02157 Functional programming

Michael R. Hansen
DTU Compute
November 19, 2018

Mandatory assignment 4

This is the fourth of four mandatory assignments in 02157 Functional programming. It is a requirement for exam participation that 3 of the 4 mandatory assignments are approved. The mandatory assignments can be solved individually or in groups of 2 or 3 students.

Acceptance of a mandatory assignment from previous years does NOT apply this year.

- Your solution should be handed in **no later than Thursday, November 29, 2018**. Submissions handed in after the deadline will face an *administrative rejection*.
- The assignment contains two independent programming problems: **Scoreboards** and **Paths in trees**. You should solve both problems and your solutions should be contained in a single F# file (*file*`.fsx` or *file*`.fs`). In your solution you are allowed to introduce helper functions; but you must also provide a declaration for each of the required functions, so that it has exactly the type and effect asked for. Each function declaration should be accompanied by a few illustrative test cases.
- Do not use imperative features, like assignments, arrays and so on in your programs. Failure to comply with that will result in an *administrative rejection of your submission*.
- To submit you should upload a single F#-file to Inside under Assignment 4: The file should start with full names and study numbers for all members of the group. If the group members did not contribute equally to the solution, then the role of each student must be explicitly stated.
- Your F# solution must be a complete program that can be uploaded to F# Interactive without encountering compilation errors. Failure to comply with that will result in an *administrative rejection of your submission*.
- Be careful that you submit the right file. A submission of a wrong file will result in an *administrative rejection of your submission*.
- **DO NOT COPY solutions** from others and **DO NOT SHARE your solution** with others. Both cases are considered as fraud and will be reported.

Scoreboards

We consider here *scoreboards* containing information about the *scores*, where a score describes the *points* a *named person* has obtained in an *event*. This is modelled by the following type declarations:

```
type Name = string
type Event = string
type Point = int
type Score = Name * Event * Point

type Scoreboard = Score list

let sb = [("Joe", "June Fishing", 35); ("Peter", "May Fishing", 30);
          ("Joe", "May Fishing", 28); ("Paul", "June Fishing", 28)];;
```

The example scoreboard `sb` describes four scores, where Joe, for example, has obtained 35 points in the fishing event in June and 28 points in the May fishing event. The other two scores have similar explanations.

1. The points occurring in scores must be non-negative integers, and scores must occur in scoreboards in a sequence respecting weakly decreasing points, that is, if (n, e, p) occurs before $(n1, e1, p1)$ in a scoreboard, then $p \geq p1$. Hence, a score with the highest number of points occurs first and a score with the lowest number of points occurs last in a scoreboard. Declare a function: `inv: Scoreboard -> bool`, that checks whether a scoreboard satisfies this constraint.

The functions below must respect the invariant `inv`, that is, it can be assumed that argument scoreboards satisfy `inv`, and it is required that scoreboard results of functions must satisfy `inv`.

2. Declare a function `insert: Score -> Scoreboard -> Scoreboard`, so that `insert` s sb gives the scoreboard obtained from sb by insertion of s. The result must satisfy `inv`.

3. Declare a function `get: Name*Scoreboard -> (Event*Point) list`, where the value of `get`(n, sb) is a list of pairs of events and points obtained from n's scores in sb. For example `get("Joe",sb)` must be a list with the two elements: `("June Fishing", 35)` and `("May Fishing", 28)`.

4. Declare a function `top: int -> Scoreboard -> Scoreboard option`. The value of `top` k sb is `None` if $k < 0$ or sb does not contain k scores; otherwise the value is `Some` sb', where sb' contains the first k scores of sb.

Paths in trees

We consider now trees where nodes can have an arbitrary number of subtrees:

```
type T<'a> = N of 'a * T<'a> list;;

let td = N("g", []);;
let tc = N("c", [N("d",[]); N("e",[td])]);;
let tb = N("b", [N("c",[])]);;
let ta = N("a", [tb; tc; N("f",[])])
```

The tree $t = \texttt{N}(v, [t_0; \ldots; t_{n-1}])$ describes a node that contains the *value* v and has n (immediate) subtrees t_i, for $0 \leq i < n$. For example, the four trees `ta` - `td` illustrate trees having 3, 1, 2 and 0 immediate subtrees, where the values contained in the nodes are the seven strings `"a"` - `"g"`.

1. Declare a function `toList` t which returns a list of all the values occurring in the nodes of the tree t. The order in which values occur in the list is of no significance.
2. Declare a function `map` f t, which returns the tree obtained from the t by applying the function f to the values occurring in the nodes of t. Give the type of `map`.

We shall use integer lists to denote paths in trees:

```
type Path = int list;;
```

A *path* (type `Path`) in the tree $t = \texttt{N}(v, [t_0; \ldots; t_i; \ldots; t_{n-1}])$ is a list of integers that *identifies* a subtree of t in the following recursive manner:

- The empty path (list) [ ] identifies the entire tree t.
- If *is* identifies t' in t_i, then $i :: is$ identifies t' in $\texttt{N}(v, [t_0; \ldots; t_i; \ldots; t_{n-1}])$.

For example, the path `[0]` identifies the subtree `tb` of `ta` and the path `[1; 1; 0]` identifies the subtree `td` of `ta`.

3. Declare a function `isPath` *is* t that checks whether *is* is a path in t.
4. Declare a function `get: Path` $\rightarrow$ `T<'a>` $\rightarrow$ `T<'a>`. The value of `get` *is* t is the subtree identified by *is* in t.
5. Declare a function `tryFindPathto` : `'a` $\rightarrow$ `T<'a>` $\rightarrow$ `Path option`. When v occurs in some node of t, then the value of `tryFindPathto` v t is `Some` *path*, where v occurs in the node of t identified by *path*. The value of `tryFindPathto` v t is `None` when v does not occur in a node of t. There is no restriction concerning which path the function should return when v occurs more than once in t.