Exercises 8: Functors, Parsing, and Yet More Trees

CIS 623

Complete this by class time Monday April 16

Part I: More Tree Problems

❖ ***Problem 1 (Index Binary Search Trees points)*** ❖

Background. We consider binary search trees (BSTs) that have characters as their key values and which each node keeps tract of the number of elements in its *left* subtree. *Example:* See Figure 1(a) below.

For generalizations of this, see `https://www.codementor.io/haskell/tutorial/monoids-fingertrees-implement-abstract-data`.

$t_0 =$

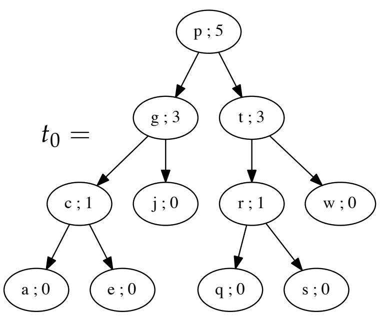

Figure 1(a)

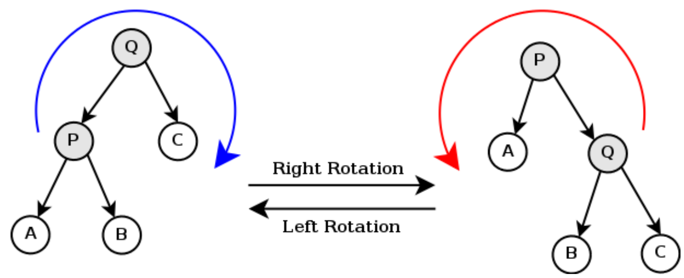

Figure 1(b)

Figure 1: The t_0 tree and an illustration of a tree rotation

The *depth* of a node N in a BST t is the length of the path from t's root node to N, e.g., in the tree of Figure 1, the j-node is of depth 2.

The *index* of a node N in a BST t is the number of nodes to the left of N in t. *Example:* Here are the indices of t_0's elements.

node with character	a	c	e	g	j	p	q	r	s	t	w
the t_0-index of that node	0	1	2	3	4	5	6	7	8	9	10

Table 1: Indices for elements of t_0

Note: If t_1 is the t_0-subtree rooted at the `g`-node then, for each t_1 node N, the t_1-index of N = the t_0-index of N; whereas, if t_2 is the t_0-subtree rooted at the `t`-node, then, for each t_2 node N, the t_2-index of N = (the t_0-index of N) $-\ (5+1)$.

Suppose we use the following data structure to represent these sorts of trees.

```
data BinTree = Empty | Fork BinTree Char BinTree Int
```

Problems.

(a) Write a function

```
add :: Tree -> Char -> Tree
```

such that, if `t` is a BST, then `(add t c)` is the result of adding Char `c` to `t` (with updated left subtree counts). If `c` is an element of `t` to start with, then `(add t c)` just returns `t`. Your function should run in $O(h)$ time, where h is the height of t.

(b) Write a function

```
index :: Tree -> Char -> Maybe Int
```

such that `(index t c)` returns `(Just i)`, if `c` occurs in `t` with index `i`, and returns `Nothing`, if `c` fails to occur in `t`. Your function should run in $O(d)$ time where d is the depth c's node in t.

(c) Write a function

```
fetch :: Tree -> Int -> Maybe Char
```

such that `(fetch t i)` returns `(Just c)` if c is the Char at index `i` in `t`, and returns `Nothing` if there is no character at that index in `t`. Your function should run in $O(d)$ time where d is the depth in the tree of the node with index i.

(d) Write a function

```
reroot :: Tree -> Char -> Tree
```

such that `(reroot t c)` returns the result of altering t to make c's node the root (while updating left tree counts). If `c` is not in `t`, then `(reroot t c)` returns `t`. Your function should run in $O(d)$ time where d is the depth of c's node in t.

Hint: Tree rotations may be helpful. See Figure 1(b) above.

❖ ***Problem 2 (Tries points)*** ❖

BACKGROUND: A *trie* is a tree structure for representing a *lexicon*, i.e., collection of strings. Each node is either gray or black and can have any number of edges leaving it. Each edge is labeled by a character. For any given node, Each edge leaving it is labeled by a different character. A string is in the trie's lexicon when there is a path from the trie's root to a black node and the characters along the path make up the string. For example, the tree in Figure 2 represents the lexicon {"a", "at", "ate", "on", "one", "out", "me", "mud", "my"}. Here are three Haskell type definitions for representing tries together with the representation of the sample trie of Figure 2.

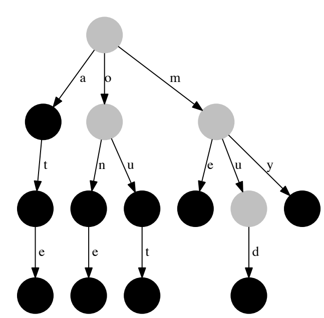

Figure 2: A sample trie

```
type Edge  = (Char,Trie)
data Color = W | B                  deriving (Show,Eq)
data Trie  = Node Color [Edge]      deriving (Show,Eq)
```

```
t0 = Node W
         [('a',Node B [('t',Node B [('e',Node B [])])]),
          ('o',Node W [('n',Node B [('e',Node B [])]),
                      ('u',Node W [('t',Node B [])])]),
          ('m',Node W [('e',Node B []),
                      ('u',Node W [('d',Node B [])]),
                      ('y',Node B [])])]
```

PROBLEMS.

(a) Write a Haskell function

```
search :: String -> Trie -> Bool
```

such that (`search str tr`) tests whether `str` is in the lexicon represented by `tr`. EXAMPLES: Let `t0` be the Trie given above. Then (`search "one" t0`) should return `True` and both (`search "owl" t0`) and (`search "ou" t0`) should return `False`. *(Hint: The built-in function* `lookup` *is handy here.)*

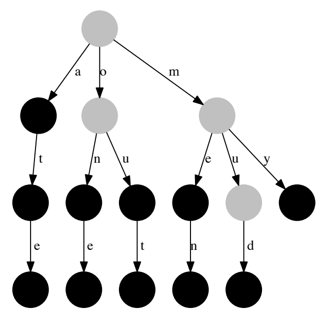

Figure 3: `t1 = (add "men" t0)`

(b) Write a Haskell function

```
add :: String -> Trie -> Trie
```

such that (`add str tr`) returns the new trie that results from adding `str` to the Trie `tr`'s lexicon. If `str` is in `tr`'s lexicon to start with, then the function simply returns `tr`. EXAMPLE: See Figure 3.

(c) Write a Haskell function

```
remove :: String -> Trie -> Trie
```

such that (`remove str tr`) returns the new trie that results from removing `str` from the Trie `tr`'s lexicon. If `str` is not in `tr`'s lexicon to start with, then the function simply returns `tr`. EXAMPLES: (`remove "a" t0`) would turn the leftmost, level-1 node Gray, and, for `t1` of Figure 3, (`remove "men" t1`) would result in `t0`.

(d) Write a Haskell function

```
lexicon :: Trie -> [String]
```

such that (`lexicon tr`) returns the list of all strings in the lexicon that `tr` represents. EXAMPLE: For `t0`, the Trie given above, (`lexicon t0`) should return `["a", "at", "ate", "on", "one", "out", "me", "mud", "my"]` (or some permutation of that list).

Both `map` and `concatMap` can be handy here. Also, a trie's lexicon contains the empty string (i.e., `""`) **if and only if** the trie's root node is black.

Part II: Simple Monadic Programming

◆ ***Problem 3*** ◆
From http://www.seas.upenn.edu/~cis194/fall16/hw/08-functor-applicative.html do Exercises 1, 2, and 4.

◆ ***Problem 4*** ◆
Make a copy of `Parsing.hs`, Hutton's parsing library. Change the type definition

```
newtype Parser a = P (String -> [(a,String)])
```

to

```
newtype Parser a = P (String -> Maybe (a,String))
```

then, to reflect the change of the type `Parser`, revise each of: *(i)* `Parser`'s instance of `Functor`, *(ii)* `Parser`'s instance of `Monad`, *(iii)* `Parser`'s instance of `MonadPlus`, *(iv)* `item`, *(v)* the type-declaration of `parse`, and *(vi)* `eval`. Tese out the revised code to see if things work as expected.

Part III: Simple Parsing Problems

◆ ***Problem 5*** ◆
Use Hutton's parsing library to write a parser for the following variation of the Dyck language[1] given by the context free grammar

[1] See https://en.wikipedia.org/wiki/Dyck_language.

$$S \ ::= \ A\# \qquad A \ ::= \ [A]A \mid \epsilon$$

The parser should return the maximum nesting of brackets in the parsed string. EXAMPLES: Each of: `"#"`, `"[]#"`, `"[][]#"`, and `"[][[][]]#"` is in the language and they have maximum nesting depths of 0, 1, 1, and 2, respectively.

◆ ***Problem 6*** ◆
Consider the context free grammar where $\langle \mathsf{STM} \rangle$ is the start symbol.[2]

[2] Things in pointy-brackets (e.g., $\langle \mathsf{thing} \rangle$) are nonterminals and things in **bold** are terminals.

$$\begin{aligned}
\langle \mathsf{STM} \rangle &::= \langle \mathsf{ACT} \rangle \mid \langle \mathsf{IF} \rangle \\
\langle \mathsf{ACT} \rangle &::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \\
\langle \mathsf{IF} \rangle &::= \mathbf{if}\ \langle \mathsf{TST} \rangle\ \mathbf{then}\ \langle \mathsf{STM} \rangle \mid \mathbf{if}\ \langle \mathsf{TST} \rangle\ \mathbf{then}\ \langle \mathsf{STM} \rangle\ \mathbf{else}\ \langle \mathsf{STM} \rangle \\
\langle \mathsf{TST} \rangle &::= \mathbf{p} \mid \mathbf{q} \mid \mathbf{r}
\end{aligned}$$

Write a parser for this language with the `ReadP` parser library. Do you notice a problem when you do:

```
ghci> readP_to_S stm "if p then if q then a else b"
```

where `stm` is your parser for this language? If so, how can you fix the grammar to avoid this problem?

◆ ***Problem 7*** ◆

Consider the context free grammar where S is the start symbol.

$$S ::= A\,X \mid T\,C \qquad A ::= \epsilon \mid a\,A \qquad C ::= \epsilon \mid c\,C$$
$$X ::= \epsilon \mid b\,X\,c \qquad Y ::= \epsilon \mid a\,Y\,b$$

Write a parser for this language with the `ReadP` parser library. **Note:** Strings of the form $\texttt{a}^n\texttt{b}^n\texttt{c}^n$ (e.g., `"abc"`, `"aabbcc"`, `"aaabbbccc"`, etc.) *should* have multiple parses since this is an inherently ambiguous context free language.

◆ ***Problem 8*** ◆

Here is a grammar for a simplified version of Lisp s-expressions:

$$\langle \mathsf{Expr} \rangle ::= \langle \mathsf{atom} \rangle \mid (\langle \mathsf{Expr} \rangle^{*})$$

For example, `()`, `(a b c)`, and `(a b (c (d e)) f)` are all valid s-expressions. Note that this is a tokenized[3] language in that whitespace characters can act as delimiters. Use the following as a starter to build a parser for s-expressions.

[3] See: https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization.

```
import Text.ParserCombinators.ReadP
import Data.Char
import Data.List

data Expr = Atom String | SExp [Expr] deriving (Eq)

instance Show Expr where
  show (Atom s) = s
  show (SExp es) = "("++(intercalate " " (map show es))++")"

parse = readP_to_S
```