Final Exam (100 pts)

Burak Ekici

Assigned : January the 6th, 10h00
Duration : 150 minutes

Name-Surname:

1 User Defined Data Types (45 pts)

Q1. A **binary tree** is a data structure in which each node contains at most two nodes. In Haskell, we develop an algebraic data type `BTree` of binary trees, for some arbitrarily given type `a` (inhabiting keys of nodes), with a pair of constructors named `Empty` and `Node` as listed below.

```
data BTree a where
    Empty :: BTree a
    Node  :: a -> BTree a -> BTree a -> BTree a
```

The task is to implement below stated functions in Haskell without employing Prelude functions and list comprehension.

a) (**15 pts**) `splitMaxFromTree :: Ord a => BTree a -> Maybe (a, BTree a)` takes a **binary search tree** `t` (a binary tree such that the key of each internal node is greater than all the keys in the respective node's left subtree and less than the ones in its right subtree), splits the maximum element from `t` and returns a `Maybe` wrapped pair of the maximum element and the rest of the tree.

Expected behavior of the function:

```
splitMaxFromTree Empty                                                  ==  Nothing
splitMaxFromTree [10,[1,_,[5,[3,[2,_,_],[4,_,_]],[7,[6,_,_],_]]],_]   ==
    Just (10,[1,_,[5,[3,[2,_,_],[4,_,_]],[7,[6,_,_],_]]])
```

b) (**15 pts**) `removeFromTree :: Ord a => a -> BTree a -> BTree a` takes a key `x` of some type `a` alongside a **binary search tree** `t` and removes the node with the key `x` from `t`. Note that the function returns the tree as it is if no node with the key `x` appears on `t`.

Expected behavior of the function:

```
removeFromTree 10 [10,[1,_,[5,[3,[2,_,_],[4,_,_]],[7,[6,_,_],_]]],_]   ==
    [7,[1,_,[5,[3,[2,_,_],[4,_,_]],[6,_,_]]],_]
removeFromTree 15 [10,[1,_,[5,[3,[2,_,_],[4,_,_]],[7,[6,_,_],_]]],_]   ==
    [10,[1,_,[5,[3,[2,_,_],[4,_,_]],[7,[6,_,_],_]]],_]
```

c) (**15 pts**) `differenceTree :: Ord a => BTree a -> BTree a` takes **binary search trees** t_1 and t_1, and returns the difference tree $t_1 - t_2$, namely a tree t that contains nodes of t_1 but not those of t_2.

Expected behavior of the function:

```
differenceTree [10,[1,_,[5,[3,[2,_,_],[4,_,_]],[7,[6,_,_],_]]],_] Empty   ==
    [10,[1,_,[5,[3,[2,_,_],[4,_,_]],[7,[6,_,_],_]]],_]
differenceTree Empty [10,[1,_,[5,[3,[2,_,_],[4,_,_]],[7,[6,_,_],_]]],_]   ==   Empty
```

A1. Find aforementioned functions implemented in `A1.hs`

2 Simply Typed λ-Calculus, $\lambda^{\rightarrow}$ (35 pts)

Q2.(**25 pts**) Given simple (ground) types A, B and C, derive the type of every single $\lambda^{\rightarrow}$-term given below

a) (**6 pts**) $\lambda x : (A \rightarrow B) \times (B \rightarrow C).\, \lambda a : A.\, \lambda b : B.\, \big((\text{fst}\ x)\, b, (\text{snd}\ x)\, a\big)$

b) (**6 pts**) $\lambda x : A \rightarrow B \rightarrow C \rightarrow B.\, \lambda a : A.\, \lambda b : B.\, (x\, a)\, b$

c) (**6 pts**) $\lambda x : A \times B \rightarrow C.\, \lambda y : C \rightarrow B.\, \lambda a : A.\, \lambda b : B.\, y\, (x\, (a, b))$

d) (**7 pts**) $\lambda x : A \rightarrow (B \times C).\, \lambda y : A.\, ((\text{fst}\ x)\ y, (\text{snd}\ x)\ y)$

under the empty context, $[\,]$.

A2.

a) $[\,] \vdash \lambda x : (A \rightarrow B) \times (B \rightarrow C).\, \lambda a : A.\, \lambda b : B.\, \big((\text{fst}\ x)\, b, (\text{snd}\ x)\, a\big)$ is ill-typed.

b) $[\,] \vdash \lambda x : A \rightarrow B \rightarrow C \rightarrow B.\, \lambda a : A.\, \lambda b : B.\, (x\, a)\, b : (A \rightarrow B \rightarrow C \rightarrow B) \rightarrow A \rightarrow B \rightarrow (C \rightarrow B)$.

c) $[\,] \vdash \lambda x : A \times B \rightarrow C.\, \lambda y : C \rightarrow B.\, \lambda a : A\, \lambda b : B.\, y\, (x\, (a, b)) : ((A \times B) \rightarrow C) \rightarrow (C \rightarrow B) \rightarrow A \rightarrow B \rightarrow B$.

c) $\lambda x : A \rightarrow (B \times C).\, \lambda y : A.\, ((\text{fst}\ x)\ y, (\text{snd}\ x)\ y)$ is ill-typed.

Q3.(10 pts) Why do we need typing? Please comment, and support your reasoning with an example.

A3. In a typed reduction, evaluation or computing system, every single term is assigned to a type with the use of assertions named *typing judgments* which form a basis for *typing rules*. This set of rules mainly aim at preventing unintended cases that may potentially take

place during the term evaluation. For instance, the application $(f\ a)$ is evaluated in $\lambda^{\rightarrow}$ only when the term f has an arrow type such that the type of the term a matches its domain. That is to say, the application $(1\ 2)$ is disallowed in $\lambda^{\rightarrow}$ as the natural number 1 is definitely not an instance of an arrow type. Of course, proving the soundness and completeness properties of type systems are crucial as (1) an unsound type system may disallow the evaluation of a well-typed term, and (2) an incomplete type system may let ill-typed terms evaluate.

3 Structural Induction (20 pts)

Q4. A binary tree is said to be **perfect** if all leaf nodes have same depth.

```
height :: BTree a -> Int
height t =
    case t of
        Empty      -> 0
        Node _ l r -> max (height l) (height r) + 1

size :: BTree a -> Int
size t =
    case t of
        Empty      -> 0
        Node _ l r -> size l + size r + 1

perfect :: BTree a -> Bool
perfect t =
    case t of
        Empty      -> True
        Node _ l r -> height l == height r && perfect l && perfect r
```

Prove by structural induction that if a binary tree (with the implementation in Section 1) of height n is perfect then it exactly has $2^n - 1$ nodes. That is, for any Haskell data type `a`,

$$\forall(\texttt{t} :: \texttt{BTree a}),\ \texttt{perfect t} \Longrightarrow \texttt{size t} = 2^{\texttt{n}}\texttt{-1}$$

holds.

A4.

Proof. proceeds by structural induction over the term t.

- base case: $t = \texttt{Empty}$

$$\begin{aligned} \texttt{size (Empty)} &= 0 \\ &= 2^0 - 1 \\ &= 2^{\texttt{height t}} - 1 \end{aligned}$$

- step case: $t = \texttt{Node x l r}$

given IHl : perfect l $\Longrightarrow$ size l = $2^{\texttt{height l}} - 1$
IHr : perfect r $\Longrightarrow$ size r = $2^{\texttt{height r}} - 1$
H : perfect (Node x l r)

> H destructs into
> Ha: perfect l
> Hb: perfect r
> Hc: height l = height r

show : size (Node x l r) = $2^{\texttt{height (Node x l r)}} - 1$

We close the goal with the following inference schema.

size (Node x l r)	=	size l + size r + 1	(by definition of size)
	=	$2^{\texttt{height l}} - 1 + 2^{\texttt{height r}} - 1 + 1$	(by (IHl Ha) and (IHr Hb))
	=	$2^{\texttt{height l}} - 1 + 2^{\texttt{height l}} - 1 + 1$	(by Hc)
	=	$2^{\texttt{height l + 1}} - 1 - 1 + 1$	(by arithmetic)
	=	$2^{\texttt{height l + 1}} - 1$	(by arithmetic)
	=	$2^{\texttt{max (height l, height l) + 1}} - 1$	(by arithmetic)
	=	$2^{\texttt{max (height l, height r) + 1}} - 1$	(by Hc)
	=	$2^{\texttt{height (Node x l r)}} - 1$	(by definition of height)

□