
```

which checks whether the given `Tree` is a valid BST.

Exercise 6 (Level 2) Ensure that your `isBST` function runs in $O(n)$ time.

Proof trees

Consider the following Haskell definitions, which encode the simple proof system we considered as a first example in class, with only propositional variables. For example, a rule of this system might look like

$$\frac{A \qquad B}{C}.$$

Since everything is a tree, we can easily encode these proof trees as values of an algebraic data type in Haskell.

These definitions are available in `Proof.hs`. If you download `Proof.hs` and put it in the same folder as your `.hs` or `.lhs` file, you can add `import Proof` at the top of your `.hs` file in order to make use of the types it defines.

```
-- Prop is a synonym for String, and represents
-- logical propositions, like A, B, C in the example above.
type Prop = String

-- An inference rule is a list of premises and a conclusion.
-- For example, the rule
--
--     A     B     C
--     -------------
--           D
--
-- would be represented as (R ["A", "B", "C"] "D").
data Rule where
  R :: [Prop] -> Prop -> Rule

-- A rule system is a list of rules.
type System = [Rule]

-- A proof is a tree where each node contains a rule and
-- a list of proofs of the rule's premises.  For example,
-- the proof tree
--
--          ---
--           A
--    ---   ---   ---
--     A     B     C
--     -------------
--           D
--
-- would be represented as
--   (PNode (R ["A", "B", "C"] "D")
--     [ PNode (R [] "A") []
--     , PNode (R ["A"] "B") [PNode (R [] "A") []]
--     , PNode (R [] "C") []
--     ]
--   )
data Proof where
  PNode
    :: Rule      -- ^ The rule used as the bottommost (root) rule in
                 --   the proof tree
    -> [Proof]   -- ^ A list of proofs of each of the premises of the
                 --   rule (in the same order)
    -> Proof
```

Note that the `type` keyword creates a *type synonym*, *i.e.* `Prop` and `String` can now be used completely interchangeably (and similarly for `System` and `[Rule]`).

Exercise 7 Write a function

```
checkProof :: Proof -> Prop -> Bool,
```

which, given a purported proof and a proposition, checks whether the given proof is actually a valid proof of the given proposition. (A proof might not be valid because, *e.g.*, the final conclusion is not the requested proposition, or because some node contains proofs whose conclusions do not match the stated premises of its rule.) You may assume that in a valid proof node, the premises of the rule match up with the given proofs *in order*, that is, the first proof should be a proof of the first premise of the rule, the second proof of the second premise, and so on (this makes your job a bit easier, and is a not unreasonable requirement).

Exercise 8 (Level 2) Write a function

```
findProof :: System -> Prop -> Maybe Proof.
```

Given a rule system and a goal proposition, it should either return a valid proof of the proposition using only rules from the system, or `Nothing` if there is no valid proof.

Propositional logic

Exercise 9 Give formal derivations (proof trees) for each of the following judgments.

You will probably want to draw these by hand and then turn them in on paper. If you are a really hard-core LaTeX user and want to typeset them, try the `mathpartir` package, available from `http://cristal.inria.fr/~remy/latex/mathpartir.sty`, with documentation at `http://cristal.inria.fr/~remy/latex/mathpartir.html`. You might also want to use the `lscape` or `pdflscape` packages to put individual pages in landscape mode, since the proof trees tend to be much wider than they are tall.

(a) $(P \implies (Q \implies R)) \vdash (Q \implies (P \implies R))$

(b) $((P \wedge Q) \implies R) \vdash (P \implies (Q \implies R))$

(c) $((P \vee Q) \implies R) \vdash ((P \implies R) \wedge (Q \implies R))$

Exercise 10 (Level 2) Take each of the above three judgments and replace $\wedge$ by multiplication, $\vee$ by addition, and replace $\implies$ by (backwards) exponentiation, *i.e.* replace $P \implies Q$ by Q^P. What do you notice?

Exercise 11 (Level 2) We did not talk about negation $(\neg P)$ in class, since it turns out that for our purposes, it is possible to encode negation using other logical connectives. In particular, consider defining

$$\neg P :\equiv (P \implies \bot).$$

Using this definition, for each of the following judgments, either give a formal derivation (*i.e.* a proof tree), or explain why it is not possible.

(a) $\vdash P \wedge \neg P \implies \bot$

(b) $\vdash P \vee \neg P$

(c) $\vdash P \implies \neg(\neg P)$

(d) $\vdash \neg(\neg P) \implies P$

(e) $\vdash \neg(P \vee Q) \implies (\neg P \wedge \neg Q)$

References

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to Algorithms*, volume 2. MIT Press, Cambridge, 2001.