



## 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
```

- 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  $\geq$  des *größten* Elements des linken, und  $\leq$  des *kleinsten* Elements des rechten Teilbaums ist, wobei das kleinste Element eines BST das linkeste, und das größte das rechteste ist.

Ein erster naiver 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`
```

- 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)))`

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

```
type BTreeish<'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
```

- e) Schreiben Sie eine Funktion `letIt (grow : 'b -> BTreeish<'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, bis `Emptyish` produziert 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 - 2)) 3N =
```

```
  Node(Node (Node (Empty, 1N, Empty), 2N, Empty), 3N, Node (Empty, 1N, Empty))
```

- 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 ex1 = [2N; 4N; 6N]
```

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

```
toList ex2 = [3N; 5N; 7N]
```

- 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]
```

### 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
```

- 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)
```

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

*Hinweis:* Wenn Sie in Übungsgruppe 2, 4, 6, 8 oder 10 sind, dann steht Ihnen zur Bearbeitung dieser Aufgabe unser Beweisbaum Werkzeug zur Verfügung, siehe Hinweis auf der ersten Seite von Übungsblatt 4. Sie müssen das Sprachfeature “Wertdefinitionen & Funktionen” aktivieren. Nach dem Einloggen im ExClaim-System führt [dieser Link](#) direkt zum vorbereiteten Baum für Teilaufgabe a). Wenn Sie möchten, können Sie einen Screenshot des erstellten Beweisbaumes in Ihre Abgabe mit einbinden. Falls Sie bereits Erfahrung mit L<sup>A</sup>T<sub>E</sub>X haben, dürfen Sie die entsprechende Exportfunktion nutzen und eine kompilierte pdf-Datei abgeben. Der L<sup>A</sup>T<sub>E</sub>X-Quellcode oder die durch den Button “Baum speichern” erzeugte json-Datei wird jedoch nicht als Abgabe anerkannt! Alternativ können Sie die Abgabe wie gewohnt auf Papier aufschreiben.

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

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

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

$$\emptyset \vdash \text{let } f \text{ } x = x \text{ true in } f \text{_____} \Downarrow \text{_____}$$

zeigt, wobei die erste Lücke mit demselben 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.

- 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.

## 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]
```

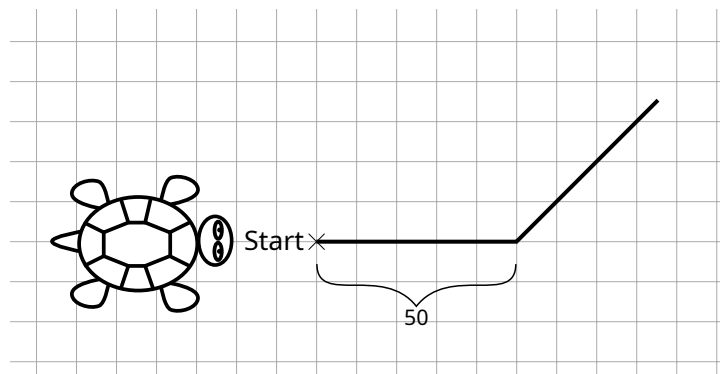


Abbildung 1: Darstellung des Programms `Turtle.ex`. Initial ist unsere Schildkröte nach rechts ausgerichtet. Sie setzt den Stift ab, bewegt sich um 50 Längeneinheiten vorwärts, dreht sich um 45 Grad nach links (gegen den Uhrzeigersinn) und bewegt sich erneut um 50 Längeneinheiten vorwärts.

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.

- 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.

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

In den folgenden Aufgabenteilen verwenden wir dafür Abkürzungen:

- 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  $45^\circ$  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.

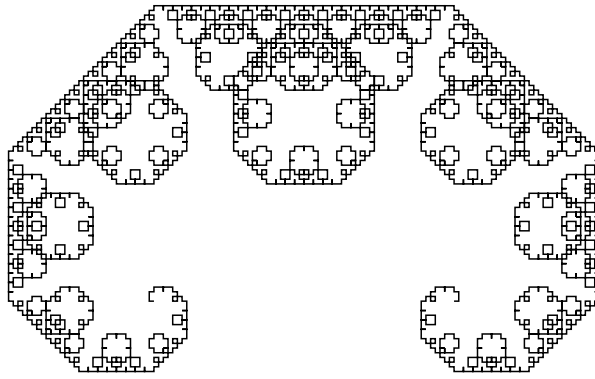


Abbildung 2: Lévy-C-Kurve, 12 Iterationen

- d) Implementieren Sie die Funktionen `kochflockeStart` und `kochflockeTransform`. Die Symbole `+` und `-` ändern den Winkel um  $60^\circ$ . `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  $1 : 1/3$ .

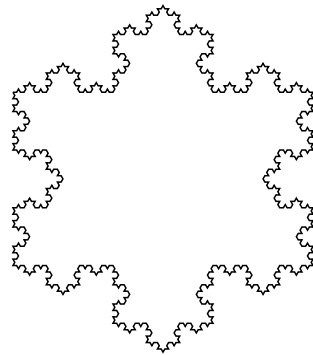


Abbildung 3: Koch-Flocke, 4 Iterationen

- e) Implementieren Sie die Funktionen `pentaplexityStart` und `pentaplexityTransform`. Die Symbole `+` und `-` ändern den Winkel um  $36^\circ$ . `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  $180^\circ$ . Das Längenverhältnis beträgt  $1 : 1/\phi^2$ , wobei  $\phi := \frac{1+\sqrt{5}}{2}$ .



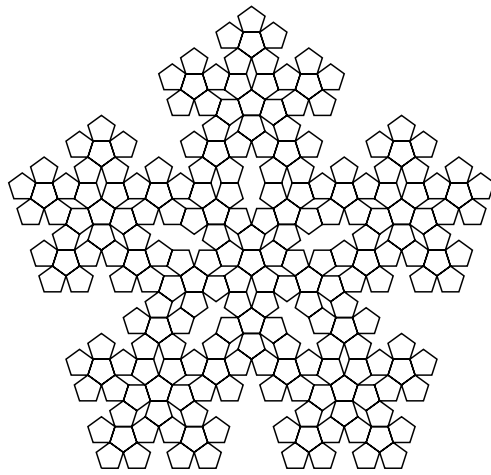


Abbildung 4: Penta Plexity, 3 Iterationen