```fsharp
//1.1
let rec repeat s = function
    | 0 -> ""
    | n when n > 0 -> s + (repeat s (n-1))
    | _ -> failwith "negative input";;

//1.2
let f s1 s2 = function
    | 0 -> ""
    | n when (n > 0 && n%2=0) ->
        repeat (s1 + "\n" + s2 + "\n") (n/2)
    | n when (n > 0 && n%2=1) ->
        (repeat (s1 + "\n" + s2 + "\n") (n/2)) + s1
    | _ -> failwith "negative input";;

f "XO" "OX" 3

//1.3
let viz m n =
    if (m > 0 && n > 0) then f (repeat "XO" m) (repeat "OX" m) n
    else failwith "something went wrong";;

"\n" + viz 5 6;;

//1.4.1 - tail recursive
let rec repeatA s a = function
    | 0 -> a
    | n when n > 0 -> (repeatA s (s + "" + a) (n-1))
    | _ -> failwith "negative input";;

repeatA "abc" "" 5

//1.4.2 - continuation
let rec repeatC s k = function
    | 0 -> k ""
    | n when n > 0 -> (repeatC s (fun v -> v + "" + k s) (n-1))
    | _ -> failwith "negative input";;

repeatC "abc" id 5

//2.1
let mixMap f xs ys = List.map f (List.zip xs ys);;

//2.2
let unMixMax f g lst =
    let (xs,ys) = List.unzip lst
    (f xs g ys);;

//('a * 'b ->  'c) -> 'a list -> 'b list -> 'c list
//('a -> 'b) -> ('c -> 'd) -> ('a * 'c) list -> ('b list) * ('d list)

//3.1
```

```
53  type Tree<'a> = Lf | Br of Tree<'a> * 'a * Tree<'a>;;
54  let t = Br(Br(Br(Lf,1,Lf),2,Br(Lf,3,Lf)),4,Br(Br(Lf,5,Lf),6,Br(Lf,7,Lf)));;
55
56  let rec reflect = function
57      | Lf -> Lf
58      | Br (tl,n,tr) -> Br ((reflect tr), n, (reflect tl));;
59
60  reflect t
61
62  //3.2
63  let rec travel a = function
64      | Lf -> Lf,a
65      | Br (tl,n,tr) ->   let anew = a+n
66                          let (tlnew, l) = travel anew tl
67                          let (trnew, m) = travel l tr
68                          Br (tlnew, m, trnew), m;;
69
70  let accumulate tree =
71      let (res, acc) = travel 0 tree
72      res;;
73
74  accumulate t
75
76  //3.3
77  //k: 'a -> Tree<'a> -> Tree<'a>
78  //'a can be either int or float.
79  //produces a new tree where the number of each node
80  //is multiplied by i^d, where d is the depth of the node.
81
82
83  //q: int -> T<'a> -> 'a list
84  //q uses h to create a list of all the nodes in the input tree,
85  //traversing the tree in-order.
86
87  let rec k i tree =
88      match tree with
89      | Lf -> Lf
90      | Br(tl,a,tr) -> Br(k (i*i) tl, i*a, k (i*i) tr);;
91
92  let rec h n m =
93      function
94      | Br(tl,a,tr) when n=m -> h n 1 tl @ [a] @ h n 1 tr
95      | Br(tl,_,tr) -> h n (m+1) tl @ h n (m+1) tr
96      | Lf -> []
97
98  let q n t = h n n t;;
99
100 k 2 t
101 q 2 t
102
103
104
```

```
105 //4.1
106 type CourseNo = int
107 type Title = string
108 type ECTS = int
109 type CourseDesc = Title * ECTS
110 type CourseBase = Map<CourseNo, CourseDesc>
111
112 let isValidCourseDesc = function
113     | (cn, ects)-> (ects%5 = 0) && (ects >= 5);
114     | _ -> false;;
115
116 //4.2
117 let isValidCourseBase cb = Map.forall (fun cn cd -> isValidCourseDesc cd) cb;;
118
119 //4.3
120 let disjoint s1 s2 = Set.isEmpty (Set.intersect s1 s2);;
121
122 //4.4
123 let sumECTS cs cb =
124     let found = Map.filter (fun cn cd -> Set.contains cs cn) cb
125     Map.fold (fun acc cn (title,ects) -> acc+ects) 0 found;;
126
127 //4.5
128 type Mandatory = Set<CourseNo>
129 type Optional = Set<CourseNo>
130 type CourseGroup = Mandatory * Optional
131
132 let isValidCourseGroup cb (man,opt) =
133     let manECTS = sumECTS man cb
134     let optECTS = sumECTS opt cb
135     (disjoint man opt) &&
136     (manECTS <= 45) &&
137     (manECTS = 45 && Set.isEmpty opt) &&
138     (manECTS + optECTS) >= 45;;
139
140 //4.6
141 type BasicNaturalScience = CourseGroup
142 type TechnologicalCore = CourseGroup
143 type ProjectProfessionalSkill = CourseGroup
144 type Elective = CourseNo -> bool
145 type FlagModel = BasicNaturalScience * TechnologicalCore
146                     * ProjectProfessionalSkill * Elective
147 type CoursePlan = Set<CourseNo>
148
149 let CG_Disjoint bns tc pps =
150     let bnsUnion = Set.union (fst bns) (snd bns)
151     let tcUnion =  Set.union (fst tc) (snd tc)
152     let ppsUnion =  Set.union (fst pps) (snd pps)
153     (disjoint bnsUnion tcUnion) &&
154     (disjoint bnsUnion ppsUnion) &&
155     (disjoint tcUnion ppsUnion);;
156
```

```
157 let All_Elective bns tc pps ep =
158     Set.forall ep (fst bns) && Set.forall ep (snd bns) &&
159     Set.forall ep (fst tc) && Set.forall ep (snd tc) &&
160     Set.forall ep (fst pps) && Set.forall ep (snd pps);;
161
162 let isValid (bns,tc,pps,ep) cb =
163     isValidCourseGroup cb bns &&
164     isValidCourseGroup cb tc &&
165     isValidCourseGroup cb pps &&
166     CG_Disjoint bns tc pps &&
167     All_Elective bns tc pps ep;;
168
169 //4.7
170 let checkPlan cp fm cb =
171     isValid fm cb && //isvalid already checks than all courses are elective, and ↵
          that each courseset is at least 45 points
172     (sumECTS cp cb = 180);;
173
174
```