BFNP–F2016, Functional Programming The IT University, Spring 2016

Exercise 3

Last update 2016-02-09

This exercise sheet must be handed in via LearnIt by February 25th.

You are welcome to solve the assignments in pairs.

Your name must be part of the filename, e.g., `BFNP-03-<name1>-<name2>.fsx`, where `<name1>` and `<name2>` are the names of the two working together. Both `name1` and `name2` must upload the same file. An example: `BFNP-03-MadsAndersen-ConnieHansen.fsx`.

You can only upload one file and it must be of type `fs` or `fsx`.

It is important that you annotate your own code with comments. It is also important that you apply a functional style, i.e., no loops and no mutable variables.

For this hand-in you also need to consider scenarios where your solutions should return an error, i.e., an exception. The requirement is, that no matter what input you pass to your function that fulfils the function type, then the function should return the intended answer or an exception. It is up to you to define the exceptions and whether they should carry extra information, like error messages.

The exercises 3.1 to 3.3 refer to code defined on slides 30 to 32 on the slide deck from lecture 5 about finite trees. You need to look there for the definition of `'a BinTree` and in-order traversal.

Exercise 3.4, 3.5 and 3.6 refer to the slides from the same slide deck, lecture 5, starting with slide 17.

Exercise 3.1 Consider the definition of type `'a BinTree` on slide 30. Write a function

`inOrder : 'a BinTree -> 'a list`

that makes an in-order traversal of the tree and collect the elements in a result list. In-order traversal is defined on slide 32.

Exercise 3.2 Write a function

`mapInOrder : ('a -> 'b) -> 'a BinTree -> 'b BinTree`

that makes an in-order traversal of the binary tree and apply the function on all nodes in the tree.

Can you give an example of why `mapInOrder` might give a result different from `mapPostOrder`, but the result tree retruned in both cases is still the same.

Exercise 3.3 Write a function

`foldInOrder : ('a -> 'b -> 'b) -> 'b -> 'a BinTree -> 'b`

that makes an in-order traversal of the tree and folds over the elements.

For instance, given the tree

```
let floatBinTree = Node(43.0,Node(25.0, Node(56.0,Leaf, Leaf), Leaf),
                             Node(562.0, Leaf, Node(78.0, Leaf,Leaf)))
```

the application

```
foldInOrder (fun n a -> a + n) 0.0 floatBinTree
```

returns 764.0.

Exercise 3.4 Complete the program skeleton for the interpreter presented on slide 28 in the slide deck from the lecture 5 about finite trees.
Define 5 examples and evaluate them.
The declaration for the abstract syntax for *arithmetic expressions* follows the grammar (slide 23):

```
type aExp =             (* Arithmetical expressions *)
  | N of int            (* numbers *)
  | V of string         (* variables *)
  | Add of aExp * aExp  (* addition *)
  | Mul of aExp * aExp  (* multiplication *)
  | Sub of aExp * aExp  (* subtraction *)
```

The declaration of the abstract syntax for *boolean expressions* is defined as follows (slide 25).

```
type bExp =             (* Boolean expressions *)
  | TT                  (* true *)
  | FF                  (* false *)
  | Eq of aExp * aExp   (* equality *)
  | Lt of aExp * aExp   (* less than *)
  | Neg of bExp         (* negation *)
  | Con of bExp * bExp  (* conjunction *)
```

The conjunction of two boolean values returns true if both values are true.
The abstract syntax for the statements are defined as below (slide 26):

```
type stm =                    (* statements *)
  | Ass of string * aExp      (* assignment *)
  | Skip
  | Seq of stm * stm          (* sequential composition *)
  | ITE of bExp * stm * stm   (* if-then-else *)
  | While of bExp * stm       (* while *)
```

Exercise 3.5 Extend the abstract syntax and the interpreter with *if-then* and *repeat-until* statements.
Again we refer to slide 28 in the slide deck from the lecture 5.

Exercise 3.6 Suppose that an expression of the form $inc(x)$ is added to the abstract syntax. It adds one to the value of x in the current state, and the value of the expression is this new value of x.

How would you refine the interpreter to cope with this construct?

Again we refer to slide 28 in the slide deck from the lecture 5

Exercise 3.7 HR exercise 6.2

Exercise 3.8 HR exercise 6.8

Exercise 3.9 HR exercise 7.2