Prof. Dr. Thomas Breuel
Dipl.-Inf. Patrick Michel (AG ST)
Dipl.-Ing. Ludwig Schmidt-Hackenberg

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Lösungshinweise/-vorschläge zum Übungsblatt 8: Software-Entwicklung 1 (WS 2010/11)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Aufgabe 1 Terminierung (Präsenzaufgabe)

In dieser Aufgabe wollen wir die Terminierung von Funktionen mit dem Verfahren aus der Vorlesung beweisen. Wir betrachten wieder einmal die Fibonacci Funktion, implementiert durch:

```
f :: Integer -> Integer
f 0 = 0
f 1 = 1
f n = f (n - 1) + f (n - 2)
```

Beweisen Sie, dass diese Funktion terminiert, denken Sie aber daran, dass wir uns nur für Parameter aus den natürlichen Zahlen interessieren.

Das erste Problem ist, dass die Funktion gar nicht für negative Eingaben terminiert. Wir müssen also erst einmal definieren, was der zulässige Parameterbereich ist, nämlich $P = \mathbb{N}$. Nun können wir annehmen, dass Parameter alle aus P sind und da 0 und 1 speziell behandelt werden gilt im dritten Pattern $n \geq 2$. Beide rekursive Aufrufe verlassen also P nicht und wir haben einen wichtigen Teil des Beweises erledigt: $G(n) \in P$.

Was uns nun noch fehlt, ist eine noethersche Ordnung, eine Funktion $\delta : P \mapsto M$ und der Beweis, dass Parameter echt kleiner werden: $\delta(G(n)) < \delta(n)$.

Als noethersche Ordnung nehmen wir die natürlichen Zahlen mit der üblichen Ordnung: $(\mathbb{N}, \leq)$. Damit ist $M = \mathbb{N}$, und wir wählen unser $\delta : \mathbb{N} \mapsto \mathbb{N}$ als die Identitätsfunktion. Die zwei Beweise für die rekursiven Aufrufe sind dann trivial:

$$\delta(n-1) \overset{\text{Def}}{=} n - 1 < n \overset{\text{Def}}{=} \delta(n)$$

$$\delta(n-2) \overset{\text{Def}}{=} n - 2 < n \overset{\text{Def}}{=} \delta(n)$$

Aufgabe 2 Terminierung (Einreichaufgabe)

a) Beweisen Sie, dass die Funktion `merge` terminiert:

```
merge :: ([Integer], [Integer]) -> [Integer]
merge ([], b) = b
merge (a, []) = a
merge (a : al, b : bl) = if a < b
  then a : merge (al, b : bl)
  else b : merge (a : al, bl)
```

- Als noethersche Ordnung nehmen wir $(\mathbb{N}, \leq)$.
- Der zulässige Parameterbereich ist der Datentyp `([Integer], [Integer])`, eingeschränkt auf endliche Listen in beiden Komponenten:

$$P = \{(a, b) \mid a, b \in \texttt{[Integer]} \land a, b \text{ sind endlich}\}$$

- Die Funktion δ : `([Integer], [Integer])` $\mapsto \mathbb{N}$ kann man als die Summer der beiden Längen wählen:

$$\delta(a, b) = \text{length a} + \text{length b}$$

  Auch hier brauchen wir die Endlichkeit, damit die Funktion wohldefiniert ist.
- Zu zeigen, dass rekursive Aufrufe nur auf zulässigen Parametern passieren, ist in diesem Fall einfach, da Haskell schon über das Typsystem garantiert, dass Aufrufe wieder mit Paaren von Listen vom passenden Typ passieren. Die Endlichkeit ist gegeben, da wir ausschließlich Teillisten von endlichen Listen verwenden.
- Bleibt zu zeigen, dass die beiden rekursiven Aufrufe echt kleiner werden:

$$\begin{aligned}
\delta(\texttt{al, b : bl}) &\overset{\text{Def } \delta}{=} \texttt{length al + length (b : bl)} \\
&< \texttt{1 + length al + length (b : bl)} \\
&\overset{\text{Def length}}{=} \texttt{length (a : al) + length (b : bl)} \\
&\overset{\text{Def } \delta}{=} \delta\texttt{(a : al, b : bl)}
\end{aligned}$$

$$\begin{aligned}
\delta(\texttt{a : al, bl}) &\overset{\text{Def } \delta}{=} \texttt{length (a : al) + length bl} \\
&< \texttt{length (a : al) + 1 + length bl} \\
&\overset{\text{Def length}}{=} \texttt{length (a : al) + length (b : bl)} \\
&\overset{\text{Def } \delta}{=} \delta\texttt{(a : al, b : bl)}
\end{aligned}$$

  Da alle rekursiven Aufrufe ausschließlich im dritten Fall passieren, können die anderen beiden Fälle der Funktion ignoriert werden.

b) Wir betrachten folgende Implementierung von `mapreduce` und wollen ihre Terminierung beweisen.

```
mapreduce :: (a -> b) -> (b -> b -> b) -> [a] -> b
mapreduce g f = fix . map g  where

  reduce []           = []
  reduce [x]          = [x]
  reduce (a : b : r) = f a b : reduce r

  fix [x] = x
  fix l   = fix (reduce l)
```

*Hinweis: In dieser Aufgabe betrachten wir ausschließlich **endliche** Werte, d.h. Listen sind immer endlich lang und auch Werte von anderen rekursiven Datentypen sind nicht unendlich tief. Auch wenn wir also Aussagen wie beispielsweise "für alle Listen" verwenden, ist implizit damit "für alle endlich langen Listen mit endlichen Elementen" gemeint.*

Beweisen Sie die Terminierung von `reduce`, wobei Sie den Parameterbereich als gegeben annehmen dürfen und auch nicht beweisen müssen, dass er nicht verlassen wird.

- Als noethersche Ordnung nehmen wir $(\mathbb{N}, \leq)$.
- Als δ-Funktion nehmen wir die Länge der Eingabe, also δ = `length`
- Die Parameter werden kleiner, da

$$\delta\texttt{(r)} \overset{\text{Def } \delta}{=} \texttt{length r} < \texttt{2 + length r} \overset{\text{Def length}}{=} \texttt{length (a : b : r)} \overset{\text{Def } \delta}{=} \delta\texttt{(a : b : r)}$$

c) Beweisen Sie per Induktion über die Länge von `l`, dass

$$\texttt{length (reduce l)} \;=\; \texttt{(length l + 1) `div` 2}$$

Tipp: Setzen Sie den Induktionsanfang bei Länge Null ***und*** *Eins und führen Sie dann den Schritt von Länge n auf Länge n + 2.*

Tipp: Sie dürfen im Beweis einfach verwenden, dass `1 + a `div` b = (a + b) `div` b` *gilt.*

I.A. Länge 0 und 1, also `l = []` und `l = [x]`

$$\begin{aligned}
\texttt{length (reduce [])} &\overset{\text{Def reduce}}{=} \texttt{length []} \\
&\overset{\text{Def length}}{=} \texttt{0} \\
&\overset{\text{Def div}}{=} \texttt{1 `div` 2} \\
&\overset{\text{Def length}}{=} \texttt{(length [] + 1) `div` 2}
\end{aligned}$$

$$\begin{aligned}
\texttt{length (reduce [x])} &\overset{\text{Def reduce}}{=} \texttt{length [x]} \\
&\overset{\text{Def length}}{=} \texttt{1} \\
&\overset{\text{Def div}}{=} \texttt{2 `div` 2} \\
&\overset{\text{Def length}}{=} \texttt{(length [x] + 1) `div` 2}
\end{aligned}$$

I.V. Für ein `l` gilt

$$\texttt{length (reduce l) = (length l + 1) `div` 2}$$

I.S. Schritt von Länge n auf $n+2$, also `l` nach `(a : b : l)`

$$\begin{aligned}
\texttt{length (reduce (a : b : l))} &\overset{\text{Def reduce}}{=} \texttt{length (f a b : reduce l)} \\
&\overset{\text{Def length}}{=} \texttt{1 + length (reduce l)} \\
&\overset{\text{I.V.}}{=} \texttt{1 + (length l + 1) `div` 2} \\
&\overset{\text{Def div}}{=} \texttt{(length l + 3) `div` 2} \\
&\overset{\text{Def length}}{=} \texttt{(length (a : b : l) + 1) `div` 2}
\end{aligned}$$

d) Versuchen Sie nun die Terminierung von `fix` zu zeigen, indem Sie das Ergebnis aus c) verwenden. Passen Sie den möglichen Parameterbereich $P_{\texttt{fix}}$ wenn nötig an und vergessen Sie dann nicht zu zeigen, dass er durch rekursive Aufrufe nicht verlassen wird.

- Als noethersche Ordnung nehmen wir $(\mathbb{N}, \leq)$.
- Als δ-Funktion nehmen wir die Länge der Eingabe, also $\delta = \texttt{length}$
- Damit die Parameter nun auch kleiner werden, muss für den rekursiven Aufruf gelten:

$$\begin{aligned}
\delta(\texttt{reduce l}) &\overset{\text{Def } \delta}{=} \texttt{length (reduce l)} \\
&\overset{\text{c)}}{=} \texttt{(length l + 1) `div` 2} \\
&\overset{!}{<} \texttt{length l} \\
&\overset{\text{Def } \delta}{=} \delta(\texttt{l})
\end{aligned}$$

  Damit die markierte Ungleichung wirklich gilt, muss `length l ≥ 2` gelten. Die Länge von `l` kann nicht eins sein, da dieser Fall in `fix` getrennt behandelt wird. Wir müssen also leere Eingaben für `fix` ausschließen.

- Der Parameterbereich $P_{\texttt{fix}}$ ist also

$$P_{\texttt{fix}} = \left\{ l \in \texttt{[a]} \mid \texttt{a passend zu g} \;\wedge\; \texttt{length l} \;\geq\; 1 \right\}$$

- Der Parameterbereich wird nicht verlassen, da für `length l ≥ 1` gilt:

$$\begin{aligned} \texttt{length (reduce l)} &\overset{\text{c)}}{=} \texttt{(length l + 1) `div` 2} \\ &\geq \texttt{(1 + 1) `div` 2} \\ &\overset{\text{Def div}}{=} 1 \end{aligned}$$

Damit terminiert natürlich auch `mapreduce` nur für nicht-leere Listen und der Parameterbereich muss entsprechend angepasst werden.

Aufgabe 3 Einfache prozedurale Programmierung (Präsenzaufgabe)

a) *Keine Hinweise.*

b) Schreiben Sie eine Java-Prozedur, die von der Kommandozeile eine Zahl `n` liest und, wenn `n` gerade ist, diese Zahl wieder ausgibt, ansonsten `n` verdoppelt und ausgibt.

```
static void proz1() {
    int n = IO.readInt();
    if (n % 2 != 0) {
        n = 2 * n;
    }
    IO.println(n);
}
```

c) Schreiben Sie eine Java-Prozedur, die von der Kommandozeile einen `String` `s` und eine Zahl `n` liest, und den `String` dann `n` mal ausgibt.

```
static void proz2() {
  String s = IO.readString();
  int n = IO.readInt();
  for (int i = 0; i < n; i++) {
    IO.println(s);
  }
}
```

d) Schreiben Sie eine Java-Prozedur, die eine Variable `n` verwaltet, die anfangs auf Eins steht. Die Prozedur liest dann von der Kommandozeile so lange Zahlen ein, bis Null eingegeben wird. Jede eingegebene Zahl, außer der Null am Ende, soll auf `n` multipliziert werden. Geben Sie `n` am Ende aus.

```
static void proz3() {
    int n = 1;
    int m = IO.readInt();
    while (m != 0) {
        n = n * m;
        m = IO.readInt();
    }
    IO.println(n);
}
```

e) Programmieren Sie eine äquivalente Prozedur `int fac(int n) {...}` in Java. Verwenden Sie dabei eine **`while`**-Schleife, um die Rekursion der gegebenen Version zu ersetzen.

```
static int fac(int n) {
  int ergebnis = 1;

  while (n != 0) {
    ergebnis = ergebnis * n;
    n = n - 1;
  }

  return ergebnis;
}
```

Aufgabe 4 Variablen, Zustand und Schleifen (Einreichaufgabe)

Wir betrachten folgendes prozedurale Programm, das den ggT von zwei Zahlen berechnet:

```
01 public class GGT {
02
03   public static void main(String[] args) {
04     int m;
05     int n;
06
07     m = IO.readInt();
08     n = IO.readInt();
09
10     while (m > 0) {
11       int v;
12
13       v = n % m;
14       n = m;
15       m = v;
16     }
17
18     IO.println(n);
19   }
20 }
```

a) Geben Sie für die Eingaben `1029` und `1071` die Sequenz der *Ausführungszustände* an. Als *Speicherzustand* können Sie die Belegung der Variablen – sofern sie lebendig sind – verwenden und für den *Steuerungszustand* die Zeilennummer der *nächsten* Aktion. Betrachten Sie Zuweisungen und Deklarationen als Aktionen.

```
 4
 5   m =    ?
 7   m =    ?   n =    ?
 8   m = 1029   n =    ?
11   m = 1029   n = 1071
13   m = 1029   n = 1071  v1 =  ?
14   m = 1029   n = 1071  v1 = 42
15   m = 1029   n = 1029  v1 = 42
11   m =   42   n = 1029
13   m =   42   n = 1029  v2 =  ?
14   m =   42   n = 1029  v2 = 21
15   m =   42   n =   42  v2 = 21
11   m =   21   n =   42
13   m =   21   n =   42  v3 =  ?
14   m =   21   n =   42  v3 =  0
15   m =   21   n =   21  v3 =  0
18   m =    0   n =   21
```

b) Schreiben Sie das Programm um, so dass es bei gleicher Funktionalität eine **`for`**-Schleife benutzt.

Allgemein wandelt man **`while`**- in **`for`**-Schleifen um, indem man nur den Bedingungsteil der **`for`**-Schleife überhaupt verwendet und den Rest einfach leer lässt (und auch sonst nichts ändert):

```
for (; m > 0;) {
    int v;

    v = n % m;
    n = m;
    m = v;
}
```

Auch ungewöhnliche Lösungen sind möglich:

```
for (int v; m > 0; v = n % m, n = m, m = v);
```

Man muss sich dabei allerdings bewusst machen, dass man die Lebensdauer von v damit verändert.

c) Schreiben Sie das Programm um, so dass es bei gleicher Funktionalität eine **do-while**- Schleife benutzt.

Allgemein wandelt man **while** in **do-while**-Schleifen um, indem man die Bedingung ans Ende stellt, aber entsprechend selbst mit einer bedingten Anweisung vor dem Einstieg in die Schleife überprüft:

```
if (m > 0) do {
    int v;

    v = n % m;
    n = m;
    m = v;
} while (m > 0);
```

Hier ist es also wichtig sich bewusst zu machen, welchen Unterschied es macht eine Bedingung beim Ein- bzw. Austritt aus der Schleife zu prüfen.

d) Geben Sie an, wie man allgemein eine **for**-Schleife der Form

```
for(I; C; U) { B }
```

in eine **while**-Schleife überführen kann.

Mit den Meta-Variablen ist übrigens *Initialization*, *Condition*, *Update*, bzw. *Body* gemeint. Umformen kann man dies dann einfach so:

```
{ I; while(C) { B; U; } }
```

Also zuerst führt man die Initialisierung aus, dann durchläuft man die Schleife solange die Bedingung gilt. In der Schleife wird zunächst der alte Rumpf ausgeführt und dann vor dem nächsten Test und Durchlauf die Aktualisierung ausgeführt.

e) Geben Sie eine sinnvolle Abstraktion für den Anweisungsblock von Zeile 10 bis 16 an, indem Sie eine neue Prozedur einführen. Ersetzen Sie den Block entsprechend durch einen Aufruf dieser Prozedur.

Hier geht es mal wieder darum einen wichtigen Zusammenhang zu begreifen, nämlich das Prozeduren ein Abstraktionsmittel über Anweisungen sind, mit denen man also Blöcke wiederverwenden kann. Was man hier in Java tut ist vollkommen analog zu Haskell, wo wir mittels Funktionen über Ausdrücken abstrahiert haben:

```
public class GGTAbstr {

    static int ggt(int m, int n) {
        while (m > 0) {
            int v;

            v = n % m;
            n = m;
            m = v;
        }

        return n;
    }

    public static void main(String[] args) {
        int m;
        int n;

        m = IO.readInt();
        n = IO.readInt();

        n = ggt(m, n);

        IO.println(n);
    }
}
```

Wer die Prozedur als **void** deklariert und den Aufruf von `ggt` nicht an `n` bindet (oder direkt ins `println` schreibt), wird sich wundern warum `m` und `n` nicht verändert wurden. Hätten wir in Java auf Basistypen *call-by-reference*, hätte das funktionieren können...

Aufgabe 5 Prozedurale Programmierung (Einreichaufgabe)

In dieser Aufgabe sollen Sie mit Hilfe von einfachen Prozeduren Probleme lösen. Sie brauchen dazu lediglich Variablen, Schleifen und bedingte Anweisungen, insbesondere also keine Arrays.

a) Schreiben Sie eine Java-Prozedur `void primfaktoren(int n)`, die alle Primfaktoren der Zahl `n` mit Hilfe von `IO.println(..)` ausgibt.

```
void primfaktoren(int n) {
  for (int i = 2; i <= n; i++) {
    while (n % i == 0) {
      IO.println(i);
      n = n / i;
    }
  }
}
```

Zu beachten ist hier, dass in der Lösung der Parameter nicht kopiert, sondern direkt verändert wird! Ein Effekt ist damit auch, dass nicht nur die Schleifenvariable `i` hoch gezählt wird, sondern auch `n` aus der Schleifenbedingung immer kleiner wird! Das ganze ist in diesem Fall aber sogar eine starke Optimierung und sehr sinnvoll.

b) Schreiben Sie eine Java-Prozedur `void primzahlen(int n)`, die alle Primzahlen bis zur Zahl `n` mit Hilfe von `IO.println(..)` ausgibt.

```
static void primzahlen(int m) {
  for (int n = 2; n <= m; n++) {
    boolean isPrime = true;
    for (int i = 2; i < n; i++) {
      if (n % i == 0) {
        isPrime = false;
      }
    }
    if (isPrime) {
      IO.println(n);
    }
  }
}
```

Dadurch, dass wir uns die bisherigen Primzahlen nicht merken können und nur ausgeben, wird die Berechnung natürlich sehr ineffizient.

c) Schreiben Sie eine Java-Funktion `double sqrt(double n, double p)`, welche die Quadratwurzel von `n` mit einer Abweichung von höchstens `p` annährt, ohne dabei Funktionen der Standardbibliothek zu verwenden.

```
static double sqrt(double n, double p) {
  double a = 1.0;
  double b = n;
  while (a - b > p || b - a > p) {
     a = (a + b) / 2.0;
     b = n / a;
  }
  return a;
}
```

Eine alternative Abbruchbedingung ist hier auch das Produkt von `a` und `b` mit `n` zu vergleichen, es wurde aber in der Aufgabe nicht so beschrieben und würde auch zu einer anderen Genauigkeit führen.

Aufgabe 6 Imperative Programmierung mit Java

Kreuzen Sie an, ob folgende Aussagen wahr oder falsch sind. Bereiten Sie diese Aufgabe bis zu Ihrer nächsten Übungsstunde vor, so dass Sie bei Unklarheiten nachfragen und die Antworten diskutieren können.

wahr	falsch	
☒	☐	Ausdrücke in Java erzeugen oft Seiteneffekte.
☐	☒	Jeder Ausdruck ist eine Anweisung. *Ausdrücke können Teile von Anweisungen sein, sind jedoch selber keine Anweisungen (mit Ausnahme der Zuweisung).*
☐	☒	Es gibt keine Anweisungen, welche die Abfolge, in der Anweisungen abgearbeitet werden, beeinflussen. *Beispielsweise ermöglicht eine bedingte Anweisung, Teile des Programms zu überspringen oder eine Schleife ermöglicht die Mehrfachausführung einer Anweisung.*
☒	☐	Auf der rechten Seite einer Zuweisung können keine Prozeduraufrufe stehen. *Hängt von der Definition des Begriffs ab; Nach den diesjährigen Folien ist dies wahr.*
☒	☐	Überall dort wo eine einzelne Anweisung stehen kann, kann auch ein Anweisungsblock stehen.
☒	☐	Der Rumpf einer `do-while`-Anweisung wird mindestens einmal ausgeführt. *Da die Schleifenbedingung erst am Ende des Rumpfs ausgewertet wird, wird der Rumpf garantiert einmal ausgeführt.*
☐	☒	Die Schleifenvariable einer for-Schleife kann nur um einen konstanten Wert erhöht werden. *Es ist möglich, in der zweiten Ausdrucksanweisung einen beliebigen Ausdruck auf der rechten Seite der Zuweisung zu verwenden. Natürlich kann hier auch z.B. `i*i` stehen, wenn `i` die Schleifenvariable ist.*
☒	☐	Geschachtelte Fallunterscheidungen lassen sich nicht immer durch eine Auswahlanweisung ersetzen. *Im Allgemeinen geht dies nicht, da Auswahlanweisungen einen Ausdruck mit verschiedenen Konstanten vergleichen und daraus bestimmen, welcher Zweig ausgeführt wird und die Bedingungen in Fallunterscheidungen hingegen sehr viel komplizierter sein können. In vielen konkreten Anwendungsfällen sind die Bedingungen jedoch so einfach, dass man statt geschachtelter Fallunterscheidungen eine Auswahlanweisung nutzen kann und wegen der besseren Lesbarkeit auch sollte.*
☐	☒	Prozeduren enthalten Anweisungen und liefern immer ein Ergebnis. *Je nach Definition können oder dürfen Prozeduren kein Ergebnis haben, auf jeden Fall müssen sie nicht. Anweisungen müssen sie auch nicht enthalten.*
☐	☒	Eine Prozedur kann sich in ihrem Rumpf nicht selbst aufrufen. *Rekursive und verschränkt rekursive Prozedurdeklarationen sind erlaubt.*
☒	☐	Jede Prozedurinkarnation hat ihre eigenen lokalen Variablen.
☒	☐	Im allgemeinen gibt es für eine lokale Variable `v` einer Prozedur mehr als eine Speichervariable, die von ihr angesprochen wird. *In jeder Prozedurinkarnation wird mit `v` eine andere Speichervariable bezeichnet.*
☐	☒	Auf jede Variable kann während der gesamten Programmausführung zugegriffen werden. *Für globale Variablen ist dies der Fall, lokale Variablen leben jedoch nur während der zugehörigen Prozedurinkarnation und können auch nur während der Ausführung dieser Inkarnation verwendet werden.*
☐	☒	Eine Variable des Typs `boolean[]` speichert ein Feld. *Variablen für Felder speichern immer nur eine Referenz auf das Feld, in diesem Fall auf ein Feld mit Einträgen vom Typ `boolean`.*
☒	☐	Ein Programm kann Felder erzeugen, auf die es später nicht mehr zugreifen kann. *Ein Feld „lebt“ von seiner Erzeugung bis zum Programmende. Die Variablen, mit denen auf das Feld zugegriffen wird, können jedoch kürzere Lebensdauern haben, so dass zu einem Zeitpunkt keine Variable mehr vorhanden ist, die auf das Feld verweist, das Feld selbst jedoch noch im Speicher liegt.*
☒	☐	Es kann mehrere Variablen geben, die auf dasselbe Feld zeigen.

Aufgabe 7 Weihnachtsaufgabe (freiwillige Zusatzaufgabe)

a) Zeichnen Sie ein Dreieck, eine eckige Spirale und einen Kreis:

```
dreieck :: Turtle ()
dreieck = do
  forward 100
  turn 120
  forward 100
  turn 120
  forward 100

-- example invocation: spirale 64
spirale :: Integer -> Turtle ()
spirale 0 = return ()
spirale n = do
  forward $ fromInteger n
  turn 90
  spirale (n - 1)

-- example invocation: kreis 360
kreis :: Int -> Turtle ()
kreis 0 = return ()
kreis n = do
  forward 1
  turn 1
  kreis (n - 1)
```

b) Schreiben Sie ein rekursives Turtle-Programm `tree :: Int -> Double -> Turtle ()`, welches einen Baum dieser Art zeichnet.

```
-- example invocation: left 90 >> tree 9 100
tree :: Int -> Double -> Turtle ()
tree level size = when (level > 0) $ do
  forward size
  left 33
  tree (level - 1) (0.8 * size)
  right 37
  tree (level - 1) (0.82 * size)
  right 38
  tree (level - 1) (0.81 * size)

  -- clean up
  right 138
  forward size
  turn 180
```

c) Schreiben Sie ein Turtle-Programm `sierpinski :: Integer -> Bool -> Turtle ()`, welches das aktuelle Level, sowie die aktuelle Drehrichtung nimmt und das Sierpinski-Dreieck als Kurve realisiert.

```
-- example invocation: sierpinski 10 False
sierpinski :: Integer -> Bool -> Turtle ()
sierpinski 0 _ = forward 10
sierpinski x l = do
  sierpinski (x - 1) $ not l
  if l then left 60 else right 60
  sierpinski (x - 1) l
  if l then left 60 else right 60
  sierpinski (x - 1) $ not l

-- helper function, using the right orientation for each level
sierpinski_curve :: Integer -> Turtle ()
sierpinski_curve n = sierpinski n $ n `mod` 2 == 1
```

d) Informieren Sie sich über die *Drachenkurve* und implementieren Sie sie als Turtle-Programm:

```
-- example invocation: left 30 >> dx 16
dx, dy :: Int -> Turtle ()
dx n = when (n > 0) $ do
  dx $ n - 1
  left 90
  dy $ n - 1
  forward 10
  left 90

dy n = when (n > 0) $ do
  right 90
  forward 10
  dx $ n - 1
  right 90
  dy $ n - 1
```

e) Zeichnen Sie einen Farn:

```
-- example invocation: left 40 >> farn 60
farn :: Double -> Turtle ()
farn size = when (size > 1.0) $ do
  forward size
  right 2.5
  farn $ size * 0.9
  left 90
  farn $ size * 0.45
  turn 180
  farn $ size * 0.42

  -- clean up
  right 87.5
  forward size
  turn 180
```