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 8: Grundlagen der Programmierung (WS 2024/25)

Sprechstunden zu den Übungen Sie haben Schwierigkeiten mit den Übungsaufgaben und machen sich Sorgen, dass es Ihnen nicht gelingen wird die zur Klausurzulassung nötigen 60% der erreichbaren Punkte zu erlangen?

Dann besuchen Sie unsere Sprechstunden zu den Übungen! Dort erhalten Sie Tipps und Lösungshinweise, wenn Sie mit einer Aufgabe nicht weiterkommen. Sie können dort auch zu früheren Aufgaben Fragen stellen. Alle Informationen zu den Übungssprechstunden finden Sie auf unserer Homepage.

Aufgabe 1 Reguläre Ausdrücke (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie sich mit regulären Ausdrücken beschäftigen. Die Aufgabe soll Ihnen dabei helfen den Weg von einem regulären Ausdruck schrittweise bis zu einer Akzeptorfunktion nachzuvollziehen. Sie können sich an den Vorlesungsfolien 553 bis 623 sowie am Skript Kapitel 6.1 und 6.2 orientieren.

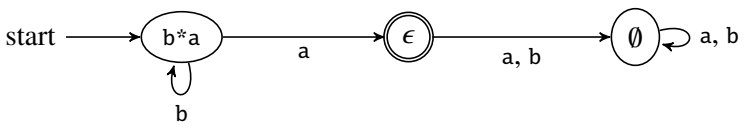
Wir betrachten den regulären Ausdruck `b*a` über dem Alphabet Alphabet $A = \{a, b\}$.

a) Bestimmen Sie **alle** Rechtsfaktoren (inkl. Rechtsfaktoren der Ergebnisse). Geben Sie dabei in der Rechnung jeweils den ersten Schritt explizit an, nachfolgende Zwischenschritte dürfen Sie zusammenfassen.

```
a \ b*a = (a\b*)a | a\a
        = (a\b)b*a | ϵ
        = ∅ | ϵ
        = ϵ

b \ b*a = (b\b*)a | b\a
        = (b\b)b*a | ∅
        = b*a

a \ ϵ = ∅
b \ ϵ = ∅

a \ ∅ = ∅
b \ ∅ = ∅
```

b) Zeichnen Sie den Aufrufgraphen für den Akzeptor (wie auf Vorlesungsfolie 599).

Umranden Sie Ausdrücke, die nullable sind, doppelt. Wenn wir beim Einlesen eines Wortes an einem solchen nullable Ausdruck landen und keine weitere Eingabe mehr folgt, gehört das eingelesene Wort zur durch den regulären Ausdruck beschriebenen Sprache. Durch die doppelte Umrandung können wir einfacher ablesen, dass wir an einem möglichen Ende angekommen sind (daher werden solche Knoten auch „Endzustände“ genannt).



c) Implementieren Sie die Akzeptorfunktionen. Gehen Sie dabei **streng nach dem Verfahren aus der Vorlesung** vor (Folie 600). Nutzen Sie für das Alphabet den Typ `type Alphabet = | A | B`.

Hinweis: Wir empfehlen die einzelnen Akzeptorfunktionen als verschränkt rekursive Hilfsfunktionen innerhlab von `accept` zu definieren und am Ende die Start-Akzeptorfunktion mit der Eingabe aufzurufen.

```
let accept (input: List<Alphabet>): Bool =
    let rec accept0 (input: List<Alphabet>): Bool = // B*A
        match input with
        | [] -> false
        | A::rest -> accept1 rest
        | B::rest -> accept0 rest
    and accept1 (input: List<Alphabet>): Bool = // ϵ
        match input with
        | [] -> true
        | A::rest -> accept2 rest
        | B::rest -> accept2 rest
    and accept2 (input: List<Alphabet>): Bool = // ∅
        match input with
        | [] -> false
        | A::rest -> accept2 rest
        | B::rest -> accept2 rest
    accept0 input
```

Aufgabe 2 Weihnachtsbaum schmücken (Einreichaufgabe, 11 Punkte)

Motivation: In dieser Aufgabe arbeiten Sie in weihnachtlicher Stimmung mit Bäumen und Listen.

Schreiben Sie Ihre Lösungen in die Datei `Weihnachtsbaum.fs` *aus der Vorlage* `Aufgabe-8-2.zip`.

Wir betrachten folgenden Typ für Bäume:

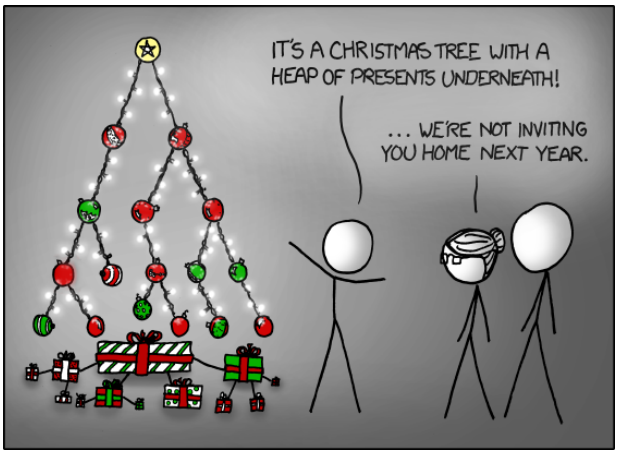
```
type Tree<'a> =
    | Leaf
    | ENode of Tree<'a> *      Tree<'a>
    | Node  of Tree<'a> * 'a * Tree<'a>
```

`ENode` Knoten (empty node) können, genauso wie Blätter, keine Elemente aufnehmen. `Node` Knoten möchten wir mit folgendem Schmuck versehen:

```
type Schmuck =
    | Kugel
    | Lametta
```

Ein Weihnachtsbaum ist ein geschmückter Baum:

```
type Weihnachtsbaum = Tree<Schmuck>
```



Tree, xkcd.com/835, CC BY-NC 2.5
Der dargestellte Baum entspricht *nicht* unserem Datentyp!

Ziel ist es, einen Baum so zu schmücken, dass er balanciert ist, sodass er nicht umfällt. Dazu muss das Gewicht des Schmucks beachtet werden: Eine `Kugel` hat ein Gewicht von `2` und `Lametta` das Gewicht `1`.

a) Schreiben Sie eine Funktion `schmuckGewicht: Schmuck -> Nat`, die das Gewicht des Schmucks berechnet.

```
let schmuckGewicht (schmuck: Schmuck): Nat =
    match schmuck with
    | Kugel   -> 2N
    | Lametta -> 1N
```

b) Schreiben Sie eine Funktion `baumGewicht: Weihnachtsbaum -> Nat`, die das Gesamtgewicht des Schmucks am Baum berechnet.

```
let rec baumGewicht (tree: Weihnachtsbaum): Nat =
    match tree with
    | Leaf            -> 0N
    | ENode (l,    r) -> baumGewicht l +                    baumGewicht r
    | Node  (l, s, r) -> baumGewicht l + schmuckGewicht s + baumGewicht r
```

c) Schreiben Sie eine Funktion `istBalanciert: Weihnachtsbaum -> Bool`, die überprüft, ob der Baum balanciert ist. Ein Baum ist balanciert, wenn das Gewicht des linken Teilbaums gleich dem Gewicht des rechten Teilbaums ist und beide Teilbäume ebenfalls balanciert sind.

```
let rec istBalanciert (tree: Weihnachtsbaum): Bool =
    match tree with
    | Leaf -> true
    | ENode (l, r) | Node  (l, _, r) ->
        baumGewicht l = baumGewicht r && istBalanciert l && istBalanciert r

// Effizientere Lösung, jeder Knoten wird nur einmal betrachtet
let istBalanciert' (tree: Weihnachtsbaum): Bool =
    // Rückgabe enthält auch das Gewicht, wenn der Baum balanciert ist
    let rec balanciertMitGewicht (tree: Weihnachtsbaum): Option<Nat> =
        match tree with
        | Leaf -> Some 0N
        | ENode (l, r) ->
            match (balanciertMitGewicht l, balanciertMitGewicht r) with
            | (Some gl, Some gr) when gl = gr -> Some (gl + gr)
            | _                              -> None
        | Node (l, s, r) ->
            match (balanciertMitGewicht l, balanciertMitGewicht r) with
            | (Some gl, Some gr) when gl = gr -> Some (gl + schmuckGewicht s + gr)
            | _                              -> None

    Option.isSome (balanciertMitGewicht tree)
```

d) Schreiben Sie eine Funktion `moeglicheGewichte<'a>: Tree<'a> -> List<Nat>`, die eine Liste aller möglichen Gewichte des Baums zurückgibt, wenn man ihn so schmückt, dass er balanciert ist.

Hinweis: Überlegen Sie, welche Gewichte möglich sind, wenn Sie die möglichen Gewichte des rechten und linken Teilbaums kennen.

```
let rec moeglicheGewichte<'a> (tree: Tree<'a>): List<Nat> =
    // Hilfsfunktion zur Berechnung der Schnittmenge von zwei Listen
    let intersect (xs: List<Nat>) (ys: List<Nat>): List<Nat> =
        xs |> List.filter (fun x -> List.contains x ys)

    match tree with
    | Leaf           -> [0N]
    | ENode (l,   r) -> intersect (moeglicheGewichte l) (moeglicheGewichte r)
                        |> List.map (fun g ->
                            g + g // Gesamtgewicht: links + rechts (beides g)
                        )
    | Node (l, _, r) -> intersect (moeglicheGewichte l) (moeglicheGewichte r)
                        |> List.collect (fun g -> [
                            g + 1N + g; // Lametta hinzufügen erhöht Gewicht um 1
                            g + 2N + g  // Kugel   hinzufügen erhöht Gewicht um 2
                        ])
                        |> List.distinct // Optional: Duplikate entfernen
```

e) Schreiben Sie eine Funktion `schmuecken: Tree<Unit> -> Nat -> Option<Weihnachtsbaum>`, die einen Baum und ein Zielgewicht nimmt und den Baum geschmückt zurückgibt, sodass das Gesamtgewicht des Baums gleich dem Zielgewicht ist und der geschmückte Baum balanciert ist. Falls dies nicht möglich ist, soll `None` zurückgegeben werden.

```
let rec schmuecken (tree: Tree<Unit>) (g: Nat): Option<Weihnachtsbaum> =
    match tree with
    | Leaf when g = 0N -> Some Leaf

    | ENode (l, r) when g % 2N = 0N -> // Nur möglich bei geradem Zielgewicht
        let g' = g / 2N // Zielgewicht hälftig auf beide Teilbäume verteilen

        // Rekursiv beide Teilbäume schmücken und dann mit ENode zusammensetzen
        match (schmuecken l g', schmuecken r g') with
        | (Some l', Some r') -> Some (ENode (l', r'))
        | _                  -> None

    | Node (l, _, r) when g > 0N -> // Nur möglich bei positivem Zielgewicht
        // Bestimme Schmuck im Node und Zielgewicht der beiden Teilbäume
        let (s, g') = if g % 2N = 0N then (Kugel,   (g - 2N) / 2N)
                                     else (Lametta, (g - 1N) / 2N)

        // Rekursiv beide Teilbäume schmücken und dann mit Node zusammensetzen
        match (schmuecken l g', schmuecken r g') with
        | (Some l', Some r') -> Some (Node (l', s, r'))
        | _                  -> None

    | _ -> None // falls die "when" Bedingungen nicht erfüllt sind
```

f) Schreiben Sie eine Funktion `schmueckungen: Tree<Unit> -> List<Weihnachtsbaum>`, die alle möglichen balancierten Schmückungen des Baums zurückgibt.

```
let schmueckungen (tree: Tree<Unit>): List<Weihnachtsbaum> =
    tree |> moeglicheGewichte |> List.choose (schmuecken tree)
```

Statt zuerst die möglichen Gewichte zu berechnen und dann in einem nächsten Schritt den Baum entsprechend zu schmücken, kann man auch direkt in einem Durchgang die möglichen balancierten Schmückungen berechnen. Wir erweitern dazu den Rückgabetyp um das Gewicht.

```
let rec schmueckungenMitGewicht (tree: Tree<'a>): List<Nat * Weihnachtsbaum> =
  // Hilfsfunktion: Finde in zwei Listen (mögliche linke und rechte Teilbäume)
  // die Baum-Paare mit gleichem Schmuck-Gewicht. Rückgabe-Liste enthält Tupel
  // der Form (Gesamtgewicht, linker Teilbaum, rechter Teilbaum).
  let findePaare
    (ls: List<Nat * Weihnachtsbaum>)
    (rs: List<Nat * Weihnachtsbaum>): List<Nat * Weihnachtsbaum * Weihnachtsbaum> =
      // Kreuzprodukt von ls und rs bilden
      ls |> List.collect (fun l -> rs |> List.map (fun r -> (l, r)))
      // Filter: Nur Paare mit gleichem Schmuck-Gewicht
      |> List.filter (fun ((gl, _), (gr, _)) -> gl = gr)
      // Umformung: Schmuck-Gewichte heraus ziehen und addieren
      |> List.map (fun ((gl, l), (gr, r)) -> (gl + gr, l, r))

  match tree with
  | Leaf           -> [(0N, Leaf)]
  | ENode (l,   r) -> findePaare (schmueckungenMitGewicht l) (schmueckungenMitGewicht r)
                      |> List.map (fun (g, l', r') ->
                          // Geschmückte Teilbäume wieder mit ENode zusammensetzen
                          (g, ENode (l', r'))
                      )
  | Node (l, _, r) -> findePaare (schmueckungenMitGewicht l) (schmueckungenMitGewicht r)
                      |> List.collect (fun (g, l', r') -> [
                          // Geschmückte Teilbäume wieder mit Node zusammensetzen
                          (g + 1N, Node (l', Lametta, r')); // Lametta im Node wiegt 1
                          (g + 2N, Node (l', Kugel,   r'))  // Kugel   im Node wiegt 2
                      ])
```

Daraus lassen sich nun die drei Teilaufgaben d, e und f ableiten:

```
let moeglicheGewichte'<'a> (tree: Tree<'a>): List<Nat> =
    schmueckungenMitGewicht tree
    |> List.map fst // Gewicht extrahieren

let schmuecken' (tree: Tree<Unit>) (g: Nat): Option<Weihnachtsbaum> =
    schmueckungenMitGewicht tree
    |> List.tryFind (fun (g', _) -> g' = g) // Nach Zielgewicht filtern
    |> Option.map snd // Geschmückten Baum extrahieren

let schmueckungen' (tree: Tree<Unit>): List<Weihnachtsbaum> =
    schmueckungenMitGewicht tree
    |> List.map snd // Geschmückten Baum extrahieren
```

Aufgabe 3 Reguläre Ausdrücke (Einreichaufgabe, 10 Punkte)

Motivation: In dieser Aufgabe sollen Sie sich mit regulären Ausdrücken beschäftigen. Die Aufgabe soll Ihnen dabei helfen den Weg von einem regulären Ausdruck schrittweise bis zu einer Akzeptorfunktion nachzuvollziehen. Sie können sich an den Vorlesungsfolien 553 bis 623 sowie am Skript Kapitel 6.1 und 6.2 orientieren.

Praxistipp: Das UNIX- bzw. Linuxprogramm `grep`[1] erlaubt die Suche in Dateien und Datenströmen anhand von regulären Ausdrücken. Unter Windows stellt das PowerShell Kommando `Select-String -Pattern` eine ähnliche Funktionalität zur Verfügung.

Hinweis: Die Präzedenz der Operatoren ist wie folgt definiert: Die Alternative (`|`) bindet am schwächsten, dann folgt die Konkatenation ($\cdot$) und zuletzt der Stern (`*`).

Schreiben Sie Ihre Lösungen in die Datei `RegExp.fs` aus der Vorlage `Aufgabe-8-3.zip`.

Wir betrachten den regulären Ausdruck $((\mathsf{a}|\mathsf{aa})|\varepsilon)((\mathsf{b}|\mathsf{bb})(\mathsf{a}|\mathsf{aa}))*$ über dem Alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}\}$.

a) Bestimmen Sie **alle** Rechtsfaktoren (inkl. Rechtsfaktoren der Ergebnisse). Geben Sie dabei in der Rechnung jeweils den ersten Schritt explizit an, nachfolgende Zwischenschritte dürfen Sie zusammenfassen.

Hinweis: Verwenden Sie als Abkürzung `A = (a|aa)` und `B = (b|bb)`, sonst wird die Berechnung unübersichtlich. Wir merken ausserdem an, dass:

```
a \ A = (ε|a)
b \ A = ∅
b \ B = (ε|b)
a \ B = ∅
```

Wir erinnern zusätzlich an die *Rechenregeln* für reguläre Ausdrücke, die es erlauben Ausdrücke zu vereinfachen.

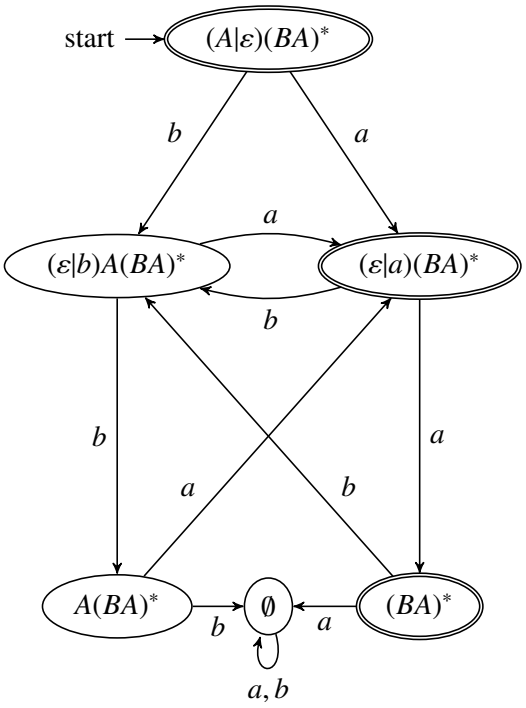
```
a \ (A|ε)(BA)* <nullable (A|ε) = true>
  = (a \ (A|ε) (BA)*) | (a \ (BA)*)
  = (ε|a)(BA)*
b \ (A|ε)(BA)*
  = (b \ (A|ε) (BA)*) | (b \ (BA)*)
  = (ε|b)A(BA)*

a \ (ε|a)(BA)*
  = (a \ (ε|a)(BA)*) | (a \ (BA)*)
  = (BA)*
b \ (ε|a)(BA)* <nullable (ε|a) = true>
  = (b \ (ε|a))(BA)* | (b \ (BA)*)
  = (ε|b)A(BA)*

a \ (BA)* = a \ (BA)(BA)*
          = ∅
b \ (BA)* = b \ (BA))(BA)*
          = (ε|b)A(BA)*

a \ (ε|b)A(BA)* = (a \ (ε|b)A)(BA)*
                = (ε|a)(BA)*
b \ (ε|b)A(BA)* = (b \ (ε|b)A)(BA)*
                = A(BA)*

a \ A(BA)* = (a \ A)(BA)*
           = (ε|a)(BA)*
b \ A(BA)* = (b \ A)(BA)*
           = ∅
```

Erinnerung: ε ist das neutrale Element der Konkatenation, $\emptyset$ ist das neutrale Element der Alternative, und das Nullelement der Konkatenation.

[1] https://de.wikipedia.org/wiki/Grep

b) Zeichnen Sie den Aufrufgraphen für den Akzeptor (wie auf Vorlesungsfolie 599).

Umranden Sie Ausdrücke, die nullable sind, doppelt. Wenn wir beim Einlesen eines Wortes an einem solchen nullable Ausdruck landen und keine weitere Eingabe mehr folgt, gehört das eingelesene Wort zur durch den regulären Ausdruck beschriebenen Sprache. Durch die doppelte Umrandung können wir einfacher ablesen, dass wir an einem möglichen Ende angekommen sind (daher werden solche Knoten auch „Endzustände“ genannt).



c) Implementieren Sie die Akzeptorfunktionen. Gehen Sie dabei **streng nach dem Verfahren aus der Vorlesung** vor (Folie 601). Nutzen Sie für das Alphabet den Typ `type Alphabet = | A | B`.

Hinweis: Wir empfehlen die einzelnen Akzeptorfunktionen als verschränkt rekursive Hilfsfunktionen innerhlab von `accept` *zu definieren und am Ende die Start-Akzeptorfunktion mit der Eingabe aufzurufen.*

```
let accept (input: List<Alphabet>): bool =
    let rec accept0 (input: List<Alphabet>): bool = // (ε|b)A(BA)*
        match input with
        | [] -> false
        | A::rest -> accept1 rest // (ε|a)(BA)*
        | B::rest -> accept2 rest // A(BA)*
    and accept1 (input: List<Alphabet>): bool = // (ε|a)(BA)*
        match input with
        | [] -> true
        | A::rest -> accept5 rest // (BA)*
        | B::rest -> accept0 rest // (ε|b)A(BA)*
    and accept2 (input: List<Alphabet>): bool = // A(BA)*
        match input with
        | [] -> false
        | A::rest -> accept1 rest // (ε|a)(BA)*
        | B::rest -> accept4 rest // ∅
    and accept3 (input: List<Alphabet>): bool = // (A|ε)(BA)*
        match input with
        | [] -> true
        | A::rest -> accept1 rest // (ε|a)(BA)*
        | B::rest -> accept0 rest // (ε|b)A(BA)*
    and accept4 (input: List<Alphabet>): bool = // ∅
        match input with
        | [] -> false
        | A::rest -> accept4 rest // ∅
        | B::rest -> accept4 rest // ∅
    and accept5 (input: List<Alphabet>): bool = // (BA)*
        match input with
        | [] -> true
        | A::rest -> accept4 rest // ∅
        | B::rest -> accept0 rest // (ε|b)A(BA)*
    accept3 input
```