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

RPTU
Fachbereich Informatik
AG Programmiersprachen

Lösungshinweise/-vorschläge zum Übungsblatt 6: Grundlagen der Programmierung (WS 2023/24)

Aufgabe 1 Funktionen höherer Ordnung (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie sich mit Funktionen höherer Ordnung auf Listen beschäftigen. Sie können sich an den Vorlesungsfolien 394 bis 404 bzw. am Skript Kapitel 4.3.2 orientieren.

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

Hinweis: Verwenden Sie in Ihrer Lösung ***nicht*** *das List-Modul aus der Standardbibliothek. Die hier zu definierenden Funktionen gibt es alle so im List-Modul.*

a) Schreiben Sie eine Funktion `map`, die eine Funktion `f: 'a -> 'b` auf alle Elemente einer Liste `xs` vom Typ `List<'a>` anwendet und die Ergebnisliste vom Typ `List<'b>` zurückgibt.

   Verwenden Sie `map`, um...

   - eine Funktion `double` zu schreiben, die alle Zahlen einer Liste natürlicher Zahlen verdoppelt,
   - eine Funktion `firstComponents` zu schreiben, die aus einer Liste von Tupeln die Liste der ersten Tupelkomponenten zurückgibt.

```
let rec map<'a, 'b> (f: 'a -> 'b) (xs: List<'a>): List<'b> =
    match xs with
    | [] -> []
    | y::ys -> (f y)::(map f ys)

// Anwendungen
let double (xs: List<Nat>): List<Nat> =
    map (fun x -> 2N * x) xs

let firstComponents<'a, 'b> (xs: List<'a * 'b>): List<'a> =
    map (fun x -> fst x) xs // map fst xs
```

b) Schreiben Sie eine Funktion `collect`, die eine Funktion `f: 'a -> List<'b>` auf alle Elemente einer Liste `xs` anwendet, die Einzelergebnisse konkateniert und eine Ergebnisliste vom Typ `List<'b>` zurückgibt.

Verwenden Sie `collect`, um eine Funktion `cloneElements` zu schreiben, die jedes Element einer Liste noch einmal in die Liste aufnimmt, Beispiel `cloneElements [1N; 2N; 3N] = [1N; 1N; 2N; 2N; 3N; 3N]`.

```
let rec collect (f: 'a -> List<'b>) (xs: List<'a>): List<'b> =
    match xs with
    | [] -> []
    | x::xs -> (f x) @ (collect f xs)

let rec concat<'a> (xs: List<List<'a>>): List<'a> =
    match xs with
    | [] -> []
    | x::xs -> x @ (concat xs)

let collect' (f: 'a -> List<'b>) (xs: List<'a>): List<'b> =
    concat (map f xs)

// Anwendung
let cloneElements (xs: List<'a>): List<'a> =
    collect (fun x -> [x; x]) xs
```

Aufgabe 2 Binärbäume (Präsenzaufgabe)

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

Schreiben Sie Ihre Lösungen in die Datei `Tree.fs` *aus der Vorlage* `Aufgabe-6-2.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 3 Heaps (Einreichaufgabe, 14 Punkte)

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

Schreiben Sie Ihre Lösungen in die Datei `Heaps.fs` *aus der Vorlage* `Aufgabe-6-3.zip`.

Ein Heap ist eine Datenstruktur in Form eines Binärbaums: Ein Heap ist entweder leer oder ein Knoten mit einem Eintrag, einem linken Teilbaum und einem rechten Teilbaum. Die Datenstruktur dient dazu, Elemente partiell geordnet abzuspeichern. Daher muss ein gültiger Heap die *Heap-Bedingung* erfüllen: Der Eintrag jedes Knotens muss kleiner oder gleich der Einträge seiner beiden Teilbäumen sein.

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

Beispiele für gültige Heaps:

```
let ex1 = Node(Node(Empty,6N,Empty), 2N, Node(Empty,4N,Empty))
let ex2 = Node(Node(Empty,7N,Empty), 3N, Node(Empty,5N,Empty))
let ex3 = Node(Node(Empty,1N,Empty), 1N, Empty)
```

Beispiele für ungültige Heaps:

```
let inv1 = Node(Node(Empty,2N,Empty), 3N, Empty)
let inv2 = Node(Node(Node(Empty,4N,Empty), 5N, Empty), 3N, Empty)
```

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

   Beispiele mit den oben definierten Heaps:

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

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

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

b) Schreiben Sie eine Funktion `isHeap: Heap<'a> -> bool`, die überprüft ob der gegebene Heap die Heap-Bedingung erfüllt.

Beispiele:

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

```
let isHeap<'a when 'a: comparison> (root: Heap<'a>): Bool =
    let rec isHeapMin (s: 'a) (root: Heap<'a>): Bool =
        match root with
        | Empty -> true
        | Node (left, x, right) -> x >= s && isHeapMin x left && isHeapMin x right
    match root with
    | Empty -> true
    | Node (left, x, right) -> isHeapMin x left && isHeapMin x right

// Alternativ: Disjunktives Muster und Einschränkung mit when
// Zuerst wird das erste Muster (Empty) abgeglichen.
// Wenn es nicht passt wird Node (x, Node (y, _, _), _) abgeglichen und x > y geprüft.
// Wenn das Muster nicht passt oder die Bedingung nicht erfüllt war,
// dann wird Node (x, _, Node (y, _, _) abgeglichen und wieder x > y geprüft.
// Falls bislang kein Muster gepasst hat oder die Bedingungen nicht erfüllt waren
// wird das letzte Muster (Node (_, left, right)) abgeglichen.
let rec isHeap'<'a when 'a: comparison> (root: Heap<'a>): Bool =
    match root with
    | Empty -> true
    | Node (Node (_, y, _), x, _) | Node (_, x, Node (_, y, _)) when x > y -> false
    | Node (left, _, right) -> isHeap' left && isHeap' right
```

c) Schreiben Sie eine Funktion `head: Heap<'a> -> Option<'a>`, die aus einem Heap das kleinste Element bestimmt. Da der Heap leer sein kann, verwenden wir wieder den `Option`-Typ. Das heißt, wenn der Heap leer ist, soll `None` zurückgegeben werden.

Sie können davon ausgehen, dass der gegebene Heap die Heap-Bedingung erfüllt ist.

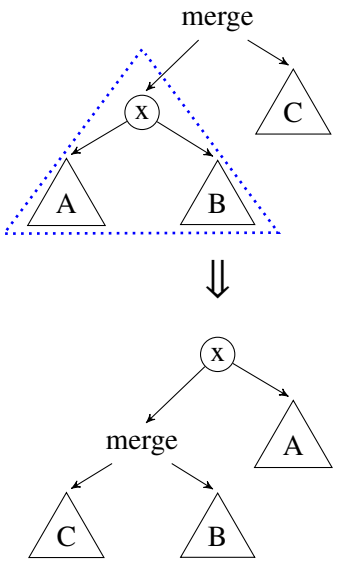
Beispiel: `head ex1 = Some 2N`

```
let head<'a> (root: Heap<'a>): Option<'a> =
    match root with
    | Empty -> None
    | Node (_, x, _) -> Some x
```

d) Schreiben Sie eine Funktion `merge: Heap<'a> -> Heap<'a> -> Heap<'a>`, die zwei gegebene Heaps in einen Heap zusammenführt. Nutzen Sie für Ihre Implementierung den folgenden Algorithmus[1]:

- Wenn einer der beiden gegebenen Heaps leer ist, dann ist das Ergebnis der jeweils andere Heap.
- Ansonsten wähle als `p` den gegebenen Heap mit dem kleineren `head` und als `q` den anderen gegebenen Heap. Konstruiere den Ergebnis-Heap `r` als `Node` wie folgt:
  - Die erste Komponente (der Eintrag) von `r` ist `head p`.
  - Die zweite Komponente (linker Teilbaum) von `r` wird berechnet, indem `q` und der rechte Teilbaum von `p` rekursiv zusammengeführt werden.
  - Die dritte Komponente (rechter Teilbaum) von `r` ist der linke Teilbaum von `p`.

  Die Abbildung rechts verdeutlicht den Algorithmus nochmals. Kreise sind Knoten und Dreiecke sind (Teil-)Bäume. Der blau umrandete Teil ist `p`. Der Baum C ist `q`.



Beispiel: `merge ex1 ex2 = Node(`

```
                        Node(
                          Node(Node(Empty,5N,Empty), 4N, Empty),
                          3N,
                          Node(Empty,7N,Empty)),
                        2N,
                        Node(Empty,6N,Empty))
```

```
let rec merge<'a when 'a: comparison> (root1: Heap<'a>) (root2: Heap<'a>): Heap<'a> =
    match (root1, root2) with
    | (Empty, t) | (t, Empty) -> t
    | (Node (l1, x1, r1), Node (l2, x2, r2)) ->
        let join x l r t = Node (merge t r, x, l)
        if x1 <= x2 then join x1 l1 r1 root2
        else join x2 l2 r2 root1
```

Tipp: Die `merge` Funktion können Sie gut in den weiteren Teilaufgaben verwenden.

e) Schreiben Sie eine Funktion `tail: Heap<'a> -> Heap<'a>`, die aus einem Heap das kleinste Element entfernt und einen gültigen Heap zurückgibt, der aus den restlichen Elementen besteht. Ist der Heap leer, soll der leere Heap wieder zurückgegeben werden. Sie können davon ausgehen, dass der gegebene Heap die Heap-Bedingung erfüllt ist.

Beispiel: `tail ex3 = Node(Empty,1N,Empty)`

```
let tail<'a when 'a: comparison> (root: Heap<'a>): Heap<'a> =
    match root with
    | Empty -> Empty
    | Node (left, _, right) -> merge left right
```

[1] Skew Heap Verschmelzung, siehe https://de.wikipedia.org/wiki/Skew_Heap

f) Schreiben Sie eine Funktion `insert: Heap<'a> -> 'a -> Heap<'a>`, die in den gegebenen Heap das gegebene Element einfügt. Sie können davon ausgehen, dass der gegebene Heap die Heap-Bedingung erfüllt. Der Ergebnis-Heap muss die Heap-Bedingung erfüllen. *Tipp:* Verwenden Sie Ihre `merge` Funktion!

Beispiel: `insert Empty 3N = Node(Empty,3N,Empty)`

```
let insert<'a when 'a: comparison> (root: Heap<'a>) (x: 'a): Heap<'a> =
    merge root (Node (Empty, x, Empty))
```

g) Schreiben Sie Funktionen `ofList: List<'a> -> Heap<'a>` und `toList: Heap<'a> -> List<'a>`. Erstere nimmt eine Liste und erstellt einen gültigen Heap, der die Elemente der Liste enthält. Die Funktion `toList` nimmt einen gültigen Heap und erstellt daraus eine **sortierte** Liste der Elemente des Heaps.

Beispiele:

```
ofList<Nat> [] = Empty
ofList [3N] = Node(Empty,3N,Empty)
toList<Nat> Empty = []
toList ex1 = [2N; 4N; 6N]
toList ex2 = [3N; 5N; 7N]
```

```
let rec ofList<'a when 'a: comparison> (xs: List<'a>): Heap<'a> =
    match xs with
    | [] -> Empty
    | x::xs' -> insert (ofList xs') x

let rec toList<'a when 'a: comparison> (root: Heap<'a>): List<'a> =
    match root with
    | Empty -> []
    | Node (left, x, right) -> x :: toList (merge left right)

// oder mit head und tail:
let rec toList'<'a when 'a: comparison> (root: Heap<'a>): List<'a> =
    match head root with
    | None -> []
    | Some x -> x :: toList' (tail root)
```

h) Schreiben Sie eine Funktion `heapsort: List<'a> -> List<'a>`, die eine Liste von Elementen nimmt und diese sortiert. Verwenden Sie dazu die beiden Funktionen aus der vorherigen Teilaufgabe.

Beispiele: `heapsort<Nat> [] = []` `heapsort [2N; 3N; 1N; 2N] = [1N; 2N; 2N; 3N]`

Wenn Sie die Funktion `merge` wie vorgesehen implementiert haben und in den anderen Funktionen sinnvoll verwenden, dann sollten Sie Listen der Länge n mit höchstens $2 * n * log_2(n)$ Vergleichen sortieren können. Dies wird durch den Testfall `heapsort Zufall Effizienz` abgeprüft.

```
let heapsort<'a when 'a: comparison> (xs: List<'a>): List<'a> =
    toList (ofList xs)
```

Aufgabe 4 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-6-4.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]
```

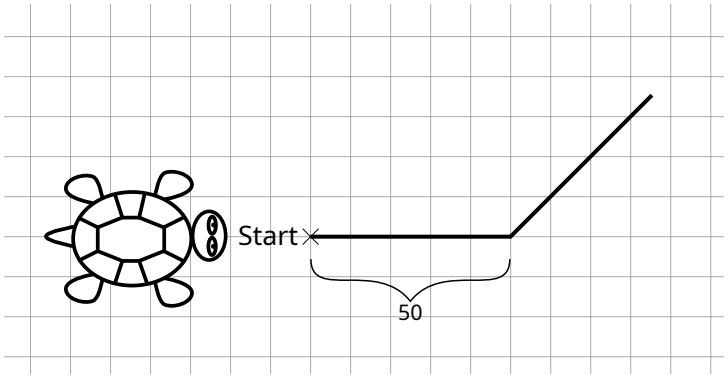


Abbildung 1: Darstellung des Programms `Turtle.ex`. Initial ist unsere Schildkröte nach rechts ausgerichtet. Sie setzt den Stift ab, bewegt sich um 50 Längeneinheiten vorwärts, dreht sich um 45 Grad nach links (gegen den Uhrzeigersinn) und bewegt sich erneut um 50 Längeneinheiten vorwärts.

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.

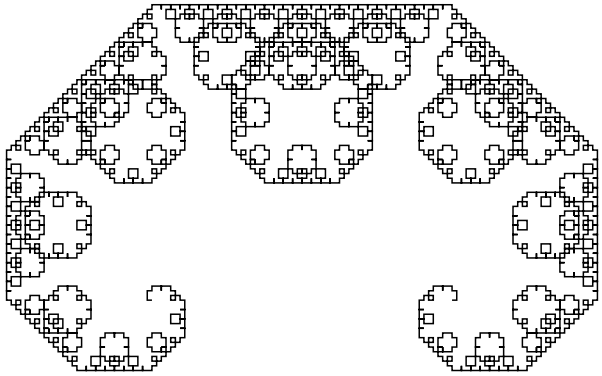


Abbildung 2: Lévy-C-Kurve, 12 Iterationen

```
// 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.

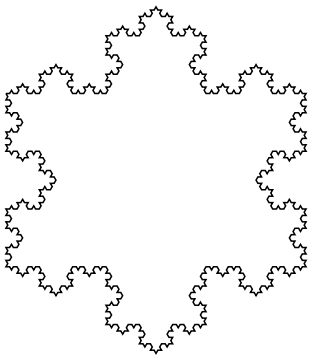


Abbildung 3: Koch-Flocke, 4 Iterationen

```
// 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 $\phi := \frac{1+\sqrt{5}}{2}$.

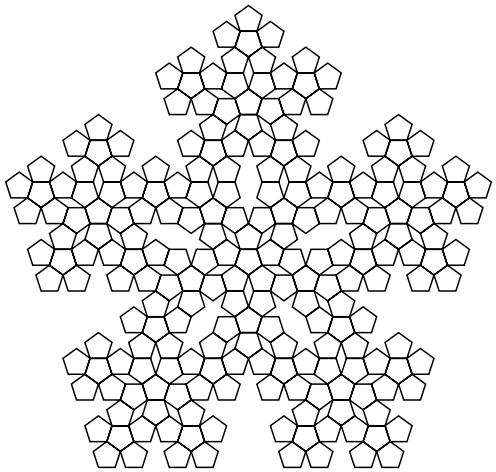


Abbildung 4: Penta Plexity, 3 Iterationen

```
// 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]
```

Aufgabe 5 Statische und dynamische Semantik (Trainingsaufgabe)

a) Bestimmen Sie den Typ des Ausdrucks `f (f 7N)` bezüglich der Signatur $\Sigma := \{\texttt{f} \mapsto (Nat \to Nat)\}$. Geben Sie einen vollständigen Beweisbaum auf Grundlage der Regeln der statischen Semantik aus der Vorlesung an.

$$\dfrac{\dfrac{}{\Sigma \vdash \texttt{f} : Nat \to Nat} \quad \dfrac{\dfrac{}{\Sigma \vdash \texttt{f} : Nat \to Nat} \quad \dfrac{}{\Sigma \vdash \texttt{7N} : Nat}}{\Sigma \vdash \texttt{f 7N} : Nat}}{\Sigma \vdash \texttt{f (f 7N)} : Nat}$$

b) Bestimmen Sie den Typ des Ausdrucks `f (a + b)` bezüglich der folgenden Signatur.

$$\Sigma := \{\texttt{a} \mapsto Nat, \texttt{b} \mapsto Nat, \texttt{f} \mapsto (Nat \to Bool)\}$$

Geben Sie einen vollständigen Beweisbaum auf Grundlage der Regeln der statischen Semantik aus der Vorlesung an.

$$\dfrac{\dfrac{}{\Sigma \vdash \texttt{f} : Nat \to Bool} \quad \dfrac{\dfrac{}{\Sigma \vdash \texttt{a} : Nat} \quad \dfrac{}{\Sigma \vdash \texttt{b} : Nat}}{\Sigma \vdash \texttt{a + b} : Nat}}{\Sigma \vdash \texttt{f (a + b)} : Bool}$$

c) Werten Sie den Ausdruck `f (a + b)` bezüglich der folgenden Umgebung aus.

$$\delta := \{\texttt{a} \mapsto 1, \texttt{b} \mapsto 3, \texttt{f} \mapsto \langle \emptyset, \texttt{x}, \texttt{x > 10N} \rangle\}$$

Geben Sie einen vollständigen Beweisbaum auf Grundlage der dynamischen Semantik aus der Vorlesung an.

$$\dfrac{\dfrac{}{\delta \vdash \texttt{f} \Downarrow \langle \emptyset, \texttt{x}, \texttt{x > 10N} \rangle} \quad \dfrac{\dfrac{}{\delta \vdash \texttt{a} \Downarrow 1} \quad \dfrac{}{\delta \vdash \texttt{b} \Downarrow 3}}{\delta \vdash \texttt{a + b} \Downarrow 4} \quad \dfrac{\dfrac{}{\{\texttt{x} \mapsto 4\} \vdash \texttt{x} \Downarrow 4} \quad \dfrac{}{\{\texttt{x} \mapsto 4\} \vdash \texttt{10N} \Downarrow 10}}{\{\texttt{x} \mapsto 4\} \vdash \texttt{x > 10N} \Downarrow false}}{\delta \vdash \texttt{f (a + b)} \Downarrow false}$$

d) Bestimmen Sie den Typ des Ausdrucks `if f true then 2N else 1N` bezüglich der folgenden Signatur.

$$\Sigma := \{\texttt{f} \mapsto (\mathit{Bool} \to \mathit{Bool})\}$$

Geben Sie einen vollständigen Beweisbaum auf Grundlage der Regeln der statischen Semantik aus der Vorlesung an.

$$\dfrac{\dfrac{\dfrac{}{\Sigma \vdash \texttt{f} : \mathit{Bool} \to \mathit{Bool}} \quad \dfrac{}{\Sigma \vdash \texttt{true} : \mathit{Bool}}}{\Sigma \vdash \texttt{f true} : \mathit{Bool}} \quad \dfrac{}{\Sigma \vdash \texttt{2N} : \mathit{Nat}} \quad \dfrac{}{\Sigma \vdash \texttt{1N} : \mathit{Nat}}}{\Sigma \vdash \texttt{if f true then 2N else 1N} : \mathit{Nat}}$$

e) Werten Sie den Ausdruck `if f true then 2N else 1N` bezüglich der folgenden Umgebung aus.

$$\delta := \{\texttt{f} \mapsto \langle \emptyset, \texttt{x}, \texttt{false} \rangle\}$$

Geben Sie einen vollständigen Beweisbaum auf Grundlage der dynamischen Semantik aus der Vorlesung an.

$$\dfrac{\dfrac{\dfrac{}{\delta \vdash \texttt{f} \Downarrow \langle \emptyset, \texttt{x}, \texttt{false} \rangle} \quad \dfrac{}{\delta \vdash \texttt{true} \Downarrow \mathit{true}} \quad \dfrac{}{\{\texttt{x} \mapsto \mathit{true}\} \vdash \texttt{false} \Downarrow \mathit{false}}}{\delta \vdash \texttt{f true} \Downarrow \mathit{false}} \quad \dfrac{}{\delta \vdash \texttt{1N} \Downarrow 1}}{\delta \vdash \texttt{if f true then 2N else 1N} \Downarrow 1}$$