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 7: Konzepte der Programmierung (WS 2025/26)

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 $(a \mid b) \mid (a \mid b)^*$ ab. Geben Sie die gesamte Reduktionsfolge an.

$(a \mid b) \mid (a \mid b)^*$
$\rightarrow (a \mid b)^*$
$\rightarrow (a \mid b) \cdot (a \mid b)^*$
$\rightarrow a \cdot (a \mid b)^*$
$\rightarrow a \cdot (a \mid b) \cdot (a \mid b)^*$
$\rightarrow a \cdot b \cdot (a \mid b)^*$
$\rightarrow a \cdot b \cdot \epsilon$
$\rightarrow ab$

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.

   $(a \mid b \mid c) \cdot (a \mid b \mid c) \cdot (a \mid b \mid c)$

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

   $(b \mid c)^* \cdot a \cdot (b \mid c)^* \cdot a \cdot (b \mid c)^*$

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

   $((a \mid b \mid c) \cdot (a \mid b \mid c))^*$

Aufgabe 2 Endliche Abbildungen I (Einreichaufgabe, 10 Punkte)

Motivation: In dieser Aufgabe sollen Sie Ihr algorithmisches Denken üben sowie noch einmal mit komplexeren rekursiven Varianten zu programmieren. Sie können sich an den Vorlesungsfolien 499 bis 542 sowie am Skript Kapitel 5.3 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `MapSortedList.fs` *aus der Vorlage* `Aufgabe-7-2.zip`.

In dieser und der folgenden Aufgabe wollen wir endliche Abbildungen in F# repräsentieren. Unter einer endlichen Abbildung verstehen wir die Zuordnung von Elementen eines Definitionsbereichs („Schlüssel“ bzw. *keys*) zu Elementen einer Zielmenge („Werte“ bzw. *values*). Davon ausgehend definieren wir als abstrakten Datentyp (vgl. Folie 501) für endliche Abbildungen `type Map<'k, 'v>`, welcher Schlüsseln vom Typ `'k` Werte vom Typ `'v` zuordnet.

In dieser Aufgabe implementieren wir endliche Abbildungen mithilfe von *sortierten* Listen vom Typ

```
type MapSortedList<'k, 'v when 'k: comparison> = List<'k * 'v>
```

Sortiert wird immer nur nach dem Schlüssel, in aufsteigender Reihenfolge. Um eine Sortierung anhand der Schlüssel durchzuführen, müssen wir diese vergleichen können. Dazu wird `when 'k: comparison` gefordert.

Hinweis: Sie finden entsprechende Beispiele in den Vorlagen.

Hinweis: Sie dürfen die Standardbibliothek in Ihrer Lösung ***nicht*** *verwenden.*

a) Definieren Sie den Wert `empty<'k, 'v>` vom Typ `MapSortedList<'k, 'v>`, welcher eine leere Abbildung darstellt.

```
let empty<'k, 'v when 'k: comparison> : MapSortedList<'k, 'v> =
    []
```

b) Schreiben Sie eine Funktion `lookup<'k, 'v>: 'k -> MapSortedList<'k, 'v> -> Option<'v>`, welche einen Schlüssel und eine Abbildung entgegennimmt und den Wert zurückgibt, auf den der Schlüssel in der Abbildung abgebildet wird. Falls der Schlüssel nicht in der Abbildung enthalten ist, soll `None` zurückgegeben werden.

```
let rec lookup<'k, 'v when 'k: comparison> (key: 'k) (m: MapSortedList<'k, 'v>):
    Option<'v> =
    match m with
    | [] -> None
    | (k, v)::rest ->
        if k = key then Some v
        elif key < k then None
        else lookup key rest
```

c) Schreiben Sie eine Funktion `set<'k, 'v>: 'k -> 'v -> MapSortedList<'k, 'v> -> MapSortedList<'k, 'v>`, welche einen Schlüssel, einen Wert und eine Abbildung entgegennimmt und eine Abbildung zurückgibt, welche den Schlüssel auf den Wert abbildet. Alle anderen Schlüssel sollen auf die gleichen Werte wie in der Eingabeabbildung abgebildet werden. Falls der Schlüssel bereits in der Abbildung enthalten ist, soll der Wert überschrieben werden.

```
let rec set<'k, 'v when 'k: comparison> (key: 'k) (value: 'v)
    (m: MapSortedList<'k, 'v>): MapSortedList<'k, 'v> =
    match m with
    | [] -> [(key, value)]
    | (k, v)::rest ->
        if k = key then (key, value)::rest
        elif key < k then (key, value)::(k, v)::rest
        else (k,v)::(set key value rest)
```

d) Schreiben Sie eine Funktion `comma<'k, 'v>: MapSortedList<'k, 'v> -> MapSortedList<'k, 'v> -> MapSortedList<'k, 'v>`, welche den Kommaoperator implementiert. Die Funktion nimmt also zwei Abbildungen und gibt eine Abbildung zurück, welche alle Schlüssel und Werte aus beiden Abbildungen enthält. Falls ein Schlüssel in beiden Abbildungen enthalten ist, soll der Wert aus der zweiten Abbildung verwendet werden.

```
let rec comma<'k, 'v when 'k: comparison> (m1: MapSortedList<'k, 'v>)
                  (m2: MapSortedList<'k, 'v>): MapSortedList<'k, 'v> =
    match m2 with
    | [] -> m1
    | (k, v)::rest -> set k v (comma m1 rest)
```

e) Schreiben Sie eine Funktion `delete<'k, 'v>: 'k -> MapSortedList<'k, 'v> -> MapSortedList<'k, 'v>`, welche einen Schlüssel und eine Abbildung entgegennimmt und eine Abbildung zurückgibt, welche alle Schlüssel und Werte aus der Eingabeabbildung enthält, außer dem Schlüssel, der in der Eingabeabbildung auf den Wert abgebildet wird. Falls der Schlüssel nicht in der Abbildung enthalten ist, soll die Eingabeabbildung zurückgegeben werden.

```
let rec delete<'k, 'v when 'k: comparison> (key: 'k) (m: MapSortedList<'k, 'v>):
    MapSortedList<'k, 'v> =
    match m with
    | [] -> []
    | (k, v)::rest ->
        if k = key then rest
        elif key < k then (k, v)::rest
        else (k, v)::(delete key rest)
```

Aufgabe 3 Endliche Abbildungen II (Einreichaufgabe, 11 Punkte)

Motivation: In dieser Aufgabe sollen Sie den Umgang mit Funktionen höherer Ordnung sowie mit endlichen Abbildungen üben.

Dies ist die Fortsetzung von Aufgabe 2.

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

In dieser Aufgabe implementieren wir endliche Abbildungen mithilfe von partiellen Funktionen, also Funktionen vom Typ

```
type MapPartialFunction<'k, 'v> = 'k -> Option<'v>
```

Hinweis: Sie dürfen die Standardbibliothek in Ihrer Lösung ***nicht*** *verwenden.*

a) Definieren Sie den Wert `empty<'k, 'v>` vom Typ `MapPartialFunction<'k, 'v>`, welcher eine leere Abbildung darstellt.

```
let empty<'k,'v> : MapPartialFunction<'k, 'v> =
    fun (_: 'k) -> None
```

b) Schreiben Sie eine Funktion `lookup<'k, 'v>: 'k -> MapPartialFunction<'k, 'v> -> Option<'v>`, welche einen Schlüssel und eine Abbildung entgegennimmt und den Wert zurückgibt, auf den der Schlüssel in der Abbildung abgebildet wird. Falls der Schlüssel nicht in der Abbildung enthalten ist, soll `None` zurückgegeben werden.

```
let lookup<'k, 'v> (key: 'k) (map: MapPartialFunction<'k, 'v>): Option<'v> =
    map key
```

c) Schreiben Sie eine Funktion `set<'k, 'v>: 'k -> 'v -> MapPartialFunction<'k, 'v> -> MapPartialFunction <'k, 'v>`, welche einen Schlüssel, einen Wert und eine Abbildung entgegennimmt und eine Abbildung zurückgibt, welche den Schlüssel auf den Wert abbildet. Alle anderen Schlüssel sollen auf die gleichen Werte wie in der Eingabeabbildung abgebildet werden. Falls der Schlüssel bereits in der Abbildung enthalten ist, soll der Wert überschrieben werden. (Diese Funktion implementiert im Prinzip den Kommaoperator für einelementige Abbildungen im zweiten Argument des Kommaoperators.)

```
let set<'k,'v when 'k: equality> (key: 'k) (value: 'v)
      (map: MapPartialFunction<'k, 'v>): MapPartialFunction<'k, 'v> =
    fun (input_key: 'k) -> if input_key = key
                           then Some value
                           else map input_key
```

d) Schreiben Sie eine Funktion `comma<'k, 'v>: MapPartialFunction<'k, 'v> -> MapPartialFunction<'k, 'v> -> MapPartialFunction<'k, 'v>`, welche den Kommaoperator implementiert.

```
let comma<'k, 'v> (map1: MapPartialFunction<'k, 'v>)
                  (map2: MapPartialFunction<'k, 'v>): MapPartialFunction<'k, 'v>
    =
    fun (input_key: 'k) ->
        match map2 input_key with
        | None -> map1 input_key
        | Some x -> Some x
```

e) Schreiben Sie eine Funktion `delete<'k, 'v>: 'k -> MapPartialFunction<'k, 'v> -> MapPartialFunction<'k, 'v>`, welche einen Schlüssel und eine Abbildung entgegennimmt und eine Abbildung zurückgibt, welche alle Schlüssel und Werte aus der Eingabeabbildung enthält, außer dem Schlüssel, der in der Eingabeabbildung auf den Wert abgebildet wird. Falls der Schlüssel nicht in der Abbildung enthalten ist, soll die Eingabeabbildung zurückgegeben werden.

```
let rec delete<'k, 'v when 'k: equality> (key: 'k)
          (map: MapPartialFunction<'k, 'v>): MapPartialFunction<'k, 'v> =
    fun (input_key: 'k) ->
        if input_key = key then None else map input_key
```