



Hausaufgabe 8 – Programmieren mit Rekursiven Datentypen

Bearbeiten Sie dieses Hausaufgabenblatt alleine. Stellen Sie dabei sicher, dass Sie die gesetzten Lernziele erreichen.

Verwenden Sie wieder die Sprache BSL (How to Design Programs – Anfänger).

Viel Erfolg!

Lernziele

Durch diese Hausaufgabe lernen Sie

- ✓ das Entwurfsrezept für rekursive Datentypen anzuwenden
- ✓ mit rekursiven Datentypen zu programmieren
- ✓ mit Listen und Baumstrukturen zu programmieren

Nach dieser Übung sollen Sie in der Lage sein, selbstständig Funktionen zu programmieren, die mit rekursiven Datentypen arbeiten.

Aufgabe 1: Programmieren mit Listen

Zur Implementierung der folgenden Aufgaben, benutzen Sie aus Abschnitt 1.8 der BSL-Definition (<http://docs.racket-lang.org/htdp-langs/beginner.html>) nur die Funktionen `list`, `empty?`, `cons`, `cons?`, `first`, `rest` und die Konstante `empty`. Selbstverständlich können Sie ihre selbst definierten Funktionen in der Implementierung von anderen Funktionen wiederverwenden.

Tipp: Definieren Sie sich zunächst Konstanten mit verschiedenen Beispiellisten, die Sie dann in Tests verwenden können.

Programmieren Sie die folgenden Funktionen auf Listen. Gehen Sie dabei nach dem Entwurfsrezept für rekursive Datentypen vor.

a.

```
; List-Of-Numbers -> Number
; multiplies all numbers contained in the
; list, returns 1 if the list is empty.
(define (product lst) ...)
```

b.

```
; List-Of-Numbers -> Boolean
; returns whether all numbers in the list
; are positive.
(define (all-positive? lst) ...)
```

c.

```
; Number List-Of-Numbers -> Boolean
; determines whether all elements in a list are greater than or equal a given number.
(define (all-greater-than? num lst) ...)
```

d.

```
; List-Of-Numbers -> Boolean
; determines whether a given list of numbers is sorted in ascending order.
(define (sorted? lst) ...)
```

Bewertungsschema (Gesamtpunktzahl 38 pt)

Für jede der sechs Teilaufgaben:

- ☐ 3pt Die Funktion wurde richtig implementiert. Es wurden nur die erwähnten sowie selbst definierte Funktionen verwendet.

Auf die ganze Aufgabe bezogen:

- ☐ 10pt Es sind sinnvolle Tests mit repräsentativen Eingaben vorhanden, die Testabdeckung ist gut.
- ☐ 10pt Die Funktionen sind gut strukturiert, wie es von der Anwendung des Entwurfsrezeptes und DRY erwartet wäre.

e. `; List-Of-Numbers List-Of-Numbers -> Boolean`
`; determines whether two lists contain the exactly same elements in the same order.`
`(define (list=? lst1 lst2) ...)`

f. `; List-Of-Numbers -> List-Of-Numbers`
`; returns a list with the same elements, but every element is incremented by 1.`
`(define (add-one/list lst) ...)`

Aufgabe 2: Listen als Mengen

Listen können auch als Mengen aufgefasst werden, indem Elemente, die doppelt vorkommen, sowie die Ordnung der Elemente ignoriert werden. Die Liste `(list 1 3 1 4 3)` repräsentiert in dieser Interpretation die Menge `{1, 3, 4}`.

Verwenden Sie nur die in Aufgabe 1 erwähnten sowie selbst programmierte Funktionen und programmieren Sie die folgenden Funktionen:

a. `; List-Of-Numbers List-Of-Numbers -> Boolean`
`; determines whether all elements in lst1`
`; are also contained in lst2`
`(define (subset? lst1 lst2) ...)`

b. `; List-Of-Numbers List-Of-Numbers -> Boolean`
`; determines whether lst1 and lst2 contain`
`; the same elements`
`(define (set=? lst1 lst2) ...)`

c. `; List-Of-Numbers List-Of-Numbers -> List-Of-Numbers`
`; returns a set, containing the elements that occur in lst1, but not in lst2`
`(define (difference lst1 lst2) ...)`

Bewertungsschema (Gesamtpunktzahl 19 pt)

Für jede der drei Teilaufgaben:

☐ 3pt Die Funktion wurde richtig implementiert. Es wurden nur die in Aufgabe 1 erwähnten sowie selbst definierte Funktionen verwendet.

Auf die ganze Aufgabe bezogen:

☐ 5pt Es sind sinnvolle Tests mit repräsentativen Eingaben vorhanden, die Testabdeckung ist gut.

☐ 5pt Die Funktionen sind gut strukturiert, wie es von der Anwendung des Entwurfsrezeptes und DRY erwartet wäre.

Aufgabe 3: Programmieren mit Bäumen

Bäume als rekursive Datenstruktur

```
(define-struct node (left value right))
(define-struct empty-tree ())

; An EmptyTree is a structure (make-empty-tree))
; interp. an empty tree
(define EMPTY (make-empty-tree))

; A Tree-Of-Numbers is one of:
; - (make-node Tree-Of-Numbers Number Tree-Of-Numbers)
; - EmptyTree
; interp. a tree with numbers stored at its nodes.

; Example 1.
;      6
;     / \
;    42  3
;   /  / \
;  8 15  2
(define EX-TREE
  (make-node (make-node (make-node EMPTY 8 EMPTY)
                        42
                        EMPTY)
            6
            (make-node (make-node EMPTY 15 EMPTY)
                        3
                        (make-node EMPTY 2 EMPTY))))
```

Voraussetzung für diese Aufgabe ist wieder die Definition von Bäumen aus dem Tutorium (siehe Kasten "Bäume als rekursive Datenstruktur").

Programmieren Sie die folgenden Funktionen:

- ```
; Tree-Of-Numbers -> Number
; sums all numbers in a tree of numbers.
(define (sum/tree tree) ...)
```
- ```
; Tree-Of-Numbers -> Number
; multiplies all numbers in a tree.
(define (mul/tree tree) ...)
```
- ```
; Tree-Of-Numbers -> Tree-Of-Numbers
; returns the same tree, but with 1 added
; to every element.
(define (add-one/tree tree) ...)
```

### Bewertungsschema (Gesamtpunktzahl 25 pt)

Für jede der drei Teilaufgaben:

- ☐ 3pt Die Funktion wurde richtig implementiert.

Auf die ganze Aufgabe bezogen:

- ☐ 8pt Es sind sinnvolle Tests mit repräsentativen Eingaben vorhanden, die Testabdeckung ist gut.
- ☐ 8pt Die Funktionen sind gut strukturiert, wie es von der Anwendung des Entwurfsrezeptes und DRY erwartet wäre.

## Aufgabe 4: Arithmetische Ausdrücke auswerten

Gegeben seien folgende Struktur und Datendefinitionen, die den abstrakten Syntaxbaum arithmetischer Ausdrücke repräsentieren.

```
(define-struct literal (value))
(define-struct addition (lhs rhs))
(define-struct subtraction (lhs rhs))
(define-struct multiplication (lhs rhs))
(define-struct division (lhs rhs))

; An Expression is one of:
; - (make-literal Number)
; - (make-addition Expression Expression)
; - (make-subtraction Expression Expression)
; - (make-multiplication Expression Expression)
; - (make-division Expression Expression)
; interp. abstract syntax tree of arithmetic
; expressions.
```

1. Programmieren Sie mindestens drei Konstantendefinitionen mit Beispielen für `Expression`, mit Namen `EXPRESSION-1`, `EXPRESSION-2`, ...
2. Folgen Sie dem Entwurfsrezept für rekursive Datentypen und implementieren Sie die Funktion

```
; Expression -> Number
; computes the numeric value of an arithmetic expression
(define (evaluate expr) ...)
```

Vergessen Sie nicht ausreichend Tests zu schreiben, bevor Sie mit der Implementierung beginnen. Sie können für die Tests die Konstanten aus dem ersten Aufgabenteil verwenden.

### Bewertungsschema (Gesamtpunktzahl 18 pt)

- ☐ 6 pt Es sind mindestens drei Beispiele für `Expression` angegeben; min. einer der Beispielausdrücke hat eine Schachtelungstiefe von min. 3
- ☐ 12 pt Die Funktion `evaluate` ist richtig implementiert (6pt), es wurde dem Entwurfsrezept (4pt) gefolgt und es sind mindestens drei sinnvolle Tests vorhanden (2pt).