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 7: Grundlagen der Programmierung (WS 2024/25)

Aufgabe 1 Reguläre Ausdrücke (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie sich mit Funktionen höherer Ordnung auf Listen beschäftigen. Sie können sich an den Vorlesungsfolien 547 bis 582 bzw. am Skript Kapitel 6.1 orientieren.

a) Leiten Sie das Wort ab aus dem regulären Ausdruck $(\mathsf{a} \mid \mathsf{b}) \mid (\mathsf{a} \mid \mathsf{b})^*$ ab. Geben Sie die gesamte Reduktionsfolge an.

$$
\begin{aligned}
& (\mathsf{a} \mid \mathsf{b}) \mid (\mathsf{a} \mid \mathsf{b})^* \\
& \to (\mathsf{a} \mid \mathsf{b})^* \\
& \to (\mathsf{a} \mid \mathsf{b}) \cdot (\mathsf{a} \mid \mathsf{b})^* \\
& \to \mathsf{a} \cdot (\mathsf{a} \mid \mathsf{b})^* \\
& \to \mathsf{a} \cdot (\mathsf{a} \mid \mathsf{b}) \cdot (\mathsf{a} \mid \mathsf{b})^* \\
& \to \mathsf{a} \cdot \mathsf{b} \cdot (\mathsf{a} \mid \mathsf{b})^* \\
& \to \mathsf{a} \cdot \mathsf{b} \cdot \epsilon \\
& \to \mathsf{ab}
\end{aligned}
$$

b) Geben Sie für die folgenden Beschreibungen in natürlicher Sprache einen regulären Ausdruck über dem Alphabet `{a, b, c}` an:

1. Die Sprache, deren Wörter aus genau drei Buchstaben bestehen.

   $(\mathsf{a} \mid \mathsf{b} \mid \mathsf{c}) \cdot (\mathsf{a} \mid \mathsf{b} \mid \mathsf{c}) \cdot (\mathsf{a} \mid \mathsf{b} \mid \mathsf{c})$

2. Die Sprache, deren Wörter genau zwei `a` enthalten.

   $(\mathsf{b} \mid \mathsf{c})^* \cdot \mathsf{a} \cdot (\mathsf{b} \mid \mathsf{c})^* \cdot \mathsf{a} \cdot (\mathsf{b} \mid \mathsf{c})^*$

3. Die Sprache, deren Wörter aus einer geraden Anzahl an Buchstaben bestehen.

   $((\mathsf{a} \mid \mathsf{b} \mid \mathsf{c}) \cdot (\mathsf{a} \mid \mathsf{b} \mid \mathsf{c}))^*$

c) Beschreiben Sie in natürlicher Sprache jeweils die Sprache, die durch die folgenden regulären Ausdrücke beschrieben wird:

1. $(\mathtt{a} \mid \mathtt{b})^*$

> Die Sprache, deren Wörter aus `a` und `b` bestehen.

2. $(\mathtt{a} \mid \mathtt{b} \mid \mathtt{c})^* \cdot \mathtt{a} \cdot (\mathtt{a} \mid \mathtt{b} \mid \mathtt{c})^*$

> Die Sprache, deren Wörter mindestens ein `a` enthalten.

3. $(\mathtt{a}^* \cdot \mathtt{b}^* \cdot \mathtt{c}^*)^*$

> Die Sprache, die alle Wörter enthält.

Aufgabe 2 Reguläre Ausdrücke: Reduktionssemantik erweitern (Einreichaufgabe, 6 Punkte)

Motivation: In der Praxis können Standardbibliotheken, die mit Regulären Ausdrücken arbeiten, meist mehr Features als das in der Vorlesung eingeführte Minimum. In dieser Aufgabe möchten wir ein solches Feature selbst einführen und die Semantik entsprechend erweitern.

Die auf auf Vorlesungsfolie 558 eingeführte Syntax von Regulären enthält r^* für *beliebige* Wiederholungen. Der Ausdruck a^* beschreibt also die Sprache mit den Wörtern ϵ, a, $\mathrm{a}\cdot\mathrm{a}$, $\mathrm{a}\cdot\mathrm{a}\cdot\mathrm{a}$, ... (unendlich viele Wörter).

Wir führen nun die Syntax $r\{n, m\}$ für mindestens n-fache und höchstens m-fache Wiederholung ein. Die Syntax verlangt an dieser Stelle, dass n und m natürliche Zahlen sind.

Die Denotationelle Semantik unserer Erweiterung ist wie folgt definiert:

$$[\![r\{n,m\}]\!] = \bigcup \left\{ [\![r]\!]^i \;\middle|\; n \le i \le m \right\}$$

Hinweis: Falls $n > m$ ist, beschreibt $r\{n, m\}$ somit die leere Sprache.

Der Ausdruck $\mathrm{a}\{3, 5\}$ beschreibt also die Sprache mit den Wörtern $\mathrm{a}\cdot\mathrm{a}\cdot\mathrm{a}$ (3 Vorkommen), $\mathrm{a}\cdot\mathrm{a}\cdot\mathrm{a}\cdot\mathrm{a}$ (4 Vorkommen) und $\mathrm{a}\cdot\mathrm{a}\cdot\mathrm{a}\cdot\mathrm{a}\cdot\mathrm{a}$ (5 Vorkommen). Die Sprache enthält nur genau diese drei Wörter.

Erweitern Sie nun auch die Reduktionssemantik (siehe Vorlesungsfolien 568 und 570) um unser neu eingeführtes Konstrukt.

Wir brauchen drei Regeln. Die erste Regel erlaubt es, r einmal auszurollen. Die restliche Wiederholung muss dann mit um eins dekrementierten Werten erfolgen. Dabei müssen wir durch entsprechende Voraussetzungen sicherstellen, dass die neuen Werte dann noch natürliche Zahlen sind.

$$\frac{n > 0 \qquad m > 0}{r\{n,m\} \to r \cdot r\{n-1, m-1\}}$$

Die zweite Regel erlaubt es, das Ausrollen zu beenden, wenn n bei 0 angekommen ist. Wir müssen m hier nicht prüfen, weil per Definition aus der Syntax $m \in \mathbb{N}$ und damit $0 = n \le m$ ist.

$$\frac{}{r\{0,m\} \to \epsilon}$$

Auch wenn n bei 0 angekommen ist, können wir r weiter ausrollen, solange $m > 0$ ist.

$$\frac{m > 0}{r\{0,m\} \to r \cdot r\{0, m-1\}}$$

Die erste Regel braucht keine Einschränkung $n \le m$, weil man auch so mit $n > m$ nach m Anwendungen der ersten Regel in einer Situation ist, in der keine der Regeln passt. Somit lässt sich kein Wort ableiten.

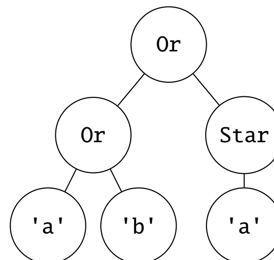
Aufgabe 3 Reduktionssemantik implementieren (Einreichaufgabe, 11 Punkte)

Motivation: In dieser Aufgabe sollen Sie sich mit regulären Ausdrücken beschäftigen und außerdem Ihr algorithmisches Denken üben. Sie können sich an den Vorlesungsfolien 552 bis 582 bzw. am Skript Kapitel 6.1 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Reduktionssemantik.fs` *aus der Vorlage* `Aufgabe-7-3.zip`.

In dieser Aufgabe wollen wir die Reduktionssemantik für reguläre Ausdrücke (Vorlesungsfolien 568 und 570) implementieren. Dazu definieren wir den Datentyp `RegEx<'a>`, der reguläre Ausdrücke über dem Alphabet `'a` repräsentiert (vgl. Vorlesungsfolie 558):

```
type RegEx<'a> =
    | Eps
    | Lit of 'a
    | Cat of RegEx<'a> * RegEx<'a>
    | Empty
    | Or of RegEx<'a> * RegEx<'a>
    | Star of RegEx<'a>
```



Beispielsweise ist der gegebene reguläre Ausdruck `Or (Or (Lit 'a', Lit 'b'), Star (Lit 'a'))` äquivalent zu `(a|b)|(a)*`. Der Ausdruck lässt sich als Baum darstellen, siehe die Abbildung oben rechts (`Lit 'a'` wird mit `'a'` abgekürzt).

Hinweis: Sie dürfen in Ihrer Lösung die Standardbibliothek verwenden.

a) Schreiben Sie eine Funktion `isWord<'a>: RegEx<'a> -> Option<List<'a>>`, die einen regulären Ausdruck nimmt und prüft, ob dieser ein Wort repräsentiert (siehe Vorlesungsfolie 567). Falls ja, soll die Funktion `Some` mit dem Wort (Liste der Zeichen im Wort) zurückgeben, falls nein, `None`.

   Beispiele:

   ```
   isWord Eps = Some []
   isWord (Lit 'a') = Some ['a']
   isWord (Cat (Lit 'a', Lit 'b')) = Some ['a'; 'b']
   isWord (Or (Lit 'a', Lit 'b')) = None
   ```

```
let rec isWord<'a>(r: RegEx<'a>): Option<List<'a>> =
    match r with
    | Eps -> Some []
    | Lit a -> Some [a]
    | Cat (r1, r2) ->
        match (isWord r1, isWord r2) with
        | (Some l1, Some l2) -> Some (l1 @ l2)
        | _ -> None
    | _ -> None
```

b) Schreiben Sie eine Funktion `reduceStep<'a>: RegEx<'a> -> List<RegEx<'a>>`, die alle regulären Ausdrücke zurückgibt, zu denen der gegebene reguläre Ausdruck *in einem Schritt* reduzieren kann (siehe Vorlesungsfolien 568 und 570). Wenn keine Reduktionsregel anwendbar ist, dann soll die leere Liste zurückgegeben werden.

Hinweis: Da die Reihenfolge der regulären Ausdrücke in der Liste keine Rolle spielt, gelten die Beispiele bis auf Permutationen.

Beispiel:

```
reduceStep Eps = []
reduceStep (Lit 'a') = []
reduceStep (Cat (Lit 'a', Eps)) = [Lit 'a']
reduceStep (Cat (Lit 'a', Lit 'b')) = []
reduceStep (Or (Lit 'a', Lit 'b')) = [Lit 'a'; Lit 'b']
reduceStep (Star (Lit 'a')) = [Eps; Cat (Lit 'a', Star (Lit 'a'))]
```

```
let rec reduceStep<'a>(r: RegEx<'a>): List<RegEx<'a>> =
    match r with
    | Cat (r, Eps) | Cat (Eps, r) -> [r]
    | Cat (r1, r2) ->
        List.map (fun r1' -> Cat (r1', r2)) (reduceStep r1) @
        List.map (fun r2' -> Cat (r1, r2')) (reduceStep r2)
    | Or (r1, r2) -> [r1; r2]
    | Star r -> [Eps; Cat (r, Star r)]
    | _ -> []
```

c) Schreiben Sie eine Funktion `reduce<'a>: RegEx<'a> -> Nat -> List<RegEx<'a>>`, die einen regulären Ausdruck und eine natürliche Zahl `n` nimmt und alle möglichen Reduktionen des regulären Ausdrucks, die in höchstens `n` Schritten erreicht werden können, zurückgibt.

Zur Vermeidung doppelter Rechenarbeit soll die in den einzelnen Rekursionsschritten zurückgegebene Liste keine Duplikate enthalten. Verwenden Sie hierzu die Bibliotheksfunktion `List.distict`[1].

Beispiel:

```
reduce (Or (Lit 'a', Lit 'b')) 0 = [Or (Lit 'a', Lit 'b')]
reduce (Or (Lit 'a', Lit 'b')) 1 = [Lit 'a'; Lit 'b'; Or (Lit 'a', Lit 'b')]
reduce (Or (Lit 'a', Lit 'b')) 2 = [Lit 'a'; Lit 'b'; Or (Lit 'a', Lit 'b')]
reduce (Cat (Or (Lit 'a', Lit 'b'), Lit 'c')) 2 =
  [Cat (Lit 'a', Lit 'c'); Cat (Lit 'b', Lit 'c'); Cat (Or (Lit 'a', Lit 'b'); Lit 'c')]
reduce (Star (Lit 'a')) 2 =
  [Star (Lit 'a'); Cat (Lit 'a', Star (Lit 'a')); Eps;
   Cat (Lit 'a', Cat (Lit 'a', Star (Lit 'a'))); Cat (Lit 'a', Eps)]
```

```
let rec reduce<'a when 'a: equality>(r: RegEx<'a>) (n: Nat): List<RegEx<'a>> =
    if n = 0N then [r]
    else r :: List.collect reduceStep (reduce r (n - 1N)) |> List.distinct
```

[1] https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-listmodule.html#distinct

d) Schreiben Sie eine Funktion `words<'a>: RegEx<'a> -> Nat -> List<List<'a>>`, die einen regulären Ausdruck und eine natürliche Zahl `n` nimmt und alle Wörter zurückgibt, zu denen der reguläre Ausdruck in höchstens `n` Schritten reduzieren kann.

Beispiel:

```
words Empty 10 = []
words (Or (Lit 'a', Lit 'b')) 0 = []
words (Or (Lit 'a', Lit 'b')) 1 = [['a']; ['b']]
words (Cat (Or (Lit 'a', Lit 'b'), Lit 'c')) 1 = [['a'; 'c']; ['b'; 'c']]
words (Star (Or (Lit 'a', Lit 'b'))) 2 = [[]]
words (Star (Or (Lit 'a', Lit 'b'))) 3 = [[]; ['a']; ['b']]
words (Star (Or (Lit 'a', Lit 'b'))) 5 =
  [[]; ['a']; ['b']; ['a'; 'a']; ['a'; 'b']; ['b'; 'a']; ['b'; 'b']]
```

```
let rec words<'a when 'a: equality>(r: RegEx<'a>) (n: Nat): List<List<'a>> =
    reduce r n |> List.choose isWord |> List.distinct

// List.choose wendet isWord auf alle Elemente der Liste an, entfernt die None
// Einträge und gibt die verbleibenden Einträge jeweils ohne das Some zurück.
// Statt List.choose können wir auch eine eigene Hilfsfunktion verwenden:
let rec words'<'a when 'a: equality>(r: RegEx<'a>) (n: Nat): List<List<'a>> =
    let rec choose (xs: List<RegEx<'a>>): List<List<'a>> =
        match xs with
        | []    -> []
        | x::xs -> match isWord x with
                   | Some w -> w :: choose xs
                   | None   ->      choose xs
    reduce r n |> choose |> List.distinct
```

e) Schreiben Sie eine Funktion `generates<'a>: RegEx<'a> -> List<'a> -> Nat -> Bool`, die einen regulären Ausdruck, ein Wort und eine natürliche Zahl `n` nimmt und prüft, ob das Wort in höchstens `n` Schritten durch Reduktion des regulären Ausdrucks erreicht werden kann.

Beispiel:

```
generates (Or (Lit 'a', Lit 'b')) ['a'] 0 = false
generates (Or (Lit 'a', Lit 'b')) ['a'] 1 = true
generates (Or (Lit 'a', Lit 'b')) ['b'] 1 = true
generates (Or (Lit 'a', Lit 'b')) ['c'] 1 = false
generates (Cat (Or (Lit 'a', Lit 'b'), Lit 'c')) ['a'; 'c'] 1 = true
generates (Star (Or (Lit 'a', Lit 'b'))) ['a'] 2 = false
generates (Star (Or (Lit 'a', Lit 'b'))) ['a'; 'a'] 5 = true
```

```
let rec generates<'a when 'a: equality>(r: RegEx<'a>) (word: List<'a>) (n: Nat): Bool =
    words r n |> List.contains word
```

f) *Freiwillige Zusatzaufgabe:* Warum brauchen wir bei `reduce`, `words` und `generates` eine Begrenzung der Schritte? Was wäre, wenn wir diese Begrenzung weg lassen würden?

Die Reduktionsregel für r^* erlaubt es, den Ausdruck r beliebig oft zu wiederholen. Ohne eine Begrenzung der Schritte würde `reduce (Star (Lit 'a'))` nicht terminieren, da es immer wieder zu einem rekursiven Aufruf mit unverändertem Parameter kommt. Die anderen beiden Funktionen bauen auf `reduce` auf und müssen die Begrenzung an `reduce` weiterreichen.

g) *Freiwillige Zusatzaufgabe:* Erweitern Sie den Typ `RegEx<'a>` um das neue Konstrukt aus Aufgabe 2. Passen Sie außerdem die Funktionen aus den vorherigen Teilaufgaben entsprechend an.

Hinweis: Zu dieser Teilaufgabe gibt es keine Testfälle, es werden nur Eingaben mit dem ursprünglichen Typ getestet. Die ursprünglichen Varianten dürfen Sie auch nicht umbenennen/verändern, sonst gibt es Kompilierfehler bei den Testfällen. Fügen Sie nur die neue Variante zu dem bestehenden Typ hinzu.

Der erweiterte Typ kann wie folgendermaßen aussehen:

```
type RegEx<'a> =
    | Eps
    | Lit of 'a
    | Cat of RegEx<'a> * RegEx<'a>
    | Empty
    | Or of RegEx<'a> * RegEx<'a>
    | Star of RegEx<'a>
    | Range of Nat * Nat * RegEx<'a>
```

Sinnvollerweise sind reguläre Ausdrücke der Form $r\{n,m\}$ keine Wörter, weshalb keine Erweiterung von `isWord` notwendig ist. Der neue Fall ist bereits durch das Muster _ mit abgedeckt.

Die Funktion `reduceStep` erweitern wir entsprechend der Lösung von Aufgabe 2 um folgende Muster vor dem Fehlerfall `_ -> []`:

```
  | Range (n, m, r) when n>0N && m>0N    -> [Cat (r, Range (n - 1N, m - 1N, r))]
  | Range (n, m, r) when n=0N && m>0N    -> [Eps; Cat (r, Range (0N, m - 1N, r))]
  | Range (n, m, r)  (*  n=0N && m=0N *) -> [Eps]
```

Die anderen Funktionen müssen nicht angepasst werden.

Aufgabe 4 Statische Semantik (Einreichaufgabe, 8 Punkte)

Harry Hacker sollte folgende Aussage der Statischen Semantik vervollständigen und einen vollständigen Beweisbaum angeben, der seine Aussage zeigt:

$$\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}\} \vdash \mathtt{let\ f\ (x:\ Nat):}\ ____ = \mathtt{f\ x} : ______$$

Harry Hacker's Beweisbaum sieht wie folgt aus (er hat unterstrichen, was er in die Lücken eingefüllt hat):

$$\dfrac{\dfrac{\overline{\{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}, \mathtt{x} \mapsto \text{Nat}\} \vdash \mathtt{f} : \text{Nat} \to \text{Bool}} \quad \overline{\{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}, \mathtt{x} \mapsto \text{Nat}\} \vdash \mathtt{x} : \text{Nat}}}{\{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}, \mathtt{x} \mapsto \text{Nat}\} \vdash \mathtt{f\ x} : \text{Bool}}}{\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}\} \vdash \mathtt{let\ f\ (x:\ Nat):}\ \underline{\text{Bool}} = \mathtt{f\ x} : \underline{\{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}\}}}$$

Lisa Lista hat jedoch den Verdacht, dass sich da ein Fehler eingeschlichen hat ...

a) Finden Sie den Fehler in Harry Hacker's Beweisbaum. Geben Sie an, bei welcher Regelanwendung der Fehler passiert ist und was genau falsch gemacht wurde.

Direkt nach der ersten Regelanwendung ist die Signatur falsch. Es muss die obere Regel von Vorlesungsfolie 172 angewendet werden, hierbei ergibt sich für die Voraussetzung die Signatur

$$\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}\}, \{\mathtt{x} \mapsto \text{Nat}\} = \{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}, \mathtt{x} \mapsto \text{Nat}\}$$

b) Korrigieren Sie Harry Hacker's Fehler. Füllen Sie die Lücken richtig aus und geben Sie einen vollständigen Beweisbaum an. Markieren Sie die Unterschiede im Baum (Textmarker oder andere Schriftfarbe).

Link zum Baum

$$\dfrac{\dfrac{\overline{\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}, \mathtt{x} \mapsto \text{Nat}\} \vdash \mathtt{f} : \text{Nat} \to \text{Nat}} \quad \overline{\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}, \mathtt{x} \mapsto \text{Nat}\} \vdash \mathtt{x} : \text{Nat}}}{\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}, \mathtt{x} \mapsto \text{Nat}\} \vdash \mathtt{f\ x} : \text{Nat}}}{\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}\} \vdash \mathtt{let\ f\ (x:\ Nat):}\ \underline{\text{Nat}} = \mathtt{f\ x} : \underline{\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}\}}}$$

c) Harry Hacker möchte die Lücken unbedingt mit Bool und $\{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}\}$ ausfüllen. Welche Änderungen kann er *am Ausdruck bzw. der Deklaration* in der untersten Zeile seines Beweisbaumes vornehmen, damit der Beweisbaum ohne weitere Änderungen gültig wird? Die von Harry Hacker ausgefüllten Lücken sowie die Start-Signatur $\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}\}$ dürfen nicht verändert werden!

Geben Sie nicht den kompletten Beweisbaum, sondern nur die unterste Zeile ab. Erklären Sie außerdem, warum der Beweisbaum durch Ihre Änderung gültig geworden ist.

$\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}\} \vdash \mathtt{let\ rec\ f\ (x:\ Nat):\ Bool} = \mathtt{f\ x} : \{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}\}$

Mit einer rekursiven Funktionsdeklaration kommt die obere Regel statt der unteren Regel von Vorlesungsfolie 195 zum Einsatz. Hierbei wird die Signatur nicht nur um einen Eintrag für den Bezeichner x des Funktionsparameters ergänzt, sondern auch um einen Eintrag für den Bezeichner f des Funktionsnamens. Es ergibt sich somit genau die Signatur, die Harry Hacker in seinem fehlerhaften Beweisbaum hatte:

$$\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}\}, \{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}, \mathtt{x} \mapsto \text{Nat}\} = \{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}, \mathtt{x} \mapsto \text{Nat}\}$$

Der Rest seines Beweisbaumes war bereits korrekt, sodass der Beweisbaum nun insgesamt korrekt ist. Hier der komplette Beweisbaum (sollte nicht abgegeben werden!): Link zum Baum

$$\dfrac{\dfrac{\overline{\{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}, \mathtt{x} \mapsto \text{Nat}\} \vdash \mathtt{f} : \text{Nat} \to \text{Bool}} \quad \overline{\{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}, \mathtt{x} \mapsto \text{Nat}\} \vdash \mathtt{x} : \text{Nat}}}{\{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}, \mathtt{x} \mapsto \text{Nat}\} \vdash \mathtt{f\ x} : \text{Bool}}}{\{\mathtt{f} \mapsto \text{Nat} \to \text{Nat}\} \vdash \mathtt{let\ rec\ f\ (x:\ Nat):\ Bool} = \mathtt{f\ x} : \{\mathtt{f} \mapsto \text{Nat} \to \text{Bool}\}}$$

Aufgabe 5 Rekursion mit Wörtern aus Zahlen (Einreichaufgabe, 5 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 2, 4, 6, 8 oder 10 sind, dann steht Ihnen zur Bearbeitung dieser Aufgabe unser Rekursions-Tutor zur Verfügung, siehe Hinweis auf der ersten Seite von Übungsblatt 4.

Betrachten Sie den folgenden Typ, der Wörter aus natürlichen Zahlen modelliert:

```
type Number =
    | Empty
    | Cons of Nat * Number
```

Betrachten Sie nun die folgende Funktion, die eine 'Number' übergeben bekommt und 'true' zurückgeben soll, wenn die 'Number' ein bestimmtes Schema bzw. eine bestimmte Sprache erfüllt und 'false' wenn nicht.

```
let rec f (n: Number): Number =
    match n with
    | Cons (x, rest) ->
        if x = 1 then match rest with
                     | Cons (y, rest2) -> if y = 2 then f rest2 else n
                     | Empty           -> n
        else n

let rec g (n: Number): Number =
    match n with
    | Cons (x, rest) -> if x = 3 then g rest else n

let rec matchNumber (n: Number): Bool =
    match n with
    | Empty -> false
    | Cons _ ->
        match g (f n) with
        | Cons (x, rest) -> if x = 1 then match rest with
                                          | Empty  -> true
                                          | Cons _ -> false
                            else false
        | Empty -> false
```

a) Welche Sprache wird von der Funktion `matchNumber` akzeptiert? Begründen Sie Ihre Antwort mit der Funktionsweise der Funktion.

Die Funktion akzeptiert die Sprache $(1 \cdot 2)^* \cdot 3^* \cdot 1$. Die Funktion `f` verkürzt die Zahl um die Sequenz $(1 \cdot 2)$ so oft es geht und bricht ab, sobald dieses Muster nicht mehr erfüllbar ist. Die Funktion `g` verkürzt die Zahl um beliebig viele 3. Die Funktion `matchNumbers` überprüft, ob die Zahl, nachdem `f` und `g` angewendet wurden, nur noch aus einer 1 besteht und gibt nur genau dann **`true`** zurück.

b) Nennen Sie je zwei Zahlen, die von der Funktion akzeptiert werden, wobei eine der beiden Zahlen mindestens 10 Ziffern lang sein muss. Sie dürfen die Zahlen abkürzen: 'Cons (3, Cons (2, Cons (1, Empty)))' dürfen Sie z.B. als 321 schreiben.

Beispiele: 1, 1231, 121231, 12331, 12121, 1212123331, ...

c) Nenne Sie eine Zahl, die **nicht** von der Funktion akzeptiert wird.

Beispiele: 2, 112, 131, 123, 311, ...