Prof. Dr. Ralf Hinze
Cass Alexandru, M.Sc.
Alexander Dinges, M.Sc.
Sebastian Schloßer, M.Sc.

RPTU
Fachbereich Informatik
AG Programmiersprachen

Lösungshinweise/-vorschläge zum Übungsblatt 6: Grundlagen der Programmierung (WS 2024/25)

Probeklausur Sie können sich ab sofort im ExClaim System für die Probeklausur am 17.12.2024 anmelden. Klicken Sie dazu unten auf der "GdP24" Seite bei der Probeklausur auf den Button "anmelden". Die Anmeldung schließt am 11.12.2024 um 23:59 Uhr.

Aufgabe 1 Binärbäume (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie üben mit komplexeren rekursiven Varianten zu programmieren. Sie können sich an den Vorlesungsfolien 512 bis 526 sowie am Skript Kapitel 5.2.3 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Tree.fs` *aus der Vorlage* `Aufgabe-6-1.zip`.

Bisher sind Ihnen in erster Linie Listen als Beispiel für rekursive Variantentypen begegnet. Es ist jedoch auch möglich mit Hilfe rekursiver Varianten komplexere Datenstrukturen zu konstruieren. Wir werden in dieser Aufgabe exemplarisch den Typ der Bäume betrachten:

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

Ein Baum (`Tree`) besteht entweder aus einem Blatt (`Leaf`) oder aus einem Knoten (`Node`), welcher einen linken Teilbaum, ein Element und einen rechten Teilbaum hat.

Abgesehen von der Struktur der Konstruktoren stellen wir in dieser Aufgabe keine weiteren Anforderungen an den Baum. Sie haben in der Vorlesung bereits Bäume kennengelernt, die durch Hinzunahme bestimmter Invarianten nützliche Eigenschaften erhalten, mit deren Hilfe sich z. B. Suchalgorithmen effizient implementieren lassen.

Bei den Teilaufgaben verwenden wir folgenden Beispielbaum:

```
let ex = Node (Node (Leaf, 1N, (Node (Leaf, 2N, Leaf))), 3N, (Node (Leaf, 4N, Leaf)))
```

a) Schreiben Sie eine Funktion `countLeaves`, welche die Anzahl der Blätter in einem Baum zurückgibt.

Beispiele:

```
countLeaves Leaf = 1N                          countLeaves ex = 5N
```

```
let rec countLeaves<'a> (t: Tree<'a>): Nat =
    match t with
    | Leaf -> 1N
    | Node (l, _, r) -> countLeaves l + countLeaves r
```

Wir folgen dem Struktur Entwurfsmuster: Elemente des Typs `Tree<'a>` können nur mit den Konstruktoren `Leaf` und `Node` konstruiert werden, mit **`match`** führen wir den Musterabgleich durch. Liegt ein Blatt vor, wird `1N` zurückgegeben (hier kann es insbesondere keinen rekursiven Aufruf mehr geben, da ein Blatt nach Konstruktion keine Kindelemente haben kann). Falls ein Knoten vorliegt, erhöht sich die Anzahl der Blätter nicht, wir können uns vorstellen, dass `0N` zur Summe der rekursiven Aufrufe hinzuaddiert wird.

b) Schreiben Sie eine Funktion `height`, welche die Höhe eines Baumes berechnet.

Beispiele:

```
height Leaf = 0N                               height ex = 3N
```

```
let rec height<'a> (t: Tree<'a>): Nat =
    match t with
    | Leaf -> 0N
    | Node (l, _, r) -> 1N + max (height l) (height r)
```

Ein einzelnes Blatt hat die Höhe 0, ein Knoten die Höhe 1. Da die maximale Höhe gesucht ist, berechnen wir, sofern ein Knoten vorliegt, das Maximum der Höhen des linken und rechten Teilbaums.

c) Schreiben Sie eine Funktion `map`, die eine Funktion auf alle Knotenelemente eines Baums anwendet.

Beispiele:

```
map (fun x -> x * 2N) Leaf = Leaf
map (fun x -> x * 2N) ex = Node ( Node (Leaf, 2N, (Node (Leaf, 4N, Leaf)))
                                , 6N, (Node (Leaf, 8N, Leaf)))
```

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

Wenn ein Knoten vorliegt, wenden wir `f` auf das Knotenelement an und rufen `map` rekursiv auf den Teilbäume auf.

Aufgabe 2 Binäre Suchbäume (Einreichaufgabe, 12 Punkte)

Motivation: In dieser Aufgabe sollen Sie üben mit komplexeren rekursiven Varianten zu programmieren. Sie können sich an den Vorlesungsfolien 470 bis 473, 512 bis 525 sowie am Skript Kapitel 5.2.3 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `BSTs.fs` *aus der Vorlage* `Aufgabe-6-2.zip`.

Ein binärer Suchbaum (binary search tree, oder BST) ist eine Datenstruktur in Form eines Binärbaums: Ein BST ist entweder leer oder ein Knoten mit einem Eintrag, einem linken Teilbaum und einem rechten Teilbaum. Die Datenstruktur dient dazu, Elemente geordnet abzuspeichern. Daher muss ein gültiger BST die *BST-Bedingung* erfüllen: Der Eintrag jedes Knotens muss größer/gleich der Einträge seines linken, und kleiner/gleich der Einträge seines rechten Teilbaums sein.

```
type BST<'a> =
    | Empty
    | Node of BST<'a> * 'a * BST<'a>
```

Beispiele für gültige BSTs:

```
let ex1 = Node(Node(Empty,2N,Empty), 2N, Node(Empty,4N,Empty))
let ex2 = Node(Node(Empty,2N,Empty), 3N, Node(Empty,5N,Empty))
let ex3 = Node(Empty, 1N, Node(Empty,1N,Empty))
```

Beispiele für ungültige BSTs:

```
let inv1 = Node(Node(Empty,3N,Empty), 2N, Empty)
let inv2 = Node(Node(Node(Empty,4N,Empty), 2N, Empty), 3N, Empty)
```

a) Schreiben Sie Funktionen `size` und `height` jeweils vom Typ `BST<'a> -> Nat`, die die Größe bzw. Höhe eines BSTs berechnen. Die Größe entspricht der Anzahl an Einträgen. Die Höhe ist die Länge des längsten Pfades von der Wurzel des BSTs bis zu einem leeren Teilbaum.

   Beispiele mit den oben definierten BSTs:

```
size Empty = 0N          size ex3 = 2N          height ex1 = 2N
size ex1 = 3N            height Empty = 0N      height ex3 = 2N
```

```
let rec size<'a> (root: BST<'a>): Nat =
    match root with
    | Empty -> 0N
    | Node (left, _, right) -> 1N + size left + size right

let rec height<'a> (root: BST<'a>): Nat =
    match root with
    | Empty -> 0N
    | Node (left, _, right) -> 1N + max (height left) (height right)
```

b) Schreiben Sie eine Funktion `isBST: BST<'a> -> Bool`, die überprüft ob der gegebene BST die BST-Bedingung erfüllt.

Beispiele:

```
isBST<Nat> Empty = true        isBST ex2 = true        isBST inv1 = false
isBST ex1 = true               isBST ex3 = true        isBST inv2 = false
```

Hinweis: Um zu etablieren, dass das Knotenelement größer/gleich aller Einträge im linken Teilbaum ist, und kleiner/gleich aller Einträge im rechten Teilbaum, reicht es, dass es >= des *größten* Elements des linken, und <= des *kleinsten* Elements des rechten Teilbaums ist, wobei das kleinste Element eines BST das linkeste, und das größte das rechteste ist.

Ein erster naïver Ansatz dies zu berücksichtigen wäre, im Rekursionsschritt mit Hilfsfunktionen immer jeweils diese Extrema zu berechnen. Dies würde jedoch einen in der Baumtiefe quadratischen Faktor zur Komplexität beitragen (warum?).

Ein anderer Ansatz ist, eine Hilfsfunktion zu implementieren, die statt `Bool` auch Informationen über die Extrema eines BST zurückgibt. Dabei stellt der leere Baum einen Spezialfall dar. Falls Sie diesen Ansatz verfolgen wollen, haben wir als Hinweis in `BSTType.fs` einen parametrisierten Typen `Range<'a>` bereitgestellt, der die Extrema eines BST modelliert.

```
type Range<'a> =
    | EmptyR
    | Twixt of 'a * 'a //Zwischen `lo` und `hi`
```

```
let isBST<'a when 'a: comparison> (root: BST<'a>): Bool =
 let rec bounds (root: BST<'a>) : Option<Range<'a>> =
  match root with
  | Empty -> Some(EmptyR)
  | Node(left, p, right) ->
   match (bounds left, bounds right) with
   | Some EmptyR      , Some EmptyR       -> Some(Twixt(p, p))
   | Some EmptyR      , Some(Twixt(c, d)) -> if p <= c then Some(Twixt(p, d)) else None
   | Some(Twixt(a, b)), Some EmptyR       -> if b <= p then Some(Twixt(a, p)) else None
   | Some(Twixt(a, b)), Some(Twixt(c, d)) ->
       if b <= p && p <= c then Some(Twixt(a, d)) else None
   | _                                    -> None
 in match bounds root with
    | None -> false
    | _    -> true
```

c) Schreiben Sie eine Funktion `deleteMin: BST<'a> -> Option<'a * BST<'a>>`, die aus einem BST das kleinste Element entfernt und einen gültigen BST zurückgibt, der aus den restlichen Elementen besteht. Ist der BST leer, soll der leere BST wieder zurückgegeben werden. Da der BST leer sein kann, verwenden wir wieder den `Option`-Typ. Das heißt, wenn der BST leer ist, soll `None` zurückgegeben werden.

Sie können davon ausgehen, dass der gegebene Baum die BST-Bedingung erfüllt.

Beispiel: `deleteMin ex2 = Some (2N, Node (Empty, 3N, Node (Empty, 5N, Empty)))`

```
let rec deleteMin<'a> (root: BST<'a>): Option<'a * BST<'a>> =
    match root with
    | Empty -> None
    | Node(l, x, r) ->
        match deleteMin l with
        | None -> Some (x, r) // l = Empty
        | Some(a, lMinusA) -> Some(a, Node(lMinusA, x, r))
```

d) Schreiben Sie eine Funktion `partition : List<'a> -> BSTreeish<'a,List<'a>>`, die die Elemente einer Liste um das *letzte* Element als Pivot teilt. Der Typ `BSTreeish<'a,'b>` ähnelt dem Typ `BST<'a>`, nur dass in den rekursiven Positionen statt rekursiv `BST<'a>` der Typparameter `'b` vorkommt.

```
type BSTreeish<'a, 'b> =
    | Emptyish
    | Nodeish of 'b * 'a * 'b
```

Beispiele:

```
partition [1N;1N;7N;4N] = Nodeish ([1N; 1N], 4N, [7N])
partition [5N; 16N; 1N; 4N; 2N; 4N] = Nodeish ([1N; 4N; 2N], 4N, [5N; 16N])
partition [1N;1N;1N] = Nodeish ([1N; 1N], 1N, [])
partition<Nat> [] = Emptyish
```

```
let rec partition<'a when 'a: comparison> (xs: List<'a>) : BSTreeish<'a, List<'a
    >> =
    match xs with
    | [] -> Emptyish
    | x :: xs ->
        match partition xs with
        | Emptyish -> Nodeish([], x, [])
        | Nodeish(smaller, p, greater) ->
            if x <= p then
                Nodeish(x :: smaller, p, greater)
            else
                Nodeish(smaller, p, x :: greater)
```

e) Schreiben Sie eine Funktion `letIt (grow : 'b -> BSTreeish<'a,'b>) (seed : 'b) : BST<'a>`, die einen Baum produziert, indem sie die Funktion `grow` auf `seed` und dann sukzessive auf die neuen „seed“ Werte anwendet, bit `Emptyish` prozuziert wird.

Beispiele:

```
letIt partition [2N;3N;1N;4N] =
  Node (Node (Empty, 1N, Node (Node (Empty, 2N, Empty), 3N, Empty)), 4N, Empty)
letIt (fun n -> if n = 0N then Emptyish else Nodeish(n - 1N, n, n - 2N)) 3N =
  Node(Node (Node (Empty, 1N, Empty), 2N, Empty), 3N, Node (Empty, 1N, Empty))
```

```
let rec letIt (grow : 'b -> BSTreeish<'a,'b>) (seed : 'b) : BST<'a> =
    match grow seed with
    | Emptyish -> Empty
    | Nodeish(seedL,x,seedR) -> Node(letIt grow seedL, x, letIt grow seedR)
```

f) Schreiben Sie eine Funktion `toList: BST<'a> -> List<'a>`, die einen gültigen BST nimmt und daraus eine sortierte Liste der Elemente des BSTs erstellt.

Hinweis: Sie können eine sortierte Liste durch sukzessives Entfernen des Minimums (die Funktion `deleteMin` aus Teilaufgabe a) aus einem binären Suchbaum erstellen. Dieses Vorgehen weist Ähnlichkeiten mit der Funktion `trace` aus Blatt 4, Aufgabe 3a), auf.

Beispiele:

```
toList<Nat> Empty = []
```

```
toList ex1 = [2N; 4N; 6N]
toList ex2 = [3N; 5N; 7N]
```

```
//Hilfsfunktion
let rec tracer (grow : 'b -> Option<'a * 'b>) (seed : 'b) : List<'a> =
    match grow seed with
    | None -> []
    | Some(a, seed') -> a :: tracer grow seed'

let rec toList<'a when 'a: comparison> (root: BST<'a>): List<'a> =
    tracer deleteMin root
```

g) Schreiben Sie eine Funktion `quickSort: List<'a> -> List<'a>`, die eine Liste von Elementen nimmt und diese sortiert. Diese soll erst mit hilfe von `letIt` und `partition` einen binären Suchbaum aufbauen, und ihn dann mit der Funktion `toList` der vorherigen Teilaufgabe in eine sortierte Liste umwandeln.

Beispiele: `quicksort<Nat> [] = []`

`quicksort [2N; 3N; 1N; 2N] = [1N; 2N; 2N; 3N]`

```
let quickSort<'a when 'a: comparison> (xs: List<'a>): List<'a> =
    toList (letIt partition xs)
```

Aufgabe 3 Rekursion mit Binärbäumen (Einreichaufgabe, 7 Punkte)

Motivation: Als Teil einer vorlesungsbegleitenden Studie wollen wir untersuchen, welches mentale Modell von Rekursion die Studierenden zu verschiedenen Zeitpunkten im Verlauf des Semesters haben. Bitte bearbeiten Sie diese Aufgabe daher gewissenhaft und ohne fremde Hilfe.

Hinweis: Wenn Sie in Übungsgruppe 1, 3, 5, 7, 9 oder 11 sind, dann steht Ihnen zur Bearbeitung dieser Aufgabe unser Rekursions-Tutor zur Verfügung, siehe Hinweis auf der ersten Seite von Übungsblatt 4.

Betrachten Sie die folgende Definition von Bäumen, bei der natürliche Zahlen in den Blättern gespeichert werden und Knoten sich in zwei Teilbäume aufteilen.

```
type Tree =
  | Leaf of Nat           // Blatt
  | Node of Tree * Tree   // Knoten
```

a) Betrachten Sie die folgende Funktion und beschreiben Sie in Ihren Worten, was die Funktion `f` tut. `height` bezieht sich auf die Teilaufgabe b) der Präsenzaufgabe und gibt die Höhe des übergebenen Baums zurück. Skizzieren Sie einen Baum für den die Funktion **`true`** zurückgibt und einen für den sie **`false`** zurückgibt.

```
let rec f (t: Tree): Bool =
  match t with
  | Leaf _      -> true
  | Node (l, r) ->    (height l - height r) <= 1
                   && (height r - height l) <= 1
                   && f l
                   && f r
```

Die Funktion gibt zurück, ob der übergebene Baum **balanciert** ist. Balanciert bedeuter hier, dass sich die Höhe bzw. maximale Tiefe der beiden Teilbäume nicht um mehr als 1 unterscheidet.

b) Betrachten Sie den folgenden Beispielbaum:

```
let t = Node (Node (Node (Leaf 4, Leaf 1), Leaf 6), Node (Leaf 5, Node (Leaf 2, Leaf 3)))
```

Was gibt der Funktionsaufruf `g t` zurück? Zeigen Sie Ihre Vorgehensweise, indem Sie alle gemachten Schritte aufschreiben und gegebenenfalls erklären. Sie dürfen die Bäume skizzieren. **Das Endergebnis allein gibt keine Punkte!**

```
let h (n: Nat): Nat =
  if (n % 2) = 0 then 2 * n
  else n - 1

let rec g (t: Tree): Tree =
  match t with
  | Leaf n      -> Leaf (h n)
  | Node (l, r) -> Node (g r, g l)
```

Die Funktion spiegelt den Baum und wendet auf die Blätter eine Funktion an, die gerade Zahlen verdoppelt und ungerade Zahlen um eins verringert. Korrektes Endergebnis: `Node (Node (Node (Leaf 2, Leaf 4), Leaf 4), Node (Node (Leaf 8, Leaf 0), Leaf 12))`

```
g (Node (Node (Node (Leaf 4, Leaf 1), Leaf 6), Node (Leaf 5, Node (Leaf 2, Leaf
   3))))
= Node (g (Node (Leaf 5, Node (Leaf 2, Leaf 3))), g (Node (Node (Leaf 4, Leaf
  1), Leaf 6)))
= Node (Node (g (Node (Leaf 2, Leaf 3)), g (Leaf 5)), Node (g (Leaf 6), g (Node
   (Leaf 4, Leaf 1))))
= Node (Node (Node (g (Leaf 3), g (Leaf 2)), Leaf (h 5)), Node (Leaf (h 6),
  Node (g (Leaf 1), g (Leaf 4))))
= Node (Node (Node (Leaf (h 3), Leaf (h 2)), Leaf 4), Node (Leaf 12, Node (Leaf
   (h 1), Leaf (h 4))))
= Node (Node (Node (Leaf 2, Leaf 4), Leaf 4), Node (Leaf 12, Node (Leaf 0, Leaf
   8)))
```

Aufgabe 4 Statische und Dynamische Semantik (Einreichaufgabe, 7 Punkte)

a) Füllen Sie die Lücken in folgender Aussage der Statischen Semantik und geben Sie einen vollständigen Beweisbaum an, der die Korrektheit dieser Aussage zeigt.

$$\emptyset \vdash \texttt{let f (x: } \_\_\_\_\_\texttt{): } \_\_\_\_\_ \texttt{ = x true in f } \_\_\_\_\_ : \text{Nat}$$

Aus Platzgründen vergeben wir Bezeichner für folgende Signaturen:
$\Sigma_0 := \{\texttt{x} \mapsto \text{Bool} \to \text{Nat}\}$ $\Sigma_1 := \{\texttt{f} \mapsto (\text{Bool} \to \text{Nat}) \to \text{Nat}\}$ $\Sigma_2 := \{\texttt{f} \mapsto (\text{Bool} \to \text{Nat}) \to \text{Nat}, \texttt{x} \mapsto \text{Bool}\}$ Link zum Baum

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_0 \vdash \texttt{x} : \text{Bool} \to \text{Nat}} \quad \dfrac{}{\Sigma_0 \vdash \texttt{true} : \text{Bool}}}{\Sigma_0 \vdash \texttt{x true} : \text{Nat}}}{\emptyset \vdash \texttt{let f (x: Bool} \to \texttt{Nat): Nat = x true} : \Sigma_1} \quad \dfrac{\dfrac{}{\Sigma_1 \vdash \texttt{f} : (\text{Bool} \to \text{Nat}) \to \text{Nat}} \quad \dfrac{\dfrac{}{\Sigma_2 \vdash \texttt{4711} : \text{Nat}}}{\Sigma_1 \vdash \texttt{fun (x: Bool)} \to \texttt{4711} : \text{Bool} \to \text{Nat}}}{\Sigma_1 \vdash \texttt{f (fun (x: Bool)} \to \texttt{4711)} : \text{Nat}}}{\emptyset \vdash \texttt{let f (x: } \underline{\text{Bool} \to \text{Nat}}\texttt{): } \underline{\text{Nat}}\texttt{ = x true in f } \underline{\texttt{(fun (x: Bool)} \to \texttt{4711)}} : \text{Nat}}$$

Die Lösung für die ersten beiden Lücken ist durch die Aufgabenstellung fest vorgegeben, für die dritte Lücke gibt es weitere Lösungsmöglichkeiten.

b) Zu welchem Wert wertet dieser Ausdruck mit den von Ihnen ausgefüllten Lücken aus? Geben Sie einen vollständigen Beweisbaum an, der

$$\emptyset \vdash \texttt{let f x = x true in f } \_\_\_\_\_ \Downarrow \_\_\_\_\_$$

zeigt, wobei die erste Lücke mit denselben Inhalt wie die dritte Lücke aus der vorherigen Teilaufgabe gefüllt werden muss. Für die Dynamische Semantik sind die Typangaben im Ausdruck (also die ersten beiden Lücken aus der vorherigen Teilaufgabe) irrelevant, daher lassen wir sie hier weg, um Platz und Schreibarbeit zu sparen.v

Aus Platzgründen vergeben wir Bezeichner für folgende Umgebungen:
$\delta_0 := \{\texttt{f} \mapsto \langle \emptyset, \texttt{x}, \texttt{x true}\rangle\}$ $\delta_1 := \{\texttt{x} \mapsto \langle \delta_0, \texttt{x}, \texttt{4711}\rangle\}$ $\delta_2 := \{\texttt{f} \mapsto \langle \emptyset, \texttt{x}, \texttt{x true}\rangle, \texttt{x} \mapsto \textit{true}\}$ Link zum Baum

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{let f x = x true} \Downarrow \delta_0} \quad \dfrac{\dfrac{}{\delta_0 \vdash \texttt{f} \Downarrow \langle \emptyset, \texttt{x}, \texttt{x true}\rangle} \quad \dfrac{}{\delta_0 \vdash \texttt{fun x} \to \texttt{4711} \Downarrow \langle \delta_0, \texttt{x}, \texttt{4711}\rangle} \quad \dfrac{\dfrac{}{\delta_1 \vdash \texttt{x} \Downarrow \langle \delta_0, \texttt{x}, \texttt{4711}\rangle} \quad \dfrac{}{\delta_1 \vdash \texttt{true} \Downarrow \textit{true}} \quad \dfrac{}{\delta_2 \vdash \texttt{4711} \Downarrow \texttt{4711}}}{\delta_1 \vdash \texttt{x true} \Downarrow \texttt{4711}}}{\delta_0 \vdash \texttt{f (fun x} \to \texttt{4711)} \Downarrow \texttt{4711}}}{\emptyset \vdash \texttt{let f x = x true in f } \underline{\texttt{(fun x} \to \texttt{4711)}} \Downarrow \underline{\texttt{4711}}}$$

c) Ist die Aussage der Statischen Semantik (Teilaufgabe a) bzw. Dynamischen Semantik (Teilaufgabe b) weiterhin gültig, wenn wir `let f` durch `let rec f` ersetzen? **Sie müssen weder die geänderten Beweisbäume noch eine Begründung abgeben!** "Ja" oder "Nein" genügt.

Die Aussagen bleiben gültig. In den Beweisbäumen wird neben dem Schlüsselwort **`rec`** nur ein nicht verwendeter Eintrag in der Signatur/Umgebung ergänzt und bei der Dynamischen Semantik wird aus dem Funktionsabschluss ein Rekursiver Funktionsabschluss. Hier sind die (nicht verlangten) Beweisbäume. Die Änderungen sind rot hervorgehoben:

Statische Semantik Aus Platzgründen vergeben wir Bezeichner für folgende Signaturen:
$\Sigma_0 := \{\texttt{f} \mapsto (\text{Bool} \to \text{Nat}) \to \text{Nat}, \texttt{x} \mapsto \text{Bool} \to \text{Nat}\}$ $\Sigma_1 := \{\texttt{f} \mapsto (\text{Bool} \to \text{Nat}) \to \text{Nat}\}$ $\Sigma_2 := \{\texttt{f} \mapsto (\text{Bool} \to \text{Nat}) \to \text{Nat}, \texttt{x} \mapsto \text{Bool}\}$ Link zum Baum

$$\dfrac{\dfrac{\dfrac{}{\Sigma_0 \vdash \texttt{x} : \text{Bool} \to \text{Nat}} \quad \dfrac{}{\Sigma_0 \vdash \texttt{true} : \text{Bool}}}{\Sigma_0 \vdash \texttt{x true} : \text{Nat}}}{\dfrac{\emptyset \vdash \texttt{let rec f (x: Bool} \to \texttt{Nat): Nat = x true} : \Sigma_1}{\emptyset \vdash \texttt{let rec f (x: Bool} \to \texttt{Nat): Nat = x true in f (fun (x: Bool} \to \texttt{4711)} : \text{Nat}} \quad \dfrac{\dfrac{}{\Sigma_1 \vdash \texttt{f} : (\text{Bool} \to \text{Nat}) \to \text{Nat}} \quad \dfrac{\dfrac{}{\Sigma_2 \vdash \texttt{4711} : \text{Nat}}}{\Sigma_1 \vdash \texttt{fun (x: Bool)} \to \texttt{4711} : \text{Bool} \to \text{Nat}}}{\Sigma_1 \vdash \texttt{f (fun (x: Bool)} \to \texttt{4711)} : \text{Nat}}}$$

Dynamische Semantik Aus Platzgründen vergeben wir Bezeichner für folgende Umgebungen:
$\delta_0 := \{\texttt{f} \mapsto \langle \emptyset, \texttt{f}, \texttt{x}, \texttt{x true}\rangle\}$ $\delta_1 := \{\texttt{f} \mapsto \langle \emptyset, \texttt{f}, \texttt{x}, \texttt{x true}\rangle, \texttt{x} \mapsto \langle \delta_0, \texttt{x}, \texttt{4711}\rangle\}$ $\delta_2 := \{\texttt{f} \mapsto \langle \emptyset, \texttt{f}, \texttt{x}, \texttt{x true}\rangle, \texttt{x} \mapsto \textit{true}\}$ Link zum Baum

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{let rec f x = x true} \Downarrow \delta_0} \quad \dfrac{\dfrac{}{\delta_0 \vdash \texttt{f} \Downarrow \langle \emptyset, \texttt{f}, \texttt{x}, \texttt{x true}\rangle} \quad \dfrac{}{\delta_0 \vdash \texttt{fun x} \to \texttt{4711} \Downarrow \langle \delta_0, \texttt{x}, \texttt{4711}\rangle} \quad \dfrac{\dfrac{}{\delta_1 \vdash \texttt{x} \Downarrow \langle \delta_0, \texttt{x}, \texttt{4711}\rangle} \quad \dfrac{}{\delta_1 \vdash \texttt{true} \Downarrow \textit{true}} \quad \dfrac{}{\delta_2 \vdash \texttt{4711} \Downarrow \texttt{4711}}}{\delta_1 \vdash \texttt{x true} \Downarrow \texttt{4711}}}{\delta_0 \vdash \texttt{f (fun x} \to \texttt{4711)} \Downarrow \texttt{4711}}}{\emptyset \vdash \texttt{let rec f x = x true in f (fun x} \to \texttt{4711)} \Downarrow \texttt{4711}}$$

Auch bei anderen gültigen Lösungen für Teilaufgabe a würde sich nichts an der Gültigkeit der beiden Aussagen ändern, da der Rumpf der (nun rekursiven) Funktion in der Aufgabenstellung fest vorgegeben ist und keinen rekursiven Aufruf enthält.

Aufgabe 5 Turtle-Grafik (Trainingsaufgabe)

Motivation: In dieser Aufgabe sollen Sie noch einmal den Umgang mit Listen einüben. Sie können sich an den Vorlesungsfolien 379 bis 404 sowie am Skript Kapitel 4.3 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Turtle.fs` aus der Vorlage `Aufgabe-6-5.zip`.

Als Turtle-Grafik wird eine Bildbeschreibungssprache verstanden, bei der man sich vorstellt, dass eine mit einem Stift ausgestattete Schildkröte (oder ein Roboter) sich über eine Zeichenebene bewegt. Die Schildkröte versteht verschiedene Kommandos, mit deren Hilfe sich ganze Programme zusammensetzen lassen, um ein Bild zu erstellen.

Dazu verwenden wir folgende Typen:

```
type Command =
    | D           // Drop:    Stift absetzen/anfangen zu zeichnen
    | F of Double // Forward: Vorwärts bewegen
    | L of Double // Left:    Nach links/gegen den Uhrzeigersinn drehen

type Program = List<Command>
```

Hinweis: Anders als sonst, arbeiten wir in dieser Aufgabe nicht mit natürlichen Zahlen. Stattdessen verwenden wir den Typ `Double`, um mit Fließkommazahlen zu arbeiten. Wenn Sie eine Zahl vom Typ `Double` angeben, müssen Sie darauf achten, den Dezimaltrenner mit anzugeben. Zum Beispiel wäre `2` eine Ganzzahl vom Typ `Int`, jedoch `2.0` eine Fließkommazahl.

Damit wir die Turtle-Grafiken auch tatsächlich betrachten können, ist in der Programmvorlage das Modul `Draw` enthalten, das Turtle-Grafiken in SVG-Bilder umwandeln kann. Sie können SVG-Dateien mit allen gängigen Webbrowsern öffnen. Im `Draw`-Modul gibt es eine Funktion `draw`, die ein Turtle-Programm als Argument erwartet und daraus eine SVG-Datei mit dem Namen `image.svg` im aktuellen Verzeichnis generiert. Das zu konvertierende Programm können Sie in der `main` Funktion auswählen und das gesamte Programm mit dem Befehl `dotnet run` ausführen.

Folgendes Turtle-Programm wird damit wie in Abbildung 1 dargestellt, in eine Grafik überführt.

```
let ex = [D; F 50.0; L 45.0; F 50.0]
```

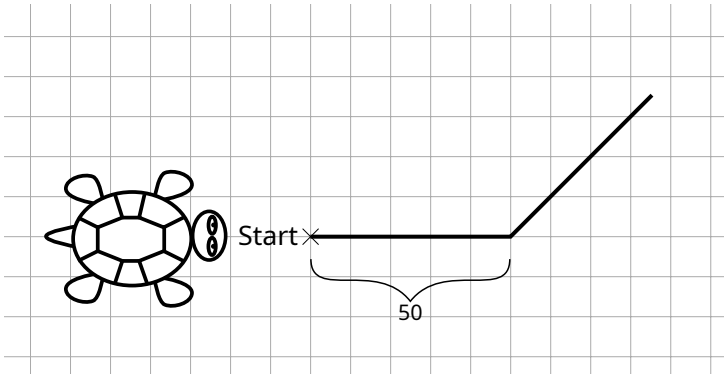

Abbildung 1: Darstellung des Programms `Turtle.ex`. Initial ist unsere Schildkröte nach rechts ausgerichtet. Sie setzt den Stift ab, bewegt sich um 50 Längeneinheiten vorwärts, dreht sich um 45 Grad nach links (gegen den Uhrzeigersinn) und bewegt sich erneut um 50 Längeneinheiten vorwärts.

Die Schildkröte startet ohne abgesetzten Stift und ist nach rechts ausgerichtet.

a) Implementieren Sie einen „smarten Konstruktor“ (eine Funktion, die den Konstruktor eines bestimmten Typs aufruft und dabei ggf. noch zusätzliche Prüfungen durchführt, die das Typsystem nicht durchführen kann - das tun wir hier jedoch nicht), der ein Element des Typs `Command` zurückgibt, das eine Drehung um den Winkel `angle` nach rechts modelliert.

Tipp: Eine Drehung nach rechts entspricht einer Drehung nach links um einen Winkel mit negativem Vorzeichen.

```
let right (angle: Double): Command = L (- angle)
```

b) Unsere Turtle-Programme eröffnen uns eine spannende Möglichkeit: Wir können Teile eines gegebenen Programms `p` anhand bestimmter Regeln ersetzen. Wir betrachten hier die Funktion `substF`, die alle Vorkommen der Vorwärtsbewegung `F` ersetzt. Implementieren Sie die Funktion `substF` und rufen Sie die Funktion `transformF: Double -> Program` mit der Länge des jeweiligen `F` Konstruktors auf, um die Substitution durchzuführen. Die Transformationsfunktion `transformF` arbeitet mit der Länge des bisherigen `F` Segments. So ist es möglich, Längenverhältnisse in der Transformationsfunktion zu berücksichtigen.

```
let rec substF (transformF: Double -> Program) (p: Program): Program =
    match p with
    | []          -> []
    | (F len)::ps -> transformF len @ substF transformF ps
    | cmd::ps     -> cmd::(substF transformF ps)
```

c) **Lévy-C-Kurve** Wir starten mit einer geraden Linie der Länge `s`. Implementieren Sie dazu die Funktion `levyStart`, die den Stift absetzt und diesen um die Länge `len` vorwärts bewegt.

Schreiben Sie nun eine Transformation `levyTransform`, welche eine Vorwärtsbewegung um die Länge `len` ersetzt durch

1. eine Drehung nach links um 45°
2. eine Vorwärtsbewegung der Länge $len/\sqrt{2}$
3. eine Drehung nach rechts um 90°
4. eine Vorwärtsbewegung der Länge $len/\sqrt{2}$
5. und noch eine Drehung nach links um 45°.

In den folgenden Aufgabenteilen verwenden wir dafür Abkürzugen:

- Den Buchstaben `F` für eine Vorwärtsbewegung (Skalierungsfaktor beachten)
- `+` für eine Drehung nach links, also gegen den Uhrzeigersinn (Winkel beachten)
- `-` für eine Drehung im Uhrzeigersinn (Winkel beachten)
- `->` gibt an, dass das links vom Pfeil ersetzt wird durch das, was rechts davon steht

Im vorliegenden Fall der Lévy-C-Kurve wäre die Kurzschreibweise für die Transformation `F -> +F--F+`, wobei das Längenverhältnis (vor Transformation vs nach Transformation) $1 : 1/\sqrt{2}$ beträgt. `+` und `-` ändern den Winkel jeweils um 45° gegen den bzw. im Uhrzeigersinn.

Hinweis: In der `main` Funktion des `Draw` Moduls wird mit Hilfe der Funktion `iterate` die Transformation `n` mal angewendet.

Hinweis: Sie können die Funktion `sqrt` zur Berechnung der Quadratwurzel verwenden.

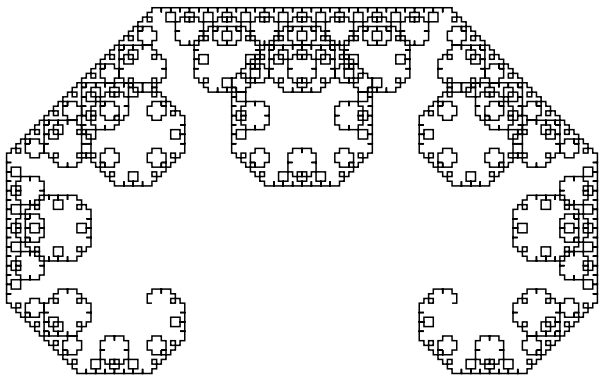

Abbildung 2: Lévy-C-Kurve, 12 Iterationen

```
// F
let levyStart (len: Double) = [D; F len]

// +F--F+
let levyTransform (len: Double): Program =
    let l = len / sqrt(2.0)
    [L 45.0; F l; right 90.0; F l; L 45.0]
```

d) Implementieren Sie die Funktionen `kochflockeStart` und `kochflockeTransform`. Die Symbole `+` und `-` ändern den Winkel um 60°. `kochflockeStart` soll mit der Sequenz `F--F--F` ein Dreieck zeichnen.

`kochflockeTransform` soll die Transformationsregel `F -> F+F--F+F` implementieren. Das Längenverhältnis beträgt dabei 1 : 1/3.

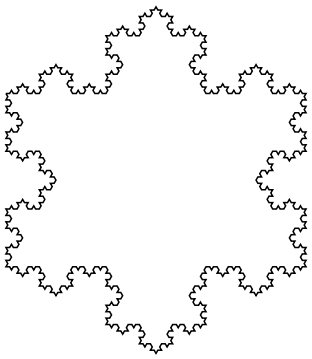

Abbildung 3: Koch-Flocke, 4 Iterationen

```
// F--F--F
let kochflockeStart (len: Double) =
    let a = 60.0
    [D; F len
      ; right (2.0*a); F len
      ; right (2.0*a); F len]

// F -> F+F--F+F
let kochflockeTransform (len: Double): Program =
    let l = len / 3.0
    let a = 60.0
    let flf = [F l; L a; F l]
    flf @ [right (2.0*a)] @ flf
```

e) Implementieren Sie die Funktionen `pentaplexityStart` und `pentaplexityTransform`. Die Symbole `+` und `-` ändern den Winkel um 36°. `pentaplexityStart` soll mit der Sequenz `F++F++F++F++F` ein Pentagon zeichnen.

`pentaplexityTransform` soll die Transformationsregel `F -> F++F++F|F-F++F` implementieren. Das Symbol `|` repräsentiert eine Drehung um 180°. Das Längenverhältnis beträgt $1 : 1/\phi^2$, wobei $\phi := \frac{1+\sqrt{5}}{2}$.

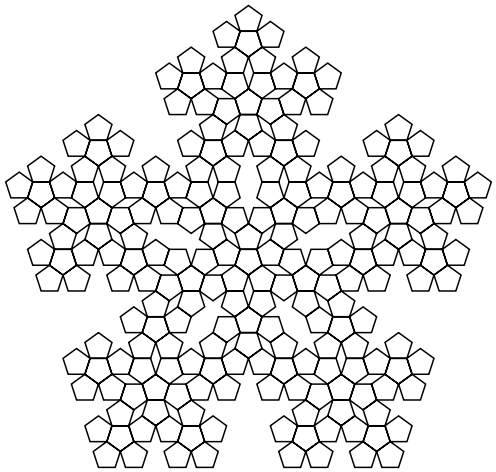

Abbildung 4: Penta Plexity, 3 Iterationen

```
// F++F++F++F++F
let pentaplexityStart (len: Double) =
    let a = 36.0
    let lf = [L (2.0*a); F len]
    [D; F len] @ lf @ lf @ lf @ lf

// F -> F++F++F|F-F++F
let pentaplexityTransform (len: Double): Program =
    let phi = (1.0 + sqrt 5.0) / 2.0
    let l = len / (phi ** 2.0)
    let a = 36.0
    [F l; L (2.0*a); F l
        ; L (2.0*a); F l
        ; L 180.0;   F l
        ; right a;   F l
        ; L (2.0*a); F l]
```