02157 Functional programming

Michael R. Hansen
DTU Compute
October 10, 2018

Mandatory assignment 2

This is the second 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, October 24, 2019**. Submissions handed in after the deadline will face an *administrative rejection*.
- The assignment has two tasks:

  Task 1: This task has the form of a paper and pencil exercise. You may consider scanning in you hand-written solution to this task and submit it in the form of a pdf-file.

  Tasks 2: This task concerns a programming problem. A solution should have the form of 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 two files to Inside under Assignment 2: a pdf-fil containing the solution to Task 1 and a single F# file (*file*`.fsx` or *file*`.fs`) containing the solutions to Task 2. The files 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 files. 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.

Task 1

1. Declare a (tree) type `T` so that

   ```
   A []
   B true
   C(A [B false; B true],B true)
   C(A [B true; C(A [B true],B true);B false], A [B true; C(A [B true],B true);B false])
   ```

   are four values of type `T`.

2. Rules for generating values of type `T` can be derived from your type declaration. Formulate these rules in your own words.

3. Show how your rules can be used to generate each of the above four values.

4. Give figures showing trees for the above four values.

Task 2

Make a solution to the problem presented on the next two pages.

Balance bikes

A *balance bike* (Danish: "løbecykel") is a bike used by small children, where they can learn how to ride without having to worry about pedals and brakes. In its barest form, a balance bike is constructed from a frame, two wheels, a saddle, handlebars, nuts and bolts, which constitute the *parts* of the bike. The manufacturing of a balance bike may be divided into a sequence of three *tasks*. For example, first mount two wheels on the frame, then mount the saddle, and finally, mount handlebars.

An *assembly line* is a manufacturing process where parts successively are added to an assembly at *workstations* until the final product is obtained. A *part register* (type `PartReg`) associates costs with parts, and a *task register* (type `TaskReg`) associates a pair (d, c) with a task *tsk*, where d is the time needed to perform *tsk* and c is the associated cost. We also call d the *duration* of the task:

```
type Part     = string
type Task     = string
type Cost     = int                 (* can be assumed to be positive *)
type Duration = int                 (* can be assumed to be positive *)
type PartReg  = Map<Part, Cost>
type TaskReg  = Map<Task, Duration*Cost>

(* Part and task registers for balance bikes *)
let preg1 = Map.ofList [("wheel",50); ("saddle",10); ("handlebars",75);
("frame",100); ("screw bolt",5); ("nut",3)];;
let treg1 = Map.ofList [("addWheels",(10,2)); ("addSaddle",(5,2));
("addHandlebars",(6,1))]
```

We observe, from the two example registers, that the cost of a wheel is 50 (say Danish kr.) and mounting a saddle (task `"addSaddle"`) takes 5 time units and costs 2 kr.

A workstation is described by a task (like `"addSaddle"`) and a part list, describing the number of the various parts that are needed to perform the task. Furthermore, an assembly line is a list of workstations:

```
type WorkStation = Task * (Part*int) list
type AssemblyLine = WorkStation list

let ws1 = ("addWheels",[("wheel",2);("frame",1);("screw bolt",2);("nut",2)])
let ws2 = ("addSaddle",[("saddle",1);("screw bolt",1);("nut",1)])
let ws3 = ("addHandlebars",[("handlebars",1);("screw bolt",1);("nut",1)])
let al1 = [ws1; ws2; ws3];;
```

We see that the assembly line for balanced bikes consists of 3 workstations, where, for example, the work station for mounting the saddle requires one piece of each of the parts: saddle, screw bolt and nut.

A *workstation* $(tsk, [(p_1, k_1); \ldots; (p_n; k_n)])$ *is well-defined* for given part register *preg* and task register *treg*, if (1) there is an entry for *tsk* in *treg*, (2) there is an entry in *preg* for every p_i, where $1 \leq i \leq n$, and (3) the numbers $k_1, \ldots k_n$ are all positive.

Furthermore, an *assembly line is well-defined* for given part register *preg* and task register *treg* if every workstation in the assembly line is well-defined.

1. Declare a function `wellDefWS: PartReg -> TaskReg -> WorkStation -> bool` that checks the well-definedness of a workstation for given part and task registers.

2. Declare a function `wellDefAL: PartReg -> TaskReg -> AssemblyLine -> bool` that checks the well-definedness of an assembly line for given part and task registers. This function should be declared using `List.forall`.

In your answers to the following questions, you can assume that workstations and assembly lines are well-defined.

3. Declare a function `longestDuration`$(al, treg)$, where *al* is an assembly line and *treg* a task register. The value of `longestDuration`$(al, treg)$ is the longest duration of a task in *al*. What is the type of `longestDuration`?

   For example, the longest duration of a task in the assembly line for balanced bikes is 10 (the duration of `"addWheels"`).

4. Declare a function `partCostAL: PartReg -> AssemblyLine -> Cost`, that computes the accumulated cost of all parts needed for one final product of an assembly line for a given part register. For example, the accumulated cost of all parts of a balanced bike is 317 - the cost of one frame, two wheels, one saddle, handlebars, 4 nuts and 4 screw bolts.

   Hint: You may introduce helper functions to deal with part lists $[(p_1, k_1); \ldots; (p_n; k_n)]$ and workstations.

5. Declare a function `prodDurCost: TaskReg -> AssemblyLine -> Duration*Cost`, that for a given assembly line and task register, computes a pair $(totalDuration, totalCost)$, where *totalDuration* is the accumulated duration of all durations of tasks in the assembly line and *totalCost* is the accumulated cost of the costs of all tasks in the assembly line (where the cost of parts is ignored). For the balanced bike example, the accumulated duration of the three tasks is 21 and the accumulated cost is 5.

A *stock* is mapping from parts to number of pieces:

```
type Stock = Map<Part, int>
```

6. Declare a function `toStock: AssemblyLine -> Stock`, that for a given assembly line, computes the stock needed to produce a single product.