Prof. Dr. Ralf Hinze
Cass Alexandru, M.Sc.
Alexander Dinges, M.Sc.
Felix Winkler, M.Sc.

RPTU
Fachbereich Informatik
AG Programmiersprachen

Lösungshinweise/-vorschläge zum Übungsblatt 8: Konzepte der Programmierung (WS 2025/26)

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 552 bis 622 sowie am Skript Kapitel 6.1 und 6.2 orientieren.

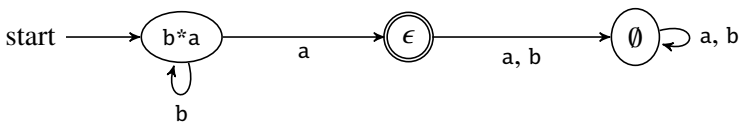
Wir betrachten den regulären Ausdruck `b*a` über dem Alphabet $A = \{a, b\}$.

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 innerhalb 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 Santa Claus is Coming to Town (Einreichaufgabe, 17 Punkte)

Motivation: In dieser Aufgabe beschäftigen wir uns mit dem algorithmischen Problem des kürzesten Pfades zwischen einer Menge von Orten. Die Problemstellung wird mit Hilfe von Listen und Records modelliert. Sie können sich an den Vorlesungsfolien 428 bis 542 sowie am Skript Kapitel 5 orientieren.

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

Santa Claus muss an Weihnachten seine Geschenke in einer Stadt verteilen, deren Straßen ein perfektes Quadratraster bilden. Er hat eine Liste von Personen, die er beschenken möchte und da er nur wenig Zeit hat, müssen wir ihm helfen den kürzesten Pfad zwischen den Wohnorten dieser Personen zu finden.

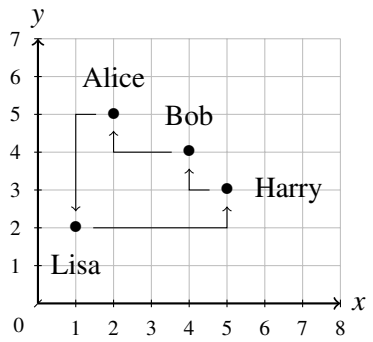
Santa stellt uns als Eingabe eine Liste von Personen mit deren Wohnort zur Verfügung, daher verwenden wir folgende Typdefinitionen:

```
type Point = {
    x: Nat
    y: Nat
}

type Person = {
    name: String
    location: Point
}
```

Seine Liste könnte wie folgt aussehen:

```
let lisa  = {name="Lisa" ; location = {x = 1N; y = 2N}}
let alice = {name="Alice"; location = {x = 2N; y = 5N}}
let harry = {name="Harry"; location = {x = 5N; y = 3N}}
let bob   = {name="Bob"  ; location = {x = 4N; y = 4N}}
let santasList = [lisa; alice; harry; bob]
```

In der nebenstehenden Abbildung ist ein kürzester Pfad für diese Liste dargestellt. Im Rahmen der Aufgabe spielt es keine Rolle von welchem Ort der Liste Santa startet.



Hinweise:

- Mit den Funktionen aus dem List-Modul[1] können Sie an einigen Stellen kürzeren und besser verständlichen Programmcode schreiben.
- Zur besseren Lesbarkeit von verketteten Funktionsaufrufen gibt es in F# den sogenannten *Forward Pipe Operator*. Der Ausdruck `x |> f` ist eine syntaktisch andere Schreibweise für `f x`, die Funktion `f` wird also auf den Wert `x` angewandt. Zum Beispiel lässt sich damit der Ausdruck

  ```
  List.filter (fun x -> x % 2 = 0) (List.map (fun y -> y * 3) [1..10])
  ```

  umschreiben zu

  ```
  [1..10] |> List.map (fun y -> y * 3) |> List.filter (fun x -> x % 2 = 0)
  ```

  oder anders formatiert:

  ```
  [1..10]
  |> List.map (fun y -> y * 3)
  |> List.filter (fun x -> x % 2 = 0)
  ```

  Der Ausdruck nimmt die Liste der Zahlen 1 bis 10, multipliziert jede Zahl mit 3 und filtert die daraus resultierende Liste nach geraden Zahlen. Das Ergebnis ist also `[6; 12; 18; 24; 30]`.

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

a) Schreiben Sie eine Funktion `distance: Person -> Person -> Nat`, die den Abstand zwischen zwei Personen berechnet. Wir nehmen an, dass Santa mit seinem Schlitten durch eine Stadt fährt, die rasterförmig angelegt ist. Die Wegstrecke zwischen zwei Punkten ist also durch die 1-Norm[2] gegeben.

Beispiel: `distance lisa harry = 5N`

```
let distance (a: Person) (b: Person): Nat =
    let ax = a.location.x
    let ay = a.location.y
    let bx = b.location.x
    let by = b.location.y
    ((max ax bx) - (min ax bx)) + ((max ay by) - (min ay by))
```

b) Schreiben Sie eine Funktion `pathlength: List<Person> -> Nat`, die aus einer Liste von Personen die Weglänge berechnet, die sich ergibt, wenn deren Wohnorte nacheinander besucht werden. Beachten Sie, dass Santa am Ende wieder zum Ausgangspunkt zurückkehren muss.

Beispiel: `pathlength santasList = 16N`

```
let rec pathlength (persons: List<Person>): Nat =
    let rec pathlength' (path: List<Person>): Nat =
        match path with
        | [] | [_] -> 0N
        | x :: (y :: ys) -> (distance x y) + pathlength' (y :: ys)
    match persons with
    | [] -> 0N
    | x :: xs -> pathlength' (persons @ [x])
```

Die Konkatenation ist notwendig, da wir einen geschlossenen Pfad benötigen.

c) Schreiben Sie eine Funktion `prepend: 'a -> List<List<'a>> -> List<List<'a>>`, die ein Element `elem` und eine Liste von Listen `xss` nimmt und das Element `elem` jeder in `xss` enthaltenen Liste voranstellt.

Beispiel: `prepend 1N [[2N; 3N]; [4N; 5N]] = [[1N; 2N; 3N]; [1N; 4N; 5N]]`

```
// rekursiv
let rec prepend (elem: 'a) (xss: List<List<'a>>): List<List<'a>> =
    match xss with
    | [] -> []
    | zs :: zss -> (elem :: zs) :: (prepend elem zss)

// mit map
let prepend' (elem: 'a) (xss: List<List<'a>>): List<List<'a>> =
    List.map (fun xs -> elem :: xs) xss
```

[2] https://de.wikipedia.org/wiki/P-Norm#Summennorm

d) Schreiben Sie eine Funktion `insert: 'a -> List<'a> -> List<List<'a>>`, die ein Element `elem` sowie eine Liste `xs` nimmt und eine Liste aller Möglichkeiten zurückgibt, wie das Element `elem` in die Liste `xs` eingefügt werden kann.

Beispiel: `insert 1N [2N; 3N] = [[1N; 2N; 3N]; [2N; 1N; 3N]; [2N; 3N; 1N]]`

```
let rec insert (elem: 'a) (xs: List<'a>): List<List<'a>> =
    match xs with
    | [] -> [[elem]]
    | y :: ys -> (elem :: xs) :: (prepend y (insert elem ys))
```

e) Schreiben Sie eine Funktion `permute: List<'a> -> List<List<'a>>`, die eine Liste aller Permutationen der gegebenen Liste berechnet.

Beispiel:

```
permute [1N; 2N; 3N] = [  [1N; 2N; 3N]; [2N; 1N; 3N]; [2N; 3N; 1N]
                        ; [1N; 3N; 2N]; [3N; 1N; 2N]; [3N; 2N; 1N]]
```

```
// mit collect
let rec permute (ls: List<'a>): List<List<'a>> =
    match ls with
    | [] -> [[]]
    | x :: xs -> List.collect (fun a -> insert x a) (permute xs)

// rekursiv
let rec permute' (ls: List<'a>): List<List<'a>> =
    let rec ins (elem: 'a) (permutationsOfTail: List<List<'a>>): List<List<'a>> =
        match permutationsOfTail with
        | [] -> []
        | (p :: ps) -> (insert elem p) @ (ins elem ps)
    match ls with
    | [] -> [[]]
    | x :: xs -> ins x (permute' xs)
```

Wir permutieren zuerst den tail `xs` und fügen dann in jede Permutation `x` an alle möglichen Stellen ein.

f) Schreiben Sie eine Funktion `shortestPath: List<Person> -> List<Person> * Nat`, die den kürzesten Weg zwischen den übergebenen Personen und dessen Länge zurückgibt. Der Pfad der Lösung ist nicht eindeutig, z.B. ändert sich die Pfadlänge nicht, wenn dieser rückwärts durchlaufen wird.

Hinweis: Die Funktionen `List.minBy` und `List.map` könnten hilfreich sein.

Beispiel einer möglichen Lösung für `santasList`:

```
shortestPath santasList =
    ([  { name = "Alice"; location = { x = 2N; y = 5N} }
      ; { name = "Lisa";  location = { x = 1N; y = 2N} }
      ; { name = "Harry"; location = { x = 5N; y = 3N} }
      ; { name = "Bob";   location = { x = 4N; y = 4N} }]
     , 14N)
```

```
// Lösung mit map und minBy
let shortestPath (persons: List<Person>): (List<Person> * Nat) =
    permute persons
    |> List.map (fun x -> (x, pathlength x))
    |> List.minBy snd

// rekursive Lösung
let rec shortestPath' (persons: List<Person>): List<Person> * Nat =
    let rec sp (possiblePaths: List<List<Person>>) =
        match possiblePaths with
        | [] -> ([], 0N)
        | [path] -> (path, pathlength path)
        | path :: paths -> let curr = (path, pathlength path)
                           let rest = sp paths
                           if snd curr <= snd rest then curr
                           else rest
    sp (permute persons)
```

Wir berechnen zuerst alle möglichen Pfade mit `permute`. Nachdem wir jedem Pfad seine Länge zugeordnet haben, geben wir einen Pfad kürzester Länge zurück.

g) Kommentieren Sie Ihren Code so, dass Ihre Lösung einfach nachzuvollziehen ist. Außerdem sollte Ihr Code keine unnötig komplexen Ausdrücke enthalten, siehe auch Aufgabe 3 auf Übungsblatt 5. Dafür vergeben wir hier drei Punkte.

Aufgabe 3 Reguläre Ausdrücke (Einreichaufgabe, 9 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 552 bis 622 sowie am Skript Kapitel 6.1 und 6.2 orientieren.

Praxistipp: Das UNIX- bzw. Linuxprogramm `grep`[3] 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.

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

Wir betrachten den regulären Ausdruck `c(a|c)*(a|(bc)*)` über dem Alphabet $A = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$.

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.

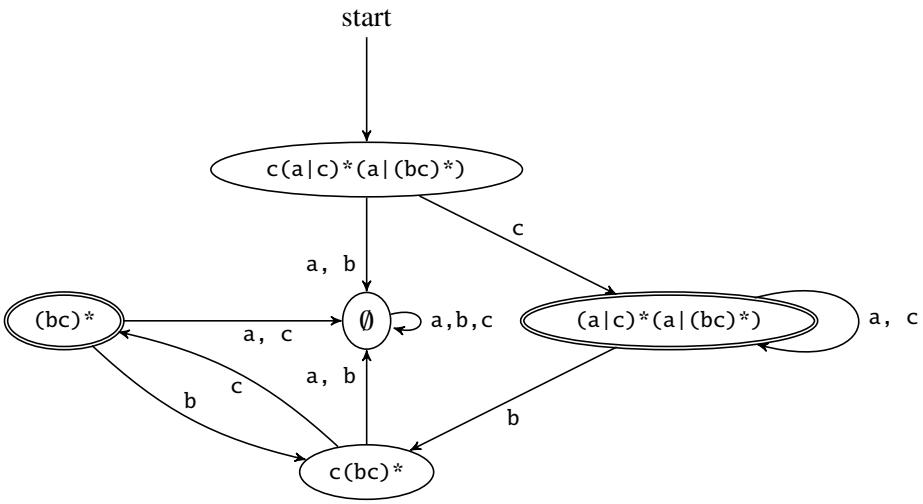
```
a \ c(a|c)*(a|(bc)*) = (a \ c) (a|c)*(a|(bc)*)
                     = ∅
b \ c(a|c)*(a|(bc)*) = (b \ c) (a|c)*(a|(bc)*)
                     = ∅
c \ c(a|c)*(a|(bc)*) = (c \ c) (a|c)*(a|(bc)*)
                     = (a|c)*(a|(bc)*)

a \ (a|c)*(a|(bc)*) = (a \ (a|c)*)(a|(bc)*) | a\(a|(bc)*)
                    = (a|c)*(a|(bc)*) | ϵ
                    = (a|c)*(a|(bc)*)
b \ (a|c)*(a|(bc)*) = (b \ (a|c)*)(a|(bc)*) | b\(a|(bc)*)
                    = c(bc)*
c \ (a|c)*(a|(bc)*) = (c \ (a|c)*)(a|(bc)*) | c\(a|(bc)*)
                    = (a|c)*(a|(bc)*)

a \ c(bc)* = (a \ c) (bc)* = ∅
b \ c(bc)* = (b \ c) (bc)* = ∅
c \ c(bc)* = (c \ c) (bc)* = (bc)*

a \ (bc)* = (a \ (bc)) (bc)* = ∅
b \ (bc)* = (b \ (bc)) (bc)* = c(bc)*
c \ (bc)* = (c \ (bc)) (bc)* = ∅

a \ ∅ = ∅
b \ ∅ = ∅
c \ ∅ = ∅
```

ϵ ist das neutrale Element der Konkatenation. $\emptyset$ ist das neutrale Element der Alternative.

[3] 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 600). Nutzen Sie für das Alphabet den Typ `type Alphabet = | A | B | C`.

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 = // C(A|C)*(A|(BC)*)
        match input with
        | [] -> false
        | A::rest -> accept3 rest // ∅
        | B::rest -> accept3 rest // ∅
        | C::rest -> accept1 rest // (A|C)*(A|(BC)*)
    and accept1 (input: List<Alphabet>): Bool = // (A|C)*(A|(BC)*)
        match input with
        | [] -> true
        | A::rest -> accept1 rest // (A|C)*(A|(BC)*)
        | B::rest -> accept2 rest // C(BC)*
        | C::rest -> accept1 rest // (A|C)*(A|(BC)*)
    and accept2 (input: List<Alphabet>): Bool = // C(BC)*
        match input with
        | [] -> false
        | A::rest -> accept3 rest // ∅
        | B::rest -> accept3 rest // ∅
        | C::rest -> accept4 rest // (BC)*
    and accept3 (input: List<Alphabet>): Bool = // ∅
        match input with
        | [] -> false
        | A::rest -> accept3 rest // ∅
        | B::rest -> accept3 rest // ∅
        | C::rest -> accept3 rest // ∅
    and accept4 (input: List<Alphabet>): Bool = // (BC)*
        match input with
        | [] -> true
        | A::rest -> accept3 rest // ∅
        | B::rest -> accept2 rest // C(BC)*
        | C::rest -> accept3 rest // ∅
    accept0 input
```