Aufgabe 1 Statische Semantik (__ /20 Punkte)

a) Füllen Sie die Lücken, sodass sich **gültige Aussagen der Statischen Semantik** ergeben. Alle vorkommenden Zahlen sind vom Typ `Nat`. Sie müssen **keine Beweisbäume** angeben!

Hinweis: In einigen Teilaufgaben gibt es mehrere richtige Lösungen. Die angegebene Lösung ist nur ein Beispiel.

i) $\emptyset \vdash$ **true** : _____ **__ /10**

$\emptyset \vdash$ **true** : Bool

ii) $\emptyset \vdash$ _____ : Bool

$\emptyset \vdash$ **true** : Bool

iii) $\emptyset \vdash$ **if true then** _____ **else** _____ : Nat

$\emptyset \vdash$ **if true then** 1 **else** 2 : Nat

iv) $\{ \texttt{f} \mapsto \text{Bool} \to \text{Nat} \} \vdash$ **let** g (x: Bool): Nat = f x : _____

$\{ \texttt{f} \mapsto \text{Bool} \to \text{Nat} \} \vdash$ **let** g (x: Bool): Nat = f x : $\{ \texttt{g} \mapsto \text{Bool} \to \text{Nat} \}$

v) $\emptyset \vdash$ **let rec** f (x: Nat): Bool = f x **in** _____ : Nat → Bool

$\emptyset \vdash$ **let rec** f (x: Nat): Bool = f x **in** f : Nat → Bool

b) Geben Sie einen **vollständigen Beweisbaum** für folgende **Aussage der Statischen Semantik** an: **__ /10**

$$\emptyset \vdash \texttt{let f (x: Bool): Nat = if x then 47 else 11} : \{\texttt{f} \mapsto \text{Bool} \to \text{Nat}\}$$

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{x} \mapsto \text{Bool}\} \vdash \texttt{x} : \text{Bool}} \quad \dfrac{}{\{\texttt{x} \mapsto \text{Bool}\} \vdash 47 : \text{Nat}} \quad \dfrac{}{\{\texttt{x} \mapsto \text{Bool}\} \vdash 11 : \text{Nat}}}{\{\texttt{x} \mapsto \text{Bool}\} \vdash \texttt{if x then 47 else 11} : \text{Nat}}}{\emptyset \vdash \texttt{let f (x: Bool): Nat = if x then 47 else 11} : \{\texttt{f} \mapsto \text{Bool} \to \text{Nat}\}}$$

Aufgabe 2 Dynamische Semantik (__ /20 Punkte)

a) Füllen Sie die Lücken, sodass die Ausdrücke **typkorrekt** sind und sich **gültige Aussagen der Dynamischen Semantik** ergeben. Alle vorkommenden Zahlen sind vom Typ `Nat`. Sie müssen **keine Beweisbäume** angeben!

Hinweis: In einigen Teilaufgaben gibt es mehrere richtige Lösungen. Die angegebene Lösung ist nur ein Beispiel.

i) $\emptyset \vdash$ `(if` ____ `then 4 + 7 else 11) + 100` $\Downarrow 111$ **__ /10**

$$\dfrac{\dfrac{\dfrac{}{\texttt{false} \Downarrow \textit{false}} \quad \dfrac{}{11 \Downarrow 11}}{\texttt{if false then 4 + 7 else 11} \Downarrow 11} \quad \dfrac{}{100 \Downarrow 100}}{\texttt{(if false then 4 + 7 else 11) + 100} \Downarrow 111} \qquad \dfrac{\dfrac{\dfrac{}{\texttt{true} \Downarrow \textit{true}} \quad \dfrac{\dfrac{}{4 \Downarrow 4} \quad \dfrac{}{7 \Downarrow 7}}{\texttt{4 + 7} \Downarrow 11}}{\texttt{if true then 4 + 7 else 11} \Downarrow 11} \quad \dfrac{}{100 \Downarrow 100}}{\texttt{(if true then 4 + 7 else 11) + 100} \Downarrow 111}$$

ii) `fst (snd (fst ((1, (2, 3)), 4)))` $\Downarrow$ ____

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{1 \Downarrow 1} \quad \dfrac{\dfrac{}{2 \Downarrow 2} \quad \dfrac{}{3 \Downarrow 3}}{\texttt{(2, 3)} \Downarrow (2,3)}}{\texttt{(1, (2, 3))} \Downarrow (1,(2,3))} \quad \dfrac{}{4 \Downarrow 4}}{\texttt{((1, (2, 3)), 4)} \Downarrow ((1,(2,3)),4)}}{\texttt{fst ((1, (2, 3)), 4)} \Downarrow (1,(2,3))}}{\texttt{snd (fst ((1, (2, 3)), 4))} \Downarrow (2,3)}}{\texttt{fst (snd (fst ((1, (2, 3)), 4)))} \Downarrow \underline{2}}$$

iii) $\emptyset \vdash$ `let rec f (x: Nat): Bool = if x = 0 then true else f (x - 1)` $\Downarrow$ ____

$$\dfrac{}{\texttt{let rec f (x: Nat): Bool = if x = 0 then true else f (x - 1)} \Downarrow \underline{\{\texttt{f} \mapsto \langle \emptyset, \texttt{f}, \texttt{x}, \texttt{if x = 0 then true else f (x - 1)}\rangle\}}}$$

Man kann also einfach einen Strich über die ausgefüllte Aussage malen, um Teilaufgabe b zu lösen.

iv) $\{\texttt{f} \mapsto \langle \emptyset, \texttt{x}, \texttt{if}$ ____ `then 47 else 11`$\rangle\} \vdash \texttt{f false} \Downarrow 11$

Mögliche Eintragungen in die Lücke: `x` oder `false`
Definiere aus Platzgründen: $\delta := \{\texttt{f} \mapsto \langle \emptyset, \texttt{x}, \texttt{if x then 47 else 11}\rangle\}$

$$\dfrac{\dfrac{}{\delta \vdash \texttt{f} \Downarrow \langle \emptyset, \texttt{x}, \texttt{if x then 47 else 11}\rangle} \quad \dfrac{}{\delta \vdash \texttt{false} \Downarrow \textit{false}} \quad \dfrac{\dfrac{}{\{\texttt{x} \mapsto \textit{false}\} \vdash \texttt{x} \Downarrow \textit{false}} \quad \dfrac{}{\{\texttt{x} \mapsto \textit{false}\} \vdash 11 \Downarrow 11}}{\{\texttt{x} \mapsto \textit{false}\} \vdash \texttt{if x then 47 else 11} \Downarrow 11}}{\delta \vdash \texttt{f false} \Downarrow 11}$$

v) $\emptyset \vdash$ `fun (x: Nat) → fun (y: Nat) →` ____ $\Downarrow \langle \emptyset, \texttt{x}, \texttt{fun (y: Nat) → x + y}\rangle$

$$\dfrac{}{\emptyset \vdash \texttt{fun (x: Nat) → fun (y: Nat) →} \underline{\texttt{x + y}} \Downarrow \langle \emptyset, \texttt{x}, \texttt{fun (y: Nat) → x + y}\rangle}$$

Man kann also einfach einen Strich über die ausgefüllte Aussage malen, um Teilaufgabe b zu lösen.

b) **Wählen Sie** von den fünf Aussagen aus Teilaufgabe a) **eine aus** und geben Sie **nur für diese Aussage** einen **vollständigen Beweisbaum** mit den Regeln der Dynamischen Semantik an.
Tipp: Die Bäume für die fünf Aussagen werden unterschiedlich groß. Machen Sie sich kurz Gedanken darüber, welche Aussage zu einem kleinen Baum führt. **__ /10**
Sie können Ihren Beweisbaum oben in die Zwischenräume schreiben und dabei die eigentliche Aussage als Teil des Baumes verwenden. Falls Ihnen der Platz nicht ausreicht können Sie die Rückseite benutzen.

Aufgabe 3 Entwurfsmuster (__ /20 Punkte)

Lösen Sie diese Aufgabe **funktional**, d. h. `mutable` und `ref` dürfen in Ihrer Lösung nicht vorkommen. Verwenden Sie **keine Bibliotheksfunktionen!**

a) Schreiben Sie eine Funktion `mappedInterval<'a>: (Nat -> 'a) -> Nat -> List<'a>`, die eine Funktion `f` und eine natürliche Zahl `n` nimmt und eine Liste der Form `[f n; f (n - 1); ...; f 0]` zurückgibt. Gehen Sie **strikt nach Peano Entwurfsmuster** vor.

Beispiele:

```
mappedInterval (fun n -> n + 1)     0 = [1]
mappedInterval (fun n -> n + 1)     1 = [2; 1]
mappedInterval (fun n -> n % 2 = 0) 3 = [false; true; false; true]
```

__ /5

```
let rec mappedInterval<'a> (f: Nat -> 'a) (n: Nat): List<'a> =
    if n = 0N then [f 0N]
    else f n :: mappedInterval f (n - 1N)
```

b) Schreiben Sie die Funktion `contains7: Nat -> Bool`, die prüft, ob die Ziffer 7 in der gegebenen Zahl vorkommt. Orientieren Sie sich am **Leibniz Entwurfsmuster**.

Beispiele:

```
contains7 815 = false                    contains7 4711 = true
```

__ /5

```
let rec contains7 (n: Nat): Bool =
    if n = 0N then false
    else (n % 10N = 7N) || contains7 (n / 10N)
```

Für die nächsten beiden Teilaufgaben verwenden wir folgenden Typen für Binärbäume:

```
type Tree<'a> = | Leaf of 'a | Node of Tree<'a> * Tree<'a>
```

Darüber hinaus verwenden wir folgende Funktionsdefinition:

```
let foldTree<'a, 'b> (f: 'b -> 'b -> 'b) (g: 'a -> 'b): Tree<'a> -> 'b =
    let rec helper (t: Tree<'a>): 'b =
        match t with
        | Leaf x -> g x
        | Node (l, r) -> f (helper l) (helper r)
    in helper
```

Die Funktion `foldTree` abstrahiert das Struktur-Entwurfsmuster für Bäume, ähnlich zu `peano-pattern` aus der Vorlesung. Sie nimmt zwei Funktionen `f` und `g`. Die Funktion `g` wird im Basisfall aufgerufen, während `f` die Rückgabewerte der rekursiven Aufrufe verarbeitet.

c) Schreiben Sie eine Funktion `inorder<'a>: Tree<'a> -> List<'a>`, die die Inorder-Traversierung eines Baums zurückgibt. Verwenden Sie hierfür die Funktion `foldTree`.

Hinweis: Die Inorder-Traversierung besucht für jeden Knoten zuerst den linken und dann den rechten Teilbaum.

Beispiele:
```
inorder (Leaf 5) = [5]
inorder (Node (Leaf 5, Leaf 4)) = [5; 4]
inorder (Node (Node (Leaf 4, Leaf 7), Node (Leaf 1, Leaf 1))) = [4; 7; 1; 1]
```

```
let inorder<'a> : Tree<'a> -> List<'a> =
    foldTree (@) (fun x -> [x])
```

__/5

d) Schreiben Sie eine Funktion `mirror<'a>: Tree<'a> -> Tree<'a>`, die einen Baum spiegelt. Verwenden Sie hierfür die Funktion `foldTree`.

Beispiele:
```
mirror (Leaf 5) = Leaf 5
mirror (Node (Leaf 5, Leaf 4)) = Node (Leaf 4, Leaf 5)
mirror (Node (Node (Leaf 4, Leaf 7), Node (Leaf 1, Leaf 1))) =
        Node (Node (Leaf 1, Leaf 1), Node (Leaf 7, Leaf 4))
```

```
let mirror<'a> : Tree<'a> -> Tree<'a> =
    foldTree (fun l r -> Node (r, l)) (Leaf)
                                    // (fun x -> Leaf x)
```

__/5

Aufgabe 4 Rekursion auf Listen (__ /20 Punkte)

Lösen Sie diese Aufgabe **funktional**, d. h. `mutable` und `ref` dürfen in Ihrer Lösung nicht vorkommen. Verwenden Sie **keine Bibliotheksfunktionen!**

a) Wir betrachten die aus der Vorlesung bekannte Peano-Darstellung für natürliche Zahlen:

```
type Peano =
    | Zero
    | Succ of Peano
```

Schreiben Sie eine Funktion `take<'a>: Peano -> List<'a> -> List<'a>`, die eine natürliche Zahl `n` in Peano-Darstellung sowie eine Liste `xs` nimmt und die ersten `n` Elemente der Liste zurückgibt. Sollte `xs` weniger als `n` Elemente haben, dann soll die komplette Liste `xs` zurückgegeben werden.

Beispiele:
```
take (Succ Zero)              [4; 7; 1; 1] = [4]
take (Succ (Succ (Succ Zero))) ['a'; 'b']   = ['a'; 'b']
```

__ /5

```
let rec take<'a> (n: Peano) (xs: List<'a>): List<'a> =
    match (n, xs) with
    | (Zero  , _        ) -> []
    | (_     , []       ) -> []
    | (Succ m, x :: tail) -> x :: take m tail
```

b) Schreiben Sie eine Funktion `reverse<'a>: List<'a> -> List<'a>`, die die gegebene Liste spiegelt.

Beispiele:
```
reverse []         = []                    reverse [1; 2; 1]    = [1; 2; 1]
reverse ['a'; 'b'] = ['b'; 'a']            reverse [4; 7; 1; 1] = [1; 1; 7; 4]
```

__ /5

```
// Naive Lösung mit quadratischer Laufzeit
let rec reverse<'a> (xs: List<'a>): List<'a> =
    match xs with
    | []        -> []
    | x :: rest -> reverse rest @ [x]

// Lösung mit linearer Laufzeit
let reverse2<'a> (xs: List<'a>): List<'a> =
    let rec reverseCat (xs: List<'a>) (ys: List<'a>): List<'a> =
        match xs with
        | []        -> ys
        | x :: rest -> reverseCat rest (x :: ys)
    reverseCat xs []
```

c) Betrachten Sie folgende Implementierung für eine Funktion `isPalindrom: List<Nat> -> Bool`, die überprüfen soll, ob die gegebene Liste ein Palindrom ist. Eine Liste `xs` ist ein Palindrom, wenn `xs = reverse xs` ist. Hierbei ist `reverse` die Funktion aus der vorherigen Teilaufgabe, sie spiegelt die Liste.

```
1 let rec isPalindrom (xs: List<Nat>): Bool =
2     match xs with
3     | []  -> true
4     | [_] -> true
5     | head :: tail  ->
6         match reverse xs with
7         | last :: ys -> head = last && isPalindrom ys
8         | _ -> false
```

Die gegebene Implementierung ist fehlerhaft, wie folgende Beispiele zeigen:

Aufruf	**Gewünschtes Ergebnis**	**Tatsächliches Ergebnis**
`isPalindrom []`	true	true
`isPalindrom [1]`	true	true
`isPalindrom [1; 1]`	true	true
`isPalindrom [1; 2; 1]`	true	false (Fehler!)
`isPalindrom [1; 2; 3]`	false	false
`isPalindrom [1; 2; 2; 1]`	true	false (Fehler!)

Finden Sie den Fehler in der gegebenen Implementierung. Geben Sie an, in welcher Zeile der Fehler ist und wie diese Zeile richtig lauten muss. Erklären Sie in maximal 50 Wörtern, wieso Ihre Korrektur den Fehler behebt.

Es ist nicht erlaubt, die gesamte Implementierung zu ersetzen durch zum Beispiel `xs = reverse xs`. Sie dürfen nur eine einzelne Zeile korrigieren!

Zeile 6 muss wie folgt korrigiert werden: **__ /10**

```
6         match reverse tail with
```

Eine Liste ist ein Palindrom, wenn das erste (`head`) gleich dem letzten (`last`) Element ist und alles dazwischen (sollte `ys` sein) auch ein Palindrom ist. `ys` enthält in der fehlerhaften Implementierung aber zusätzlich `head` nochmal. Mit `tail` statt `xs` verhindern wir die Duplizierung von `head`.

Aufgabe 5 Endliche Abbildungen (__ /20 Punkte)

Lösen Sie diese Aufgabe **funktional**, d. h. `mutable` und `ref` dürfen in Ihrer Lösung nicht vorkommen. Verwenden Sie **keine Bibliotheksfunktionen!**

Wir betrachten folgenden Typen, um endliche Abbildungen zu modellieren:

```
type Map<'k, 'v> = 'k -> Option<'v>
```

Ist also eine endliche Abbildung φ gegeben und `key` $\in$ dom(φ), dann gibt die zugehörige Abbildung vom Typ `Map<'k, 'v>` den Wert φ(`key`) zurück. Hierbei ist `'k` der Typ des Definitions- und `'v` der Typ des Wertebereichs. Ist `key: 'k` gegeben mit `key` $\notin$ dom(φ), gibt die zugehörige Abbildung `None` zurück.

Hinweis: Sie dürfen davon ausgehen, dass die Schlüssel vom Typ `'k` mit dem Gleichheitsoperator '=' vergleichbar sind.

a) Geben Sie die leere Abbildung $\emptyset$ an:

__ /5

```
let empty<'k,'v> : Map<'k, 'v> =
    fun (_: 'k) -> None
```

b) Schreiben Sie eine Funktion `insert<'k, 'v>: 'k * 'v -> Map<'k, 'v> -> Map<'k, 'v>`, die das gegebene Schlüssel-/Wertepaar in die gegebene endliche Abbildung einfügt.

__ /5

```
let insert<'k,'v when 'k: equality>
  (key: 'k, value: 'v) (map: Map<'k, 'v>): Map<'k, 'v> =
    fun (k: 'k) -> if k = key then Some value else map k
```

c) Schreiben Sie eine Funktion `lookup<'k, 'v>: 'k -> Map<'k, 'v> -> Option<'v>`, die den zu dem gegebenen Schlüssel in der gegebenen endliche Abbildung zugehörigen Wert bestimmt. Wenn der Schlüssel nicht in der Abbildung enthalten ist, soll `None` zurückgegeben werden.

__ /5

```
let lookup<'k, 'v> (key: 'k) (map: Map<'k, 'v>): Option<'v> =
    map key
```

d) Schreiben Sie eine Funktion `comma<'k, 'v>: Map<'k, 'v> -> Map<'k, 'v> -> Map<'k, 'v>`, die den Kommaoperator für endliche Abbildungen implementiert. Wenn ein Schlüssel in beiden endlichen Abbildungen enthalten ist, wird also der zweiten endlichen Abbildung Vorrang gegeben.

__ /5

```
let comma<'k, 'v> (map1: Map<'k, 'v>) (map2: Map<'k, 'v>): Map<'k, 'v> =
    fun (input_key: 'k) ->
        let v = map2 input_key
        match v with
        | None -> map1 input_key
        | Some x -> Some x
```

Aufgabe 6 Dünne Bäume (__ /20 Punkte)

Sie dürfen zur Lösung dieser Aufgabe Bibliotheksfunktionen verwenden.

Gegeben ist der folgende Typ für Bäume:

```
type Tree<'a> = | Leaf | Node of Tree<'a> * Option<'a> * Tree<'a>
```

Ein Knoten kann also ein Element vom Typ `'a` enthalten, muss es aber nicht.

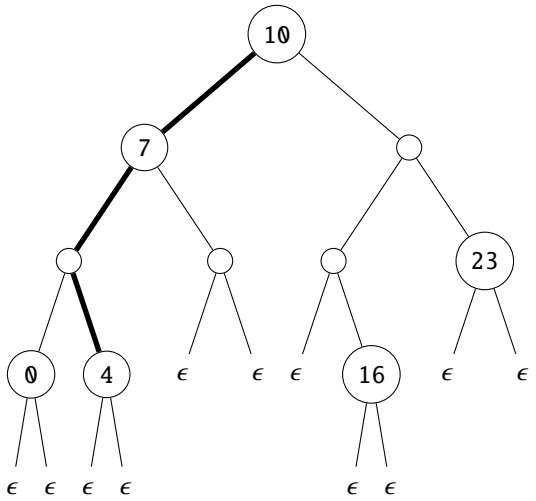
Darüber hinaus ist der folgende Typ für Pfade gegeben:

```
type Dir = | Left | Right

type Path = List<Dir>
```

Ein Pfad ist also eine Liste von Richtungen (`Dir`). Pfade beginnen immer in der Wurzel des Baums.

Beispiel: Rechts sehen Sie den Code zu dem hier dargestellten Baum und dem hervorgehobenen Pfad mit den Knoten `Some 10`, `Some 7`, `None`, `Some 4`.




```
let exTree: Tree<Nat> =
  Node (
    Node (
      Node (
        Node (Leaf, Some 0, Leaf),
        None,
        Node (Leaf, Some 4, Leaf)
      ),
      Some 7,
      Node (Leaf, None, Leaf)
    ),
    Some 10,
    Node (
      Node (
        Leaf,
        None,
        Node (Leaf, Some 16, Leaf)
      ),
      None,
      Node (Leaf, Some 23, Leaf)
    )
  )

let exPath: Path = [Left; Left; Right]
```

a) Schreiben Sie eine Funktion `toList<'a>: Tree<'a> -> List<'a * Path>`, die einen Baum in eine Liste von Knotenwerten mit den dazugehörigen Pfaden überführt. Die Reihenfolge der Elemente in der Liste ist dabei nicht relevant.

__/10

Beispiele:

```
toList Leaf = []
toList (Node (Leaf, Some 5, Leaf)) = [(5, [])]
toList (Node (Leaf, None, Leaf)) = []
toList exTree = [(0, [Left; Left; Left]); (4, [Left; Left; Right]); (7, [Left]);
                 (10, []); (16, [Right; Left; Right]); (23, [Right; Right])]
```

```
let rec toList<'a> (t: Tree<'a>): List<'a * Path> =
  let addDir d = List.map (fun (x, p) -> (x, d :: p))
  match t with
  | Leaf           -> []
  | Node (l, x, r) -> match x with
                      | None   -> []
                      | Some x -> [(x, [])]
                      @ addDir Left  (toList l)
                      @ addDir Right (toList r)

// Alternativ: Pfad in zusätzlichem Parameter aufbauen
let toList'<'a> (t: Tree<'a>): List<'a * Path> =
  let rec h (t: Tree<'a>) (p: Path): List<'a * Path> =
    match t with
    | Leaf -> []
    | Node (l, x, r) -> match x with
                        | None   -> []
                        | Some x -> [(x, p)]
                        @ h l (p @ [Left])
                        @ h r (p @ [Right])
  h t []
```

b) Schreiben Sie eine Funktion `fromList<'a>: List<'a * Path> -> Tree<'a>`, die eine Liste von Paaren von Werten und Pfaden nimmt und einen Baum zurückgibt, der an den Stellen der Pfade die Werte enthält. Alle anderen Knoten des Baums sind `None`. Sie dürfen davon ausgehen, dass jeder Pfad nur einmal in der Liste enthalten ist. **Erklären Sie kurz Ihren Lösungsweg in ein bis zwei Sätzen, zum Beispiel als Kommentar im Code**.

__/10

Beispiele:

```
fromList [] = Leaf
fromList [(5, [])] = Node (Leaf, Some 5, Leaf)
fromList [(5, [Left]); (6, [])] = Node (Node (Leaf, Some 5, Leaf), Some 6, Leaf)
```

Erster Lösungsweg: Nach der Struktur der Eingabeliste Wir starten mit einem Leaf und fügen nacheinander die einzelnen Einträge aus der Liste in den Baum ein. Immer wenn wir beim Einfügen in den Baum auf ein Leaf treffen, erweitern wir den Baum an dieser Stelle vorerst um ein Node ohne Element und mit leeren Teilbäumen. Die Einfügeoperation wiederum arbeitet rekursiv auf dem durch das erste Pfadelement vorgegebenen Teilbaum.

```
let fromList<'a> (xs: List<'a * Path>): Tree<'a> =
  let rec insert (tree: Tree<'a>) (elem: 'a, path: Path): Tree<'a> =
    match tree with
    | Leaf           -> insert (Node (Leaf, None, Leaf)) (elem, path)
    | Node (l, x, r) -> match path with
                        | []         -> Node (l, Some elem, r)
                        | Left  :: p -> Node (insert l (elem, p), x, r)
                        | Right :: p -> Node (l, x, insert r (elem, p))
  List.fold insert Leaf xs
```

Zweiter Lösungsweg: Nach der Struktur des zu erstellenden Baumes Wenn die Eingabeliste nicht leer ist, dann splitten wir sie in drei Teile, nämlich eine Teilliste für alle Einträge deren Pfad mit `Left` beginnt, eine Teilliste für alle Einträge deren Pfad mit `Right` beginnt, sowie das optionale Element mit dem leeren Pfad für den Wurzelknoten. Wir erstellen rekursiv die beiden Teilbäume und können dann den Baum zusammensetzen.

```
type State<'a> = {
  root:   Option<'a>
  lefts:  List<'a * List<Dir>>
  rights: List<'a * List<Dir>>
}

let rec fromList2<'a> (xs: List<'a * Path>): Tree<'a> =
  match xs with
  | [] -> Leaf
  | _  ->
    let state =
      List.fold (fun state (elem, path) ->
          match path with
          | []         -> { state with root   = Some elem                }
          | Left  :: p -> { state with lefts  = (elem, p) :: state.lefts  }
          | Right :: p -> { state with rights = (elem, p) :: state.rights }
      ) { root = None; lefts = []; rights = [] } xs
    Node (fromList2 state.lefts, state.root, fromList2 state.rights)
```

Wir können auch ohne die Typdefinition auskommen, wenn wir ein Tupel statt dem Record verwenden:

```
let rec fromList3<'a> (xs: List<'a * Path>): Tree<'a> =
    let (root, lefts, rights) =
      List.fold (fun (root, lefts, rights) (elem, path) ->
          match path with
          | []         -> (Some elem, lefts, rights)
          | Left  :: p -> (root, (elem, p) :: lefts, rights)
          | Right :: p -> (root, lefts, (elem, p) :: rights)
      ) (None, [], []) xs
    Node (fromList3 lefts, root, fromList3 rights)
```

Alternativ kann man wie in Aufgabe 2 von Übungsblatt 6 vorgehen. Dort hatten wir gesehen, dass man rekursive Datatentypen produzieren kann, indem man eine Funktion iteriert, die jeweils eine Rekursionsebene des Typs produziert. Wir definieren dazu hier den Typen `Treeish` und die Funktion `letIt`. Wir verwenden ein disjunktives Muster um im Basisfall den Akkumulator mit `([], None, [])` zu „initialisieren“. Bei den Teillisten entfernen wir jeweils das erste Pfadelement, und geben die Restlisten als neue „seed“ Werte zurück.

N.B.: Die rekursion *nach* `Tree` wird über `letIt` gemacht, die *von* `List` über `List.fold`, sodass die eigentliche Business Logic komplett nicht-rekursiv dasteht!

```
type Treeish<'a,'b> = | SLeaf | SNode of 'b * Option<'a> * 'b
let rec letIt (grow : 'b -> Treeish<'a,'b>) (seed : 'b) : Tree<'a> =
    match grow seed with
    | SLeaf -> Leaf
    | SNode(seedL,x,seedR) -> Node(letIt grow seedL, x, letIt grow seedR)

let part<'a> (res : Treeish<'a, List<'a * Path>>) ((elem, path) : 'a * Path)
  : Treeish<'a, List<'a * Path>> =
    match (res                      , ([],    None,     [])) with
        | (SLeaf                    , (lefts, root, rights))
        | (SNode(lefts, root, rights),  _                  ) ->
            match path with
            | []         -> SNode(lefts, Some elem, rights)
            | Left  :: p -> SNode((elem,p) :: lefts, root, rights)
            | Right :: p -> SNode(lefts, root, (elem,p) :: rights)

let partition<'a> : List<'a * Path> -> Treeish<'a, List<'a * Path>> =
    List.fold part SLeaf

let fromList4<'a> (xs: List<'a * Path>): Tree<'a> = letIt partition xs
```