

```
1 (* Daniel F. Hauge - Studentnumber: s201186 - DTU Course: 02157 Functional programming *)
2 open System
3
4 (* Problem 1 types *)
5
6 type Person = string
7 type Contacts = Person list
8 type Register = (Person * Contacts) list
9
10
11 (* Problem 3 types *)
12
13 type Name = string
14 type Part =
15     | S of Name // Simple part
16     | C of Name * Part list // Composite part
17
18 type OccurrenceCount = Map<Name,int>
19
20
21 [<EntryPoint>]
22 let main argv =
23
24     (* Problem 1 *)
25
26     let reg1 = [("p1", ["p2"; "p3"]);
27                 ("p2", ["p1"; "p4"]);
28                 ("p3", ["p1"; "p4"; "p7"]);
29                 ("p4", ["p2"; "p3"; "p5"]);
30                 ("p5", ["p2"; "p4"; "p6"; "p7"]);
31                 ("p6", ["p5"; "p7"]);
32                 ("p7", ["p3"; "p5"; "p6"])]
33
34     (* Question 1.1 *)
35     let inv1 (reg:Register): bool =
36         let contacts = List.map (fun x -> snd x) reg
37         List.forall (fun x -> List.length (List.distinct x) = List.length x ) contacts
38
39     printfn "Question 1.1: %A" (inv1 reg1)
40
41     (* Question 1.2 *)
42     let inv2 (reg:Register) : bool =
43         let people = List.map (fun x -> fst x) reg
44         List.forall (fun x -> not (List.isEmpty (snd x))) reg && List.length (List.distinct people) = List.length people
45
46     printfn "Question 1.2: %A" (inv2 reg1)
```

```

47
48
49     let rec insert p ps = if List.contains p ps then ps else p::ps
50
51     let rec combine ps1 ps2 = List.foldBack insert ps1 ps2
52
53
54     (* Question 1.3 *)
55     let rec immediateContacts (p:Person) (reg:Register) : Contacts =
56         match reg with
57         | (pr, c)::_ when p = pr -> c
58         | _::tail -> immediateContacts p tail
59         | _ -> []
60
61     printfn "Question 1.3: %A" (immediateContacts "p1" reg1)
62
63     (* Question 1.4
64         Assuming adding contacts is bi-directional. ie. Adding p1 to p2 as a
65         close contact implies adding p2 to p1 as a close contact aswell.
66     *)
67     let rec addContacts (p1:Person) (p2:Person) (reg:Register) : Register =
68         match reg with
69         | (p, c)::xs when p = p1 -> (p, insert p2 c)::addContacts p1 p2 xs
70         | (p, c)::xs when p = p2 -> (p, insert p1 c)::addContacts p1 p2 xs
71         | h::tail -> h::addContacts p1 p2 tail
72         | _ -> []
73
74     printfn "Question 1.4: %A" (addContacts "p1" "p2" reg1)
75
76     (* Question 1.5 *)
77     let contacts (p:Person) (reg:Register) : Contacts =
78         let imC = immediateContacts p reg
79         let depth2contacts = List.map (fun x-> snd x) (List.filter (fun x->
80             List.contains (fst x) imC) reg)
81         combine imC (List.reduce (fun a b -> combine a b) depth2contacts)
82
83     printfn "Question 1.5: %A \n" (contacts "p1" reg1)
84
85
86
87     (* Problem 2 *)
88
89     (* Question 2.1
90         j is used with an addition operation, hence j is inferred to be an
91         integer.
92         xs can be inferred to be some sort of list, by the pattern matching.
93         (Can easily be seen by the [] -> [] case)

```

```

92      output is also inferred to be some sort of list by the pattern matching. ↗
          (Can easily be seen by the [] -> [] case)
93      f is a function which is used with j and an element of xs.
94      there is no information for what kind of type the elements within xs ↗
          has to be, also there is no information about what type f has as ↗
          output,
95      hence the generic types 'a for elements within xs and 'b for elements ↗
          after f has been applied.

96
97      Therefor:
98      j : int
99      f : (int -> 'a -> 'b)
100     xs : 'a list
101     output : 'b list
102
103     h : (int -> 'a -> 'b) -> 'a list -> int -> 'b list
104
105     *)
106
107     (* Question 2.2
108
109         given
110         f : (fun i x -> (i,x))
111         xs : ["a";"b";"c"]
112         (and 0 as j in h, as declared in mapi)
113
114         => mapi (fun i x -> (i,x)) ["a";"b";"c"]    (=>) h (fun i x -> (i,x)) ↗
            ["a";"b";"c"] 0
115         => (0,"a")::h (fun i x -> (i,x)) ["b";"c"] 1
116         => (0,"a")::(1,"b")::h (fun i x -> (i,x)) ["c"] 2
117         => (0,"a")::(1,"b")::(2,"c")::h (fun i x -> (i,x)) [] 3
118         => (0,"a")::(1,"b")::(2,"c")::[]    (Concatinating elements from here.)
119         => [(0,"a");(1,"b");(2,"c")]
120
121     *)
122
123     (* Question 2.3
124
125         (fun i x -> (i,x)) : 'a -> 'b -> ('a * 'b)
126         The expression is a function which takes 2 inputs respectively i and x, ↗
            putting them together in a tuple. Types for i and x can be whatever, ↗
            there is no operations which forces behavior or type restrictions.

127
128         ["a";"b";"c"] : string list
129         Double qoutes is in many programming languages including F# the way to ↗
            indicate literal string values.

130
131         'a in h is forced to be an integer because of the previously mentioned ↗
            addition operation, as a result forces 'a or rather i in f to be an ↗

```

```

integer aswell.
132     Using a list of strings will give string in 'b' or rather x in f.
        Therefor the result of f will give a tuple of integer and string, 'b
        in h.
133
134     *)
135
136
137     (* Question 2.4 *)
138
139     let rec h f xs j = match xs with
140     | []      -> []
141     | x::rest -> f j x :: h f rest (j+1)
142
143     let rec h_tail_rec f xs j acc =
144     match xs with
145     | []      -> acc
146     | x::rest -> h_tail_rec f rest (j+1) (acc@[(f j x)])
147
148     let mapi f xs = h f xs 0
149     let mapi_h_trec f xs = h_tail_rec f xs 0
150
151     printfn "Question 2.4: %A - %A \n" (mapi_h_trec (fun i x -> (i,x))
        ["a";"b";"c"] []) (mapi (fun i x -> (i,x)) ["a";"b";"c"])
152
153     (* Problem 3 *)
154
155     (* Question 3.1 -> Assuming that all composite sub-parts of p also has to
        comply, and all subsequent composite sub-parts also comply. *)
156     let rec inv (p:Part) : bool =
157     match p with
158     | S(_) -> true
159     | C(_, []) -> false
160     | C(_,s) -> List.forall inv s
161
162
163     (* Question 3.2 *)
164     let rec depth (p:Part) : int =
165     match p with
166     | S(_) -> 0
167     | C(_,s) -> 1 + List.max (List.map (fun x -> depth x) s)
168
169     (* Question 3.3
170
171     "Name" is a string type. (Declared as string type)
172
173     "Part" is the name of the type being declared, which for F# is given
        after the keyword "type".
174

```

```

D:\development\development_lib\Project\F#\DanielFHaugeS201186.fs 5
175 "S" is one of valid union cases of Part. Part is a discriminated union ↗
    type, a type which may contain a set of different kind of types, ie. ↗
    The full set of all possible values within each union case.
176 S consists of Name which is a string type, hence S is the union case ↗
    where Part is just a simple part.
177
178 "*" indicating a relation, said simply: making a record or tuple.
179
180 "list" is a keyword for indicating a list type. ex. string list.
181
182 "C" is the other valid union case of Part. The valid type for C is a ↗
    tuple consisting of Name in the first value and a list of Parts in ↗
    the second value.
183 Part is recursive in nature as the union case C contain the same type ↗
    in the form of a Part list.
184
185 *)
186
187
188 (* Question 3.4 Assuming conditions are atleast 5 different simple parts ↗
    and atleast 4 different composite parts but can have more.*)
189 let specialPart = C("C1",
190                     [C("C2",
191                       [C("C3", [S("S1");S("S2");S("S5");S("S3");S("S7");S ↗
192                             ("S7");S("S7");S("S7");])];S("S9");S("S7");
193                       C("C4", [S("S7");S("S4");S("S5");]);S("S2");S("S2");S ↗
194                             ("S1");S("S15");
195                       S("S5");S("S5");S("S1");S("S15");S("S15");])])
196
197 (* Question 3.5 Assuming all names recursively in p (exluding names of ↗
    composite parts) *)
198 let simpleNames (p:Part) : Set<Name> =
199     let rec simpleNamesRec (p:Part) (acc:Set<Name>) : Set<Name> =
200         match p with
201         | S(n) -> Set.add n acc
202         | C(_,s) -> (Set.unionMany (List.map (fun x-> simpleNamesRec x acc) ↗
203           s))
204     simpleNamesRec p Set.empty
205
206 (* Question 3.6 *)
207
208 let computeOccurences (p:Part) : OccurrenceCount =
209
210     let addOccurences (i:int) (n:Name) (o:OccurrenceCount) : ↗
211         OccurrenceCount =
212         match Map.tryFind n o with
213         | None -> Map.add n i o
214         | Some a -> Map.add n (i+a) (Map.remove n o)

```

```

212
213
214     let mergeOccurrences (o1:OccurrenceCount) (o2:OccurrenceCount) :      ↗
        OccurrenceCount = Map.fold (fun s k v -> addOccurrences v k s) o1 o2
215
216     let rec computeOccurrenceRec (p:Part) (acc:OccurrenceCount) :          ↗
        OccurrenceCount =
217         match p with
218         | S(n) -> addOccurrences 1 n acc
219         | C(n,s) -> mergeOccurrences (addOccurrences 1 n acc) (List.reduce ↗
            (fun a b -> mergeOccurrences a b) (List.map (fun x -> ↗
                computeOccurrenceRec x acc) s))
220
221     computeOccurrenceRec p Map.empty
222
223     printfn "Question 3.1: %A" (inv specialPart)
224     printfn "Question 3.2 & 3.4: depth = %A" (depth specialPart)
225     printfn "Question 3.5: %A" (simpleNames specialPart)
226     printfn "Question 3.6: %A\n" (computeOccurrences specialPart)
227
228
229
230     (* Problem 4 *)
231
232     let rec gC i k =
233         if i=0 then k 0
234         else if i=1 then k 1
235         else gC (i-1) (fun v1 -> gC (i-2) (fun v2 -> k(v1+v2)))
236
237
238     (* Question 4.1 *)
239     let rec g i = if i = 0 then 0 else if i = 1 then 1 else (g (i-1))+(g (i-2)) ↗
240
241     printfn "Question 4.1: %A = %A" (g 15) (gC 15 id)
242
243     (* Question 4.2 *)
244     let seq1 = seq { for i in Seq.initInfinite id do if (i%2 = 0) then yield ↗
        (2*i+1) else yield -(2*i+1) }
245     printfn "Question 4.2: %A" seq1
246
247
248     (* Question 4.3 *)
249     let seq1float = seq { for i in seq1 do yield float i}
250     let seq2 = seq { for i in seq1float do yield (1.0/i)}
251     printfn "Question 4.3: %A" seq2
252
253
254     (* Question 4.4 *)

```

```
255     let seq3 = seq { for i in Seq.initInfinite (fun x->x+1) do yield Seq.sum  
                        (Seq.take i seq2) }  
256     printfn "Question 4.4: %A" seq3  
257  
258     0
```