

## Übungsblatt 12: Grundlagen der Programmierung (WS 2023/24)

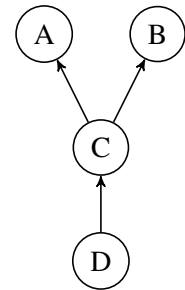
Ausgabe: 30. Januar 2024

Abgabe: 05./06./07. Februar 2024, siehe [Homepage](#)

### Aufgabe 1 Untertypen (Präsenzaufgabe)

*Motivation:* In dieser Aufgabe sollen Sie sich mit Untertypbeziehungen beschäftigen. Sie können sich an den Vorlesungsfolien 901 bis 969 sowie an den Kapiteln 8.1 und 8.2 im Skript orientieren.

Es gelten die Untertypbeziehungen  $D \leq C$ ,  $C \leq A$  und  $C \leq B$ , die in der nebenstehenden Abbildung visualisiert sind.



In der folgenden Tabelle werden je zwei Typen in Relation gesetzt:

- $t_1 < t_2$  bedeutet, dass  $t_1$  ein Untertyp von  $t_2$  ist ( $t_1 \leq t_2$ ), nicht jedoch  $t_2$  ein Untertyp von  $t_1$ .
- $t_1 > t_2$  bedeutet, dass  $t_2$  ein Untertyp von  $t_1$  ist ( $t_2 \leq t_1$ ), nicht jedoch  $t_1$  ein Untertyp von  $t_2$ .
- $t_1 \parallel t_2$  bedeutet, dass  $t_1$  und  $t_2$  unvergleichbar sind, das heißt es gilt weder  $t_1 \leq t_2$  noch  $t_2 \leq t_1$ .

Füllen Sie die Lücken in der Tabelle aus. In der ersten und dritten Spalte müssen Sie einen Typ eintragen, in der zweiten Spalte eine Relation (< oder > oder ||).

*Zur Erinnerung:* Die folgenden Deduktionsregeln gelten für die Relation  $\leq$ :

$$\frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3}$$

$$\frac{t_1 \leq t'_1 \quad t_2 \leq t'_2}{t_1 * t_2 \leq t'_1 * t'_2}$$

$$\frac{t'_1 \leq t_1 \quad t_2 \leq t'_2}{t_1 \rightarrow t_2 \leq t'_1 \rightarrow t'_2}$$

$t_1$	Relation	$t_2$
$D$		$A$
	$\succ$	$C$
$A * C$	$\prec$	
$A * B$		$B * A$
$C * D$	$\succ$	
$A \rightarrow C$		$C \rightarrow C$
$B \rightarrow D$		$C \rightarrow C$
$C \rightarrow D$	$\succ$	
$C \rightarrow A$		$B \rightarrow D$
$D \rightarrow A$		$C \rightarrow B$

## Aufgabe 2 Warteschlangen (Einreichaufgabe, 15 Punkte)

Schreiben Sie Ihre Lösungen in die Datei `Queues.fs` aus der Vorlage `Aufgabe-12-2.zip`.

Wir definieren eine Schnittstelle `IQueue`, um Warteschlangen zu beschreiben. Objekte, welche die Schnittstelle implementieren, müssen die Methoden `isEmpty`, `Add` und `Remove` definieren. Die Methode `isEmpty` gibt zurück, ob die Warteschlange leer ist. Mit Hilfe von `Add` soll ein Element in die Warteschlange eingefügt 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<'a> =  
  interface  
    abstract member isEmpty: Unit -> Bool  
    abstract member Add: 'a -> Unit  
    abstract member Remove: Unit -> Option<'a>  
  end
```

- Schreiben Sie eine Funktion `simpleQueue: Unit -> IQueue<'a>`, 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.
- Implementieren Sie die Funktion `priorityQueue: Unit -> IQueue< QElem<'a> >`, die ein `IQueue` Objekt erzeugt und zurückgibt. Der Typ

```
type QElem<'a> =  
  { priority: Nat  
    value: 'a }
```

ist Ihnen von Blatt 5, Aufgabe 2, bekannt. Entsprechend dieser Aufgabe soll auch die Warteschlange intern funktionieren.

- Implementieren Sie die Funktion `advancedQueue: Unit -> IQueue<'a>`, 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 zurückgeben. 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.

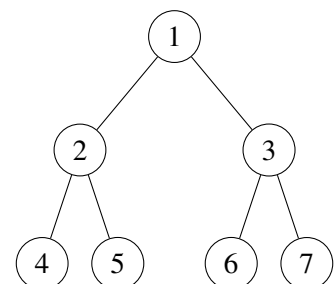
- Implementieren Sie die Funktionen `enqueue: IQueue<'a> -> List<'a> -> Unit` und `dequeue: IQueue<'a> -> List<'a>`. Die Funktion `enqueue` fügt eine Liste von Elementen der Reihe nach in eine Warteschlange ein, während `dequeue` alle Elemente aus der Warteschlange entfernt und diese als Liste zurückgibt (das erste Element der Warteschlange befindet 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.

- 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<'a> =  
  | Empty  
  | Node of Tree<'a> * 'a * Tree<'a>
```

Beim Breitendurchlauf werden die Elemente ebenenweise durchlaufen, in nebenstehendem Beispiel also 1, 2, 3, 4, 5, 6, 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<'a>> -> List<'a>` (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 3 Untertypen (Trainingsaufgabe)

*Motivation:* In dieser Aufgabe sollen Sie die Regeln der statischen Semantik von Schnittstellen und Untertypen einüben. Sie können sich an den Vorlesungsfolien 899 bis 967 sowie an den Kapiteln 8.1 und 8.2 im Skript orientieren.

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

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

type C =
  interface
    inherit A
    abstract member h: String -> 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 (s : B * D) -> ((snd s) :> C).h ((fst s).g 1N)`
- b) `fun (s: A -> D) -> (fun (b: B) -> (s b).f ())`