Übungsblatt 1

Diana Beldiman 1513454
Santino Nobile 1414417
Marco Präg 1613370

Aufgabe 1: Methoden zur Listenmanipulation

(a) Elemente tauschen:

```
(defun rotiere (liste)
(append (cdr liste) (list(car liste))))
```

(b) Element einfügen:

```
(defun neuesvorletztes (liste element)
(setq neueliste (reverse liste))
(reverse (append (list(car neueliste)) (list element) (cdr neueliste))))
```

(c) Länge einer Liste berechnen:

```
(defun my-length (liste)
(cond
  ((null liste) 0)
   ((null (cdr liste)) 1)
    (T(+ 1 (my-length (cdr liste))))))
```

(d) Länge einer geschachtelten Liste berechnen:

```
(defun my-lengthR (liste)
 (labels ((helper (liste counter)
          (cond
           ((null liste) counter)
              ((listp (car liste)) (helper (append (car liste) (cdr liste)) counter))
             (T(helper (cdr liste) (+ counter 1))))))
  (helper liste 0)))
```

Version ohne Hilfsfunktion.

```
(defun my-lengthR (liste)
  (cond
   ((null liste) 0)
         ((listp (car liste)) (my-lengthR (append (car liste) (cdr liste))))
      (T(+ 1 (my-lengthR (cdr liste))))))
```

- Wir gehen jedes Element durch und schauen ob es eine Liste ist. Falls nicht, wird der „counter“ um 1 erhöht. Falls ja, wird das erste Element der Liste mit dem Rest der ursprünglichen Liste übergeben ohne den „counter“ zu erhöhen.

- geprüft auf :
  CL-USER 25 : 5 > (my-lengthR '(((1 2))(3 4)))
  4

  CL-USER 26 : 5 > (my-lengthR '(1 2(3 4)5 6))
  6

  CL-USER 27 : 5 > (my-lengthR '((((((((((1 (((2)))))))))))((3))(1 2)))
  5

  CL-USER 30 : 5 > (my-lengthR '(eins zwei (zwei (zwei drei) eins) drei vier))
  8

(e) Listen umkehren:

```
(defun my-reverse (liste)
 (labels ((helper (liste resultList)
          (cond
           ((null liste) resultList)
            (T(helper (cdr liste) (cons (car liste) resultList))))))
  (helper li '())))
```

Version ohne Hilfsfunktion und mit „append“.

```
(defun my-reverse (liste)
 (cond
  ((null liste) '() )
   (T (append (my-reverse (cdr liste)) (list (car liste))))))
```

(f) Geschachtelte Listen umkehren:

```
(defun my-reverseR (liste)
 (cond
  ((null liste) '() )
   ((listp (car liste)) (append (my-reverseR (cdr liste)) (list (my-reverseR (car liste)))))
    (T (append (my-reverseR (cdr liste)) (list (car liste))))))
```

```
CL-USER 20 : 4 > (my-reverseR '(((1))2 4((3 ((8))))))
((((#) 3)) 4 2 ((1)))

CL-USER 21 : 4 > *print-level*
4

CL-USER 22 : 4 > (setf *print-level* 7)
7

CL-USER 23 : 4 > (my-reverseR '(((1))2 4((3 ((8))))))
(((((8)) 3)) 4 2 ((1)))
```

Quelle:
https://stackoverflow.com/questions/14683101/incomplete-output-in-lisps-list-given-with-lists-deepness-over-four

Aufgabe 2: ADT BinaryTree
(a) Darstellung eines Binärbaums:

Ein binärer Suchbaum ist eine Liste mit 3 Elementen. Die Liste beginnt mit dem Vaterknoten gefolgt von zwei Kindknoten. Die Reihenfolge in der Liste ist dann (Vater LinkesKind RechtesKind). Da die Kinder auch Väter sein können, setzt sich dieser Aufbau rekursiv fort.

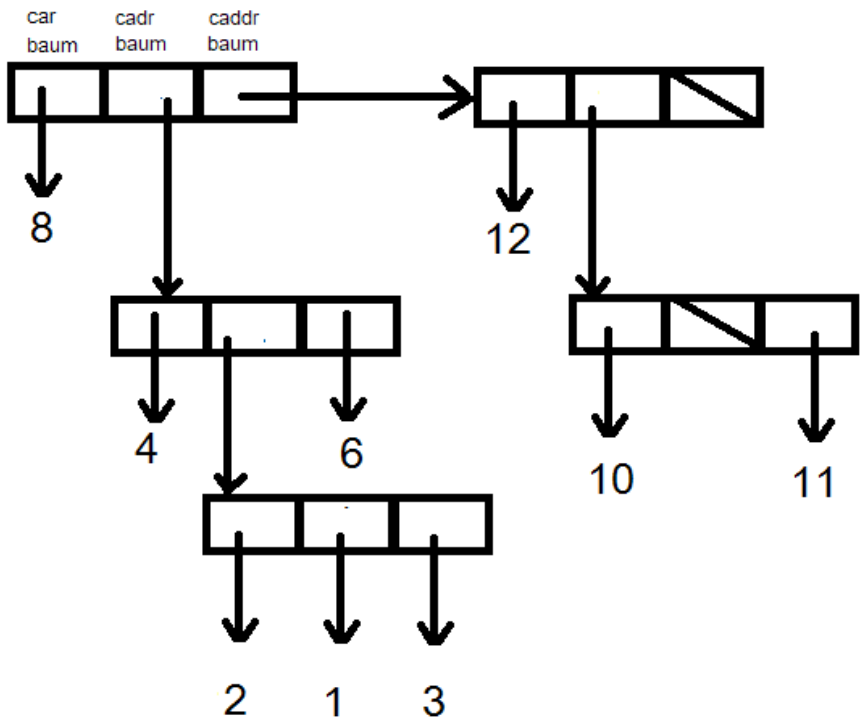
Bsp LinkesKind ist ein Vater:
(Vater (Vater LinkesKind RechtesKind) RechtesKind)

(V L R)

- (setq baum ' (8 (4 (2 (1 nil nil) (3 nil nil)) (6 nil nil)) (12 (10 nil (11 nil nil)) nil)))

(4 (2 (1 nil nil) (3 nil nil)) (6 nil nil)) (12 (10 nil (11 nil nil)) nil))

- (setq baum ' (8 (4 (2 (1) (3)) (6)) (12 (10 () (11)))))



(b) Baumtraversierung:

```
; (car(cdr baum)) => linkes Kind
; (car(cdr(cdr baum))) => rechtes Kind

(defun preorder (baum)
 (cond
   ((null baum) 'KEINE-ELEMENTE)
   (T (princ (car baum)) (princ " ")
      (preorder (car(cdr baum)))
      (preorder (car(cdr(cdr baum))))))

(defun postorder (baum)
 (cond
   ((null baum) 'KEINE-ELEMENTE)
   (T (postorder (car(cdr baum)))
      (postorder (car(cdr(cdr baum))))
      (princ (car baum))(princ " "))))

(defun inorder (baum)
 (cond
   ((null baum) 'KEINE-ELEMENTE)
   (T (inorder (car(cdr baum)) )
      (princ (car baum)) (princ " ")
      (inorder (car(cdr(cdr baum)))))))
```