Problem 2 from May 2019 (Approx. 72 minutes)

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 workstations and part lists $[(p_1, k_1); \ldots; (p_n; k_n)]$.
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.

Problem 1 from May 2017 (approx 48 minutes)

Consider the following F# declaration:

```
let rec f xs ys = match (xs,ys) with
                  | (x::xs1, y::ys1) -> x::y::f xs1 ys1
                  | _                -> [];;
```

1. Give an evaluation (using $\rightsquigarrow$) for `f [1;6;0;8] [0; 7; 3; 3]` thereby determining the value of this expression.
2. Give the (most general) type for `f`, and describe what `f` computes. Your description should focus on *what* it computes, rather than on individual computation steps.
3. The declaration of `f` is *not* tail recursive. Give a brief explanation of why this is the case and provide a declaration of a tail-recursive variant of `f` that is based on an accumulating parameter. Your tail-recursive declaration must be based on an explicit recursion.
4. Provide a declaration of a continuation-based, tail-recursive variant of `f`.

Problem 2.1 from May 2017 (approx 20 minutes)

Consider the following F# declarations:

```
let rec f  = function
             | 0          -> [0]
             | i when i>0 -> i::g(i-1)
             | _          -> failwith "Negative argument"
and g = function
        | 0 -> []
        | n -> f(n-1);;

let h s k  = seq { for a in s do
                         yield k a };;
```

1. Give the values of `f 5` and `h (seq [1;2;3;4]) (fun i -> i+10)`. Furthermore, give the (most general) types for `f` and `h`, and describe what each of these two functions computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.

Problem 3 from May 16 (approx 48 minutes)

We shall now consider *containers* that can either have the form of a *tank*, that is characterized by it length, width and height, or the form of a *ball*, that is characterized by its radius. This is captured by the following declaration:

```
type Container =
    | Tank of float * float * float // (length, width, height)
    | Ball of float                 // radius
```

1. Declare two F# values of type `Container` for a tank and a ball, respectively.
2. A tank is called *well-formed* when its length, width and height are all positive and a ball is well-formed when its radius is positive. Declare a function `isWF` : `Container` $\rightarrow$ `bool` that can test whether a container is well-formed.
3. Declare a function `volume` c computing the volume of a container c. (Note that the volume of ball with radius r is $\frac{4}{3} \cdot \pi \cdot r^3$.)

A *cylinder* is characterized by its radius and height, where both must be positive float numbers.

4. Extend the declaration of the type `Container` so that it also captures cylinders, and extend the functions `isWF` and `volume` accordingly. (Note that the volume of cylinder with radius r and height h is $\pi \cdot r^2 \cdot h$.)

A *storage* consist of a collection of uniquely named containers, each having a certain *contents*, as modelled by the type declarations:

```
type Name     = string
type Contents = string
type Storage  = Map<Name, Contents*Container>
```

where the name and contents of containers are given as strings.

Note: You may choose to solve the below questions using a list-based model of a storage (`type Storage = (Name * (Contents*Container)) list`), but your solutions will, in that case, at most count 75%.

5. Declare a value of type `Storage`, containing a tank with name `"tank1"` and contents `"oil"` and a ball with name `"ball1"` and contents `"water"`.
6. Declare a function `find` : `Name` $\rightarrow$ `Storage` $\rightarrow$ `Contents` $*$ `float`, where `find` $n\ stg$ should return the pair (cnt, vol) when cnt is the contents of a container with name n in storage stg, and vol is the volume of that container. A suitable exception must be raised when no container has name n in storage stg.

Problem 4 from May 16 (approx. 48 minutes)

Consider the following F# declarations of a type `T<'a>` for binary trees having values of type `'a` in nodes, three functions `f`, `h` and `g`, and a binary tree `t`:

```
type T<'a> = L | N of T<'a> * 'a * T<'a>

let rec f g t1 t2 =
   match (t1,t2) with
   | (L,L) -> L
   | (N(ta1,va,ta2), N(tb1,vb,tb2))
           -> N(f g ta1 tb1, g(va,vb), f g ta2 tb2);;

let rec h t = match t with
              | L           -> L
              | N(t1, v, t2) -> N(h t2, v, h t1);;

let rec g = function
            | (_,L)                    -> None
            | (p, N(t1,a,t2)) when p a -> Some(t1,t2)
            | (p, N(t1,a,t2))          -> match g(p,t1) with
                                          | None -> g(p,t2)
                                          | res  -> res;;

let t = N(N(L, 1, N(N(L, 2, L), 1, L)), 3, L);;
```

1. Give the type of `t`. Furthermore, provide three values of type `T<bool list>`.
2. Give the (most general) types of `f`, `h` and `g` and describe what each of these three functions computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.
3. Declare a function `count` $a\,t$ that can count the number of occurrences of a in the binary tree t. For example, the number of occurrences of 1 in the tree `t` is 2.
4. Declare a function `replace`, so that `replace` $a\,b\,t$ is the tree obtained from t by replacement of every occurrence of a by b. For example, `replace 1 0 t` gives the tree `N(N(L, 0, N(N(L, 2, L), 0, L)), 3, L)`.