1 Abstrakte Datentypen

Ein Abstrakter Datentyp (ADT) ist ein Datentyp, der unabhängig von einer Implementierung durch seinen Wertebereich und den darauf definierten Operationen bestimmt wird.

Die bekanntesten Beispiele für ADTs ist der Kellerspeicher (Stack) und die Warteschlange (Queue), Bäume (binäre und Mehrwegbäume) und Graphen. Man nennt den Stack auch LIFO-Struktur, denn er arbeitet nach dem Prinzip: Last in, First out. Entsprechend ist eine Queue eine FIFO-Struktur.

Aufgabe: Welche Anwendungsbeispiele findest du für Stacks, Queues, Bäume und Graphen?

Ein Abstrakter Datentyp besteht aus zwei Teilen. Zum einen aus der Signatur (Syntax) der Operationen und den darin verwendeten Basistypen, und zum anderen aus der Semantik, der den Basistypen und den definierten Operationen eine Bedeutung zukommen lässt. Basistypen können selbst wieder ADTs sein. Somit beschreibt ein ADT, *was* die Operationen tun (Semantik), aber noch nicht, *wie* sie es tun (Implementierung). Die Semantik kann verbal, über Pseudocode oder Funktionen spezifiziert werden, solange diese präzise und eindeutig sind.

Es gibt verschiedene Methoden ADTs zu definieren, also seine Spezifikation anzugeben. Bei der mathematisch-axiomatischen Definition wird die Semantik der Struktur wie in der Mathematik durch ein Axiomensystem beschrieben. Die Operationen werden vollständig durch deren Beziehungen untereinander definiert. Die mathematisch-algebraische Definition bedient sich bei der Semantik hauptsächlich prädikatenlogischer Ausdrücke (das wird hier nicht betrachtet).

Weit verbreitet ist die Beschreibung eines ADTs durch informelle Beschreibungen als Interface. Die Semantik wird dabei als Kommentar (oft in natürlicher Sprache) angegeben.

1.1 Angabe einer Spezifikation durch die mathematisch-axiomatische Methode

```
1  Basisdatentypen:
2
3    STACK      (Kellerspeicher mit LIFO-Eigenschaften)
4    ELEMENT    (Elemente die im Kellerspeicher gespeichert werden; eigener ADT)
5    BOOLEAN    (TRUE/FALSE Werte)
6
7  Operationen:
8
9    create:                  -> STACK    (legt leeren Stack an)
10   isEmpty: STACK           -> BOOLEAN  (gibt TRUE aus, wenn Stack leer ist)
11   push: STACK x ELEMENT    -> STACK    (ein Element wird zu oberst auf den Stack gepackt)
12   pop: STACK               -> STACK    (das oberste Element wird geloescht)
13   top:  STACK              -> ELEMENT  (das oberste Element wird ausgegeben)
14
15 Axiome:
16
17   s: STACK
18   x: ELEMENT
19
20   isEmpty(create())    = TRUE
21   isEmpty(push(s,x))   = FALSE
22   pop(push(s,x))       = s
```

```
23    top(push(s,x))       = x
24    push(pop(s),top(s)) = s, falls isEmpty(s) = FALSE
25    pop(create())        = ERROR
26    top(create())        = ERROR
```

1.2 Informelle Methode durch Schnittstellenbeschreibung

1. Diese Methode nutzt die starke Trennung von Spezifikation und Implementierung (Information Hiding). Zuerst wird in einer Schnittstellenbeschreibung angegeben, also wie der neue Typ heißt und was man damit machen kann.

```haskell
module Stack (Stack, empty, isEmpty, push, top, pop) where

data Stack a  -- opaque!
-- Ein strukturierter Datentyp, der Objekte vom selben Typ nach dem
-- LIFO-Prinzip verwaltet, der Stack kann unendlich gross werden

empty::  Stack a
-- Vor.: keine
-- Effekt: empty erzeugt einen neuen leeren Stack

isEmpty:: Stack a -> Bool
-- Vor.: ein Stack a
-- Effekt: ist der Stack leer, dann ist die Rueckgabe true, anonsten false
-- der Stack ist unveraendert

push :: a -> Stack a -> Stack a
-- Vor.: keine
-- Effekt: das Element ist an die oberste Stelle des Stack angefuegt

top :: Stack a -> a
-- Vor.: der Stack ist nicht leer
-- Effekt: das oberste Element wird aus dem Stack ausgelesen und
-- zurueckgegeben, der Stack ist unveraendert

pop :: Stack a -> (a,Stack a)
-- Vor.: der Stack ist nicht leer
-- Effekt: das oberste Elemente ist aus dem Stack entfernt und wird
-- zurueckgegeben
```

2. Als nächstes wird die Implementierung realisiert. Diese wird dann per `module` und Exportliste in Haskell den anderen Programmen zur Verfügung gestellt.

```haskell
module Stack (Stack, empty, isEmpty, push, top, pop) where

data Stack a = StackImpl [a] -- Nutzung des vordefinierten Listentyps

empty::  Stack a
isEmpty:: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a,Stack a)

---------- Implementierung ----------

instance (Show a) => Show (Stack a) where
    show (StackImpl []) = "-"
    show (StackImpl (x:xs))  = show x ++ "|" ++ show (StackImpl xs)

empty = StackImpl []
```

```
19 isEmpty (StackImpl s) = null s
20
21 push x (StackImpl s) = StackImpl (x:s)
22
23 top (StackImpl s) = head s
24
25 pop (StackImpl (s:ss)) = (s,StackImpl ss)
```

3. Nun noch das Anwendungsprogramm, welches den zur Verfügung gestellten Typ und dessen Funktionen importiert.

```
1 module Main (main) where
2  import Stack
3
4  s1 = push 3 empty
5  s2 = push 4 s1
6
7  main = do
8         putStrLn (show s2)
```

Aufgabe: Fertige eine vollständige Schnittstellenbeschreibung und Implementierung einer Warteschlange (Queue) an.

Lösung: siehe Unterricht