Prof. Dr. Ralf Hinze
Markus Heinrich, M.Sc.
Sebastian Schweizer, M.Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Übungsblatt 12: Grundlagen der Programmierung (WS 2019/20)

Ausgabe: 31. Januar 2020
Abgabe: 7. Februar 2020, 15 Uhr

Aufgabe 1 Warteschlangen (11 Punkte)

Schreiben Sie Ihre Lösungen in die Datei `Queues.fs` *aus der Vorlage* `Aufgabe-12-1.zip`.

Wir definieren eine Schnittstelle `IQueue`, um Warteschlangen zu beschreiben. Objekte, welche die Schnittstelle implementieren, müssen die Methoden `Add` und `Remove` definieren. Mit Hilfe von `Add` soll ein Element ans Ende der Warteschlange angehängt werden. Die Methode `Remove` entfernt das vorderste Element in einer nicht leeren Warteschlange und gibt es zurück. Falls die Warteschlange leer ist, wird `None` zurückgegeben.

```
type IQueue<'T> =
    interface
        abstract member Add: 'T -> Unit
        abstract member Remove: Unit -> 'T option
    end
```

a) Schreiben Sie eine Funktion `simpleQueue: Unit -> IQueue<'T>`, die ein `IQueue` Objekt erzeugt und zurückgibt. Verwenden Sie als Warteschlange nur eine Liste. Ein Element wird dann zur Warteschlange hinzugefügt, indem es ans Ende der Liste angehängt wird. Ein Element aus der Warteschlange zu entfernen wird dadurch realisiert, dass das erste Element aus der Liste entfernt wird.

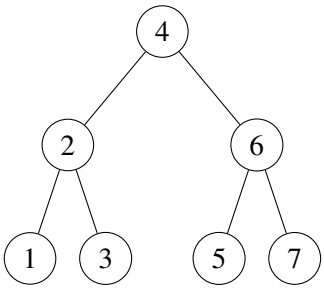
b) Implementieren Sie die Funktion `advancedQueue: Unit -> IQueue<'T>`, die ebenfalls ein `IQueue` Objekt erzeugt und zurückgibt. Hier soll die Warteschlange allerdings durch zwei Listen `front` und `rear` abgebildet werden. Ein neues Element wird zur Warteschlange hinzugefügt, indem es einfach an den Anfang der `front` Liste eingefügt wird. Wir entfernen ein Element aus der Warteschlange, indem wir das erste Element der `rear` Liste entfernen und zurückgegeben. Wenn allerdings die `rear` Liste leer ist, müssen wir zuvor die Elemente der `front` Liste in umgekehrter Reihenfolge in `rear` schreiben und alle Einträge aus `front` entfernen. Sind sowohl `front` als auch `rear` leer, dann ist die Warteschlange insgesamt leer und es soll `None` zurückgegeben werden.

   Der Vorteil hierbei ist, dass die Laufzeit der beiden Operationen besser ist als in der ersten Variante.

c) Implementieren Sie die Funktionen `enqueue: IQueue<'T> -> 'T list -> Unit` und `dequeue: IQueue<'T> -> 'T list`. `enqueue` fügt eine Liste von Elementen der Reihe nach in eine Warteschlange ein. `dequeue` entfernt alle Elemente aus der Warteschlange und gibt diese als Liste zurück (erstes Element der Warteschlange vorne in der Liste).

   Beachten Sie, dass die Funktionen unabhängig von einer konkreten Implementierung der `IQueue` Schnittstelle sind. Wir können sowohl eine `simpleQueue` als auch eine `advancedQueue` verwenden, ohne `enqueue` oder `dequeue` anpassen zu müssen.

d) Warteschlangen werden als Hilfsmittel in manchen Algorithmen benötigt. Ein Beispiel hierfür ist der Breitendurchlauf von Bäumen. Wir betrachten dazu den bekannten Typ für Binärbäume:




```
type Tree<'T> =
    | Empty
    | Node of Tree<'T> * 'T * Tree<'T>
```

Beim Breitendurchlauf werden die Elemente ebenenweise durchlaufen, in nebenstehendem Beispiel also 4, 2, 6, 1, 3, 5, 7.

Um einen Baum der Breite nach zu durchlaufen, wenden wir folgenden Algorithmus an:

1. Der Baum wird als Ganzes in die Warteschlange eingefügt.
2. Das erste Element wird aus der Warteschlange entfernt. Wenn wir `None` erhalten haben, ist die Warteschlange leer und wir sind fertig.
3. Wenn es sich bei dem entfernten Element um ein Blatt (ein Element `Empty`) handelt, gehe zurück zu Punkt 2.
4. Wenn es sich bei dem entfernten Element um einen Knoten (ein Element `Node`) handelt, füge den linken und rechten Teilbaum zur Warteschlange hinzu. Das im Knoten enthaltene Element ist das vorderste Element der Rückgabeliste. Um den Rest der Rückgabeliste zu berechnen, gehe zurück zu Punkt 2.

Schreiben Sie eine Funktion `bft: IQueue<Tree<'T>> -> 'T list` (für *breadth-first traversal*), die als Argument einen Baum in einer Warteschlange erwartet (Punkt 1 müssen Sie also nicht implementieren) und diesen der Breite nach durchläuft, um die gefundenen Elemente in einer Liste zurückzugeben.

Aufgabe 2 Untertypen (7 Punkte)

```
type A =
    interface
        abstract member f: Nat -> Nat
    end

type B =
    interface
        inherit A
        abstract member g: Nat -> String
    end
```

```
type C =
    interface
        abstract member h: Unit -> Nat
    end

type D =
    interface
        inherit C
        abstract member i: Nat -> Unit
    end
```

Verwenden Sie die Schnittstellentypdefinitionen von oben, um den Typ der folgenden Ausdrücke mit einem vollständigen Beweisbaum anzugeben. Benutzen Sie die Regeln der **statischen Semantik** aus der Vorlesung.

a) `fun (b: B) -> b.g ((b :> A).f 1N)`

b) `fun (r: A -> D) -> (fun (b: B) -> (r b).h ())`

Aufgabe 3 Geometrische Formen (12 Punkte)

Schreiben Sie Ihre Lösungen in die Datei `Shapes.fs` *aus der Vorlage* `Aufgabe-12-3.zip`.

Wir implementieren noch einmal die zweidimensionalen geometrischen Formen von Übungsblatt 5, Aufgabe 4 indem wir Objekte statt Varianten[1] verwenden.

Eine geometrische Form ist dann ein Objekt, das die Schnittstele `IShape` implementiert:

```
type IShape =
    interface
        abstract member Contains: Nat -> Nat -> Bool
        abstract member Rightmost: Nat
        abstract member Topmost: Nat
    end
```

Die Methoden sollen sich wie folgt verhalten:

- `Contains: Nat -> Nat -> Bool` erwartet als erstes Argument eine x- und als zweites Argument eine y-Koordinate und prüft, ob diese in der von der Form bedeckten Fläche enthalten ist. Die Formflächen beinhalten jeweils auch die Kanten und Eckpunkte.
- Die Eigenschaften `Rightmost: Nat` und `Topmost: Nat` sind die größte x- bzw. y-Koordinate der Form.

a) Schreiben Sie eine Funktion `square: Nat -> IShape`, welche eine Form erzeugt und zurückgibt, die ein Quadrat der Seitenlänge `size` beschreibt.

   Beispiel: `square 3N` erzeugt ein Quadrat mit linker unterer Ecke an Koordinate $(0, 0)$ und rechter oberer Ecke an Koordinate $(3, 3)$. Das durch `square 0N` erzeugte Quadrat ist der einzelne Punkt $(0, 0)$.

b) Schreiben Sie eine Funktion `rectangle: Nat * Nat -> IShape`, welche eine Form erzeugt und zurückgibt, die ein Rechteck der Breite `width` und Höhe `height` beschreibt.

   Beispiel: `rectangle (4N, 5N)` erzeugt ein Rechteck mit linker unterer Ecke an Koordinate $(0, 0)$ und rechter oberer Ecke an Koordinate $(4, 5)$.

c) Schreiben Sie eine Funktion `union: IShape * IShape -> IShape`, welche zwei Formen `s1` und `s2` als Argumente erwartet und eine Form erzeugt und zurückgibt, die die Vereinigung der beiden Formen beschreibt.

   Beispiel: `union ((square 3N), (rectangle (4N, 1N)))`
   Dabei können sich die Formen überlappen, die eine kann aber auch komplett in der anderen enthalten sein oder es gibt keinerlei Überlappung.

d) Schreiben Sie eine Funktion `move: Nat * Nat -> IShape -> IShape`, die aus einer Form `s` eine um die Koordinaten `(deltaX, deltaY)` verschobene Form erzeugt und zurückgibt.

   Beispiel: `move (1N, 9N) (square 4N)` beschreibt das Quadrat mit linker unterer Ecke an Koordinate $(1, 9)$ und rechter oberer Ecke an Koordinate $(5, 13)$.

[1] s. dazu auch S. 400 im Skript