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 11: Grundlagen der Programmierung (WS 2024/25)

Aufgabe 0 Abschluss der Studie zum Rekursions-Tutor

Bitte nehmen Sie an dieser Umfrage zum Rekursions-Tutor teil. Ihr Feedback hilft uns, die Studie abzuschließen und das Tool zu verbessern.

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 837 bis 900 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 wie 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 405 bis 428 und 837 bis 853 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, 8 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 AB = | A of Nat | B

exception E of Nat
exception F of Bool
```

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

```
try
    match f() with
    | A x -> if x < 10N then raise (E x) else x
    | B   -> raise (F true)
with
| E x -> if x = 0N then raise (F false) else x
| F x -> if x then 4711N else raise (E 7N)
```

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 (E 123)` geworfene Ausnahme also durch `E 123`.

1. `let f() = A 0`

   `F false`

2. `let f() = B`

   `4711`

3. `let f() = raise (E 99)`

   `99`

4. `let f() = raise (F false)`

   `E 7`

Aufgabe 4 Ringpuffer (Einreichaufgabe, 12 Punkte)

Motivation: In dieser Aufgabe sollen Sie das Programmieren mit Arrays einüben. Sie können sich an den Vorlesungsfolien 404 bis 427 sowie an Kapitel 4.4 im Skript orientieren.

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

Ein Ringpuffer ist ein Speicher, in den kontinuierlich Daten gespeichert und wieder herausgenommen werden können. Er funktioniert wie eine Warteschlange: Es wird immer dasjenige Element herausgenommen, das sich schon am längsten im Ringpuffer befindet. Das besondere ist, dass der Speicher eine feste Kapazität hat, die die maximale Anzahl an Elementen in der Warteschlange vorgibt.

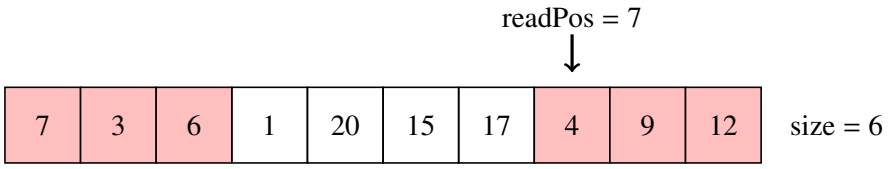
Arrays sind Speicher fixer Größe. Wir verwenden folgenden Typ für unseren Ringpuffer:

```
type RingBuffer<'a> =
    { buffer: Array<'a>
      size: Ref<Int>
      readPos: Ref<Int> }
```

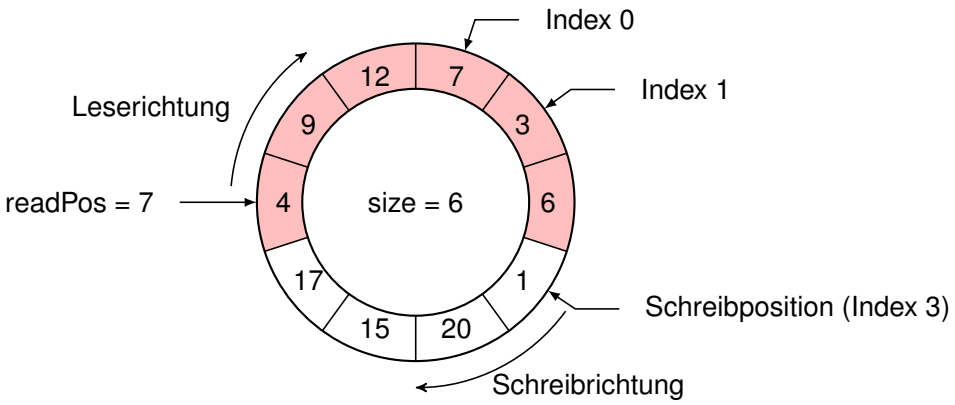
Das Array `buffer` enthält die Elemente im Ringpuffer. Es hat eine vorab festgelegte Länge (`buffer.Length`), die die Kapazität des Ringpuffers darstellt. Die Zahl der aktuell tatsächlich im Ringpuffer enthaltenen Elemente kann sich davon unterscheiden und ist in `size` gespeichert. Der Wert, der an einer bestimmten Position im Array gespeichert ist, ist nur dann im Ringpuffer enthalten, wenn der Wert zuvor hinzugefügt aber noch nicht gelesen wurde. In unserer Abbildung unten sind Positionen, die im Ringpuffer tatsächlich enthalten sind, rot eingefärbt. Nicht enthaltene Positionen sind weiß. Die Zahl `readPos` bezeichnet den Index innerhalb `buffer`, an dessen Stelle das nächste zu lesende Element steht.

Gelesen werden können die `size` Elemente beginnend ab `readPos`, wobei auf Index `buffer.Length - 1` der Index 0 folgt. Als nächstes geschrieben wird der Index `size` Positionen nach `readPos`.

Die folgende Abbildung veranschaulicht einen `RingBuffer<Int>` mit Kapazität 10 und Größe 6.



Als Ring dargestellt wird der Übergang zwischen Anfang und Ende des Arrays deutlicher:



Der Ringpuffer enthält sechs Elemente (rote Felder, gelesen wird in der Reihenfolge 4, 9, 12, 7, 3, 6) und es können noch vier weitere Elemente gespeichert werden (dabei werden die weißen Felder 1, 20, 15 und 17 überschrieben).

Hinweis: In F# haben Array-Indizes und das `Length` Attribut den Typ `Int` (Typ der ganzen Zahlen), obwohl sie natürlich nie negativ sind. Damit Sie nicht zwischen `Int` und `Nat` konvertieren müssen, verwenden wir auch für `size` und `writePos` den Typ `Int`.

Beispiele:

```
let ex1: RingBuffer<Int> = { buffer = [|0; 0; 0|]; size = ref 0; readPos = ref 0 }
let ex2: RingBuffer<String> = { buffer = [|"La"; "Le"; "Lu"|]
            size = ref 1
            readPos = ref 0 }
let ex3: RingBuffer<Int> = { buffer = [|7; 3; 6; 1; 20; 15; 17; 4; 9; 12|]
            size = ref 6
            readPos = ref 7 }
```

a) Schreiben Sie die Funktion `create: Int -> RingBuffer<'a>`, die einen Ringpuffer mit Gesamtkapazität `capacity` initialisiert. Die anfängliche Schreibposition und Größe des Puffers sollen `0` sein.

Beispiel: Das Ergebnis von `create<Int> 3` ist `ex1`.

Hinweis: Verwenden Sie die Funktion `Array.zeroCreate`[1].

```
let create<'a> (capacity: Int): RingBuffer<'a> =
    { buffer = Array.zeroCreate<'a> capacity; size=ref 0; readPos=ref 0 }
```

b) Schreiben Sie eine Funktion `get: RingBuffer<'a> -> 'a`, welche das jeweils nächste im Puffer enthaltene Element zurückgibt. Dabei soll `size` um eins verringert und `readPos` (unter Beachtung des Umbruchs am Ende des Arrays) um eins erhöht werden.

Falls keine zu lesenden Elemente (rote Felder) mehr vorhanden sind, soll der Ringpuffer nicht verändert und eine `BufferEmpty` Ausnahme geworfen werden.

```
let get<'a> (r: RingBuffer<'a>): 'a =
    if !r.size > 0 then
        let capacity = r.buffer.Length
        let readPos = !r.readPos
        r.readPos := (readPos + 1) % capacity
        r.size := !r.size - 1
        r.buffer.[readPos]
    else
        raise BufferEmpty
```

c) Schreiben Sie eine Funktion `put: RingBuffer<'a> -> 'a -> Unit`, welche ein Element `elem` auf den Ringpuffer schreibt. Falls der Ringpuffer voll ist (keine weißen Felder), soll eine `BufferFull` Ausnahme geworfen und der Ringpuffer nicht verändert werden.

Ansonsten soll der Array-Eintrag `size` Positionen nach `readPos` (Umbruch beachten!) überschrieben und`size` um eins erhöht werden.

```
let put<'a> (r: RingBuffer<'a>) (elem: 'a): Unit =
    if !r.size < r.buffer.Length then
        let capacity = r.buffer.Length
        let writePos = (!r.readPos + !r.size) % capacity
        r.buffer.[writePos] <- elem
        r.size := !r.size + 1
    else
        raise BufferFull
```

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