Functional and Logic Programming

Home Assignment 3

Due: Saturday, 5.5.2018 - 23:55

Instructions

- Please create a source file called ***HW3.hs*** and put all the answers there.
  The file should start with a comment which contains your **full name** (in English) and **ID**
  (see also example: ***hwexample.hs*** in the Moodle)

  ```
  -- John Doe
  -- 654321987
  ```

- Make sure the file **is valid** by loading it into GHCi.
  A valid file will load without any errors or warnings.

- If you need a function but you don't know how to implement it - just write it's signature (name and type) and put `undefined` in the function's body.
  That way you'll be able to load the file even though it contains references to undefined names.
  (see also example: ***hwexample.hs*** in the Moodle)

- When writing a function - write both the **type** and the **body** of the function.

- Be sure to write functions with **exactly the specified name** (and **type signature** - if it is provided) for each exercise.
  You may create additional auxiliary/helper functions with whatever names and type signatures you wish.

- Try to write **small functions** which perform just **a single task**, and then **combine** them to create more complex functions.

Exercises

1. The following BNF represents arithmetic expressions which have **variables** in them:

```
data Expr
    = Const Value    -- a constant number.
    | Add Expr Expr  -- addition of expressions.
    | Mul Expr Expr  -- multiplication of expressions.
    | Sub Expr Expr  -- subtraction of expressions.
    | Div Expr Expr  -- division of expressions.
    | Var Variable   -- a variable.
```

where:

```
type Variable = String
type Value    = Float
```

To **evaluate** the BNF - we'll also use the following type synonyms:

```
type Dictionary = [(Variable, Value)]
type EvalError  = [Variable]
type EvalResult = Either EvalError Value
```

The `Expr` expression may contain variables.
In order to evaluate the variables we will look for the value of a variable in the `Dictionary` (which maps variables to values).
If we try to evaluate an expression which contains variables which aren't in the `Dictionary` - we'll get an `EvalError`. The `EvalError` will contain **all** the variables that appear in the expression but not in the `Dictionary`.

   a) Write a function **`display`** which takes an expression and returns a string which represents the expression.
   `display` needs to follow the following rules:
   - Addition, multiplication, subtraction, and division are displayed using the infix `+`, `*`, `-`, `/` - respectively.
   - Arithmetic operations (addition, multiplication, subtraction, and division) will always appear within **parentheses**.
   - A `Variable` is always displayed as it is.
   - To display a `Value` - simply use the built-in function **`show`** on it.

```
display (Const 5) = "5.0"

display (Var "x") = "x"

display ((Var "x") `Div` (Var "y")) = "(x/y)"
```

```
display
    (((Const 2) `Mul` ((Var "x") `Mul` (Var "x"))) `Add` ((Var "a") `Mul` (Var "x")))
    = "((2.0*(x*x))+(a*x))"
```

b) Write a function **eval** with the following signature:

```
eval :: Dictionary -> Expr -> EvalResult
```

This function will take a dictionary and an expression, and will evaluate the expression by the following rules:

- Valid values are returned by wrapping them with the **Right** constructor.
  Evaluation errors are returned by wrapping them with the **Left** constructor.
- A **Value** is simply returned without further evaluation (since it is already a value).
- The **Add**, **Mul**, **Sub**, and **Div** expressions are evaluated by doing addition, multiplication, subtraction, and division (respectively) of the values of their subexpressions.
- A **Variable** is evaluated by looking for its value in the dictionary.
  You may assume the variable only appear once in the dictionary.
- If there is at least 1 variable in the expression which doesn't have a value in the dictionary - instead of a **Value** the **eval** function will return an **EvalError**, which will contain **all** the undefined variables in the expression.
- The order of the variables in the **EvalError** list **doesn't matter**.

```
eval
    [("x",5),("a",10)]
    (((Const 2) `Mul` ((Var "x") `Mul` (Var "x"))) `Add` ((Var "a") `Mul` (Var "x")))
    = Right 100.0

eval
    [("x",10),("a",5)]
    (((Const 2) `Mul` ((Var "x") `Mul` (Var "x"))) `Add` ((Var "a") `Mul` (Var "x")))
    = Right 250.0

eval
    [("x",10)]
    (((Const 2) `Mul` ((Var "x") `Mul` (Var "x"))) `Add` ((Var "a") `Mul` (Var "x")))
    = Left ["a"]

eval
    []
    (((Const 2) `Mul` ((Var "x") `Mul` (Var "x"))) `Add` ((Var "a") `Mul` (Var "x")))
    = Left ["x","x","a","x"]
```

2. The questions in this section are about the following **polymorphic tree** data type:

```
data Tree a b = Leaf b | Node a (Tree a b) (Tree a b)
```

a) Create a function **`reverseTree`** which takes a tree and reverses it (so the output will be its mirror reflection).

```
reverseTree (Leaf 't') = Leaf 't'

reverseTree (Node 5 (Leaf 'x') (Leaf 'y'))
    = Node 5 (Leaf 'y') (Leaf 'x')

reverseTree
    (Node 1 (Node 2 (Leaf 'a') (Leaf 'b')) (Node 3 (Leaf 'c') (Leaf 'd')))
    = Node 1 (Node 3 (Leaf 'd') (Leaf 'c')) (Node 2 (Leaf 'b') (Leaf 'a'))
```

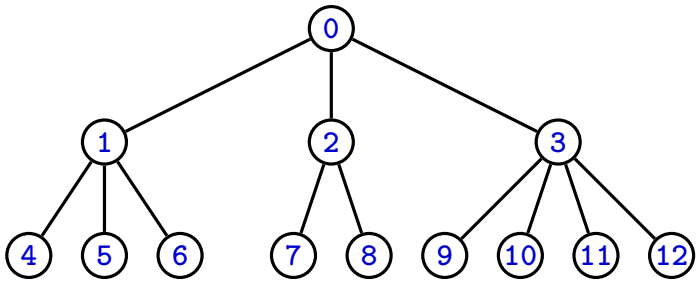
b) Create a function **`isSubtree`** which takes 2 trees of type `Tree Int Char` and returns `True` if the 1st tree is a subtree of the 2nd tree and `False` otherwise.
Note: a tree is a subtree of itself.

```
isSubtree (Leaf 'x') (Leaf 'x') = True

isSubtree (Leaf 'x') (Leaf 'y') = False

isSubtree
    (Node 3 (Leaf 'c') (Leaf 'd'))
    (Node 1 (Node 2 (Leaf 'a') (Leaf 'b')) (Node 3 (Leaf 'c') (Leaf 'd')))
    = True

isSubtree
    (Node 7 (Leaf 'c') (Leaf 'd'))
    (Node 1 (Node 2 (Leaf 'a') (Leaf 'b')) (Node 3 (Leaf 'c') (Leaf 'd')))
    = False

isSubtree
    (Node 1 (Node 2 (Leaf 'a') (Leaf 'b')) (Node 3 (Leaf 'c') (Leaf 'd')))
    (Node 3 (Leaf 'c') (Leaf 'd'))
    = False

isSubtree
    (Leaf 'c')
    (Node 1 (Node 2 (Leaf 'a') (Leaf 'b')) (Node 3 (Leaf 'c') (Leaf 'd')))
    = True
```

3. The following data type represents tree where each node can have **many** children:

```
data MTree a = MTree a [MTree a]
```

In this kind of tree, the leaves are nodes that have an empty children list.
For example - the following trees:



is represented as:

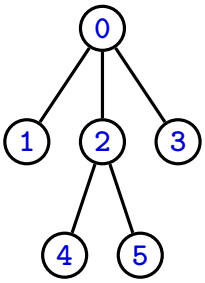
```
MTree 0
    [ MTree 1
        [ MTree 4 [], MTree 5 [], MTree 6 [] ]
    , MTree 2
        [ MTree 7 [], MTree 8 [] ]
    , MTree 3
        [ MTree 9 [], MTree 10 [], MTree 11 [], MTree 12 [] ]
    ]
```

The following questions are about the `MTree` data type:

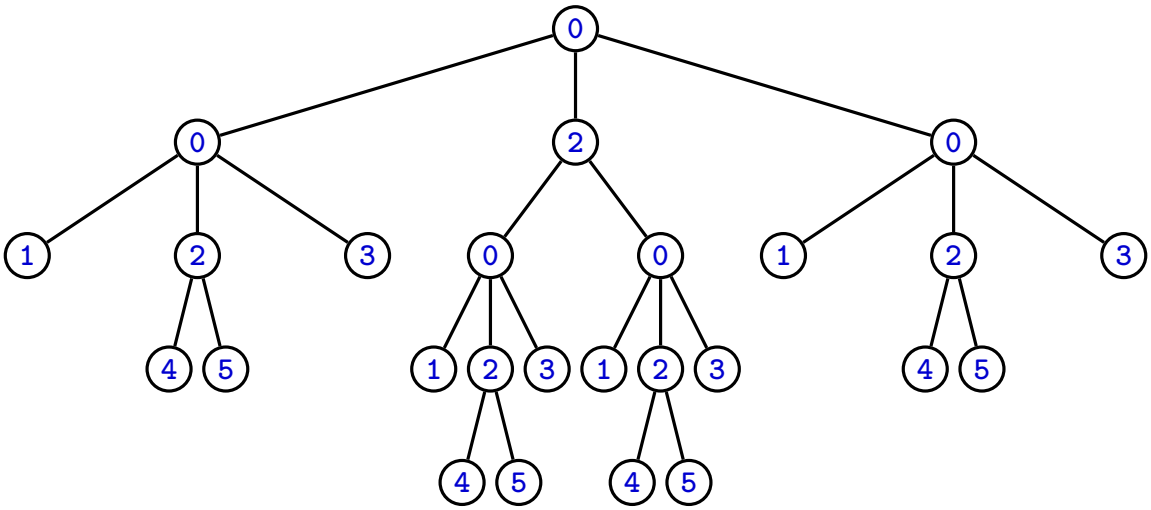
a) Create a function **`sumMTree`** which takes a tree of type `MTree Int` and computes the sum of all of its nodes.

```
sumMTree (MTree 5 []) = 5

sumMTree (MTree 1 [MTree 2 [], MTree 3 [MTree 5 [], MTree 6 []], MTree 4 []])
    = 21
```

b) Create a function **`grow`** that takes an `MTree a` and replace each leaf in the tree with a copy of the original tree.
So - the following tree:



Will become:



While a tree where the root is a leaf will stay as it is:

```
grow (MTree 5 []) = MTree 5 []
```

Another example:

```
grow (MTree 3 [ MTree 5 [], MTree 2 [] ])
    = MTree 3 [ MTree 3 [ MTree 5 [], MTree 2 [] ]
              , MTree 3 [ MTree 5 [], MTree 2 [] ]
              ]
```