Sven Eric Panitz

Lehrbriefe
Programmierung in Java

Rekursive Listen

Rekursive Listen

Sven Eric Panitz

16. April 2023

Inhaltsverzeichnis

1 Formale Spezifikation

Eine der häufigsten Datenstrukturen in der Programmierung sind Sammlungstypen. In fast jedem nichttrivialen Programm wird es Punkte geben, an denen eine Sammlung mehrerer Daten gleichen Typs anzulegen sind. Eine der einfachsten Strukturen, um Sammlungen anzulegen, sind Listen. Da Sammlungstypen oft gebraucht werden, stellt Java entsprechende Klassen als Standardklassen zur Verfügung.

Wir haben im ersten Semester bereits eine Implementierung von Listen gesehen. Dort wurde eine Reihung verwendet, sodass wir zu einer Array-basierten Liste kamen. Die Elemente wurden in einer Reihung gespeichert und die Listenklasse hat diese bei Bedarf vergößert.

In diesem Lehrbrief werden wir eine komplett andere Umsetzung sehen. Die Listen werden als rekursive Struktur definiert.

Wir spezifizieren Listen formal als abstrakten Datentyp. Ein abstrakter Datentyp (ADT) wird spezifiziert über eine endliche Menge von Methoden.

Hierzu wird spezifiziert, auf welche Weise Daten eines ADT konstruiert werden können. Dazu werden entsprechende Konstruktormethoden spezifiziert. Dann wird eine Menge von Funktionen definiert, die wieder Teile aus den konstruierten Daten selektieren können. Schließlich werden noch Testmethoden spezifiziert, die angeben, mit welchem Konstruktor ein Datum erzeugt wurde.

Der Zusammenhang zwischen Konstruktoren und Selektoren sowie zwischen den Konstruktoren und den Testmethoden wird in Form von Gleichungen spezifiziert.

Der Trick, der angewendet wird, um abstrakte Datentypen wie Listen zu spezifizieren, ist die Rekursion. Das Hinzufügen eines weiteren Elements zu einer Liste wird dabei als das Konstruieren einer neuen Liste aus der Ursprungsliste und einem weiteren Element betrachtet. Mit dieser Betrachtungsweise haben Listen eine rekursive Struktur: eine Liste besteht aus dem zuletzt vorne angehängten neuen Element, dem sogenannten Kopf der Liste, und aus der alten Teilliste, an die dieses Element angehängt wurde, dem Rest der Liste, dem sogenannten *Tail*. Wie bei jeder rekursiven Struktur bedarf es eines Anfangs der Definition. Im Falle von Listen wird dieses durch die Konstruktion einer leeren Liste spezifiziert.[1]

1.1 Konstruktoren

Abstrakte Datentypen wie Listen lassen sich durch ihre Konstruktoren spezifizieren. Die Konstruktoren geben an, wie Daten des entsprechenden Typs konstruiert werden können. In dem Fall von Listen bedarf es nach den obigen Überlegungen zweier Konstruktoren:

- einem Konstruktor für neue Listen, die noch leer sind.
- einem Konstruktor, der aus einem Element und einer bereits bestehenden Liste eine neue Liste konstruiert, indem an die Ursprungsliste das Element angehängt wird.

Wir benutzen in der Spezifikation eine mathematische Notation der Typen von Konstruktoren.[2] Dem Namen des Konstruktors folgt dabei mit einem Doppelpunkt abgetrennt der Typ. Der Ergebnistyp wird von den Parametertypen mit einem Pfeil getrennt. Typvariablen werden mit einem griechischen Buchstaben bezeichnet. Für generische Typen verwenden wir die aus Java bekannte Notation in spitzen Klammern.

Es lassen sich die Typen der zwei Konstruktoren für Listen wie folgt spezifizieren:

[1] Man vergleiche es mit der Definition der natürlichen Zahlen: die 0 entspricht der leeren Liste, der Schritt von n nach $n + 1$ dem Hinzufügen eines neuen Elements zu einer Liste.

[2] Entgegen der Notation in Java, in der der Rückgabetyp kurioser Weise vor den Namen der Methode geschrieben wird. Diese Notation wurde ursprünglich aus der Sprache C übernommen.

- Nil: () →`List<`α`>`
- Cons: (α,`List<`α`>`) →`List<`α`>`

1.2 Selektoren

Die Selektoren können wieder auf die einzelnen Bestandteile der Konstruktion zugreifen. Der Konstruktor `Cons` hat zwei Parameter. Für `Cons`-Listen werden zwei Selektoren spezifiziert, die jeweils einen dieser beiden Parameter wieder aus der Liste selektieren. Die Namen dieser beiden Selektoren sind traditioneller Weise *head* und *tail*.

- head: `List<`α`>`→ α
- tail: `List<`α`>`→`List<`α`>`

Der funktionale Zusammenhang von Selektoren und Konstruktoren lässt sich durch folgende Gleichungen spezifizieren:

$$\begin{aligned} head(Cons(x, xs)) &= x \\ tail(Cons(x, xs)) &= xs \end{aligned}$$

Wie man sieht, gibt es keine Gleichungen für Listen, die mit `Nil` konstruiert wurden. Diese Fälle sind unspezifiziert und werden in einer Implementierung zu einem Fehler führen.

1.3 Testmethoden

Um für Listen Algorithmen umzusetzen, ist es notwendig, unterscheiden zu können, welche Art der beiden Listen vorliegt: die leere Liste oder eine `Cons`-Liste. Hierzu bedarf es noch einer Testmethode, die mit einem bool'schen Wert als Ergebnis angibt, ob es sich bei der Eingabeliste um die leere Liste handelte oder nicht. Wir wollen diese Testmethode *isEmpty* nennen. Sie hat folgenden Typ:

- isEmpty: `List<`α`>`→`boolean`

Das funktionale Verhalten der Testmethode lässt sich durch folgende zwei Gleichungen spezifizieren:

$$\begin{aligned} isEmpty(Nil()) &= true \\ isEmpty(Cons(x, xs)) &= false \end{aligned}$$

Somit ist alles spezifiziert, was eine Listenstruktur ausmacht. Listen können konstruiert werden, die Bestandteile einer Liste wieder einzeln selektiert und Listen können nach der Art ihrer Konstruktion unterschieden werden.

1.4 Listenalgorithmen

Allein diese fünf Funktionen beschreiben den ADT der Listen. Wir können aufgrund dieser Spezifikation Algorithmen für Listen schreiben.

1.4.1 Länge

Es lässt sich durch zwei Gleichungen spezifizieren, was die Länge einer Liste ist:

$$\begin{aligned} length(Nil()) &= 0 \\ length(Cons(x, xs)) &= 1 + length(xs) \end{aligned}$$

Mit Hilfe dieser Gleichungen lässt sich jetzt schrittweise die Berechnung einer Listenlänge auf Listen durchführen. Hierzu benutzen wir die Gleichungen als Ersetzungsregeln. Wenn ein Unterausdruck in der Form der linken Seite einer Gleichung gefunden wird, so kann diese durch die entsprechende rechte Seite ersetzt werden. Man spricht bei so einem Ersetzungsschritt von einem Reduktionsschritt.

Beispiel 1.2 Wir errechnen in diesem Beispiel die Länge einer Liste, indem wir die obigen Gleichungen zum Reduzieren auf die Liste anwenden:

$$\begin{aligned} & length(\texttt{Cons}(a, \texttt{Cons}(b, \texttt{Cons}(c, \texttt{Nil}())))) \\ \rightarrow\ & 1 + length(\texttt{Cons}(b, \texttt{Cons}(c, \texttt{Nil}()))) \\ \rightarrow\ & 1 + (1 + length(\texttt{Cons}(c, \texttt{Nil}()))) \\ \rightarrow\ & 1 + (1 + (1 + length(\texttt{Nil}()))) \\ \rightarrow\ & 1 + (1 + (1 + 0)) \\ \rightarrow\ & 1 + (1 + 1) \\ \rightarrow\ & 1 + 2 \\ \rightarrow\ & 3 \end{aligned}$$

1.4.2 Letztes Listenelement

Wir können mit einfachen Gleichungen spezifizieren, was wir unter dem letzten Element einer Liste verstehen.

$$\begin{aligned} last(Cons(x, Nil())) &= x \\ last(Cons(x, xs))) &= last(xs) \end{aligned}$$

Beispiel 1.3 Auch die Funktion `last` können wir von Hand auf einer Beispielliste einmal per Reduktion ausprobieren:

$$\begin{aligned} & last(\texttt{Cons}(a, \texttt{Cons}(b, \texttt{Cons}(c, \texttt{Nil}())))) \\ \rightarrow\ & last(\texttt{Cons}(b, \texttt{Cons}(c, \texttt{Nil}()))) \\ \rightarrow\ & last(\texttt{Cons}(c, \texttt{Nil}())) \\ \rightarrow\ & c \end{aligned}$$

1.4.3 Listenkonkatenation

Die folgenden Gleichungen spezifizieren, wie zwei Listen aneinandergehängt werden:

$$\begin{aligned} concat(Nil(), ys) &= ys \\ concat(Cons(x, xs), ys) &= Cons(x, concat(xs, ys)) \end{aligned}$$

Beispiel 1.4 Auch diese Funktion lässt sich beispielhaft mit der Reduktion einmal durchrechnen:

$$\begin{aligned} & concat(\texttt{Cons}(i, \texttt{Cons}(j, \texttt{Nil}())), \texttt{Cons}(a, \texttt{Cons}(b, \texttt{Cons}(c, \texttt{Nil}())))) \\ \rightarrow\ & \texttt{Cons}(i, concat(\texttt{Cons}(j, \texttt{Nil}()), \texttt{Cons}(a, \texttt{Cons}(b, \texttt{Cons}(c, \texttt{Nil}()))))) \\ \rightarrow\ & \texttt{Cons}(i, \texttt{Cons}(j, concat(\texttt{Nil}(), \texttt{Cons}(a, \texttt{Cons}(b, \texttt{Cons}(c, \texttt{Nil}())))))) \\ \rightarrow\ & \texttt{Cons}(i, \texttt{Cons}(j, \texttt{Cons}(a, \texttt{Cons}(b, \texttt{Cons}(c, \texttt{Nil}()))))) \end{aligned}$$

1.5 Schachtel- und Zeiger-Darstellung

Listen lassen sich auch sehr schön graphisch visualisieren. Hierzu wird jede Liste durch eine Schachtel mit zwei Feldern dargestellt. Von diesen beiden Feldern gehen Pfeile aus. Der erste Pfeil zeigt auf das erste Element der Liste, dem `head`, der zweite Pfeil zeigt auf die Schachtel, die für den Restliste steht dem *tail*. Wenn eine Liste leer ist, so gehen keine Pfeile von der Schachtel aus, die sie repräsentiert.

Die Liste `Cons(a,Cons(b,Cons(c,Nil())))` hat somit die Schachtel- und Zeiger- Darstellung aus Abbildung 1.

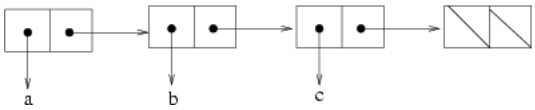

Abbildung 1: Schachtel Zeiger Darstellung einer dreielementigen Liste.

In der Schachtel- und Zeiger-Darstellung lässt sich sehr gut verfolgen, wie bestimmte Algorithmen auf Listen dynamisch arbeiten. Wir können die schrittweise Reduktion der Methode `concat` in der Schachtel- und Zeiger-Darstellung gut nachvollziehen:

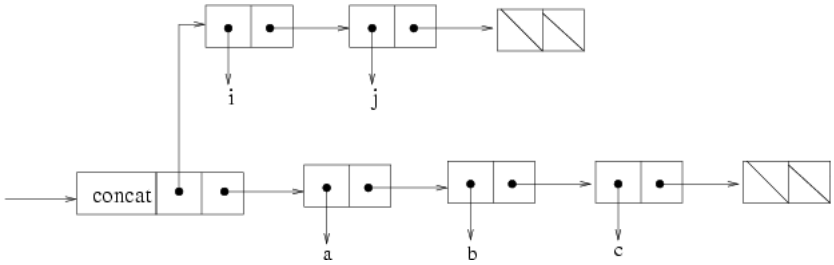

Abbildung 2: Schachtel Zeiger Darstellung der Funktionsanwendung von concat auf zwei Listen.

Abbildung 2 zeigt die Ausgangssituation. Zwei Listen sind dargestellt. Von einer Schachtel, die wir als die Schachtel der Funktionsanwendung von `concat` markiert haben, gehen zwei Zeiger aus. Der erste auf das erste Argument, der zweite auf das zweite Argument der Funktionsanwendung.

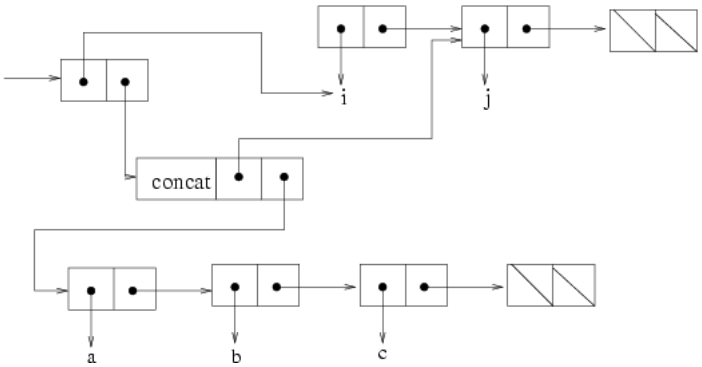

Abbildung 3: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

Abbildung 3 zeigt die Situation, nachdem die Funktion `concat` einmal reduziert wurde. Ein neuer Listenknoten wurde erzeugt. Dieser zeigt auf das erste Element der ursprünglich ersten Argumentliste. Der zweite zeigt auf den rekursiven Aufruf der Funktion `concat`, diesmal mit der Restliste des ursprünglich ersten Arguments.

Abbildung 4 zeigt die Situation nach dem zweiten Reduktionsschritt. Ein weiterer neuer Listenknoten ist entstanden und ein neuer Knoten für den rekursiven Aufruf ist entstanden.

Abbildung 5 zeigt die endgültige Situation. Der letzte rekursive Aufruf von `concat` hatte als erstes Argument eine leere Liste. Deshalb wurde kein neuer Listenknoten erzeugt, sondern lediglich der Knoten für die Funktionsanwendung gelöscht. Man beachte, dass die beiden ursprünglichen Listen noch vollständig erhalten sind. Sie wurden nicht gelöscht. Die erste Argumentliste wurde quasi kopiert. Die zweite Argumentliste teilen sich gewisser Maßen die neue Ergebnisliste der Funktionsanwendung und die zweite ursprüngliche Argumentliste.

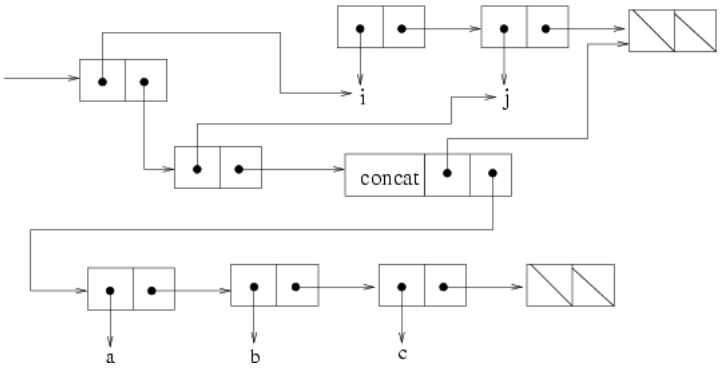

Abbildung 4: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

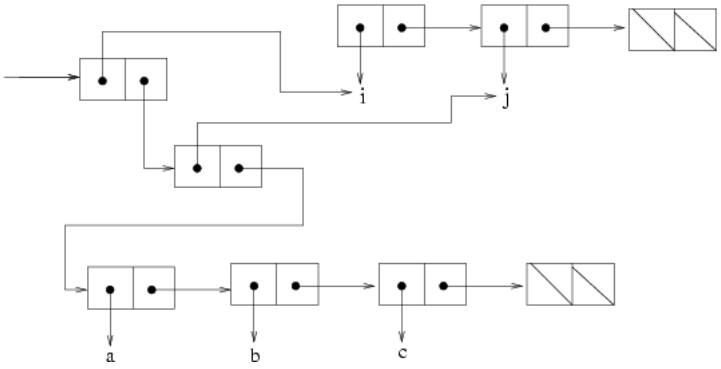

Abbildung 5: Schachtel Zeiger Darstellung des Ergebnisses nach der Reduktion.

1.6 Listen in Lisp

Die hier spezifizierten Listen sind in dieser Weise in der Programmiersprache Lisp als fundamentale ursprünglich fast einzige strukturierte Form von Daten umgesetzt. Die beiden Konstruktorfunktionen heißen in List ebenso wie bei uns *cons* und *nil*. Die Selektorfunktionen *head* und *tail* heißen in Lisp *car* bzzw. *cdr*. Diese beiden Namen gehen ursprünglich auf die Maschinenregister zurück, in denen die beiden Zeiger gespeichert wurden. Die Testfunktion, die wir als *isEmpty* bezeichnet haben, heißt in Lisp *null*.

Die leere Liste *nil* repräsentiert in Lisp auch den Wahrheitswert *false*.

Wer den Texteditor *emacs* verwendet, kann in diesem direkt einmal die Lisplisten ausprobieren, denn *emacs* ist nicht nur in Lisp geschrieben, sondern enthält auch einen integrierten Lisp-Interpreter, der mit der Tastenkombination `M-x ielm` gestartet werden kann.

Dann kann man Ausdrücke eingeben, deren Auswertungsergebnis dann direkt angezeigt wird. So wie wir es aus der JShell kennen.

Eine syntaktische Besonderheit in Lisp ist, dass Funktionsaufrufe statt `f(x,y)` als `(f x y)` notiert werden.

Öffnen wir also den Lisp-Interpreter des Emacs und tippen eine erste Liste ein.

Shell

```
*** Welcome to IELM ***  Type (describe-mode) for help.
ELISP> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
```

Mit der Funktion `null` lässt sich testen, ob eine Liste leer ist.

Shell

```
ELISP> (null nil)
t
ELISP> (null (cons 1 nil))
nil
```

Mit `car` und `cdr` lässt sich jeweils auf erstes Listenelement und die Restliste zugreifen.

Shell

```
ELISP> (car (cons 1 (cons 2 (cons 3 nil))))
1
ELISP> (cdr (cons 1 (cons 2 (cons 3 nil))))
(2 3)
```

Die von uns spezifizierten Funktionen *last* und *length* sind auch in Lisp implementiert.

Shell

```
ELISP> (last (cons 1 (cons 2 (cons 3 nil))))
(3)

ELISP> (length (cons 1 (cons 2 (cons 3 nil))))
3
```

Auch die Funktion zum Aneinanderhängen von Listen ist in Lisp vordefiniert und heißt dort *append*.

Shell

```
ELISP> (append (cons 1 (cons 2 (cons 3 nil))) (cons 4 (cons 5 6 nil)))
(1 2 3 4 5)
```

Etwas gewöhnungsbedürftig ist in Lisp die Definition von Funktionen. Als Beispiel hier die Funktion *drop*, die von einer Liste die ersten n Elemente absplittet.

Shell

```
ELISP> (defun drop (xs n)
   (if (null xs) xs
     (if (= n 0) xs
```

```
        (drop (cdr xs) (- n 1))
        )
      )
    )
drop
ELISP> (drop (cons 1(cons 2(cons 3(cons 4 nil))))  2)
(3 4)
```

2 Einfach verkettete Liste als *sealed* interface

Wir haben bereits in der Vorlesung einfach verkettete Listen betrachtet und eine kleine ad hoc Implementierung umgesetzt. In dieser Aufgabe wollen wir einfach verkettete Listen implementieren und dabei einen möglichst funktionalen Weg beschreiten.

Zunächst einmal die Imports unserer Umsetzung. Wir wollen eine Reihe von Funktionen höherer Ordnung implementieren, sodass wir uns ein paar der wichtigen funktionalen Schnittstellen implementieren.

LL.java

```
package name.panitz.util;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.Consumer;
import java.util.Comparator;

import java.util.NoSuchElementException;
```

Die eigentliche Implementierung ist die generische Schnittstelle `LL`. Sie ist generisch gehalten über die Elemente der Liste. `LL` steht dabei für *Linked List*.

Wir versiegeln die Schnittstelle mit dem neuen Schlüsselwort `sealed` und erlauben nur zwei Klassen, die die Schnittstelle implementieren.

LL.java

```
public sealed interface LL<E> permits LL.Nil, LL.Cons{
```

2.1 Implementierungen als Record-Klassen

Die beiden Klassen, die als einzige die Schnittstelle `LL` implementieren, sind die Klassen `Nil` für leere Listen und `Cons` für Listen, die aus einem Kopfelemente, dem ersten Elementen, und einer Restliste, also der Liste bis auf das erste Element gebildet werden.

Beide Klassen werden als innerer Record-Klassen umgesetzt.

Die Klasse `Nil` hat keine Felder und somit den leeren Konstruktor:

LL.java

```
record Nil<E>() implements LL<E>{
  @Override public String toString(){return "[]";}
}
```

Die Klasse `Cons` bekommt ein Kopfelement (eng.: *head*) und die Restliste (eng.: *tail*) als Argumente.

LL.java

```
record Cons<E>(E hd,LL<E> tl) implements LL<E>{
  @Override public String toString(){
    return show(true,new StringBuffer("["));
  }
}
```

Die Wahl, Record-Klassen zu verwenden, hat mehrere Konsequenzen. Zum einen bekommen wir die Konstruktoren und Implementierungen von *equals* und *toString* geschenkt. Zum anderen sind Objekte von Record-Klassen unveränderlich. Einmal erzeugt, behalten die Felder `hd` und `tl` der Objekte des Typs `Cons`, immer dieselben Objekte.

Für fast alle Algorithmen, die wir rekursiv umsetzen werden, ist eine wichtige Unterscheidung, ob es sich um eine leere Liste oder um eine Liste mit einem Kopfelement handelt. Hierfür sei eine Testfunktion vorgesehen.

LL.java

```
default boolean isEmpty(){return this instanceof Nil;}
```

2.2 Selektoren

Um Listen wieder auseinanderzusnehmen, muss man in der Lage sein, wieder auf Kopfelement und Restelement zuzugreifen. Diese gibt es nur für Cons-Liste, also nichtleeren Listen. Trotzdem implementieren wir Selektorfunktionen für alle Listen, auch für leere Listen. Allerdings wird eine Ausnahme bei Anwendung einer leeren Liste geworfen.

Einmal die Selektion auf das Kopfelement:

LL.java

```
default E head(){
  if (this instanceof Cons<E> c) return c.hd();
  throw new NoSuchElementException("head on empty list");
}
```

Und die Selektorfunktion für die Restliste:

LL.java

```
default LL<E> tail(){
  if (this instanceof Cons<E> c) return c.tl();
  throw new NoSuchElementException("tail on empty list");
}
```

Zusammen mit `isEmpty()` können mit diesen Selektorfunktionen am einfachsten rekursive Funktionen auf den Listen realisiert werden.

2.3 Konstruktorfunktionen

Da wir sehr oft einen Konstruktor aufrufen werden, werden wir dieses nicht direkt machen, sondern sehen hierfür zwei kleine statische Hilfsfunktionen vor. Damit wird das Programm lesbarer. Zusätzlich haben wir die Chance, für die leere Liste nur ein einziges Objekt zu erzeugen, dass gemeinsam für alle leere Listen verwendet wird.

Im Prinzip könnte der Compiler diese Optimierung für die Klasse `Nil` eigenständig machen, aber kurze Tests mit der JShell zeigen, dass dieses nicht geschieht.

LL.java

```
@SuppressWarnings("rawtypes")
final LL nil = new Nil<>();
@SuppressWarnings("unchecked")
static <E>LL<E> nil(){return nil;}
```

Die kleine Konstruktormethode für die Record-Klasse `Cons` ruft direkt deren Konstruktor auf:

LL.java

```
static <E>LL<E> cons(E hd,LL<E> tl){return new Cons<>(hd,tl);}
```

Als besonderen Komfort ist noch eine Funktion zur Erzeugung einer Liste aus der direkten Aufzählung der Elemente zu erzeugen.

LL.java

```
@SafeVarargs
static <E>LL<E> of(E...es){
  LL<E> r = nil();
  for (int i=es.length-1;i>=0;i--) r = cons(es[i],r);
  return r;
}
```

2.4 Stringdarstellung

Record-Klasse kommen zwar schon mit einer recht guten Methode `toString` daher, trotzdem haben wir für die beiden Listen-Recordklassen eigene `toString`-Methoden vorgesehen. Die eigentliche Implementierung folgt hier als die default-Methode `show`. Dieser Umweg ist notwendig, weil in Schnittstellen `toString` nicht als default-Methode umgesetzt werden kann.

Die Methode `show` listet die Listenelemente in eckigen Klammern mit Komma separiert auf.

LL.java

```
default String show(boolean first,StringBuffer result){
  if (isEmpty()) {
    result.append("]");
    return result.toString();
  }
  if (!first)result.append(", ");
  result.append(head());
  return tail().show(false,result);
}
```

2.5 Aufrufe auf der JShell

Einfache Testaufrufe lassen sich für diese Listenimplementierung in der JShell durchführen. Hierzu ist die Klasse mit dem Javacompiler zu übersetzen und die class-Dateien entsprechend der Paketstruktur in einem Ordner zu generieren:

Shell

```
javac -d classes files/solution/LL.java
Note: files/solution/LL.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Beim Übersetzen ist mindestens die Java-Version 15 mit angeschalteten Preview zu verwenden oder eine höhere Version von Java. Die Option `-d classes` sorgt dafür, dass in dem Ordner `classes` die generierten Klassen entsprechend der Paketstruktur gespeichert werden.

Jetzt kann man eine JShell-Session öffnen:

Shell

```
jshell --class-path classes
|  Welcome to JShell -- Version 20.0.1
|  For an introduction type: /help intro

jshell>
```

Hierbei ist der Ordner, in dem die generierten Klassen liegen, als Klassenpfad anzugeben.

Es empfiehlt sich, alle statischen Eigenschaften der Schnittstelle `LL` in der JShell-Session zu importieren.

```
jshell> import static name.panitz.util.LL.*;
```

Jetzt kann interaktiv mit den Listen gearbeitet werden.

Erzeugen einer leeren Liste.

```
nil()
```

```
[]
```

Erzeugen einer Liste mit `cons` und `nil`.

```
cons(1,cons(2,cons(3,nil())))
```

```
[1, 2, 3]
```

Verwendung der Konstruktorfunktion mit variabler Parameteranzahl.

```
of(1,2,3,4,5,6)
```

```
[1, 2, 3, 4, 5, 6]
```

Aufrufe der Selektoren.

```
of(1,2,3,4,5,6).tail().tail().head()
```

```
3
```

Aufruf der Testfunktion.

```
of(1,2,3,4,5,6).isEmpty()
```

```
false
```

Liste von Stringelementen.

```
of("hallo","welt")
```

```
[hallo, welt]
```

3 Aufgaben

Es folgen viele kleine Aufgaben. Alle zu schreibenden Funktionen lassen sich am naheliegendsten rekursiv umsetzen und fast alle kommen mit drei Zeilen im Funktionsrumpf aus. Aber es dürfen natürlich auch längere Funktionen sein.

Die Funktionen können sich natürlich gegenseitig aufrufen. Dieses ist vielfach gewünscht.

Auf keinem Fall sollten Sie allerdings die Funktionen `get` und `length` in anderen Funktionen aufrufen.

Alle Funktionen können direkt in der JShell kurz getestet werden. Ein Beispielaufruf ist jeweils bei der Aufgabenstellung mit angegeben.

Aufgabe 1

a) Schreiben Sie die Funktion, die die Länge der Liste berechnet.

LL.java

```
default int length(){
  return 0; /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,7,8,9,10).length()

$2 ==> 10
```

b) Schreiben Sie eine Funktion, die das letzte Element der Liste zurück gibt. Ist die Liste leer, so soll eine `NoSuchElementException` geworfen werden.

LL.java

```
default E last(){
  return null;   /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,7,8,9,10).last()

10
```

c) Schreiben Sie die Funktion, die eine neue Liste erstellt, die mit den Elementen der this-Liste beginnt und anschließend die Elemente der that-Liste hat. Es handelt sich dabei um die Funktion, die wir im ersten Abschnitt als `concat` spezifiziert haben.

LL.java

```
default LL<E> append(LL<E> that){
  return nil();  /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,7,8,9,10).append(of(431,432,431,21,76))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 431, 432, 431, 21, 76]
```

d) Schreiben Sie die Funktion, die eine Teilliste erzeugt, die ohne die ersten i-Elemente aus der this-Liste entsteht. Ist i größer als die Länge, dann sei das Ergebnis die leere Liste. Ist i negativ, so sei das Ergebnis die gesamte Liste.

Folgende Gleichungen sollen erfüllt sein:

$$\begin{aligned} drop(Nil(), n) &= Nil() \\ drop(xs, 0) &= xs \\ drop(Cons(x, xs), (n+1)) &= drop(xs, n) \end{aligned}$$

LL.java

```
default LL<E> drop(int i){
  return nil();   /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,7,8,9,10).drop(6)
```

```
[7, 8, 9, 10]
```

e) Schreiben Sie die Funktion, die eine Teilliste erzeugt, die aus den ersten i-Elemente aus der this-Liste entsteht. Ist i größer als die Länge, dann sei das Ergebnis die komplette Liste. Ist i negativ, dann sei das Ergebnis die leere Liste.

Folgende Gleichungen sollen erfüllt sein:

$$\begin{aligned} take(Nil(), n) &= Nil() \\ take(xs, 0) &= Nil() \\ take(Cons(x, xs), (n+1)) &= Cons(x, take(xs, n)) \end{aligned}$$

LL.java

```
default LL<E> take(int i){
  return nil();    /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,7,8,9,10).take(6)
```

```
[1, 2, 3, 4, 5, 6]
```

f) Schreiben Sie die Funktion, die eine Teilliste erzeugt, die mit dem Element an dem angegebenen Index startet und die angegebene Länge hat. Sollten nicht genug Elemente in der Liste sein, so wird die maximale Anzahl der von dem Index an kommenden Elemente genommen.

LL.java

```
default LL<E> sublist(int from, int length) {
  return nil();  /*ToDo*/
}
```

Ein Beispielaufrufe:

```
of(1,2,3,4,5,6,7,8,9,10).sublist(3,4)
```

```
[4, 5, 6, 7]
```

```
of(1,2,3,4,5,6).sublist(3,100000)
```

```
[4, 5, 6]
```

```
of(1,2,3,4,5,6).sublist(10,100000)
```

```
[]
```

g) Schreiben eine Funktion, die eine Liste mit den Elementen in umgekehrter Reihenfolge erzeugt.

LL.java

```
default LL<E> reverse(){
  return nil();   /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,7,8,9,10).reverse()
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

h) Schreiben Sie eine Funktion, die eine neue Liste erzeugt, in der zwischen den Elementen der this-Liste das übergebene Element steht.

LL.java

```
default LL<E> intersperse(E e){
  return nil();      /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6).intersperse(42)
```

```
[1, 42, 2, 42, 3, 42, 4, 42, 5, 42, 6]
```

i) Schreiben Sie eine Funktion, die prüft, ob die this-Liste der Anfang der that-Liste ist.

LL.java

```
default boolean isPrefixOf(LL<E> that){
  return false;     /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4).isPrefixOf(of(1,2,3,4,5,6,7,8,9,10))
```

```
true
```

Beachten Sie, dass die leere Liste ein Präfix von jeder Liste ist.

j) Schreiben Sie eine Funktion, die prüft, ob die this-Liste das Ende der that-Liste ist.

LL.java

```
default boolean isSuffixOf(LL<E> that){
  return false;     /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(8,9,10).isSuffixOf(of(1,2,3,4,5,6,7,8,9,10))
```

```
true
```

k) Schreiben Sie eine Funktion, die prüft, ob die this-Liste in der that-Liste enthalten ist.

LL.java

```
default boolean isInfixOf(LL<E> that){
  return false;     /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(6,7,8,9).isInfixOf(of(1,2,3,4,5,6,7,8,9,10))
```

```
true
```

l) Schreiben Sie eine Funktion, die das Element am Index i zurück gibt. Bei einem illegalen Index wird eine `IndexOutOfBoundsException` geworfen.

LL.java

```
default E get(int i){
  /*ToDo*/
  throw new IndexOutOfBoundsException();
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,7,8,9,10).get(5)
```

```
6
```

m) Schreiben Sie eine Funktion, die eine neue Liste erzeugt, bei der das ursprüngliche Kopfelement an die letzte Stelle wandert.

LL.java

```
default LL<E> rotate(){
  return nil();  /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6).rotate()
```

```
[2, 3, 4, 5, 6, 1]
```

n) Schreiben Sie eine Funktion, die die Liste aller Listen erzeugt, mit der die this-Liste endet.

LL.java

```
default LL<LL<E>> tails(){
  return of(nil());  /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4).tails()
```

```
[[1, 2, 3, 4], [2, 3, 4], [3, 4], [4], []]
```

Aufgabe 2 Als nächstes sind einige Funktionen höherer Ordnung zu entwickeln. Eine Funktion wird als Funktion höherer Ordnung bezeichnet, wenn sie ein Argument hat, das wiederum eine Funktion ist. In Java können zwar eigentlich nicht direkt Funktionen sondern immer nur Objekte als Parameter übergeben werden, aber die funktionalen Schnittstelle, die nur eine Methode enthalten, repräsentieren Funktionen.

a) Schreiben Sie eine Funktion, die ein Konsumentenobjekt auf jedes Element der Liste anwendet. Es soll also für alle Elemente der Liste eine bestimmte Methode aufgerufen werden.

LL.java

```
  default void forEach(Consumer<? super E> con) {
    /*ToDo*/
}
```

Ein Beispielaufruf:

```
var x = new int[1]; of(1,2,3,4,5,6,7,8,9,10).forEach(y->x[0]+=y); x
```

```
x ==> int[1] { 0 }
x ==> int[1] { 55 }
```

Etwas verwirrend ist der Argumenttyp `Consumer<? super E>`. Man lese ihn erst einmal als `Consumer<E>`. Man möchte ja für jedes Element die Methode `accept(E e)` aufrufen. Man braucht also ein Konsumerobjekt, das `E`-Elemente verarbeiten kann. Stellen wir uns ein Objekt des Typs `Consumer<Object>` vor. Dieses hat dann die Methode `accept(Object e)`. Diese kann natürlich auch mit einem Objekt des Typs `E` aufgerufen werden, was immer tatsächlich sich hinter `E` für ein konkreter Typ verbirgt.

Wir benötigen also gar nicht unbedingt einen Konsumenten, der nur `E`-Objekte in der Methode `accept` als Argumente akzeptiert, sondern mindestens `E`-Objekte, vielleicht aber auch mehr Objekte in Form einer Oberklasse von `E`.

Genau das drückt der Typ `Consumer<? super E>` aus. Ich benötige einen Consumer, der irgendeinen unbekannten Typ akzeptiert (das Fragezeichen), es muss nur gewährleistet sein, dass dieser Typ eine Oberklasse von `E` ist, also alle Objekte des Typs `E` mit einschließt.

b) Schreiben Sie eine Funktion, die genau dann wahr ergibt, wenn mindestens ein Element der Liste das übergebene Prädikat erfüllt.

LL.java

```
default boolean containsWith(Predicate< ? super E> p) {
  return false;  /*ToDo*/
}
```

Ein paar Beispielaufrufe:

```
of(1,2,3,4,5,6,7,8,9,10).containsWith(x->x>7 && x<10)
```

```
true
```

Und ein Aufruf, der `false` ergibt:

```
of(1,2,3,4,5,6,7,8,9,10).containsWith(x->x>7 && x%6==0)
```

```
false
```

c) Schreiben Sie jetzt eine Funktion, die testet, ob ein dem Argument gleiches Element in der Liste enthalten ist.

LL.java

```
default boolean contains(E el) {
  return false;  /*ToDo*/
}
```

Beispielaufrufe:

```
of(1,2,3,4,5,6,7,8,9,10).contains(6)
```

```
true
```

Und ein Aufruf, der `false` ergibt:

```
of(1,2,3,4,5,6,7,8,9,10).contains(42)
```

```
false
```

d) Schreiben Sie jetzt eine Funktion, die von der Liste alle Elemente vorne absplittet, für die das übergebene Prädikat wahr ist.

LL.java

```
default LL<E> dropWhile(Predicate< ? super E> p){
  return nil();  /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,7,8,9,10).dropWhile(x->x<=5)
```

```
[6, 7, 8, 9, 10]
```

e) In der nächsten Funktionen soll eine Liste aus der Liste erzeugt werden, solange die Elemente noch das Prädikat erfüllen.

LL.java

```
default LL<E> takeWhile(Predicate< ? super E> p){
  return nil();   /*ToDo*/
}
```

Ein Beispielaufruf: of(1,2,3,4,5,6,7,8,9,10).takeWhile(x-> x<8) [1, 2, 3, 4, 5, 6, 7]

f) Schreiben Sie eine Funktion, die eine Liste aus allen Listenelementen erzeugt, für die das übergebene Prädikat erfüllt ist.

LL.java

```
default LL<E> filter(Predicate<? super E> p){
  return nil();   /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,7,8,9,10).filter(x->x%2==0)
```

```
[2, 4, 6, 8, 10]
```

g) Schreiben Sie eine Funktion, die eine Liste durch Anwendung der übergebenen Funktion auf jedes Listenelement erzeugt.

LL.java

```
default <R> LL<R> map(Function<? super E, ? extends R> f){
  return nil();    /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,7,8,9,10).map(x->x*x)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Auch hier lohnt sich ein genauer Blick auf den Typ des Funktionsobjekts:
`Function<? super E, ? extends R>`

Die funktionale Schnittstelle `Function` hat zwei Typparameter. Der ersten für den Argumenttyp der Funktion, der zweite für den Ergebnistyp. Für das Argument ist der Typ `? super E`. Also ein Typ, der `E` Objekte mit einschließt.

Neu ist für uns die Notation für den Ergebnistyp: `? extends R`. Das Ergebnis soll irgendein unbekannter Typ sein, der eine Unterklasse vom Typ `R` ist. Wenn ich eine Funktion benötige, die Objekte erzeugt, bin ich vollkommen zufrieden mit einer Funktion die Stringobjekte erzeugt, denn jedes Objekt der Klasse `String` ist ja auch ein Objekt der Klasse `Object`.

Aufgabe 3 In dieser Aufgabe sollen Funktionen geschrieben werden, die auch mit Tupelobjekte arbeiten.

Hierzu sei zunächst die Record-Klasse für Paare von Objekten definiert.

LL.java

```
record Pair<A,B>(A fst,B snd){
  @Override public String toString(){return "("+fst()+", "+snd()+")";}
}
```

a) Schreiben Sie die Funktion, die die Elemente der this-Liste paarweise mit den Elementen der that-Liste zu einer Liste aus Paaren verknüpft.

LL.java

```
default <B> LL<Pair<E,B>> zip(LL<B> that){
  return nil();   /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6).zip(of("A","B","C","D"))
```

```
[(1, A), (2, B), (3, C), (4, D)]
```

b) Schreiben Sie die Funktion, die ein Paar aus zwei Listen erzeugt. Die erste Liste soll der Präfix der Elemente sein, die das Prädikat erfüllen, die zweite Liste, die Teilliste vom ersten Element an, das das Prädikat nicht erfüllt.

LL.java

```
default Pair<LL<E>,LL<E>> span(Predicate<? super E> p){
  return new Pair<>(nil(),nil());   /*ToDo*/
}
```

Ein Beispielaufruf: of(1,2,3,4,5,6,1,2,3).span(x->x<4) ([1, 2, 3], [4, 5, 6, 1, 2, 3])

c) Schreiben Sie die Funktion, die ein Paar aus zwei Listen erzeugt. Das erste Element des Paares soll die Liste aller Elemente sein, die das Prädikat erfüllen, die zweite Liste die Liste aller Elemente, die es nicht erfüllen.

LL.java

```
default Pair<LL<E>,LL<E>> partition(Predicate<? super E> p){
  return new Pair<>(nil(),nil());   /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,6,1,2,3).partition(x-> x<5)

([1, 2, 3, 4, 1, 2, 3], [5, 6])
```

Aufgabe 4 In dieser Aufgabe geht es um Sortierungen.

a) Schreiben Sie die Funktion, die prüft, ob die Liste aufsteigend sortiert ist. Die Sortiereigenschaft wird dabei als Comparator-Objekt übergeben. `Comparator` ist eine funktionale Standardschnittstelle mit der abstrakten Methode `compare`, die zwei Elemente vergleicht. Ist das erste Argument kleiner als das zweite, so ist das Ergebnis eine negative Zahl, ist es größer, dann eine positive Zahl. Sind beide Elemente in der Ordnung gleich, so ist das Ergebnis die Zahl 0.

LL.java

```
default boolean isSorted(Comparator<? super E> cmp){
  return false;   /*ToDo*/
}
```

Ein Beispielaufruf:

```
of(1,2,3,4,5,435,4,2345,33,2,3,2).isSorted((x,y)->x-y)

false
```

Ein Beispiel für eine sortierte Liste:

```
of(1,2,3,4,5).isSorted((x,y)->x-y)
```

```
true
```

b) Sie werden in diesem Semester im Modul ›Algorithmen und Datenstrukturen‹ unterschiedliche Algorithmen zum Sortieren kennenlernen. Deren Laufzeitverhalten werden Sie dort analysieren. Dort werden Sie zunächst die Elemente auf Arrays sortieren. Für unsere Listen bietet sich an, den sogenannten *quicksort*-Algorithmus zu implementieren.

Für die leere Liste sei das Ergebnis die leere Liste.

Ansonsten: Partitionieren Sie die tail-Liste, nach den Elementen, die im übergebenen Comparator kleiner sind als das head-Element.

Sie erhalten zwei Teillisten, die der Elemente, die kleiner sind als das erste Element und die der Elemente, die größer oder gleich sind zum ersten Element.

Sortieren Sie rekursiv die beiden Ergebnislisten der Partitionierung.

Erzeugen Sie eine Ergebnisliste, in dem sie die Sortierung der kleineren mit der Sortierung der größeren Elemente zusammenhängen und dazwischen noch das head-Element fügen.

Schreiben Sie die Funktion, die die Liste mit Hilfe des Quicksort-Algorithmus sortiert.

Auch dieses lässt sich in drei Zeilen realisieren.

LL.java

```
  default LL<E> qsort(Comparator<? super E> cmp){
    return nil();     /*ToDo*/
  }
```

Ein Beispielaufruf:

```
of(1,2,4,5,435,4,2345,33,3,453,423,22,0).qsort((x,y)->x-y)
```

```
[0, 1, 2, 3, 4, 4, 5, 22, 33, 423, 435, 453, 2345]
```

LL.java

```
}
```

Konstruktoren: nil, cons
Selektoren: head, tail
Funktionen höherer Ordnung