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 10: Grundlagen der Programmierung (WS 2024/25)

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 754 bis 826 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 754 bis 826 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 Hörsaal 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 751 bis 824 sowie am Skript Kapitel 7.2 orientieren.

Die Deklarationen

```
let i = ref 5N
let j = ref 8N
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:

$$\sigma = \{a_0 \mapsto 5N,\ a_1 \mapsto 8N,\ a_2 \mapsto \mathit{true},\ a_3 \mapsto a_0,\ a_4 \mapsto a_1\}$$

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 := !x && (5N > !j) // σ1
i := !j + !i         // σ2
```

Bezeichner	i	j	x	a	b
Adresse	a_0	a_1	a_2	a_3	a_4
σ	5N	8N	true	a_0	a_1
σ_1	5N	8N	false	a_0	a_1
σ_2	13N	8N	false	a_0	a_1

b)
```
!a := !(!b) - !(!b)      // σ1
!b := !i + !j + !(!b)    // σ2
b := !a                  // σ3
x := !i <> !(!a)         // σ4
```

Bezeichner	i	j	x	a	b
Adresse	a_0	a_1	a_2	a_3	a_4
σ	5N	8N	true	a_0	a_1
σ_1	0N	8N	true	a_0	a_1
σ_2	0N	16N	true	a_0	a_1
σ_3	0N	16N	true	a_0	a_0
σ_4	0N	16N	false	a_0	a_0

Aufgabe 4 Veränderbare Listen (Einreichaufgabe, 15 Punkte)

Motivation: In dieser Aufgabe sollen Sie den Umgang mit Speicherzellen anhand eines komplexeren Problems einüben. Sie können sich an den Vorlesungsfolien 751 bis 824 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> =
    { value: Ref<'a>
      next: Ref<Option<Item<'a>>> }

type NEList<'a> =
    { first: Item<'a>
      last: Item<'a>
      length: Nat }

type MList<'a> = Ref<Option<NEList<'a>>>
```

Der Typ `Item<'a>` repräsentiert einzelne Listenelemente. Jedes Element besteht aus einer Referenz zu einem in der Liste gespeicherten Wert und ggf. einer Referenz auf das nächste Listenelement. Das letzte Element einer Liste hat kein Folgeelement, daher speichert die `next` Referenz in diesem Element den Wert `None`.

Der Typ `NEList<'a>` beschreibt eine nicht-leere Liste. Neben dem ersten Listenelement speichern wir uns auch das letzte Listenelement sowie die Länge der Liste. Dadurch können wir neue Elemente effizient am Anfang und am Ende der Liste einfügen. Zur Erinnerung: Bei herkömmlichen Listen müssen wir die gesamte Liste durchlaufen, um ein Element am Ende einzufügen oder die Länge der Liste zu bestimmen.

Der allgemeine Listentyp ist nun `MList<'a>`. Dieser besteht aus einer Referenz auf `Option<NEList<'a>>`. Die leere Liste wird durch eine Referenz auf `None` dargestellt, nicht-leere Listen durch eine Referenz auf eine Liste vom Typ `NEList<'a>`.

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 `length<'a>: MList<'a> -> Nat`, die eine veränderbare Liste nimmt und deren Länge zurückgibt. Nutzen Sie den `length` Eintrag im `NEList<'a>` Record und durchlaufen Sie nicht die gesamte Liste.

```
let length<'a> (l: MList<'a>): Nat =
    match !l with
    | None -> 0N
    | Some lst -> lst.length
```

b) Schreiben Sie eine Funktion `insertFirst<'a>: 'a -> MList<'a> -> Unit`, die einen Wert `v` sowie eine veränderbare Liste nimmt und `v` vorne in die Liste einfügt.

Hinweis: Wenn die Liste bislang leer war, dann müssen Sie eine einelementige Liste erstellen. Dabei müssen `first` und `last` dasselbe `Item` enthalten.

```
let insertFirst<'a> (v: 'a) (l: MList<'a>): Unit =
    match !l with
    | None ->
        let item = { value = ref v; next = ref None }
        l := Some { first = item; last = item; length = 1N }
    | Some lst ->
        let item = { value = ref v; next = ref (Some lst.first) }
        l := Some { lst with first = item; length = lst.length + 1N }
```

Zunächst prüfen wir, ob die Liste leer ist. Im leeren Fall konstruieren wir das `item` mit dem Wert `v` und geben an, dass kein nächstes Element existiert. Wir konstruieren eine neue Liste, bei der dem Hinweis entsprechend sowohl das erste als auch das letzte Element `item` ist.

Im nicht-leeren Fall konstruieren wir wieder das `item` mit dem Wert `v` und geben als nächstes Element das bisher erste Element an. Wir aktualisieren dann `first` und `size` in der Liste, `last` bleibt unverändert.

c) Schreiben Sie eine Funktion `insertLast<'a>: 'a -> MList<'a> -> Unit`, die einen Wert `v` sowie eine veränderbare Liste nimmt und `v` ans Ende der Liste anhängt. Nutzen Sie den `last` Eintrag im `NEList<'a>` Record und durchlaufen Sie nicht die gesamte Liste.

Hinweis: Der Hinweis aus der vorherigen Teilaufgabe gilt auch hier.

```
let insertLast<'a> (v: 'a) (l: MList<'a>): Unit =
    let item = { value = ref v; next = ref None }
    match !l with
    | None ->
        l := Some { first = item; last = item; length = 1N }
    | Some lst ->
        lst.last.next := Some item
        l := Some { lst with last = item; length = lst.length + 1N }
```

Wir beginnen damit, das neue `item` zu konstruieren. Da dieses ans Ende der Liste angehängt werden soll, gibt es kein Folgeelement und `next` speichert eine Referenz auf `None`.

Dann unterscheiden wir wieder die zwei Fälle, ob die Liste leer ist oder nicht: Im leeren Fall gehen wir genauso vor wie bei `insertFirst`.

Im nicht-leeren Fall aktualisieren wir zuerst die `next` Referenz im bisherigen letzten Listenelement auf das neu erstellte Listenelement. Anschließend aktualisieren wir `last` und `size` in der Liste, `first` bleibt unverändert.

d) Schreiben Sie eine Funktion `skip<'a>: Nat -> Item<'a> -> Option<Item<'a>>`, die eine natürliche Zahl `n` sowie ein `item` nimmt. Die Funktion soll die ersten `n` Elemente der verketteten Struktur (`next` Referenzen) überspringen. Zurückgegeben wird das Element an Index `n`, wobei die Nummerierung bei 0 beginnt. Bei `n = 0` soll also `Some item` zurückgegeben werden. Wenn es nicht genügend Nachfolgeelemente gibt, dann soll `None` zurückgegeben werden.

Hinweis: Diese Funktion ist als Hilfsfunktion für die folgenden Teilaufgaben gedacht. Überlegen Sie sich jeweils, wie Sie `skip` dort sinnvoll einsetzen können.

```
let rec skip (n: Nat) (item: Item<'a>): Option<Item<'a>> =
    if n = 0N then Some item
    else
        match !item.next with
        | None -> None
        | Some next -> skip (n - 1N) next
```

Wir starten mit einer Fallunterscheidung ob `n = 0` ist. Wenn ja, dann müssen keine Elemente übersprungen werden und wir können direkt `Some item` zurückgeben.

Ansonsten muss mindestens das erste Element übersprungen werden. Wir untersuchen daher seine `next` Referenz. Enthält diese `None`, so gibt es kein Folgeelement und wir geben wie gefordert `None` zurück. Ansonsten nutzen wir einen rekursiven Aufruf: Beim Nachfolgeelement startend müssen wir nun ein Element weniger überspringen.

e) 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.

```
let get<'a> (index: Nat) (l: MList<'a>): Option<'a> =
    match !l with
    | None -> None // leere Liste
    | Some lst ->
        match skip index lst.first with
        | None -> None // index ungültig
        | Some item -> Some !item.value
```

Wir beginnen wir mit einer Fallunterscheidung danach, ob die Liste leer ist. Im leeren Fall geben wir `None` zurück, weil der gesuchte Index auf jeden Fall ungültig ist. Ansonsten starten wir beim ersten Listenelement und überspringen mit Hilfe von `skip` so viele Elemente, bis wir an der gewünschten Indexposition angekommen sind. Wenn das gewünschte Element existiert, können wir seinen Wert zurückgeben, ansonsten ebenfalls `None`.

f) 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.

```
let update<'a> (index: Nat) (v: 'a) (l: MList<'a>): Unit =
    match !l with
    | None -> () // leere Liste
    | Some lst ->
        match skip index lst.first with
        | None -> () // index ungültig
        | Some item -> item.value := v
```

Bis auf zwei kleine Änderungen gehen wir genauso vor wie im vorherigen Aufgabenteil. Wenn das gewünschte Element existiert, dann wird sein Wert aktualisiert. Ansonsten wird das leere Tupel `()` zurückgegeben.

g) Kommentieren Sie Ihren Code so, dass Ihre Lösung einfach nachzuvollziehen ist. Außerdem sollte Ihr Code keine unnötig komplexen Ausdrücke enthalten, siehe auch Aufgabe 5 auf Übungsblatt 5. Dafür vergeben wir hier drei Punkte.

h) *Freiwillige Zusatzaufgabe:* Schreiben Sie eine Funktion `remove<'a>: Nat -> MList<'a> -> Unit`, die einen Index sowie 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 =
  match !l with
  | None -> () // leere Liste
  | Some lst ->
      if index = 0N then
        // Erstes Element der Liste soll gelöscht werden
        match !lst.first.next with
        | None ->
            // Einziges Element wird gelöscht, Liste ist dann leer
            l := None
        | Some next ->
            // Erstes Element wird gelöscht, es gibt weitere Elemente
            l := Some { lst with first = next; length = lst.length - 1N }
      else
        // Finde Element, das vor dem zu löschenden Element liegt
        match skip (index - 1N) lst.first with
        | None -> () // index ungültig
        | Some prev ->
            // Untersuche das zu löschende Element
            match !prev.next with
            | None -> () // index ungültig
            | Some item ->
                // Untersuche das nächste Element
                match !item.next with
                | None ->
                    // Letztes Element der Liste wird gelöscht
                    prev.next := None
                    l := Some { lst with last = prev; length = lst.length - 1N }
                | Some next ->
                    // Zu löschendes Element ist weder das erste
                    // noch das letzte Element in der Liste
                    prev.next := Some next
                    l := Some { lst with length = lst.length - 1N}
```