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 11: Konzepte der Programmierung (WS 2025/26)

Aufgabe 1 Kontrollstrukturen und Ausnahmen (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie dasselbe algorithmische Problem aus verschiedenen Blickrichtungen betrachten, um sich mit den Unterschieden der funktionalen und der imperativen Programmierung vertraut zu machen. Sie können sich an den Vorlesungsfolien 836 bis 899 sowie an den Kapiteln 7.3 und 7.4 im Skript orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Find.fs` *aus der Vorlage* `Aufgabe-11-1.zip`.

In den folgenden Teilaufgaben sollen Sie Funktionen schreiben, die das letzte Element einer Liste zurückgeben, für welches ein vorgegebenes Prädikat zu **`true`** auswertet. In zwei der Teilaufgaben verwenden wir dabei die folgende Ausnahme:

```
exception NotFound
```

Hinweis: Da anhand derselben Problemstellung verschiedene Konzepte eingeübt werden sollen, ist es naheliegend, dass es ***nicht erlaubt ist, die Funktionen der einzelnen Teilaufgaben gegenseitig aufzurufen****. Verwenden Sie in Ihrer Lösung außerdem* ***keine Bibliotheksfunktionen****. Davon ausgenommen sind das* `Length` *Attribut von Listen, bzw.* `List.length`*, sofern Sie diese verwenden möchten.*

a) Schreiben Sie eine Funktion `tryFindLast<'a>: ('a -> Bool) -> List<'a> -> Option<'a>`, welche ein Prädikat `pred` sowie eine Liste `xs` nimmt und das letzte Element der Liste zurückgibt, für das `pred` zu **`true`** auswertet. Wenn es in der gesamten Liste kein solches Element gibt, soll `None` zurückgegeben werden. Schreiben Sie eine **rekursive Funktion**, verwenden Sie **keine Kontrollstrukturen (Schleifen) oder Ausnahmen**.

```
let rec tryFindLast<'a> (pred: 'a -> Bool) (xs: List<'a>): Option<'a> =
    match xs with
    | [] -> None
    | y::ys ->
        match tryFindLast pred ys with
        | None -> if pred y then Some y else None
        | Some z -> Some z
```

Im Fall, dass die übergebene Liste leer ist, geben wir `None` zurück. Falls nicht, rufen wir die Funktion rekursiv mit der Restliste auf und matchen auf das Ergebnis. Gibt es in der Restliste kein letztes Element, welches das Prädikat erfüllt, prüfen wir ob das Kopfelement `y` das Prädikat erfüllt. Falls ja, können wir es mit `Some y` zurückgeben (durch den rekursiven Aufruf wissen wir ja, dass es kein Element weiter hinten in der Liste geben kann, welches das Prädikat erfüllen könnte). Erfüllt auch `y` das Prädikat nicht, geben wir `None` zurück. Sofern im rekursiven Aufruf in der Restliste ein Element `z` gefunden wird, für welches das Prädikat zu **`true`** auswertet, geben wir dieses zurück (auch wenn `y` das Prädikat erfüllt, liegt `z` weiter hinten in der Liste als `y`).

b) Schreiben Sie eine Funktion `findLast<'a>: ('a -> Bool) -> List<'a> -> 'a`, welche ein Prädikat `pred` sowie eine Liste `xs` nimmt und das letzte Element der Liste zurückgibt, für das `pred` zu **`true`** auswertet. Wenn es in der gesamten Liste kein solches Element gibt, soll die **Ausnahme** `NotFound` geworfen werden. Schreiben Sie eine **rekursive Funktion**, verwenden Sie **keine Kontrollstrukturen**.

```
let rec findLast<'a> (pred: 'a -> Bool) (xs: List<'a>): 'a =
    match xs with
    | [] -> raise NotFound
    | y::ys ->
        try findLast pred ys with
        | NotFound -> if pred y then y else raise NotFound
```

Wir gehen genauso vor wie in der ersten Teilaufgabe. Falls die übergebene Liste leer ist, können wir direkt die `NotFound` Ausnahme werfen. Den rekursiven Aufruf matchen wir jedoch nicht, sondern packen ihn in einen **`try`** Block ein. Ist der rekursive Aufruf erfolgreich, wird dessen Ergebnis als Resultat zurückgegeben. Schlägt er fehl, so fangen wir die `NotFound` Ausnahme und prüfen wieder, ob `y` das Prädikat erfüllt. Falls ja, geben wir `y` zurück. Ansonsten werfen wir die Ausnahme weiter.

c) Schreiben Sie eine Funktion `tryFindLast2<'a>: ('a -> Bool) -> List<'a> -> Option<'a>`, welche ein Prädikat `pred` sowie eine Liste `xs` nimmt und das letzte Element der Liste zurückgibt, für das `pred` zu **`true`** auswertet. Wenn es in der gesamten Liste kein solches Element gibt, soll `None` zurückgegeben werden. Schreiben Sie die Funktion imperativ mit Hilfe von **Kontrollstrukturen**, verwenden Sie **keine rekursiven Funktionen oder Ausnahmen**.

```
let tryFindLast2<'a> (pred: 'a -> Bool) (xs: List<'a>): Option<'a> =
    let mutable last: Option<'a> = None
    for x in xs do
        if pred x then last <- Some x
    last
```

Zunächst definieren wir eine veränderliche Variable `last`, die wir verwenden möchten, um das Ergebnis zu speichern. Wir initialisieren sie mit `None`, da wir zu Beginn noch nicht wissen, ob wir überhaupt ein Element in der Liste finden werden, welches das Prädikat erfüllt. Mit einer **`for`** Schleife iterieren wir über die Listenelemente und prüfen, ob das aktuelle Element `x` das Prädikat erfüllt. Falls ja, sichern wir das Element in der Variablen `last` (ein ggf. vorher darin gespeichertes Element überschreiben wir), ansonsten tun wir nichts. Nachdem wir mit der Schleife die gesamte Liste durchlaufen haben, steht in der Variablen `last` das letzte Element der Liste, welches das Prädikat erfüllt (oder `None`). Als Ergebnis geben wir entsprechend `last` zurück.

d) Schreiben Sie eine Funktion `findLast2<'a>: ('a -> Bool) -> List<'a> -> 'a`, welche ein Prädikat `pred` sowie eine Liste `xs` nimmt und das letzte Element der Liste zurückgibt, für das `pred` zu **`true`** auswertet. Wenn es in der gesamten Liste kein solches Element gibt, soll die **Ausnahme** `NotFound` geworfen werden. Schreiben Sie die Funktion imperativ mit Hilfe von **Kontrollstrukturen**, verwenden Sie **keine rekursiven Funktionen**.

```
let findLast2<'a> (pred: 'a -> Bool) (xs: List<'a>): 'a =
    let mutable last: Option<'a> = None
    for x in xs do
        if pred x then last <- Some x
    match last with
    | None -> raise NotFound
    | Some x -> x
```

Wir übernehmen den Großteil der Lösung aus der vorherigen Teilaufgabe. Anstelle `last` zurückzugeben, prüfen wir, ob ein Element gefunden wurde. Falls nicht, werfen wir eine `NotFound` Ausnahme. Ansonsten geben wir das gefundene Element `x` zurück.

Aufgabe 2 Arrays (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie den Umgang mit Arrays einüben. Sie können sich an den Vorlesungsfolien 404 bis 427 und 836 bis 852 sowie an den Kapiteln 4.4 und 7.3 im Skript orientieren.

Schreiben Sie Ihre Lösungen in die Datei `ArrayMap.fs` *aus der Vorlage* `Aufgabe-11-2.zip`.

a) Schreiben Sie eine Funktion `map<'a, 'b>: ('a -> 'b) -> Array<'a> -> Array<'b>`, welche eine Funktion `f` sowie ein Array `ar` nimmt und ein *neues* Array zurückgibt, welches die Anwendung von `f` auf jedes Element von `ar` enthält.

```
let map<'a,'b> (f: 'a -> 'b) (ar: Array<'a>) : Array<'b> =
    [| for x in ar -> f x |]
```

b) Schreiben Sie eine Funktion `inplaceMap<'a>: ('a -> 'a) -> Array<'a> -> Unit`, welche eine Funktion `f` sowie ein Array `ar` nimmt und das Array `ar` in-place verändert, sodass jedes Element von `ar` durch die Anwendung von `f` auf dieses Element ersetzt wird. Warum kann `f` nicht den Typ `'a -> 'b` haben?

```
let inplaceMap<'a> (f: 'a -> 'a) (ar: Array<'a>) : Unit =
    for i in 0 .. ar.Length - 1 do
        ar.[i] <- f ar.[i]
```

`f` kann nicht den Typ `'a -> 'b` haben, da wir in-place arbeiten und das Array `ar` nicht verändern können, wenn `f` Elemente des Arrays auf Elemente eines anderen Typs abbildet. Insbesondere wäre der Typ von `ar` während der Ausführung von `inplaceMap` nicht konsistent.

Aufgabe 3 Ausnahmen (Einreichaufgabe, 6 Punkte)

Motivation: In dieser Aufgabe sollen Sie Ausnahmen einüben. Sie können sich an den Vorlesungsfolien 853 bis 899 sowie am Skript Kapitel 7.4 orientieren.

Unter Berücksichtigung dieser Typ- und Ausnahmedefinitionen

```
type A = | A1 | A2 | A3 of String | A4 of Bool

exception E
exception E1 of Nat
exception E2 of A
```

betrachten wir den folgenden Ausdruck. Dabei ist `f` eine Funktion vom Typ `Unit -> A`.

```
try
    match f() with
    | A1   -> 22N + raise (E2 A2)
    | A2   -> raise E
    | A3 x -> raise (E1 4711N)
    | A4 x when x -> 97N
    | A4 x -> raise (E2 A1)
with
| E    -> 4711N
| E1 n -> if n = 4711N then 50N else raise (E1 815N)
| E2 s -> match s with
           | A1 -> 1N
           | A2 -> 2N
           | A3 x -> 3N
           | A4 x -> raise (E2 (A4 (not x)))
```

Bestimmen Sie für die folgenden Implementierungen der Funktion `f` jeweils, zu welchem Wert obiger Ausdruck auswertet. Kennzeichnen Sie geworfene Ausnahmen dabei, wie in der Vorlesung eingeführt, mit einem Kästchen, die durch `raise (E1 4711N)` geworfene Ausnahme also durch `E1 4711N`.

a) `let f() = A1`

`2N`

b) `let f() = A4 false`

`1N`

c) `let f() = raise (E2 (A4 true))`

`E2 (A4 false)`

Aufgabe 4 Arrays und Zustand (Einreichaufgabe, 8 Punkte)

Motivation: In dieser Aufgabe sollen Sie Arrays und Kontrollstrukturen (Schleifen) einüben. Sie können sich an den Vorlesungsfolien 404 bis 427 und 836 bis 852 sowie an den Kapiteln 4.4 und 7.3 im Skript orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Arrays.fs` *aus der Vorlage* `Aufgabe-11-4.zip`.

In den folgenden Teilaufgaben betrachten wir Funktionen, die auf Arrays arbeiten. Es steht Ihnen, frei die Funktionen rekursiv oder imperativ zu implementieren.

Hinweis: Beachten Sie, dass Arrays mit nichtnegativen Ganzzahlen vom Typ `Int` *indiziert werden. Der Typ der natürlichen Zahlen* `Nat` *wird als Index leider nicht unterstützt.*

Hinweis: Es gibt mehrere Möglichkeiten ein Array mit Hilfe von Schleifen zu durchlaufen. Einerseits kann mit **`for`** *oder* **`while`** *und einer Zählvariable der Zugriff auf die Arrayelemente direkt über deren „Hausnummer" erfolgen. Andererseits kann für ein Array* `ar` *mit* **`for x in`** `ar` *auch direkt über die Arrayelemente selbst iteriert werden. Beispiele dazu finden Sie im Skript auf den Seiten 408 und 409.*

Hinweis: Sie können Funktionen aus vorherigen Teilaufgaben verwenden, wenn Sie möchten. Verwenden Sie in Ihrer Lösung ***keine Bibliotheksfunktionen****. Davon ausgenommen sind das* `Length`*-Attribut von Listen und Arrays bzw. die Funktionen* `List.length` *und* `Array.length`*. Konvertieren Sie in Ihrer Lösung nicht zwischen Arrays und Listen hin und her, außer wenn explizit gefordert.*

a) Schreiben Sie eine Funktion `swap<'a>: Array<'a> -> Int * Int -> Unit`, die ein Array sowie zwei Indizes nimmt und die Elemente an den Positionen der Indizes vertauscht.

```
let swap<'a> (ar: Array<'a>) (i: Int, j: Int): Unit =
    let tmp = ar.[i]
    ar.[i] <- ar.[j]
    ar.[j] <- tmp
```

S. Folie 851.

b) Schreiben Sie eine Funktion `insertionsort<'a when 'a: comparison>: Array<'a> -> Unit`, die ein Array nimmt und dieses in-place sortiert (also ohne dabei ein neues Array zu konstruieren). Implementieren Sie den Insertionsort[1] Algorithmus.

```
let insertionsort<'a when 'a: comparison> (ar: Array<'a>): Unit =
    let n = ar.Length
    for i in 1..(n-1) do
        let mutable j = i
        while j > 0 && ar.[j-1] > ar.[j] do
            swap ar (j-1, j)
            j <- j-1
```

Unsere Insertionsort-Implementierung sortiert das Array von links nach rechts. Die äußere Schleife gibt vor, bis wohin das Array bereits sortiert ist. Die innere Schleife nimmt das nächste Element und schiebt es durch Vertauschungen so weit nach links, bis es an der richtigen Stelle steht.

Insertionsort ist ein relativ intuitives, aber ineffizientes Sortierverfahren (Worstcase-Laufzeit quadratisch in der Länge des Arrays).

[1] s. z.B. https://en.wikipedia.org/wiki/Insertion_sort#Algorithm

c) Schreiben Sie eine Funktion `rotate<'a>: Array<'a> -> Unit`, welche das übergebene Array in-place rotiert, also das erste Element an die zweite Stelle schiebt, das zweite an die dritte, usw. Das letzte Element wird an die erste Stelle geschoben.

```
let rotate<'a> (ar: Array<'a>) : Unit =
    let n = ar.Length
    if n > 0 then
        let nth = ar.[n-1]
        for i in [n-1..(-1)..1] do
            ar.[i] <- ar.[i-1]
        ar.[0] <- nth

//alternativ:
let rotate2<'a> (ar: Array<'a>) : Unit =
    let n = ar.Length
    for i in [n-1..(-1)..1] do
        swap ar (i-1, i)
```

Wir merken uns das letzte Element und schieben alle anderen Elemente um eine Stelle nach rechts. Dann schreiben wir das letzte Element an die erste Stelle.

d) Schreiben Sie eine Funktion `same<'a when 'a: equality>: List<'a> -> Array'<a> -> Bool`, die eine Liste sowie ein Array nimmt und zurückgibt, ob die Elemente an korrespondierenden Stellen in der Liste und im Array gleich sind. Wenn die Liste und das Array unterschiedlich lang sind, ist das Ergebnis der `same` Funktion **`false`**. Konvertieren Sie dazu das Array mithilfe einer Listenbeschreibung in eine Liste.

```
let same<'a when 'a: equality> (xs: List<'a>) (ar: Array<'a>): Bool =
    xs = [ for x in ar -> x ]
```