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 10: Konzepte der Programmierung (WS 2025/26)

Zustand Bisher haben wir nur mit Bezeichnern gearbeitet, die an unveränderliche Werte gebunden sind:

```
let x = 1N // Definiert einen Bezeichner x, der an den Wert 1N gebunden ist.
let f () = print x // f schreibt den Wert x auf die Konsole.
f() // Schreibt 1N auf die Konsole.
let x = 2N // Definiert einen neuen Bezeichner mit dem gleichen Namen.
f() // Schreibt 1N auf die Konsole, da f den alten Bezeichner benutzt.
```

Nun haben wir in der Vorlesung Speicherzellen kennengelernt. Wir verändern das Programm etwas, sodass die zweite Ausführung von `f` den neuen Wert auf die Konsole schreibt:

```
let x = ref 1N // Allokiert eine Speicherzelle, die den Wert 1N enthält, und definiert
  // einen Bezeichner x, der an die Adresse dieser Speicherzelle gebunden ist.
let f () = print (!x) // f schreibt den Inhalt der Speicherzelle an x auf die Konsole.
f() // Schreibt 1N auf die Konsole.
x := 2N // Speichert einen neuen Wert in die Speicherzelle an Adresse x.
f() // Schreibt 2N auf die Konsole.
```

Eine zweite Variante sind veränderliche Bezeichner. Wir können das Programm auch so schreiben:

```
let mutable x = 1N
let f () = print x
f() // Schreibt 1N auf die Konsole.
x <- 2N
f() // Schreibt 2N auf die Konsole.
```

Aufgabe 1 Semantik mit Zustand (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie die dynamische Semantik mit einem Speicher nachvollziehen. Sie können sich an den Vorlesungsfolien 753 bis 825 sowie am Skript Kapitel 7.2 orientieren.

Die Deklarationen

```
let x = ref 1N
let y = ref 2N
let i = ref y
```

werten zu der folgenden Umgebung aus:

$$\delta = \{x \mapsto a_0,\ y \mapsto a_1,\ i \mapsto a_2\}$$

Hierbei bezeichnen a_0, a_1 und a_2 Adressen. Der dazugehörige Speicher ist:

$$\sigma = \{a_0 \mapsto 1N,\ a_1 \mapsto 2N,\ a_2 \mapsto a_1\}$$

Der folgende Ausdruck soll in der Umgebung δ und dem Speicher σ ausgewertet werden. Geben Sie für den Ausdruck den Speicherzustand nach jeder Einzelanweisung an.

```
x := !y + 3N      // σ1
y := !(!i) + !x   // σ2
i := x            // σ3
!i := !y + 1N     // σ4
```

Bezeichner	x	y	i
Adresse	a_0	a_1	a_2
σ	1N	2N	a_1
σ_1	5N	2N	a_1
σ_2	5N	7N	a_1
σ_3	5N	7N	a_0
σ_4	8N	7N	a_0

Aufgabe 2 Handzähler (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie sich mit den Grundlagen von Speicherzellen vertraut machen. Sie können sich an den Vorlesungsfolien 753 bis 825 sowie am Skript Kapitel 7.2 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Counters.fs` *aus der Vorlage* `Aufgabe-10-2.zip`.

Zur Einlasskontrolle in einem Supermarkt muss gezählt werden, wie viele Kund*innen sich darin befinden. Der Supermarktbetreiber beauftragt Lisa Lista und Harry Hacker damit, Handzähler zu entwickeln. Diese kleinen praktischen Geräte haben zwei Taster und eine Anzeige für den aktuellen Zählerstand. Der Reset-Taster setzt den Zähler auf Null zurück. Der Inkrement-Taster erhöht den Zählerstand um eins.

a) Harrys Vorschlag ist nun, den Zählerstand in einer **`mutable`** Variable zu speichern und mit drei Funktionen darauf zuzugreifen:

```
reset: Unit -> Unit      // stellt den Zähler auf Null
increment: Unit -> Unit  // erhöht den Zähler um eins
get: Unit -> Nat         // gibt den aktuellen Zählerstand zurück
```

Implementieren Sie diese drei Funktionen und verwenden Sie dazu **`let mutable`**.

```
let mutable counter = 0N

let reset(): Unit =
    counter <- 0N

let increment(): Unit =
    counter <- counter + 1N

let get(): Nat =
    counter
```

b) Lisa merkt an, dass man so aber immer nur einen Zähler gleichzeitig benutzen kann. Der Supermarkt hat jedoch zwei Türen und es wäre praktisch, wenn man an jeder der beiden Türen einen eigenen Zähler verwenden kann. Sie schlägt daher folgende Modellierung vor:

```
type Counter = Ref<Nat>
create: Unit -> Counter      // gibt einen neuen Zähler zurück
reset2: Counter -> Unit      // stellt den gegebenen Zähler auf Null
increment2: Counter -> Unit  // erhöht den gegebenen Zähler um eins
get2: Counter -> Nat         // gibt den aktuellen Stand des gegebenen Zählers zurück
```

Implementieren Sie diese Funktionen.

```
type Counter = Ref<Nat>

let create(): Counter =
    ref 0N

let reset2(c: Counter): Unit =
    c := 0N

let increment2(c: Counter): Unit =
    c := !c + 1N

let get2(c: Counter): Nat =
    !c
```

Aufgabe 3 Semantik mit Zustand (Einreichaufgabe, 6 Punkte)

Motivation: In dieser Aufgabe sollen Sie die dynamische Semantik mit einem Speicher nachvollziehen. Sie können sich an den Vorlesungsfolien 753 bis 825 sowie am Skript Kapitel 7.2 orientieren.

Die Deklarationen

```
let i = ref 2N
let j = ref 4N
let x = ref true
let a = ref i
let b = ref j
```

werten zu der folgenden Umgebung aus:

$$\delta = \{i \mapsto a_0,\ j \mapsto a_1,\ x \mapsto a_2,\ a \mapsto a_3,\ b \mapsto a_4\}$$

Hierbei bezeichnen a_0, a_1, a_2, a_3 und a_4 Adressen. Der dazugehörige Speicher ist:

$$\sigma = \{a_0 \mapsto 2N,\ a_1 \mapsto 4N,\ a_2 \mapsto \mathit{true},\ a_3 \mapsto a_0,\ a_4 \mapsto a_1\}$$

Die folgenden Ausdrücke sollen nun jeweils in der Umgebung δ und dem Speicher σ ausgewertet werden. Geben Sie für jeden Ausdruck den Speicherzustand nach jeder Einzelanweisung an.

Beachten Sie: In jeder Teilaufgabe ist der Speicher vor der Auswertung jeweils σ, die Effekte sind also über die Teilaufgaben hinweg *nicht* kumulativ, innerhalb einer Teilaufgabe jedoch schon.

a)
```
x := if !x then !(!a) < !i else !i = !j // σ1
i := !(!a) + (if !x then 1N else 2N)    // σ2
```

Bezeichner	i	j	x	a	b
Adresse	a_0	a_1	a_2	a_3	a_4
σ	2N	4N	true	a_0	a_1
σ_1	2N	4N	false	a_0	a_1
σ_2	4N	4N	false	a_0	a_1

b)
```
i := 0N ; j := 10N  // σ1
while !i < 3N do
  (if !(x := not !x; x) then j := !(!b) + !i) ; !a := !i + 1N ; // σ2, σ3, σ4
```

Bezeichner	i	j	x	a	b
Adresse	a_0	a_1	a_2	a_3	a_4
σ	2N	4N	true	a_0	a_1
σ_1	0N	10N	true	a_0	a_1
σ_2	1N	10N	false	a_0	a_1
σ_3	2N	11N	true	a_0	a_1
σ_4	3N	11N	false	a_0	a_1

Aufgabe 4 Veränderbare Listen (Einreichaufgabe, 10 Punkte)

Motivation: In dieser Aufgabe sollen Sie den Umgang mit Speicherzellen anhand eines komplexeren Problems einüben. Sie können sich an den Vorlesungsfolien 753 bis 825 sowie am Skript Kapitel 7.2 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Lists.fs` *aus der Vorlage* `Aufgabe-10-4.zip`.

Wir verwenden einen von der Vorlesung abweichenden Typ für veränderbare Listen.

```
type Item<'a> =
    { mutable elem: 'a
      mutable next: Option<Item<'a>> }

type MList<'a> =
    { mutable first: Option<Item<'a>>
      mutable last: Option<Item<'a>>
      mutable size: Nat }
```

Zunächst definieren wir einen Typ `Item<'a>`, welcher die Listenelemente repräsentieren soll. Jedes Listenelement besteht aus einem Wert und ggf. einer Referenz auf ein weiteres Listenelement. Das letzte Element einer Liste hat kein Folgeelement, daher hat für dieses Element `next` den Wert `None`. Die Liste insgesamt wird durch den Typ `MList<'a>` repräsentiert. Dieser Typ ist gegenüber herkömmlichen Listen etwas erweitert. Zum Einen speichern wir Referenzen sowohl zum ersten als auch zum letzten Element der Liste. Dadurch können wir neue Elemente sehr effizient an den Anfang und ans Ende der Liste anhängen (zur Erinnerung: bei herkömmlichen Listen müssen wir ganz durch die Liste durchgehen, um ein Element ans Ende anzuhängen). Zum Anderen merken wir uns mit `size` immer die aktuelle Länge der Liste.

Hinweis: Es ist nicht Sinn dieser Aufgabe, entsprechende Funktionen für herkömmliche Listen zu schreiben und zwischen veränderbaren Listen und herkömmlichen Listen hin- und herzukonvertieren. Solche Abgaben werden mit 0 Punkten bewertet.

a) Schreiben Sie eine Funktion `isEmpty<'a>: MList<'a> -> Bool`, die eine veränderbare Liste nimmt und zurückgibt, ob diese leer ist. Verwenden Sie das Feld `size`, um zu ermitteln, ob die Liste leer ist.

```
let isEmpty<'a> (l: MList<'a>): Bool =
    l.size = 0N
```

b) Schreiben Sie eine Funktion `appendFront<'a>: 'a -> MList<'a> -> Unit`, die einen Wert `v` sowie eine veränderbare Liste nimmt und diesen Wert vorne an die Liste anhängt.

Hinweis: Bei einer einelementigen Liste müssen `first` und `last` Referenzen auf dasselbe Objekt sein.

```
let appendFront<'a> (v: 'a) (l: MList<'a>): Unit =
    let elem = Some { elem = v; next = l.first }
    l.first <- elem
    if l.size = 0N then l.last <- elem
    l.size <- l.size + 1N
```

Zunächst konstruieren wir das neue Listenelement `elem` vom Typ `Item<'a>` mit dem Wert `v` und der Referenz auf das bisher erste Element der Liste. Danach setzen wir die Referenz für das erste Listenelement auf das neu konstruierte Element. Wenn wir in eine leere Liste einfügen (das ist der Fall, wenn `size=0N`, wir könnten alternativ z.B. auch prüfen, ob `l.last` den Wert `None` hat), müssen wir auch die Referenz für das letzte Element der Liste auf das neue Element setzen. Zu guter Letzt inkrementieren wir `size`.

c) Schreiben Sie eine Funktion `appendBack<'a>: 'a -> MList<'a> -> Unit`, die einen Wert `v` sowie eine veränderbare Liste nimmt und diesen Wert ans Ende der Liste anhängt.

Hinweis: Bei einer einelementigen Liste müssen `first` und `last` Referenzen auf dasselbe Objekt sein.

```
let appendBack<'a> (v: 'a) (l: MList<'a>): Unit =
    let elem = Some { elem = v; next = None }
    match l.last with
    | None -> l.first <- elem
    | Some lastElem -> lastElem.next <- elem
    l.last <- elem
    l.size <- l.size + 1N
```

Wir beginnen wieder damit das neue Listenelement zu konstruieren. Da dieses ans Ende der Liste angehängt werden soll, gibt es keine Referenz auf ein Folgeelement, also ist `next = None`.

Dann unterscheiden wir zwei Fälle: Gibt es in der Liste noch kein letztes Element, weil die Liste leer ist, so müssen wir zunächst die Referenz für das erste Element der Liste auf das neue Element setzen. Wenn die Liste nicht leer ist, besitzt sie ein letztes Element, dessen `next` Referenz wir auf das neue Element setzen.

Danach aktualisieren wir die Referenz für das letzte Element und inkrementieren `size`.

d) Schreiben Sie eine Funktion `get<'a>: Nat -> MList<'a> -> Option<'a>`, die einen Index sowie eine veränderbare Liste nimmt und den Wert des Listenelements an der Position des Index zurückgibt. Beachten Sie, dass das erste Element der Liste den Index 0 hat. Liegt der Index außerhalb der Liste, soll `None` zurückgegeben werden. Die Liste soll durch die `get` Funktion nicht verändert werden.

Hinweis: Um an eine bestimmte Position in der Liste zu navigieren, können Sie sich zum Beispiel eine rekursive Hilfsfunktion definieren, die in einem ihrer Argumente eine natürliche Zahl erwartet, die bei jedem Durchlauf um eins heruntergezählt wird. Wenn Sie also `index` als Argument übergeben und in jedem Schritt ein Element in der Liste weitergehen, sind Sie an der Position `index`, sobald das Argument 0 ist.

```
let get<'a> (index: Nat) (l: MList<'a>): Option<'a> =
    let rec h (i: Nat) (item: Option<Item<'a>>) =
        match item with
        | None -> None
        | Some e ->
            if i = 0N then Some e.elem
            else h (i-1N) e.next
    h index l.first
```

Dem Hinweis folgend definieren wir eine rekursive Hilfsfunktion, die wir mit dem gesuchten Index und dem ersten Listenelement aufrufen. Die Idee ist, dass wir in der Hilfsfunktion so lange mit der `next` Referenz zum jeweils nächsten Element springen und bei jedem Schritt den Zähler `i` dekrementieren, bis wir insgesamt `index` Elemente durchlaufen haben. Der Zähler steht dann auf 0 und `item` ist das gesuchte Element.

Wir prüfen hier jedoch zuerst, ob es überhaupt ein aktuelles Element gibt. Damit fangen wir den Fall ab, dass der Index außerhalb der Liste liegt. Gibt es das Element, prüfen wir, ob wir an der gesuchten Position sind und falls ja, können wir den dort hinterlegten Wert mit `Some e.elem` zurückgeben. Wenn die Position noch nicht erreicht ist, rufen wir die Funktion rekursiv auf, um zum nächsten Listenelement zu springen. Dazu übergeben wir einen dekrementierten Zähler sowie das nächste Listenelement.

e) Schreiben Sie eine Funktion `update<'a>: Nat -> 'a -> MList<'a> -> Unit`, die einen Index sowie einen Wert und eine veränderbare Liste nimmt und in der Liste das Element an der Position des Index durch den übergebenen Wert ersetzt. Liegt der übergebene Index außerhalb der Liste, so soll die Funktion `update` die Liste nicht verändern.

Tipp: Verwenden Sie eine rekursive Hilfsfunktion ähnlich wie bei Aufgabenteil d).

```
let update<'a> (index: Nat) (v: 'a) (l: MList<'a>): Unit =
    let rec h (i: Nat) (item: Option<Item<'a>>) =
        match item with
        | None -> ()
        | Some e ->
            if i = 0N then e.elem <- v
            else h (i-1N) e.next
    h index l.first
```

Bis auf zwei kleine Änderungen gehen wir genauso vor wie im letzten Aufgabenteil. Für den Fall, dass es kein Listenelement mehr gibt, obwohl der Zähler noch nicht 0 ist (Index außerhalb der Liste), geben wir direkt das leere Tupel `()` zurück. Wenn `i = 0N` ist, sind wir an der gesuchten Position und können den Wert des Elements mit `e.elem <- v` aktualisieren.

f) Freiwillige Zusatzaufgabe: Schreiben Sie eine Funktion `remove<'a>: Nat -> MList<'a> -> Unit`, die einen Index und eine veränderbare Liste nimmt und das Element an der Position des Index aus der Liste entfernt.

```
let remove<'a> (index: Nat) (l: MList<'a>): Unit =
    let rec h (i: Nat) (prev: Option<Item<'a>>) (curr: Option<Item<'a>>) =
        match curr with
        | None -> ()
        | Some c ->
            if i = 0N then
                match prev with
                | None -> l.first <- c.next
                | Some p -> p.next <- c.next
                match c.next with
                | None -> l.last <- prev
                | _ -> ()
                l.size <- l.size - 1N
            else
                h (i-1N) curr c.next

    h index None l.first
```

Wir erweitern unsere Idee mit der rekursiven Hilfsfunktion dahingehend, dass wir nun sowohl das aktuelle Element `curr` als auch das vorherige Element `prev` übergeben.

Beim Aufruf der rekursiven Hilfsfunktion übergeben wir als aktuelles Element das erste Listenelement. Da kein vorheriges Element existiert, übergeben wir für `prev` den Wert `None`

Wir behandeln zuerst wieder den Fall, dass es kein aktuelles Listenelement gibt (`index` liegt außerhalb der Liste). Anschließend prüfen wir, ob wir an der gesuchten Position stehen (`i = 0N`).

Falls nicht, rufen wir die Hilfsfunktion rekursiv auf, dabei wird der Zähler `i` dekrementiert. Als „neues vorheriges“ Element übergeben wir `curr` und als „neues aktuelles“ Element den Nachfolger `c.next` (der, falls `index` außerhalb der Liste liegt, irgendwann `None` ist und damit im rekursiven Aufruf abgefangen wird).

Falls wir uns jedoch an der gesuchten Position in der Liste befinden, können wir das Element entfernen. Dabei müssen wir einige Spezialfälle beachten. Zunächst prüfen wir, ob es ein vorheriges Element `prev` gibt. Wenn es kein vorheriges Element gibt, heißt das, dass wir das Element am Anfang der Liste löschen möchten. Dann müssen wir die Referenz der Liste für das erste Element ändern. Wir

aktualisieren es mit `l.first <- c.next` auf das vormals zweite Element in der Liste.

Existiert ein vorheriges Element, so muss dessen `next` Referenz auf die `next` Referenz des zu löschenden Elements gesetzt werden. Das zu löschende Element ist damit nicht mehr Teil der Kette - es ist aus der Liste entfernt.

Nun müssen wir noch prüfen, ob das zu löschende Element das letzte Element der Liste ist. Dies ist der Fall, wenn `c.next` den Wert `None` hat und wir aktualisieren entsprechend die Referenz der Liste für das letzte Element.

Am Ende dekrementieren wir schließlich `size`.