TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 04b:
Funktionales Programmieren: Listen

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegendes zu Listen

In Racket gibt es zwar ebenfalls Arrays wie in Java. Aber in eigentlich allen deklarativen Sprache sind *Listen* das Arbeitspferd. Daher ist Listen hier ein ganzes Kapitel gewidmet, 04b.

Grundlegendes zu Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define list1 ( list 2 5 0 -4 3 5 ) )
( define list2 ( cons 7 list1 ) )
( define list3 ( cons -3 list2 ) )
( define list4 ( cons 2 list3 ) )

→ ( list 2 -3 7 2 5 0 -4 3 5 )

( define list5 ( cons 2 empty ) )
```

In Racket sind Listen ein eingebautes und sehr einfach zu verwendendes Konstrukt. Auf dieser Folie sehen wir schon alles, war wir brauchen, um Listen entweder als Ganzes oder schrittweise einzurichten.

***Vorgriff*: In Kapitel 07 werden wir sehen, dass es auch Listen in Java gibt, allerdings nicht als eingebautes Konstrukt wie Arrays, sondern in Form von Interfaces und Klassen in der Standardbibliothek.**

Grundlegendes zu Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define list1 ( list 2 5 0 -4 3 5 ) )
( define list2 ( cons 7 list1 ) )
( define list3 ( cons -3 list2 ) )
( define list4 ( cons 2 list3 ) )

→ ( list 2 -3 7 2 5 0 -4 3 5 )

( define list5 ( cons 2 empty ) )
```

Wir haben schon die Definition von Konstanten kennen gelernt: zuerst das Wort define, dann der Name der neuen Konstanten.

Grundlegendes zu Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define list1 ( list 2 5 0 -4 3 5 ) )
( define list2 ( cons 7 list1 ) )
( define list3 ( cons -3 list2 ) )
( define list4 ( cons 2 list3 ) )

→ ( list 2 -3 7 2 5 0 -4 3 5 )

( define list5 ( cons 2 empty ) )
```

Die Funktion list kann beliebig viele Parameter haben. Sie richtet eine Liste aus ihren Parametern ein, das ist ihr Rückgabewert. Die Konstante list1 ist also eine Liste, die aus diesen sechs Zahlen in dieser Reihenfolge besteht.

Mathematisch gesprochen, ist eine Liste also eine endliche geordnete Sequenz aus irgendwelchen Elementen, die natürlich nicht nur Zahlen sein müssen. Listen aus anderen Typen von Elementen sehen wir bald.

Grundlegendes zu Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define list1 ( list 2 5 0 -4 3 5 ) )
( define list2 ( cons 7 list1 ) )
( define list3 ( cons -3 list2 ) )
( define list4 ( cons 2 list3 ) )

→ ( list 2 -3 7 2 5 0 -4 3 5 )

( define list5 ( cons 2 empty ) )
```

Mit der Funktion cons kann man aus einer Liste eine neue Liste bauen. Die Funktion cons hat zwei Parameter, und der zweite Parameter muss zwingend eine Liste sein, sonst beendet DrRacket die Ausführung mit einer Fehlermeldung. Das Ergebnis von cons ist eine Liste, deren erstes Element der erste Parameter von cons ist, und die weiteren Elemente der Ergebnisliste sind genau die Elemente im zweiten Parameter, und zwar in derselben Reihenfolge.

Grundlegendes zu Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define list1 ( list 2 5 0 -4 3 5 ) )
( define list2 ( cons 7 list1 ) )
( define list3 ( cons -3 list2 ) )
( define list4 ( cons 2 list3 ) )

→ ( list 2 -3 7 2 5 0 -4 3 5 )

( define list5 ( cons 2 empty ) )
```

Das Ergebnis der ersten vier Zeilen, das in list4 zusammengesammelt ist, ist also identisch mit dem, was wir erhalten hätten, wenn wir die drei neuen Elemente von vornherein der Funktion list mit übergeben hätten, in umgekehrter Reihenfolge natürlich.

Grundlegendes zu Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define list1 ( list 2 5 0 -4 3 5 ) )
( define list2 ( cons 7 list1 ) )
( define list3 ( cons -3 list2 ) )
( define list4 ( cons 2 list3 ) )

→ ( list 2 -3 7 2 5 0 -4 3 5 )

( define list5 ( cons 2 empty ) )
```

Wie gesagt, braucht cons immer zwei Parameter, und der zweite Parameter muss eine Liste sein. Man kann auch eine einelementige Liste mit cons einrichten, indem man die vordefinierte Konstante empty verwendet. Die Konstante empty ist ein Name für die leere Liste. In dieser Zeile wird list5 also eine Liste zugewiesen, die aus der 2 und *keinen* weiteren Elementen gebildet ist.

Grundlegendes zu Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define list1 ( list 2 5 0 -4 3 5 ) )

( first list1 )  →  2
( rest list1 )   →  ( 5 0 -4 3 5 )
```

Diese zwei nützlichen Funktionen werden wir oft benutzen. Beide haben genau einen Parameter und beide erwarten, dass der Parameter eine nichtleere Liste ist. Wie die Namen schon andeuten, liefert die Funktion first das erste Element der Liste zurück, und die Funktion rest liefert eine Liste zurück, die alle Elemente des Parameters *außer* dem ersten enthält, und zwar in derselben Reihenfolge wie im Parameter.

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursive Funktionen auf Listen

Die Verarbeitung von Listen in eigentlich allen deklarativen Sprachen verwendet Rekursion.

***Vorgriff*: In Kapitel 07 werden wir sehen, dass man in Java eher Schleifen für den Durchlauf durch Listen verwendet.**

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Summe einer Liste von Zahlen ist:

- 0, falls die Liste leer ist;
- erstes Element plus Summe der Restliste andernfalls.

```
( if ( empty? lst )
     0
     ( + ( first lst ) ( sum ( rest lst ) ) ) )
```

Eine erste einfache, beispielhafte Funktion auf Listen, die erwartet, dass alle Elemente der Liste Zahlen sind, und alle diese Zahlen aufsummiert. Unten steht der zentrale Teil der Racket-Implementation, oben eine Beschreibung des Ergebnisses in Deutsch.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Summe einer Liste von Zahlen ist:

- 0, falls die Liste leer ist;
- erstes Element plus Summe der Restliste andernfalls.

```
( if ( empty? lst )
     0
     ( + ( first lst ) ( sum ( rest lst ) ) ) )
```

Die Summe über eine leere Menge von Summanden ist in der Mathematik als 0 definiert. Das ist keine Willkür, sondern passt genau zu solchen rekursiven Definitionen wie hier.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Summe einer Liste von Zahlen ist:

- 0, falls die Liste leer ist;
- erstes Element plus Summe der Restliste andernfalls.

```
( if ( empty? lst )
     0
     ( + ( first lst ) ( sum ( rest lst ) ) ) )
```

Wenn die Liste hingegen *nicht* leer ist, dann kann man die Summe rekursiv definieren als die Summe aus dem ersten Element und der Summe der Elemente der Restliste.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Summe einer Liste von Zahlen ist:

- 0, falls die Liste leer ist;
- erstes Element plus Summe der Restliste andernfalls.

```
( if ( empty? lst )
     0
     ( + ( first lst ) ( sum ( rest lst ) ) ) )
```

Wie man sicher schon erahnen kann, ist auch hier die Racket-Definition praktisch eine direkte Umsetzung der obigen informellen Beschreibung, so wie schon bei den rekursiven Beispielen in Kapitel 04a.

Das ist genau das, was wir in Kapitel 04a als *deklaratives* Programmieren bezeichnet haben: Der Code beschreibt, wie das Ergebnis *definiert* ist, und nicht, wie das Ergebnis *berechnet* wird – genauso wie bei unseren ersten Beispielen in Kapitel 04a: Fakultät, Fibonacci-Folge. Binomialkoeffizient und Bisektionsverfahren.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> number
;; Returns: the sum of all list elements
( define ( sum lst )
   ( if ( empty? lst )
        0
        ( + ( first lst ) ( sum ( rest lst ) ) ) ) )

( check-expect ( sum ( list 5 1 3 2 1 ) ) 12 )
( check-expect ( sum empty ) 0 )
```

Eben haben wir den Kern gesehen, das ist jetzt die *gesamte* Definition dieser Funktion.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> number
;; Returns: the sum of all list elements
( define ( sum lst )
   ( if ( empty? lst )
        0
        ( + ( first lst ) ( sum ( rest lst ) ) ) ) )

( check-expect ( sum ( list 5 1 3 2 1 ) ) 12 )
( check-expect ( sum empty ) 0 )
```

Das ist neu: Der Parameter soll nicht eine Zahl sein, sondern eine *Liste* von Zahlen. Zur klaren Abgrenzung von anderen Bestandteilen des Funktionstyps setzen wir Typnamen, die aus mehreren Wörtern bestehen, zweckmäßigerweise in Klammern.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> number
;; Returns: the sum of all list elements
( define ( sum lst )
   ( if ( empty? lst )
        0
        ( + ( first lst ) ( sum ( rest lst ) ) ) ) )

( check-expect ( sum ( list 5 1 3 2 1 ) ) 12 )
( check-expect ( sum empty ) 0 )
```

Selbstverständlich überprüfen wir stichprobenartig das Ergebnis der neu definierten Funktion sum ...

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> number
;; Returns: the sum of all list elements
( define ( sum lst )
   ( if ( empty? lst )
        0
        ( + ( first lst ) ( sum ( rest lst ) ) ) ) )

( check-expect ( sum ( list 5 1 3 2 1 ) ) 12 )
( check-expect ( sum empty ) 0 )
```

... und vergessen dabei die Randfälle nicht. Randfall hier ist der Fall, dass die Liste leer ist.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> number
;; Returns: the sum of all list elements
( define ( sum lst )
   ( if ( empty? lst )
        0
        ( + ( first lst ) ( sum ( rest lst ) ) ) ) )

( check-expect ( sum ( list 5 1 3 2 1 ) ) 12 )
( check-expect ( sum empty ) 0 )
```

Die Funktion empty? ist vordefiniert. Als Parameter erwartet sie eine Liste. Sie liefert boolean zurück, und zwar genau dann true, wenn der Parameter eine leere Liste ist.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> number
;; Returns: the sum of all list elements
( define ( sum lst )
   ( if ( empty? lst )
        0
        ( + ( first lst ) ( sum ( rest lst ) ) ) ) )

( check-expect ( sum ( list 5 1 3 2 1 ) ) 12 )
( check-expect ( sum empty ) 0 )
```

Bei einer leeren Liste soll wie gesagt 0 herauskommen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> number
;; Returns: the sum of all list elements
( define ( sum lst )
   ( if ( empty? lst )
        0
        ( + ( first lst ) ( sum ( rest lst ) ) ) ) )

( check-expect ( sum ( list 5 1 3 2 1 ) ) 12 )
( check-expect ( sum empty ) 0 )
```

Das ist sozusagen die wörtliche Umsetzung der Beschreibung der Summe einer nichtleeren Liste als die Summe aus dem ersten Element und der Summe aus den weiteren Elementen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> number
;; Returns: the sum of all list elements
( define ( sum lst )
   ( if ( empty? lst )
        0
        ( + ( first lst ) ( sum ( rest lst ) ) ) ) )

( check-expect ( sum ( list 5 1 3 2 1 ) ) 12 )
( check-expect ( sum empty ) 0 )
```

Die Summe aus den weiteren Elementen – mit Rekursion lässt sich diese Formulierung eins-zu-eins in einen Racket-Ausdruck übersetzen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( sum ( list 5 1 3 2 1 ) )
( + 5 ( sum ( list 1 3 2 1 ) ) )
( + 5 ( + 1 ( sum ( list 3 2 1 ) ) ) )
( + 5 ( + 1 ( + 3 ( sum ( list 2 1 ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 ( sum list 1 ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 ( + 1 ( sum empty ) ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 1 ) ) ) )
( + 5 ( + 1 ( + 3 3 ) ) )
( + 5 ( + 1 6 ) )
( + 5 7 )
12
```

So kann man sich die Auswertung der Funktion sum für eine konkrete Liste vorstellen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( sum ( list 5 1 3 2 1 ) )
( + 5 ( sum ( list 1 3 2 1 ) ) )
( + 5 ( + 1 ( sum ( list 3 2 1 ) ) ) )
( + 5 ( + 1 ( + 3 ( sum ( list 2 1 ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 ( sum list 1 ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 ( + 1 ( sum empty ) ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 1 ) ) ) )
( + 5 ( + 1 ( + 3 3 ) ) )
( + 5 ( + 1 6 ) )
( + 5 7 )
12
```

Da die Liste nicht leer ist, ist ihre Summe gleich der Summe aus dem ersten Element und der Summe über die Elemente der Restliste.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( sum ( list 5 1 3 2 1 ) )
( + 5 ( sum ( list 1 3 2 1 ) ) )
( + 5 ( + 1 ( sum ( list 3 2 1 ) ) ) )
( + 5 ( + 1 ( + 3 ( sum ( list 2 1 ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 ( sum list 1 ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 ( + 1 ( sum empty ) ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 1 ) ) ) )
( + 5 ( + 1 ( + 3 3 ) ) )
( + 5 ( + 1 6 ) )
( + 5 7 )
12
```

Für diese Restliste mit vier Elementen gilt dasselbe.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( sum ( list 5 1 3 2 1 ) )
( + 5 ( sum ( list 1 3 2 1 ) ) )
( + 5 ( + 1 ( sum ( list 3 2 1 ) ) ) )
( + 5 ( + 1 ( + 3 ( sum ( list 2 1 ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 ( sum list 1 ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 ( + 1 ( sum empty ) ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 1 ) ) ) )
( + 5 ( + 1 ( + 3 3 ) ) )
( + 5 ( + 1 6 ) )
( + 5 7 )
12
```

Genauso für die Restliste mit drei Elementen und so weiter.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( sum ( list 5 1 3 2 1 ) )
( + 5 ( sum ( list 1 3 2 1 ) ) )
( + 5 ( + 1 ( sum ( list 3 2 1 ) ) ) )
( + 5 ( + 1 ( + 3 ( sum ( list 2 1 ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 ( sum list 1 ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 ( + 1 ( sum empty ) ) ) ) ) )
( + 5 ( + 1 ( + 3 ( + 2 1 ) ) ) )
( + 5 ( + 1 ( + 3 3 ) ) )
( + 5 ( + 1 6 ) )
( + 5 7 )
12
```

Nach insgesamt so vielen rekursiven Schritten, wie die Liste lang ist, ist die leere Liste erreicht, und das Ergebnis ist 0. Ab da ist das Ganze wieder ein ganz normaler arithmetischer Ausdruck, der ganz normal schrittweise ausgewertet wird.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Summe einer Liste von Zahlen ist:

- 0, falls die Liste leer ist;
- erstes Element plus Summe der Restliste andernfalls.

```
( if ( empty? lst )
     0
     ( + ( first lst ) ( sum ( rest lst ) ) ) )
```

Wir halten noch einmal den entscheidenden Aha-Effekt fest: Die Summe der Elemente einer Liste lässt sich auf sicherlich natürliche und leicht verständliche Weise rekursiv in Deutsch beschreiben, und diese Beschreibung lässt sich eins-zu-eins in eine rekursive Racket-Funktion umsetzen – deklarative Programmierung eben.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Liste der Quadratwurzeln einer Liste ist:

- **leer, falls die Liste leer ist;**
- **die Quadratwurzel des ersten Elements zusammen mit den Quadratwurzeln der Restliste andernfalls.**

```
( if ( empty? lst )
     empty
     ( cons ( sqrt ( first lst ) )
                    ( sqrts ( rest lst ) ) ) ) )
```

Ein zweites Beispiel für rekursive Funktionen auf Listen: Aus einer Liste von Zahlen soll eine Liste mit den Quadratwurzeln dieser Zahlen aufgebaut werden.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Liste der Quadratwurzeln einer Liste ist:

- leer, falls die Liste leer ist;
- die Quadratwurzel des ersten Elements zusammen mit den Quadratwurzeln der Restliste andernfalls.

```
( if ( empty? lst )
     empty
     ( cons ( sqrt ( first lst ) )
                    ( sqrts ( rest lst ) ) ) ) )
```

Wenn die Liste der Zahlen leer ist, ist natürlich auch die Liste der zugehörigen Quadratwurzeln leer.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Liste der Quadratwurzeln einer Liste ist:

- leer, falls die Liste leer ist;
- die Quadratwurzel des ersten Elements zusammen mit den Quadratwurzeln der Restliste andernfalls.

```
( if ( empty? lst )
     empty
     ( cons ( sqrt ( first lst ) )
                    ( sqrts ( rest lst ) ) ) ) )
```

Bei einer nichtleeren Liste kann man das Ergebnis wieder rekursiv beschreiben: die Quadratwurzel des ersten Elements gefolgt vom Ergebnis für die Restliste.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Liste der Quadratwurzeln einer Liste ist:

- **leer, falls die Liste leer ist;**
- **die Quadratwurzel des ersten Elements zusammen mit den Quadratwurzeln der Restliste andernfalls.**

```
( if ( empty? lst )
     empty
     ( cons ( sqrt ( first lst ) )
                    ( sqrts ( rest lst ) ) ) ) )
```

Und auch hier kann man wieder erahnen, dass die Racket-Definition diese informelle Beschreibung eins-zu-eins umsetzt – wieder deklarative Programmierung.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> ( list of number )
;; Returns: the square roots of the elements of list
( define ( sqrts lst )
   ( if ( empty? lst )
        empty
        ( cons ( sqrt ( first lst ) ) ( sqrts ( rest lst ) ) ) ) )

( check-expect ( sqrts ( list 36 9 64 49 ) ) ( list 6 3 8 7 ) )
( check-expect ( sqrts empty ) empty )
```

Das ist die gesamte Funktion.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> ( list of number )
;; Returns: the square roots of the elements of list
( define ( sqrts lst )
   ( if ( empty? lst )
        empty
        ( cons ( sqrt ( first lst ) ) ( sqrts ( rest lst ) ) ) ) )

( check-expect ( sqrts ( list 36 9 64 49 ) ) ( list 6 3 8 7 ) )
( check-expect ( sqrts empty ) empty )
```

Diesmal ist nicht nur der Parameter, sondern auch die Rückgabe eine Liste.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> ( list of number )
;; Returns: the square roots of the elements of list
( define ( sqrts lst )
   ( if ( empty? lst )
        empty
        ( cons ( sqrt ( first lst ) ) ( sqrts ( rest lst ) ) ) ) )

( check-expect ( sqrts ( list 36 9 64 49 ) ) ( list 6 3 8 7 ) )
( check-expect ( sqrts empty ) empty )
```

Für jedes Element der Eingabeliste soll die Ergebnisliste ein Element haben, und zwar an derselben Position in der Liste, und das Element der Ausgabeliste an einer bestimmten Position ist die Quadratwurzel aus dem Element in der Eingabeliste an derselben Position.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> ( list of number )
;; Returns: the square roots of the elements of list
( define ( sqrts lst )
   ( if ( empty? lst )
        empty
        ( cons ( sqrt ( first lst ) ) ( sqrts ( rest lst ) ) ) ) )

( check-expect ( sqrts ( list 36 9 64 49 ) ) ( list 6 3 8 7 ) )
( check-expect ( sqrts empty ) empty )
```

Die Checks dienen auch immer gut als Beispiele dafür, was die Funktion leisten soll. Wenn man sich nicht ganz sicher ist, wie das Returns im Vertrag oben genau zu verstehen ist, kann man hier noch einmal sein Verständnis überprüfen – vorausgesetzt, es sind tatsächlich aussagekräftige Checks definiert.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> ( list of number )
;; Returns: the square roots of the elements of list
( define ( sqrts lst )
   ( if ( empty? lst )
        empty
        ( cons ( sqrt ( first lst ) ) ( sqrts ( rest lst ) ) ) ) )

( check-expect ( sqrts ( list 36 9 64 49 ) ) ( list 6 3 8 7 ) )
( check-expect ( sqrts empty ) empty )
```

Und auch hier vergessen wir den Randfall nicht. Die Quadratwurzeln einer leeren Liste ergeben natürlich eine leere Liste.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> ( list of number )
;; Returns: the square roots of the elements of list
( define ( sqrts lst )
   ( if ( empty? lst )
        empty
        ( cons ( sqrt ( first lst ) ) ( sqrts ( rest lst ) ) ) ) )

( check-expect ( sqrts ( list 36 9 64 49 ) ) ( list 6 3 8 7 ) )
( check-expect ( sqrts empty ) empty )
```

Hier wird die leere Liste in Form der uns schon bekannten vordefinierten Konstanten empty zurückgeliefert, falls list leer ist.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) -> ( list of number )
;; Returns: the square roots of the elements of list
( define ( sqrts lst )
   ( if ( empty? lst )
        empty
        ( cons ( sqrt ( first lst ) ) ( sqrts ( rest lst ) ) ) ) )

( check-expect ( sqrts ( list 36 9 64 49 ) ) ( list 6 3 8 7 ) )
( check-expect ( sqrts empty ) empty )
```

Mit der Funktion cons lässt sich die Ergebnisliste aus der Quadratwurzel des ersten Elements und den Quadratwurzeln der weiteren Elemente zusammenbauen. Die Liste der Quadratwurzeln der weiteren Elemente wird wieder durch rekursiven Aufruf berechnet.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( sqrts ( list 36 9 64 ) )
( cons ( sqrt 36 ) ( sqrts ( list 9 64 ) ) )
( cons ( sqrt 36 ) ( cons ( sqrt 9 ) ( sqrts ( list 64 ) ) ) )
( cons ( sqrt 36 ) ( cons ( sqrt 9 )
                         ( cons ( sqrt 64 ) empty ) ) )
( cons ( sqrt 36 ) ( cons ( sqrt 9 ) ( list 8 ) ) )
( cons ( sqrt 36 ) ( list 3 8 ) )
( list 6 3 8 )
```

Die Auswertung ist völlig analog zur Funktion sum eben, nur dass die 0 durch empty und das + durch cons ersetzt ist und dass anstelle der Zahlen selbst nun ihre Quadratwurzeln betrachtet werden.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Liste der Quadratwurzeln einer Liste ist:

- **leer, falls die Liste leer ist;**
- **die Quadratwurzel des ersten Elements zusammen mit den Quadratwurzeln der Restliste andernfalls.**

```
( if ( empty? list )
     empty
     ( cons ( sqrt ( first list ) )
                    ( sqrts ( rest list ) ) ) ) )
```

Noch einmal der entscheidende Aha-Effekt: Das Ergebnis lässt sich rekursiv beschreiben, und diese Beschreibung lässt sich eins-zu-eins in eine rekursive Racket-Funktion übersetzen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Eine gefilterte Liste ist

- leer, falls die Liste leer ist;
- die Filterung der Restliste, falls das erste Element den Filter nicht passiert;
- das erste Element gefolgt von der Filterung der Restliste ansonsten.

Ein weiteres Beispiel für rekursive Funktionen auf Listen, eine Filter-Funktion. Sie heißt so, weil sie Elemente aus einer Liste nach einem gegebenen Kriterium herausfiltert. Auch hier nehmen wir die deutsche Beschreibung des Ergebnisses als Ausgangspunkt.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of real ) real -> ( list of real )
;; Returns: all elements of list that are smaller than x
( define ( less-than-only lst x )
   ( if ( empty? lst )
        empty
        ( if ( < ( first lst ) x )
            ( cons ( first lst ) ( less-than-only ( rest lst ) x ) )
            ( less-than-only ( rest lst ) x ) ) ) )

( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )
```

Und das ist die Umsetzung dieser Beschreibung in einer Funktion in Racket.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of real ) real -> ( list of real )
;; Returns: all elements of list that are smaller than x
( define ( less-than-only lst x )
   ( if ( empty? lst )
        empty
        ( if ( < ( first lst ) x )
            ( cons ( first lst ) ( less-than-only ( rest lst ) x ) )
            ( less-than-only ( rest lst ) x ) ) ) )

( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )
```

Hier sehen Sie, wie sinnvoll es für die Abgrenzung zu anderen Bestandteilen des Funktionstyps und somit für die Verständlichkeit ist, dass list of real in Klammern gefasst ist.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of real ) real -> ( list of real )
;; Returns: all elements of list that are smaller than x
( define ( less-than-only lst x )
   ( if ( empty? lst )
        empty
        ( if ( < ( first lst ) x )
            ( cons ( first lst ) ( less-than-only ( rest lst ) x ) )
            ( less-than-only ( rest lst ) x ) ) ) )
```

```
( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )
```

Auf die Checks kommen wir gleich noch zu sprechen. Aus Platzgründen schreiben wir auf diese Folie erst einmal nur *einen* Check. Wir sehen: Die Ergebnisliste in diesem Check besteht aus allen Elementen der Eingabeliste, die kleiner als 5 sind.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of real ) real -> ( list of real )
;; Returns: all elements of list that are smaller than x
( define ( less-than-only lst x )
   ( if ( empty? lst )
      empty
      ( if ( < ( first lst ) x )
         ( cons ( first lst ) ( less-than-only ( rest lst ) x ) )
         ( less-than-only ( rest lst ) x ) ) ) )

( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )
```

Wenn die Eingabeliste leer ist, dann ist natürlich auch die Ergebnisliste leer, das hatten wir in der Beschreibung in Umgangssprache schon festgehalten.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of real ) real -> ( list of real )
;; Returns: all elements of list that are smaller than x
( define ( less-than-only lst x )
   ( if ( empty? lst )
        empty
        ( if ( < ( first lst ) x )
             ( cons ( first lst ) ( less-than-only ( rest lst ) x ) )
             ( less-than-only ( rest lst ) x ) ) ) )

( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )
```

Der Fall, dass die Eingabeliste *nicht* leer ist, ist etwas komplexer abzuhandeln. Hier brauchen wir wieder eine Fallunterscheidung, also die if-Funktion.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of real ) real -> ( list of real )
;; Returns: all elements of list that are smaller than x
( define ( less-than-only lst x )
   ( if ( empty? lst )
        empty
        ( if ( < ( first lst ) x )
             ( cons ( first lst ) ( less-than-only ( rest lst ) x ) )
             ( less-than-only ( rest lst ) x ) ) ) )

( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )
```

In dem Fall, dass das erste Element von list kleiner als der zweite Parameter von less-than-only, also kleiner als x ist, sieht die Rekursion wie bisher aus: das erste Element plus rekursive Anwendung auf die Restliste.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of real ) real -> ( list of real )
;; Returns: all elements of list that are smaller than x
( define ( less-than-only lst x )
   ( if ( empty? lst )
        empty
        ( if ( < ( first lst ) x )
             ( cons ( first lst ) ( less-than-only ( rest lst ) x ) )
             ( less-than-only ( rest lst ) x ) ) ) )

( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )
```

Aber wenn das erste Element *nicht* kleiner als x ist, dann soll es herausgefiltert werden. Das geht ganz einfach, indem wir den rekursiven Aufruf auf der Restliste allein zum Gesamtergebnis machen, das erste Element taucht wie gewünscht in der Ergebnisliste nicht mehr auf. Auch dies setzt die Beschreibung in Umgangssprache wieder eins-zu-eins um.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( less-than-only ( list 2 7 5 3 6 ) 5 )
( cons 2 ( less-than-only ( list 7 5 3 6 ) 5 ) )
( cons 2 ( less-than-only ( list 5 3 6 ) 5 ) )
( cons 2 ( less-than-only ( list 3 6 ) 5 ) )
( cons 2 ( cons 3 ( less-than-only ( list 6 ) 5 ) ) )
( cons 2 ( cons 3 ( less-than-only empty 5 ) ) )
( cons 2 ( cons 3 empty ) )
( cons 2 ( list 3 ) )
( list 2 3 )
```

Die Auswertung mit einer konkreten Liste ist dann wieder völlig analog zu den vorherigen Beispielen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( less-than-only ( list 2 7 5 3 6 ) 5 )
( cons 2 ( less-than-only ( list 7 5 3 6 ) 5 ) )
( cons 2 ( less-than-only ( list 5 3 6 ) 5 ) )
( cons 2 ( less-than-only ( list 3 6 ) 5 ) )
( cons 2 ( cons 3 ( less-than-only ( list 6 ) 5 ) ) )
( cons 2 ( cons 3 ( less-than-only empty 5 ) ) )
( cons 2 ( cons 3 empty ) )
( cons 2 ( list 3 ) )
( list 2 3 )
```

So wie in diesem beiden Schritten, mit dem Wert 2 kleiner 5 beziehungsweise 3 kleiner 5, sieht die Auswertung aus, wenn das erste Element der Liste kleiner als der Vergleichswert ist.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( less-than-only ( list 2 7 5 3 6 ) 5 )
( cons 2 ( less-than-only ( list 7 5 3 6 ) 5 ) )
( cons 2 ( less-than-only ( list 5 3 6 ) 5 ) )
( cons 2 ( less-than-only ( list 3 6 ) 5 ) )
( cons 2 ( cons 3 ( less-than-only ( list 6 ) 5 ) ) )
( cons 2 ( cons 3 ( less-than-only empty 5 ) ) )
( cons 2 ( cons 3 empty ) )
( cons 2 ( list 3 ) )
( list 2 3 )
```

Und in diesen beiden Fällen verschwindet der Wert 7 beziehungsweise 6 einfach aus der weiteren Betrachtung – eben herausgefiltert.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Gemäß Objektmodell:

- **Eine Liste ist eine Folge von *Werten*, nicht von *Objekten***
- **Eine gefilterte Liste enthält *Kopien* von Werten**

```
( define my-list ( list ..... ) )

( define ( do-sth-with-2-lists list1 list2 ) ( ..... ) )

( do-sth-with-2-lists ( less-than-only my-list 5 ) my-list )
```

Wir sollten besser noch einmal kurz auf das Objektmodell von Racket zu sprechen kommen, wie wir es in Kapitel 04a, Abschnitt zur funktionalen Abstraktion, kurz angerissen hatten.

***Erinnerung*: Dort hatten wir gesehen, dass es in Racket keine Objektidentität gibt, das heißt, es gibt nur Werte.**

Vielleicht hilft die Übertragung auf das Java-Datenmodell: Darin können Sie sich das so vorstellen, als wäre jedes Vorkommen eines Wertes ein eigenes Objekt, also keine zwei Referenzen auf dasselbe Objekt, jede Zuweisung ist eine Kopie in ein neues Objekt, und zum Test auf Gleichheit gibt es nur den Test auf Wertgleichheit, es gibt keinen Test auf Objektidentität. So stellt sich das Objektmodell für den Racket-Programmierer dar, DrRacket ist frei darin, das intern und damit unsichtbar aus Effizienzgründen anders zu organisieren.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Gemäß Objektmodell:

- **Eine Liste ist eine Folge von *Werten*, nicht von *Objekten***
- **Eine gefilterte Liste enthält *Kopien* von Werten**

```
( define my-list ( list ..... ) )
( define ( do-sth-with-2-lists list1 list2 ) ( ..... ) )
( do-sth-with-2-lists ( less-than-only my-list 5 ) my-list )
```

Das bedeutet im konkreten Beispiel auf dieser Folie, dass die beiden aktualen Parameter der rein illustrativen Funktion do-sth-with-2-lists nichts miteinander zu tun haben, sie sind in keiner Weise miteinander verknüpft. Der einzige Zusammenhang ist inhaltlicher Natur: Jeder Wert in der ersten Liste ist eine Kopie eines Wertes in der zweiten Liste. Daher kann in do-sth-with-2-lists passieren, was will, es kann in do-sth-with-2-lists keine unerwünschten Wechselwirkungen zwischen den beiden Parametern geben, denn es sind zwei disjunkte Listen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )

( check-expect ( less-than-only ( list 6 7 5 8 6 ) 5 ) empty )

( check-expect ( less-than-only ( list 2 3 1 2 1 ) 5 )
               ( list 2 3 1 2 1 ) )

( check-expect ( less-than-only ( list -6 6 -6 6 -6 ) -5 )
               ( list -6 -6 -6 ) )

( check-expect ( less-than-only empty ) 5 ) empty )
```

Wir kommen wie versprochen jetzt noch einmal auf das stichprobenartige Überprüfen der Korrektheit der Funktion less-than-only mittels check-expect zurück. Bei dieser Funktion gibt es mehrere, grundsätzlich unterschiedliche Fälle, die vielleicht sicherheitshalber einer Überprüfung wert wären und deren Überprüfung mit check-expect auch das Verständnis des Lesers weiter fördern. Wichtig sind neben den allgemeinen Fällen wieder die Randfälle.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )
( check-expect ( less-than-only ( list 6 7 5 8 6 ) 5 ) empty )
( check-expect ( less-than-only ( list 2 3 1 2 1 ) 5 )
               ( list 2 3 1 2 1 ) )
( check-expect ( less-than-only ( list -6 6 -6 6 -6 ) -5 )
               ( list -6 -6 -6 ) )
( check-expect ( less-than-only empty ) 5 ) empty )
```

Ein potentiell relevanter Randfall ist, dass die Ergebnisliste leer ist, obwohl die Eingabeliste *nicht* leer ist.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )

( check-expect ( less-than-only ( list 6 7 5 8 6 ) 5 ) empty )

( check-expect ( less-than-only ( list 2 3 1 2 1 ) 5 )
               ( list 2 3 1 2 1 ) )

( check-expect ( less-than-only ( list -6 6 -6 6 -6 ) -5 )
               ( list -6 -6 -6 ) )

( check-expect ( less-than-only empty ) 5 ) empty )
```

Oder auch der umgekehrte Randfall, dass *alle* Elemente der Eingabeliste sich in der Ergebnisliste wiederfinden.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )
( check-expect ( less-than-only ( list 6 7 5 8 6 ) 5 ) empty )
( check-expect ( less-than-only ( list 2 3 1 2 1 ) 5 )
               ( list 2 3 1 2 1 ) )
( check-expect ( less-than-only ( list -6 6 -6 6 -6 ) -5 )
               ( list -6 -6 -6 ) )
( check-expect ( less-than-only empty ) 5 ) empty )
```

Auch wenn das Vorzeichen der beteiligten Zahlen eigentlich keine Rolle spielen sollte, kann es nicht schaden, auch hier zu variieren, einfach um sicher zu gehen, dass das wirklich keine Rolle spielt.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( check-expect ( less-than-only ( list 2 7 5 3 6 ) 5 ) ( list 2 3 ) )
( check-expect ( less-than-only ( list 6 7 5 8 6 ) 5 ) empty )
( check-expect ( less-than-only ( list 2 3 1 2 1 ) 5 )
               ( list 2 3 1 2 1 ) )
( check-expect ( less-than-only ( list -6 6 -6 6 -6 ) -5 )
               ( list -6 -6 -6 ) )
( check-expect ( less-than-only empty ) 5 ) empty )
```

Und natürlich wie immer den Fall überprüfen, dass die Eingabeliste selbst schon leer ist.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( less-than-only lst x )
   ( cond
      [ ( empty? lst ) empty ]
      [ ( < ( first lst ) x )
        ( cons ( first lst ) ( less-than-only ( rest lst ) x ) ) ]
      [ else ( less-than-only ( rest lst ) x ) ] ) )
```

Das ist noch einmal dieselbe Funktion less-than-only wie eben, nur etwas anders realisiert, mit einem neuen Konstrukt namens cond, das ineinandergeschachtelte if-Verzweigungen wie in der ersten Realisierung von less-than-only oft vereinfacht. Vergleichen Sie diese Realisierung von less-than-only beim Durchgehen Punkt für Punkt mit der ersten Realisierung vor ein paar Folien, die nur if benutzt hat!

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( less-than-only lst x )
  ( cond
     [ ( empty? lst ) empty ]
     [ ( < ( first lst ) x )
       ( cons ( first lst ) ( less-than-only ( rest lst ) x ) ) ]
     [ else ( less-than-only ( rest lst ) x ) ] ) )
```

Ein cond-Ausdruck beginnt mit dem Wort cond und ist insgesamt in runde Klammern gefasst, so wie es ja bei allen zusammengesetzten Ausdrücken der Fall ist.

***Erinnerung*: Die cond-Funktion in Racket erinnert stark an die switch-Anweisung in Java aus dem gleichnamigen Abschnitt von Kapitel 03c. Allerdings werden wir auf den nächsten Folien sehen, dass das cond-Konstrukt in Racket deutlich flexibler ist.**

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( less-than-only lst x )
   ( cond
     [ ( empty? lst ) empty ]
     [ ( < ( first lst ) x )
       ( cons ( first lst ) ( less-than-only ( rest lst ) x ) ) ]
     [ else ( less-than-only ( rest lst ) x ) ] ) )
```

Das cond-Konstrukt ist anders aufgebaut als gewöhnliche Funktionen. Nach dem cond kommen mehrere Fälle in einer Art Fallunterscheidung. Jeder Fall ist in eckige Klammern gefasst. Dies ist der erste Fall. Jeder Fall besteht aus zwei Ausdrücken.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( less-than-only lst x )
  ( cond
    [ ( empty? lst ) empty ]
    [ ( < ( first lst ) x )
      ( cons ( first lst ) ( less-than-only ( rest lst ) x ) ) ]
    [ else ( less-than-only ( rest lst ) x ) ] ) )
```

Der erste Ausdruck in einem solchen Fall muss einen Wahrheitswert zurückliefern. In diesem konkreten Fall liefert der Ausdruck genau dann true zurück, wenn die Liste leer ist.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( less-than-only lst x )
   ( cond
     [ ( empty? lst ) empty ]
     [ ( < ( first lst ) x )
       ( cons ( first lst ) ( less-than-only ( rest lst ) x ) ) ]
     [ else ( less-than-only ( rest lst ) x ) ] ) )
```

Falls der erste Ausdruck true zurückliefert, falls die Liste also leer ist, dann ist empty der Wert des gesamten cond-Ausdrucks. Also: Falls die Eingabeliste leer ist, dann ist auch die Ergebnisliste leer.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( less-than-only lst x )
   ( cond
      [ ( empty? lst ) empty ]
      [ ( < ( first lst ) x )
        ( cons ( first lst ) ( less-than-only ( rest lst ) x ) ) ]
      [ else ( less-than-only ( rest lst ) x ) ] ) )
```

Falls der erste Fall *nicht* eintritt, falls also die Eingabeliste *nicht* leer ist, dann wird der zweite Fall geprüft. An dieser Stelle wissen wir definitiv, dass die Liste nicht leer ist, denn sonst wäre der erste Fall eingetreten und der zweite Fall gar nicht geprüft worden. Also können wir Funktion first ohne Fehler auf die Eingabeliste anwenden und das erste Element mit x vergleichen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( less-than-only lst x )
  ( cond
    [ ( empty? lst ) empty ]
    [ ( < ( first lst ) x )
      ( cons ( first lst ) ( less-than-only ( rest lst ) x ) ) ]
    [ else ( less-than-only ( rest lst ) x ) ] ) )
```

Und falls der zweite Fall tatsächlich eintritt, falls also die Liste nicht leer ist und das erste Element kleiner als x ist, dann wird mit dem Rest der Liste rekursiv weitergemacht und das erste Element an die Ergebnisliste vorne angehängt.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( less-than-only lst x )
   ( cond
      [ ( empty? lst ) empty ]
      [ ( < ( first lst ) x )
        ( cons ( first lst ) ( less-than-only ( rest lst ) x ) ) ]
      [ else ( less-than-only ( rest lst ) x ) ] ) )
```

In einem letzten Fall kann man noch angeben, was der Gesamtwert des cond-Ausdrucks sein soll, falls keiner der vorangehenden, explizit abgefragten Fälle eintritt. In diesem Beispiel tritt der else-Fall ein, wenn die Eingabeliste nicht leer ist und das erste Element nicht kleiner als x ist. Dann gehört das erste Element nicht zur Ergebnisliste.

***Achtung*: Falls kein else-Teil vorhanden ist, aber keiner der vorhergehenden, abgefragten Fälle eintritt, gibt es einen Fehler bei der Ausführung, falls wirklich eine Situation eintritt, die durch keinen der Fälle abgedeckt ist. Falls die abgefragten Fälle nicht jede mögliche Situation abdecken, sollte also immer ein else-Teil dabei sein.**

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Eine gefilterte Liste ist

- **leer, falls die Liste leer ist;**
- **die Restliste, falls das erste Element den Filter nicht passiert;**
- **das erste Element gefolgt von der Filterung der Restliste ansonsten.**

```
( cond
  [ ( empty? lst ) empty ]
  [ ( < ( first lst ) x )
    ( cons ( first x ) ( less-than-only ( rest lst ) x ) ) ) ]
  [ else ( less-than-only ( rest lst ) x ) ] )
```

Auch hier wieder der Aha-Effekt noch einmal wiederholt: Die Racket-Funktion ist die Eins-zu-eins-Übersetzung der Beschreibung in Umgangssprache.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Aussage, dass zwei Listen an irgendeiner Position dasselbe Element enthalten, ist

- **false, wenn mindestens eine der beiden Listen leer ist;**
- **true, wenn das erste Element in beiden Listen dasselbe ist;**
- **andernfalls genau dann true für die beiden Gesamtlisten, wenn sie true für die beiden Restlisten ist.**

Nun noch ein Beispiel für eine rekursive Funktion auf Listen, die jetzt *zwei* Listen prozessiert, nicht nur *eine* wie in den vorhergehenden Beispielen. Es geht darum, bei zwei Listen zu prüfen, ob es mindestens eine Position gibt, an der beide Listen dasselbe Element enthalten.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Aussage, dass zwei Listen an irgendeiner Position dasselbe Element enthalten, ist

- false, wenn mindestens eine der beiden Listen leer ist;
- true, wenn das erste Element in beiden Listen dasselbe ist;
- andernfalls genau dann true für die beiden Gesamtlisten, wenn sie true für die beiden Restlisten ist.

Das einzige Element einer Liste, auf das wir direkt zugreifen können, ist das erste. Nur an der ersten Position können wir daher direkt die beiden Elemente miteinander vergleichen.

***Nebenbemerkung*: Racket bietet neben first auch noch analoge Funktionen namens second, third und so weiter bis tenth zum Zugriff auf das zweite bis zehnte Element der Liste. Aber allgemein um an das Element an irgendeiner beliebigen Position zu kommen, muss man die Liste wie bisher schrittweise – also rekursiv – mit Funktion rest demontieren. Daher interessieren uns second bis tenth nicht weiter, sie sind nur in seltenen Fällen wirklich hilfreich.**

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Aussage, dass zwei Listen an irgendeiner Position dasselbe Element enthalten, ist

- **false, wenn mindestens eine der beiden Listen leer ist;**
- **true, wenn das erste Element in beiden Listen dasselbe ist;**
- **andernfalls genau dann true für die beiden Gesamtlisten, wenn sie true für die beiden Restlisten ist.**

Wenn die Elemente an der ersten Position in beiden Listen unterschiedlich sind, dann müssen eben die anderen Positionen die Entscheidung bringen, ob das Gesamtergebnis true oder false sein soll.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Aussage, dass zwei Listen an irgendeiner Position dasselbe Element enthalten, ist

- false, wenn mindestens eine der beiden Listen leer ist;
- true, wenn das erste Element in beiden Listen dasselbe ist;
- andernfalls genau dann true für die beiden Gesamtlisten, wenn sie true für die beiden Restlisten ist.

Wir müssen aber aufpassen: Der Fall, dass die beiden ersten Elemente gleich sind, ist nur definiert, wenn beide Listen jeweils ein erstes Element haben, und der letzte Fall nur, wenn beide Liste jeweils eine Restliste nach dem ersten Element haben. Beides – Existenz eines ersten Elements und Existenz der Restliste – ist genau dann der Fall, wenn die Liste nicht leer ist. Also müssen wir vorab den Fall abfangen, dass eine der beiden Listen leer ist oder sogar beide leer sind. In diesem Fall kann es natürlich kein identisches Element in beiden Listen geben, das Ergebnis ist also unbesehen false.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Und das ist dann die Umsetzung dieser umgangssprachlichen Beschreibung.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Das sind die beiden Listen, wir nennen sie list1 und list2.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Das Wort „iff“ mit doppeltem „f“ ist eine gängige Abkürzung für das englische Konstrukt „... if, and only if,“ Es besagt also, dass die beiden miteinander verknüpften Aussagen äquivalent sind: Der erste Aussage ist genau dann wahr, wenn die zweite Aussage wahr ist.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Im ersten check-expect sehen wir zwei Listen, die an der dritten Position denselben Wert haben. In diesem Fall soll true herauskommen gemäß Returns-Klausel.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Jetzt zwei Listen, die gleich lang sind, aber keine Position mit demselben Wert haben, also erwünschtes Ergebnis false.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Auf jeden Fall sollten die beiden Fälle extra abgeprüft werden, dass die erste Liste länger als die zweite und umgekehrt ist.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Wieder eine Fallunterscheidung mit cond.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Wenn bis dahin nicht eine Position mit demselben Wert in beiden Listen gefunden wurde, dann wird irgendwann der Fall eintreten, dass eine der beiden Listen zu Ende ist, zum Beispiel die erste. Da keine weiteren gemeinsamen Positionen existieren, kann das Ergebnis dann nur false sein.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Es kann natürlich auch die zweite Liste sein, die vor der ersten Liste zu Ende ist, dann greift der zweite Fall.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Und wenn beide Listen zugleich zu Ende sind, ohne dass an irgendeiner Position bis dahin derselbe Wert gefunden wurde, dann würden die ersten beiden Fälle im cond-Ausdruck greifen. Aber der erste Fall wird vor dem zweiten Fall geprüft, was in diesem Fall egal ist, da beide Fälle dasselbe zurückliefern, nämlich false.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Im dritten Fall des cond-Ausdrucks wissen wir aufgrund der ersten beiden Fälle, dass beide Listen nicht leer sind, so dass Funktion first auf beide Listen problemlos anwendbar ist, um die beiden ersten Elemente miteinander zu vergleichen. Im ersten check-expect greift dieser Fall an der dritten Position, und das Ergebnis ist true.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of number ) ( list of number ) -> boolean
;; Returns: true iff both lists have the same value at (at least) one position
( define ( equal-at-some-position list1 list2 )
    ( cond
        [ ( empty? list1 ) #f ]
        [ ( empty? list2 ) #f ]
        [ ( = ( first list1 ) ( first list2 ) ) #t ]
        [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ) ] ) )

( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) ) #t )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) ) #f )
( check-expect ( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) ) #f )
```

Falls beide Listen nicht leer sind und die Elemente an der ersten Position der beiden Listen nicht identisch sind, ist das Gesamtergebnis gleich dem Ergebnis für die beiden Restlisten, welches wieder durch rekursiven Aufruf ermittelt wird.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) )
( equal-at-some-position ( list 2 3 ) ( list 1 3 ) )
( equal-at-some-position ( list 3 ) ( list 3 ) )
#t

( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) )
( equal-at-some-position ( list 2 3 ) ( list 1 2 ) )
( equal-at-some-position ( list 3 ) ( list 2 ) )
( equal-at-some-position empty empty )
#f
```

Hier sehen Sie die rekursive Auswertung der ersten beiden check-expect.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) )
( equal-at-some-position ( list 2 3 ) ( list 1 3 ) )
( equal-at-some-position ( list 3 ) ( list 3 ) )
#t

( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) )
( equal-at-some-position ( list 2 3 ) ( list 1 2 ) )
( equal-at-some-position ( list 3 ) ( list 2 ) )
( equal-at-some-position empty empty )
#f
```

Im dritten Durchlauf ist die ursprünglich dritte Position jetzt die erste. Die beiden Werte an der ersten Position sind identisch, der dritte Fall des cond-Ausdrucks tritt also ein, daher ist das Ergebnis true.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( equal-at-some-position ( list 1 2 3 ) ( list 2 1 3 ) )
( equal-at-some-position ( list 2 3 ) ( list 1 3 ) )
( equal-at-some-position ( list 3 ) ( list 3 ) )
#t

( equal-at-some-position ( list 1 2 3 ) ( list 2 1 2 ) )
( equal-at-some-position ( list 2 3 ) ( list 1 2 ) )
( equal-at-some-position ( list 3 ) ( list 2 ) )
( equal-at-some-position empty empty )
#f
```

Im zweiten check-expect kommen hingegen beide Listen im vierten Durchlauf bei empty an, und das Ergebnis ist false.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( equal-at-some-position ( list 1 2 3 ) ( list 2 1 ) )
( equal-at-some-position ( list 2 3 ) ( list 1 ) )
( equal-at-some-position ( list 3 ) empty )
#f

( equal-at-some-position ( list 1 2 ) ( list 2 1 2 ) )
( equal-at-some-position ( list 2 ) ( list 1 2 ) )
( equal-at-some-position empty ( list 2 ) )
#f
```

Nun die letzten beiden check-expect. In beiden Fällen kommt die eine Liste an ihr Ende und die andere noch nicht. Daher greift der zweite beziehungsweise der erste Fall im cond-Ausdruck, Gesamtergebnis also false.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Aussage, dass zwei Listen an irgendeiner Position dasselbe Element enthalten, ist

- **false, wenn mindestens eine der beiden Listen leer ist;**
- **true, wenn das erste Element in beiden Listen dasselbe ist;**
- **andernfalls genau dann true für die beiden Gesamtlisten, wenn sie true für die beiden Restlisten ist.**

```
( cond
    [ ( empty? list1 ) #f ]
    [ ( empty? list2 ) #f ]
    [ ( = ( first list1 ) ( first list2 ) #t ]
    [ else ( equal-at-some-position ( rest list1 ) ( rest list2 ) ] )
```

Und wieder abschließend noch einmal der Aha-Effekt: Man kann das Ergebnis, das herauskommen soll, rekursiv definieren und diese rekursive Definition eins-zu-eins in eine rekursive Racket-Funktion umsetzen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define abc ( list ´alpha ´bravo ´charlie ) )
( define bac ( list ´bravo ´alpha ´charlie ) )


;; Type: ( list of symbol ) ( list of symbol ) -> boolean
( define ( equal-at-some-position list1 list2 )
   ( cond
      [  ( or ( empty? list1 ) ( empty? list2 ) )  #f  ]
      [  ( symbol=? ( first list1 ) ( first list2 ) )  #t  ]
      [  else ( equal-at-some-position
                                 ( rest list1 ) ( rest list2 ) ) ] ) )
```

Dasselbe Beispiel nun mit Listen von Symbolen statt Listen von Zahlen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define abc ( list ´alpha ´bravo ´charlie ) )
( define bac ( list ´bravo ´alpha ´charlie ) )


;; Type: ( list of symbol ) ( list of symbol ) -> boolean
( define ( equal-at-some-position list1 list2 )
   ( cond
      [  ( or ( empty? list1 ) ( empty? list2 ) )  #f  ]
      [  ( symbol=? ( first list1 ) ( first list2 ) )  #t  ]
      [  else ( equal-at-some-position
                                 ( rest list1 ) ( rest list2 ) ) ] ) )
```

Eine Liste von Symbolen wird genauso kreiert wie eine Liste von Zahlen.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define abc ( list ´alpha ´bravo ´charlie ) )
( define bac ( list ´bravo ´alpha ´charlie ) )

;; Type: ( list of symbol ) ( list of symbol ) -> boolean
( define ( equal-at-some-position list1 list2 )
   ( cond
     [  ( or ( empty? list1 ) ( empty? list2 ) )  #f  ]
     [  ( symbol=? ( first list1 ) ( first list2 ) )  #t  ]
     [  else ( equal-at-some-position
                                ( rest list1 ) ( rest list2 ) ) ] ) )
```

Wir schreiben die Funktion equal-at-some-position so um, dass sie eine Liste von Symbolen anstelle einer Liste von Zahlen behandelt. Die Logik der Funktion ist ja eigentlich unabhängig vom Typ der Elemente, also passt das.

In beiden Beispiellisten oben findet sich dasselbe Symbol an der dritten Position, für diese beiden Listen soll die Funktion also true zurückliefern.

Rekursive Funktionen auf Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define abc ( list ´alpha ´bravo ´charlie ) )
( define bac ( list ´bravo ´alpha ´charlie ) )


;; Type: ( list of symbol ) ( list of symbol ) -> boolean
( define ( equal-at-some-position list1 list2 )
  ( cond
    [  ( or ( empty? list1 ) ( empty? list2 ) )  #f  ]
    [  ( symbol=? ( first list1 ) ( first list2 ) )  #t  ]
    [  else ( equal-at-some-position
                                ( rest list1 ) ( rest list2 ) ) ] ) )
```

Wie wir schon gesehen haben, liefert diese Funktion genau dann true, wenn die beiden Parameter erstens beides Symbole und zweitens dasselbe Symbol sind.

TECHNISCHE UNIVERSITÄT DARMSTADT

Structs

Wir haben primitive Datentypen gesehen, nämlich Zahlen, boolean und Symbole. Für die Zusammenfassung von Elementen haben wir zudem Listen gesehen. Jetzt sehen wir eine zweite Möglichkeit zur Zusammenfassung, die vergleichbar einer Klasse mit public-Objektattributen und ohne Methoden ist. Damit bekommen wir dann unendlich viel zusätzliches Beispielmaterial für Listen an die Hand.

Structs in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Student {
    public String lastName;
    public String firstName;
    public int enrollmentNumber;
}

Student alfTanner = new Student();
alfTanner.lastName = “Tanner“;
alfTanner.firstName = “Alf“;
alfTanner.enrollmentNumber = 123;
```

Hier sehen Sie ein Beispiel für eine solche Klasse. Dieses Beispiel in Java werden wir zur Erläuterung in Racket nachbauen.

Structs in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Student {
    public String lastName;
    public String firstName;
    public int enrollmentNumber;
}
```

```
( define-struct student
      ( last-name first-name enrollment-number ) )
```

Oben ist noch einmal der Struct namens Student in Java definiert, exakt so wie auf der letzten Folie. Unten sehen Sie den Nachbau in Racket.

Structs in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Student {
    public String lastName;
    public String firstName;
    public int enrollmentNumber;
}
```

```
( define-struct student
        ( last-name first-name enrollment-number ) )
```

Mit dem Schlüsselwort define-struct wird angezeigt, dass der folgende Name, also student, nicht der Name einer Konstanten oder einer Funktion werden soll, sondern der Name eines Struct-Typs. Wie bei define kommt die ganze Definition in ein Klammerpaar.

Structs in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Student {
    public String lastName;
    public String firstName;
    public int enrollmentNumber;
}
```

```
( define-struct student
        ( last-name first-name enrollment-number ) )
```

Gemäß der Namenskonvention in Racket, an die wir uns bisher strikt gehalten haben und auch weiter halten, wird der Bezeichner student vollständig kleingeschrieben. Das ist der Name des Struct-Typs.

Structs in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Student {
    public String lastName;
    public String firstName;
    public int enrollmentNumber;
}
```

```
( define-struct student
      ( last-name first-name enrollment-number ) )
```

In Klammern werden danach die Namen der Attribute einfach aufgezählt. Wie immer in Racket, werden die Typen nicht hingeschrieben, sondern zur Laufzeit wird bei jeder Operation auf einem Attribut geprüft, ob der Typ mit der Operation kompatibel ist. Damit ist der Struct-Typ vollständig definiert.

***Nebenbemerkung*: Im Zusammenhang mit Racket spricht man in der Regel nicht von Attributen, sondern von Feldern, aber wir bleiben hier bei dem uns geläufigen Begriff Attribut, das auch weniger mehrdeutig ist als „Feld“.**

Struct erzeugen im Vergleich

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Student alfTanner = new Student();
alfTanner.lastName = “Tanner“;
alfTanner.firstName = “Alf“;
alfTanner.enrollmentNumber = 123;
```

```
( define alf-tanner ( make-student ´Tanner ´Alf 123 ) )
```

Oben sehen wir noch einmal, wie ein Struct in Java erzeugt und seine Attribute initialisiert werden können. Für die Initialisierung der Attribute hätten wir Student natürlich auch alternativ einen Konstruktor mitgeben können.

Struct erzeugen im Vergleich

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Student alfTanner = new Student();
alfTanner.lastName = “Tanner“;
alfTanner.firstName = “Alf“;
alfTanner.enrollmentNumber = 123;
```

```
( define alf-tanner ( make-student ´Tanner ´Alf 123 ) )
```

In Racket bekommen wir für einen selbstdefinierten Struct-Typ wie student so etwas wie einen Konstruktor automatisch geschenkt. Für ein Struct namens xyz heißt dieser make-xyz und hat so viele Parameter, wie der Struct xyz Attribute hat. Diese Parameter initialisieren die Attribute, wobei die Reihenfolge der Parameter in make-xyz gleich der Reihenfolge der Attribute in der Definition von xyz ist.

Die Funktion make-student hat also drei Parameter und setzt die Parameterwerte in die Attribute. Das erste Attribut wird mit dem ersten Parameterwert initialisiert, das zweite mit dem zweiten, das dritte mit dem dritten.

Attributzugriff im Vergleich

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public String lastName ( Student studi ) {
    return studi.lastName;
}
```

```
;; Type: student -> symbol
( define ( last-name studi )
      ( student-last-name studi ) )
```

Wie greifen wir nun auf ein Attribut zu?

Attributzugriff im Vergleich

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public String lastName ( Student studi ) {
    return studi.lastName;
}
```

```
;; Type: student -> symbol
( define ( last-name studi )
        ( student-last-name studi ) )
```

Um das zu sehen, definieren wir eine illustrative Methode in Java und eine dazu völlig analoge Funktion in Racket.

Attributzugriff im Vergleich

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public String lastName ( Student studi ) {
    return studi.lastName;
}
```

```
;; Type: student -> symbol
( define ( last-name studi )
      ( student-last-name studi ) )
```

Beide Ausdrücke sehen auf den ersten Blick zwar sehr unterschiedlich aus, beinhalten aber dasselbe, nämlich den Wert des Attributs last-name von studi, wobei studi vom Typ student sein muss. In Racket kommt zuerst der Name des Struct-Typs, gefolgt vom Namen des Attributs. beide sind durch Bindestrich miteinander verbunden. Das ist die Funktion, mit der auf das Attribut last-name zugegriffen wird, auch diese Funktion wird uns durch Racket für jeden selbstdefinierten Struct und jedes Attribut dieses Structs geschenkt. Dann folgt als Parameter dieser Funktion der Name des Objektes, von dem der Attributwert zurückgeliefert werden soll.

Gemischte Liste mit Studierenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define my-list
  ( list ´a pi ( make-student ´Tanner ´Alf 123 ) #t 2.71 ) )


;; Type: ( list of ANY ) -> natural
( define ( number-of-elems lst )
    ( if ( empty? lst )
        0
        ( + 1 ( number-of-elems ( rest lst ) ) ) ) )
```

Mit Structs haben wir jetzt ausreichend Spielmaterial für einen wichtigen Aspekt von Listen, den wir anhand dieses kleinen Beispiels beleuchten. Die beispielhafte Funktion unten berechnet einfach die Länge der Liste, also die Anzahl Elemente in der Liste. Für diese Funktion ist der Typ der Listenelemente völlig unerheblich.

Gemischte Liste mit Studierenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define my-list
  ( list ´a pi ( make-student ´Tanner ´Alf 123 ) #t 2.71 ) )


;; Type: ( list of ANY ) -> natural
( define ( number-of-elems lst )
    ( if ( empty? lst )
        0
        ( + 1 ( number-of-elems ( rest lst ) ) ) ) )
```

Nun schauen wir uns als Beispiel eine Funktion auf Listen an, die *nicht nur* Studierende enthalten. Wir sehen dabei, dass Listen nicht unbedingt *homogen* sein müssen, das heißt, es müssen nicht alle Elemente vom selben Typ sein. Eine Liste mit gemischten Typen nennen wir *heterogen*.

Gemischte Liste mit Studierenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define my-list
  ( list ´a pi ( make-student ´Tanner ´Alf 123 ) #t 2.71 ) )


;; Type: ( list of ANY ) -> natural
( define ( number-of-elems lst )
    ( if ( empty? lst )
        0
        ( + 1 ( number-of-elems ( rest lst ) ) ) ) )
```

In der Spezifikation des Typs der Funktion besagt ANY per Konvention, dass die Elemente der Liste beliebigen Typs sein dürfen, wobei es auch egal ist, ob die mit (list of ANY) bezeichnete Liste homogen oder heterogen ist.

Gemischte Liste mit Studierenden

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Liste aller Studierenden extrahiert aus einer gegebenen Liste von beliebigen Elementen ist:

- **leer, falls die gegebene Liste leer ist**
- **die Liste aller Studierenden extrahiert aus der Restliste der gegebenen Liste, falls das erste Element kein(e) Studierende(r) ist**
- **das erste Element plus die Liste aller Studierenden extrahiert aus der Restliste der gegebenen Liste ansonsten**

Nun schauen wir uns als zweites Beispiel noch eine etwas komplexere Funktion auf heterogenen Listen an. Dies ist wieder eine Filterfunktion: Herausgefiltert werden alle Elemente, die nicht vom Typ student sind. Die Beschreibung in Umgangssprache ist daher inhaltlich dieselbe wie bei der Filterfunktion less-than-only weiter vorne in diesem Kapitel, nur dass das Filterkriterium ein anderes ist.

Gemischte Liste mit Studierenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( all-students lst )
  ( cond
    [  ( empty? lst )  empty  ]
    [  ( student? ( first lst ) ) ( cons ( first lst ) ( all-students ( rest lst ) ) ) ]
    [  else  ( all-students ( rest lst ) )  ] ) )
```

```
( define ( less-than-only lst x )
  ( cond
    [ ( empty? lst ) empty ]
    [ ( < ( first lst ) x ) ( cons ( first lst ) ( less-than-only ( rest lst ) x ) ) ]
    [ else ( less-than-only ( rest lst ) x ) ] ) )
```

Und das ist die Umsetzung in Racket. Es sollte nicht überraschen, dass die Struktur praktisch identisch zu der von less-than-only ist, hier zum Vergleich noch einmal danebengestellt. Nur ein paar Details sind anders.

Gemischte Liste mit Studierenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of ANY ) -> ( list of student )
( define ( all-students lst )
  ( cond
    [  ( empty? lst )  empty  ]
    [  ( student? ( first lst ) )
           ( cons ( first lst )
                  ( all-students ( rest lst ) ) ) ]
    [  else  ( all-students ( rest lst ) )  ] ) )
```

Wir konzentrieren uns auf all-students. Die Eingabeliste darf beliebig sein, die Ergebnisliste soll nur Elemente vom Typ student enthalten.

Gemischte Liste mit Studierenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of ANY ) -> ( list of student )
( define ( all-students lst )
  ( cond
    [  ( empty? lst )  empty  ]
    [  ( student? ( first lst ) )
           ( cons ( first lst )
                  ( all-students ( rest lst ) ) ) ]
    [  else  ( all-students ( rest lst ) )  ] ) )
```

Wenn die Eingabeliste leer ist, dann sind natürlich auch keine Studierenden darin, das heißt, die Ausgabeliste ist leer.

Gemischte Liste mit Studierenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of ANY ) -> ( list of student )
( define ( all-students lst )
  ( cond
    [  ( empty? lst )  empty  ]
    [  ( student? ( first lst ) )
          ( cons ( first lst )
                 ( all-students ( rest lst ) ) ) ]
    [  else  ( all-students ( rest lst ) )  ] ) )
```

In diesem Zweig prüfen wir, ob das erste Element der Eingabeliste vom Typ student ist. Die Funktion student? ist ebenfalls automatisch dadurch vordefiniert, dass wir den Struct-Typ student eingerichtet haben.

Gemischte Liste mit Studierenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of ANY ) -> ( list of student )
( define ( all-students lst )
  ( cond
    [  ( empty? lst )  empty  ]
    [  ( student? ( first lst ) )
            ( cons ( first lst )
                    ( all-students ( rest lst ) ) ) ]
    [  else  ( all-students ( rest lst ) )  ] ) )
```

Wenn das erste Element der Eingabeliste tatsächlich vom Typ student ist, dann besteht die Ergebnisliste aus diesem Element plus aller Studierenden in der Restliste – also plus dem Ergebnis des rekursiven Aufrufs von all-students auf der Restliste.

Gemischte Liste mit Studierenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( list of ANY ) -> ( list of student )
( define ( all-students lst )
  ( cond
    [  ( empty? lst )  empty  ]
    [  ( student? ( first lst ) )
          ( cons ( first lst )
                 ( all-students ( rest lst ) ) ) ]
    [  else  ( all-students ( rest lst ) )  ] ) )
```

Und wenn weder die Liste leer ist noch das erste Element vom Typ student ist, dann ist das Ergebnis für die Gesamtliste identisch mit dem Ergebnis für die Restliste und kann daher durch rekursiven Aufruf auf der Restliste berechnet werden.

TECHNISCHE UNIVERSITÄT DARMSTADT

Designmuster Akkumulatoren

Akkumulatoren sind ein wichtiges Designmuster in Racket. Das heißt, jetzt kommt kein neues Programmkonstrukt, sondern eine neue, häufige Art, wie die bisher eingeführten Programmkonstrukte, namentlich Rekursion, zielführend verwendet werden können.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( check-expect ( list-of-sums ( list 357 ) ) ( list 357 ) )

( check-expect ( list-of-sums  ( list 2 7 13 47 ) )
                               ( list 2 9 22 69 ) )
```

→ 2 = 2, 9 = 2 + 7, 22 = 2 + 7 + 13, 69 = 2 + 7 + 13 + 47

Wie üblich, schauen wir uns dazu ein konkretes Beispiel an, diesmal eine Funktion namens list-of-sums. Da der Rückgabewert der Funktion etwas komplizierter zu erklären ist, beginnen wir mit ein paar Checks, die hier nicht nur als Korrektheitstests, sondern als Zahlenbeispiele zur Erläuterung dienen.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( check-expect ( list-of-sums ( list 357 ) ) ( list 357 ) )

( check-expect ( list-of-sums  ( list 2 7 13 47 ) )
               ( list 2 9 22 69 ) )
```

→ 2 = 2, 9 = 2 + 7, 22 = 2 + 7 + 13, 69 = 2 + 7 + 13 + 47

Offensichtlich soll eine Liste von Zahlen eingegeben und eine andere Liste von Zahlen zurückgeliefert werden, und beide Listen sind gleich lang. Der Wert an einer Position der Ausgabeliste soll die Summe der Werte der Eingabeliste an allen Positionen von der ersten bis zu dieser Position sein. Das erste Element ist bei beiden Listen also gleich, und zwei aufeinanderfolgende Elemente der Ausgabeliste unterscheiden sich gerade um den Wert der Eingabeliste an der Position des zweiten dieser beiden Elemente.

Beispiele wie diese haben den Begriff „Akkumulator“ inspiriert, denn in diesem Beispiel werden ja die Zahlen der Eingabeliste sozusagen akkumuliert.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( check-expect ( list-of-sums ( list 357 ) ) ( list 357 ) )

( check-expect ( list-of-sums  ( list 2 7 13 47 ) )
                    ( list 2 9 22 69 ) )
```

→ 2 = 2, 9 = 2 + 7, 22 = 2 + 7 + 13, 69 = 2 + 7 + 13 + 47

Die Logik dieser Akkumulation erfordert, dass der Fall einer leeren Eingabeliste im Vertrag auszuschließen ist. Randfall ist daher der Fall einer einelementigen Eingabeliste. Dieser Fall sollte natürlich ebenfalls abgeprüft werden, und dieses check-expect ist sicherlich ebenfalls zum Verständnis der Funktion hilfreich.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( list-of-sums ( list 2 7 13 47 ) )
( list-of-sums-with-accu ( list 7 13 47 ) 2 )
( list-of-sums-with-accu ( list 13 47 ) 9 )
( list-of-sums-with-accu ( list 47 ) 22 )
( list-of-sums-with-accu empty 69 )

( 69 )
( 22 69 )
( 9 22 69 )
( 2 9 22 69 )
```

Bevor wir uns die Implementation der Funktion list-of-sums ansehen, betrachten wir ihre Ausführung anhand einer kleinen Beispielliste.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( list-of-sums ( list 2 7 13 47 ) )
( list-of-sums-with-accu ( list 7 13 47 ) 2 )
( list-of-sums-with-accu ( list 13 47 ) 9 )
( list-of-sums-with-accu ( list 47 ) 22 )
( list-of-sums-with-accu empty 69 )

( 69 )
( 22 69 )
( 9 22 69 )
( 2 9 22 69 )
```

Dazu nehmen wir einfach dasselbe Beispiel wie beim Check auf der letzten Folie her.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( list-of-sums ( list 2 7 13 47 ) )
( list-of-sums-with-accu ( list 7 13 47 ) 2 )
( list-of-sums-with-accu ( list 13 47 ) 9 )
( list-of-sums-with-accu ( list 47 ) 22 )
( list-of-sums-with-accu empty 69 )
( 69 )
( 22 69 )
( 9 22 69 )
( 2 9 22 69 )
```

Hier sehen Sie die einzelnen rekursiven Aufrufe der lokal in list-of-sums definierten Hilfsfunktion. Wie auch in den bisherigen Beispielen, wird in jedem rekursiven Aufruf die Liste durch den Rest der Liste ohne erstes Element ersetzt. Dabei passiert aber noch etwas mit dem zweiten Parameter: Von Aufruf zu Aufruf wächst der Wert dieses Parameters, und zwar immer um den Wert des Listenelements, das zuletzt herausgefallen ist.

Insgesamt ergibt sich als Invariante, dass der Akkumulator in jedem rekursiven Aufruf die Summe all derjenigen Elemente der ursprünglichen Eingabeliste ist, die bei diesem rekursiven Aufruf schon weggefallen sind. Aus dieser Invariante ergibt sich zwingend, dass der initiale Wert das erste Listenelement sein muss, denn in der Restliste der Eingabeliste fehlt ja nur das erste Element.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( list-of-sums ( list 2 7 13 47 ) )
( list-of-sums-with-accu ( list 7 13 47 ) 2 )
( list-of-sums-with-accu ( list 13 47 ) 9 )
( list-of-sums-with-accu ( list 47 ) 22 )
( list-of-sums-with-accu empty 69 )
```

```
( 69 )
( 22 69 )
( 9 22 69 )
( 2 9 22 69 )
```

Die Rückgabe jedes rekursiven Aufrufs von list-of-sum-with-accu ist die Rückgabeliste des darunterliegenden rekursiven Aufrufs plus dem eigenen Wert des Akkumulators als erstes Element vorneweg.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( list-of-sums ( list 2 7 13 47 ) )
( list-of-sums-with-accu ( list 7 13 47 ) 2 )
( list-of-sums-with-accu ( list 13 47 ) 9 )
( list-of-sums-with-accu ( list 47 ) 22 )
( list-of-sums-with-accu empty 69 )
( 69 )
( 22 69 )
( 9 22 69 )
( 2 9 22 69 )
```

Bei der leeren Liste im letzten rekursiven Aufruf muss daher der Wert des Akkumulators zur Ergebnisliste gemacht werden.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( list-of-sums ( list 2 7 13 47 ) )
( list-of-sums-with-accu ( list 7 13 47 ) 2 )
( list-of-sums-with-accu ( list 13 47 ) 9 )
( list-of-sums-with-accu ( list 47 ) 22 )
( list-of-sums-with-accu empty 69 )

( 69 )
( 22 69 )
( 9 22 69 )
( 2 9 22 69 )
```

In allen anderen rekursiven Aufrufen muss statt dessen der Wert im Akkumulator vorne an die Liste angehängt werden, die in den vorher abgeschlossenen, tieferen rekursiven Aufrufen schrittweise aufgebaut wurde. Die Liste aus dem rekursiven Aufruf mit leerer Liste enthält nur die 69. Daran wird vorne der Wert 22 des Akkumulators gehängt, ...

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( list-of-sums ( list 2 7 13 47 ) )
( list-of-sums-with-accu ( list 7 13 47 ) 2 )
( list-of-sums-with-accu ( list 13 47 ) 9 )
( list-of-sums-with-accu ( list 47 ) 22 )
( list-of-sums-with-accu empty 69 )

( 69 )
( 22 69 )
( 9 22 69 )
( 2 9 22 69 )
```

... an diese zweielementige Liste dann der Wert 9 im nächsthöheren rekursiven Aufruf, ...

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( list-of-sums ( list 2 7 13 47 ) )
( list-of-sums-with-accu ( list 7 13 47 ) 2 )
( list-of-sums-with-accu ( list 13 47 ) 9 )
( list-of-sums-with-accu ( list 47 ) 22 )
( list-of-sums-with-accu empty 69 )

( 69 )
( 22 69 )
( 9 22 69 )
( 2 9 22 69 )
```

... und dann die 2 an die dreielementige Liste.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( list-of-sums ( list 2 7 13 47 ) )
( list-of-sums-with-accu ( list 7 13 47 ) 2 )
( list-of-sums-with-accu ( list 13 47 ) 9 )
( list-of-sums-with-accu ( list 47 ) 22 )
( list-of-sums-with-accu empty 69 )

( 69 )
( 22 69 )
( 9 22 69 )
( 2 9 22 69 )
```

Und das Ergebnis dieses obersten rekursiven Aufrufs ist genau das, war wir als Ergebnis von list-of-sums herausbekommen wollten. Die Funktion list-of-sums muss also „nur“ die Funktion list-of-sums-with-accu mit der Restliste der Eingabeliste und dem ersten Element der Eingabeliste als Parametern aufrufen und mehr nicht.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
   ( ( define ( list-of-sums-with-accu lst accu )
      ( cond
       [  ( empty? lst )  ( list accu )  ]
       [  else  ( cons accu
                        ( list-of-sums-with-accu
                          ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
  ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Dies ist nun die Funktion list-of-sums, für die wir eben ein paar Checks und die Abarbeitung eines Beispiels gesehen haben.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
   ( ( define ( list-of-sums-with-accu lst accu )
      ( cond
       [  ( empty? lst )  ( list accu )  ]
       [  else  ( cons accu
                        ( list-of-sums-with-accu
                          ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
  ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Den Typ der Funktion haben wir schon durch die Checks erkennen können: eine Liste von Zahlen ‘rein, eine Liste von Zahlen ‘raus.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
    ( ( define ( list-of-sums-with-accu lst accu )
        ( cond
          [  ( empty? lst )  ( list accu )  ]
          [  else  ( cons accu
                           ( list-of-sums-with-accu
                             ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
    ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Normalerweise würden wir die Funktion so implementieren, dass sie auch mit leeren Listen fertig wird. Aber das ist in diesem Beispiel nicht der Punkt. Zur Vereinfachung ist die Funktion daher so implementiert, dass sie nur mit nichtleeren Listen klarkommt, weil sie ohne Abfrage auf den Rest und das erste Element zugreift. Als kleine Übung können Sie die Funktion ja so umschreiben, dass sie auch mit leeren Listen klarkommt.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
   ( ( define ( list-of-sums-with-accu lst accu )
      ( cond
       [  ( empty? lst )  ( list accu )  ]
       [  else  ( cons accu
                        ( list-of-sums-with-accu
                          ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
  ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Die Beschreibung der Rückgabe ist etwas kompliziert, daher lassen wir sie aus Platzgründen hier aus. Aus den Checks ist ja klargeworden, was die Funktion leisten soll.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
    ( ( define ( list-of-sums-with-accu lst accu )
        ( cond
          [  ( empty? lst )  ( list accu )  ]
          [  else  ( cons accu
                           ( list-of-sums-with-accu
                             ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
    ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Erinnerung: Abschnitt „Definitionen verstcken“ in Kapitel 04a.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
    ( ( define ( list-of-sums-with-accu lst accu )
      ( cond
        [  ( empty? lst )  ( list accu )  ]
        [  else  ( cons accu
                         ( list-of-sums-with-accu
                           ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
    ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Hier definieren wir die lokale Hilfsfunktion, die neben einer Liste von Zahlen noch einen Akkumulator als zweiten Parameter hat. In diesem Beispiel ist der Akkumulator zahlenwertig, das muss er aber nicht zwingend sein.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
    ( ( define ( list-of-sums-with-accu lst accu )
       ( cond
         [  ( empty? lst )  ( list accu )  ]
         [  else  ( cons accu
                          ( list-of-sums-with-accu
                            ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
    ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Der Ausdruck, der den Rückgabewert des gesamten local-Ausdrucks und damit den Rückgabewert der Funktion list-of-sums definiert, ist einfach nur ein Aufruf dieser Hilfsfunktion mit einer Liste und einem geeigneten initialen Wert für den Akkumulator. In der beispielhaften Abarbeitung haben wir schon gesehen, dass die Restliste der Eingabeliste die geeignete initiale Liste und das erste Element der Eingabeliste der geeignete initiale Wert ist.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
   ( ( define ( list-of-sums-with-accu lst accu )
      ( cond
       [  ( empty? lst )  ( list accu )  ]
       [  else  ( cons accu
                        ( list-of-sums-with-accu
                          ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
  ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

In jedem rekursiven Aufruf der Hilfsfunktion list-of-sum-with-accu ist vorne ein Element weniger im Parameter list der Hilfsfunktion. Betrachten Sie einen beliebigen rekursiven Aufruf, sagen wir den 357-ten. dann sind die ersten 357 Elemente der Eingabeliste schon nicht mehr im Parameter list dabei (es sind 357, nicht 356, denn beim allerersten Aufruf in der letzten Zeile war ja schon ein Element nicht mehr dabei).

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
    ( ( define ( list-of-sums-with-accu lst accu )
       ( cond
         [  ( empty? lst )  ( list accu )  ]
         [  else  ( cons accu
                          ( list-of-sums-with-accu
                            ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
    ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Der entscheidende Punkt ist, dass accu bei diesem Aufruf die Summe der ersten 357 Elemente enthält.

Genau das ist die schon formulierte Invariante: Bei jedem rekursiven Aufruf von list-of-sum-with-accu enthält accu immer die Summe all derjenigen Elemente von input-list, die nicht mehr in list sind. Daraus ergibt sich zwingend, dass der Akkumulator mit dem ersten Element von input-list initialisiert werden muss, denn das ist das eine Element von input-list, das nicht in der Restliste ist.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
    ( ( define ( list-of-sums-with-accu lst accu )
        ( cond
          [ ( empty? lst ) ( list accu ) ]
          [ else ( cons accu
                       ( list-of-sums-with-accu
                         ( rest lst ) ( + ( first lst ) accu ) ) ) ] ) ) )
    ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Wie wir schon bei der Abarbeitung des Beispiels gesehen haben, ist bei einer leeren Teilliste der Wert des Akkumulators als einelementige Liste zurückzuliefern.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
   ( ( define ( list-of-sums-with-accu lst accu )
      ( cond
       [  ( empty? lst )  ( list accu )  ]
       [  else  ( cons accu
                      ( list-of-sums-with-accu
                        ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
  ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Wir haben ebenfalls in der Abarbeitung des Beispiels gesehen, dass vorne immer der Wert des Akkumulators anzuhängen ist.

Beispiel mit Akkumulator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: list-of-sums: ( list of number ) -> ( list of number )
;; Precondition: input-list is not empty
;; Returns: .....
( define ( list-of-sums input-list )
  ( local
    ( ( define ( list-of-sums-with-accu lst accu )
       ( cond
         [  ( empty? lst )  ( list accu )  ]
         [  else  ( cons accu
                          ( list-of-sums-with-accu
                            ( rest lst ) ( + ( first lst ) accu ) ) )  ]  ) ) )
    ( list-of-sums-with-accu ( rest input-list ) ( first input-list ) ) ) )
```

Hier schließlich wird der Akkumulator für den nächsten rekursiven Aufruf so berechnet, wie wir das in der Abarbeitung des Beispiels gesehen hatten. Die Invariante bleibt erhalten: Ein Element geht aus der Liste ´raus, und sein Wert wird auf den Akkumulator addiert, der Akkumulator enthält also weiterhin die Summe aller nicht mehr in der Liste vorhandenen Elemente der Eingabeliste.

Damit ist dieser Abschnitt und das ganze Kapitel beendet.