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

RPTU
Fachbereich Informatik
AG Programmiersprachen

Lösungshinweise/-vorschläge zum Übungsblatt 8: Grundlagen der Programmierung (WS 2023/24)

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.

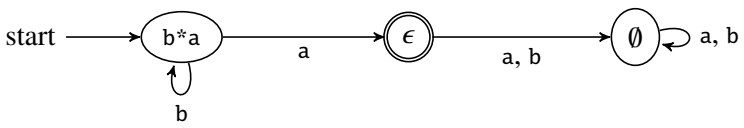
Wir betrachten den regulären Ausdruck `b*a` über dem Alphabet Alphabet $A = \{a, b\}$.

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 600).

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 = // 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 Datentypen (Einreichaufgabe, 5 Punkte)

Motivation: In dieser Aufgabe sollen Sie üben, selber geeignete Datentypen zu definieren. Sie können sich an den Vorlesungsfolien 260 bis 373 und 379 bis 386 sowie am Skript Kapitel 4 orientieren.

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

Hinweis: Zu dieser Aufgabe gibt es keine Tests!

a) Schreiben Sie einen Datentypen, der Mahlzeiten modelliert. Eine Mahlzeit besteht aus einem Namen, einem Preis (in Cent, also einer natürlichen Zahl) und aus einer unbestimmten Anzahl an Zutaten.

```
type Meal = {
    name: String;
    price: Nat;
    ingredients: List<String>;
}
```

b) Schreiben Sie einen Datentypen, der „magische Sequenzen“ modelliert. Eine magische Sequenz ist entweder ein einzelnes „A“ oder ein „B“ gefolgt von zwei magischen Sequenzen

```
type MagicSequence =
    | A
    | B of MagicSequence * MagicSequence
```

c) Schreiben Sie einen Datentypen, der parametrische Listen modelliert, die immer eine gerade Anzahl an Elementen haben.

```
type DoubleList<'a> =
    | Empty
    | Cons of 'a * 'a * DoubleList<'a>
```

d) Schreiben Sie einen Datentypen, der parametrische Bäume modelliert, deren Knoten Werte von einem Typ `'a` enthalten und deren Blätter Werte von einem Typ `'b` enthalten.

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

e) Schreiben Sie einen Datentypen, der parametrische Multiway-Bäume modelliert. Multiway-Bäume sind entweder Blätter, die einen Wert vom Typ `'a` enthalten, oder Knoten, die eine Liste aus Multiway-Bäumen enthalten.

```
type MultiTree<'a> =
    | Leaf of 'a
    | Node of List<MultiTree<'a>>
```

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 `ab(ab)*|ba(ba)*` über dem Alphabet $A = \{\mathtt{a}, \mathtt{b}\}$.

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.

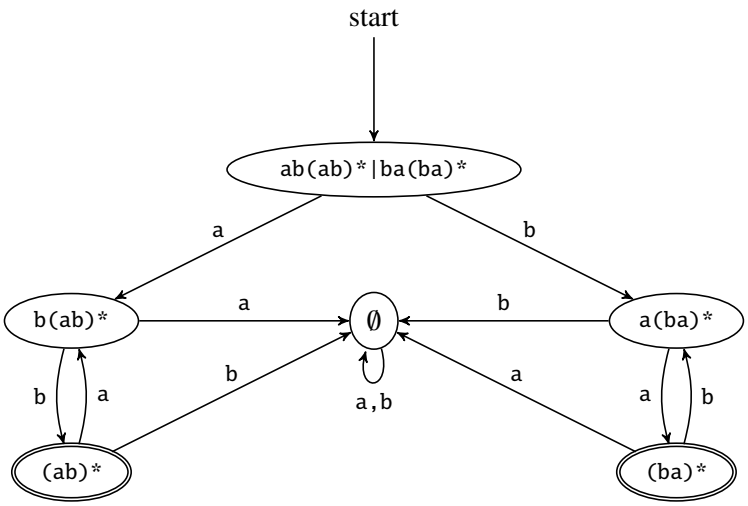
```
a \ ab(ab)*|ba(ba)* = (a \ ab(ab)*) | (a \ ba(ba)*)
                    = b(ab)*
b \ ab(ab)*|ba(ba)* = (b \ ab(ab)*) | (b \ ba(ba)*)
                    = a(ba)*

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

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

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

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

a \ ∅ = ∅
b \ ∅ = ∅
```

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

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

b) Zeichnen Sie den Aufrufgraphen für den Akzeptor (wie auf Vorlesungsfolie 600).

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