Aufgaben – Lösungen

Einführung in die Programmierung

Johannes Brauer

10. März 2022

Inhaltsverzeichnis

1 Erste Schritte in Racket

Machen Sie sich mit den NORDAKADEMIE-Rechnern vertraut und richten Sie Ihren Arbeitsplatz ein (Mail, Webbrowser, Verzeichnisse für die Vorlesungen usw.).

Finden und starten Sie DrRacket nach den Anweisungen in der Vorlesung. Werten Sie einen ersten Ausdruck aus, z. B. (* 6 7).

Welche Funktion haben die Buttons? Welche Menü-Befehle verstehen Sie schon?

Schauen Sie sich in einem Webbrowser die Seiten zu Racket unter `https://racket-lang.org/` an. Wo finden Sie Hilfe zur Bedienung von DrRacket? Wie können Sie sich über die Sprache Racket informieren? Wo finden Sie alle vordefinierten mathematischen Funktionen?

2 Auswertung arithmetischer Ausdrücke

a. Wie wird der Ausdruck
`(* (+ 2 2) (/ (* (+ 3 5) (/ 30 10)) 2))`
ausgewertet?

b. Experimentieren Sie mit verschiedenen Operatoren und Zahlenarten.

c. Werten Sie die folgenden Ausdrücke aus und vergleichen Sie die Resultate:

```
(- 1.0 0.9)
(- 1000.0 999.9)
(- #i1000.0 #i999.9)
```

3 Aufschreiben elementarer Funktionen

Schreiben Sie für die folgenden mathematischen Formeln Racket-Funktionsdefinitionen auf:

a. $n^2 + 1$

```
(define fa
  (lambda [n]
    (+ (* n n) 1)))
```

b. $\frac{1}{2}n^2 + 3$

```
(define fb
  (lambda [n]
    (+ (/ (* n n) 2) 3)))
```

c. $2 - \frac{1}{n}$

```
(define fc
  (lambda [n]
    (- 2 (/ 1 n))))
```

Geben Sie die Racket-Funktionen in das Definitionsfenster von DrRacket ein. Geben Sie anschließend in das Interaktionsfenster Funktionsaufrufe für diese Funktionen ein.

4 Aufschreiben elementarer Funktionen (2)

In der Praxis findet der Programmierer selten mathematische Formeln vor. Aufgabenstellungen sind eher als Prosatext gegeben. Die Berechnungsformeln muss er selbst entwickeln durch

- eigenes Nachdenken,
- Nachschlagen in geeigneten Quellen oder
- Nachfragen beim Auftraggeber.

Finden Sie für die folgenden Aufgabenstellungen die passenden Formeln und schreiben Sie diese als Funktionsdefinitionen in *Racket* auf:

a. Berechnung des Rauminhalts eines Quaders aus dessen Länge, Breite und Höhe.

```
(define quader-volumen
  (lambda [laenge breite hoehe]
    (* laenge breite hoehe)))
```

b. Schreiben Sie eine Funktion, die aus der Entfernung und den Geschwindigkeiten zweier Züge die Zeit ermittelt, nach der die Züge sich treffen, wenn Sie sich auf einem gemeinsamen Streckenabschnitt von ihren jeweiligen Startpunkten aus aufeinander zu bewegen.

```
(define zug-treffen
  (lambda [entfernung geschwndgkt1 geschwndgkt2]
    (/ entfernung (+ geschwndgkt1 geschwndgkt2))))
```

c. Berechnung der Miete, die eine Spielerin in Monopoly bezahlen muss, falls sie auf einen Bahnhof trifft, der einer anderen Spielerin gehört. Die Miete ist davon abhängig wie viele Bahnhöfe der anderen Spielerin gehören:

Anzahl der Bahnhöfe	Miete
1	500
2	1000
3	2000
4	4000

Hinweis: Ein Aufruf `(expt x y)` liefert x^y als Ergebnis.

```
(define bahnhofsmiete
  (lambda [anzahl-bahnhoefe]
    (* 500 (expt 2 (- anzahl-bahnhoefe 1)))))
```

5 Anwenden der Aufschreibregeln

Schreiben Sie die Funktion zur Berechnung der Bahnhofsmiete in Monopoly (s. o.) gemäß den Regeln 1 bis 3 aus der Vorlesung auf.

```
;; berechnet die Bahnhofsmiete aus der Anzahl der Bahnhöfe
(define bahnhofsmiete
  (lambda [anzahl-bahnhoefe]
    (* 500 (expt 2 (- anzahl-bahnhoefe 1)))))
;; Beispielanwendungen
(= (bahnhofsmiete 4) 4000)
(= (bahnhofsmiete 3) 2000)
(= (bahnhofsmiete 2) 1000)
(= (bahnhofsmiete 1)  500)
```

Wenn nichts anderes angeben ist, sind auch die Funktionen für die folgenden Aufgaben gemäß diesen Regeln aufzuschreiben!!!

6 Profit für den Kinobesitzer

Ein altmodisches Vorstadtkino besitzt eine einfache Formel für die Berechnung des Profits einer Vorstellung: Jeder Kinobesucher bezahlt 500 Währungseinheiten für die Eintrittskarte. Jede Vorstellung kostet das Kino 2000 Währungseinheiten plus 50 Währungseinheiten pro Besucher. Schreiben Sie eine Funktion `profit` zur Berechnung des Profits bei gegebener Besucherzahl.

```
;; berechnet den Profit eine Filmvorfuehrung aus 2000 WE Fixkosten
;; und 50 WE Kosten pro Besucher bei gegebener Besucherzahl und
;; 500 WE Kartenpreis
(define profit
  (lambda [besucherzahl]
    (- (* 500 besucherzahl)
       (+ (* 50 besucherzahl) 2000))))
;; Beispielanwendungen
(= (profit 100) 43000)
```

7 Modifikation von `kino.rkt`

a. Modifizieren Sie das Programm `kino.rkt` so, dass die Fixkosten einer Veranstaltung wegfallen und dafür 15 Währungseinheiten pro Besucher an Kosten anfallen.

b. Nehmen Sie die gleiche Modifikation auch an der Funktion `profit` aus der Vorlesung vor, die ohne Hilfsfunktionen auskommt, und vergleichen Sie die Ergebnisse.

8 Einsatz von Hilfsfunktionen

Die folgenden Aufgaben sind unter Benutzung von Hilfsfunktionen zu lösen. Befolgen Sie unbedingt alle in der Vorlesung angegebenen Regeln:

a. Schreiben Sie ein Programm, das das Volumen eines Zylinders zu berechnen erlaubt. Eingangsgrößen sind der Radius und die Höhe des Zylinders.

b. Schreiben Sie ein Programm, das die Oberfläche eines Zylinders zu berechnen erlaubt. Eingangsgrößen sind der Radius und die Höhe des Zylinders.

c. Schreiben Sie ein Programm, das die Oberfläche eines Rohrs zu berechnen erlaubt. Eingangsgrößen sind der Innenradius, die Wandstärke und die Länge des Rohrs.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; Lösung von Simon Greßmann, I18b ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;Aufgabe 8a
(define mein-pi 3.14)
;Berechnet Kreisfläche aus Radius
(define Kreisflaeche
  (lambda [radius]
    (* radius radius mein-pi)))
(= (Kreisflaeche 1) 3.14)
(= (Kreisflaeche 2) 12.56)
;Berechnet das Volumen eines Zylinders in Abhängigkeit von Radius und Höhe
(define Zylindervolumen
  (lambda [willi hoehe]
    (* (Kreisflaeche willi) hoehe)))
(= (Zylindervolumen 1 1) 3.14)

;Aufgabe 8b
;Berechnet den Umfang eines Kreises in Anhängigkeit von Pi.
(define Kreisumfang
```

```
  (lambda [radius]
    (* 2 mein-pi radius)))
(= (Kreisumfang 1) 6.28)

;Berechnet die Mantelfläche eines Zylinders in Abhängigkeit von Radius
;und Höhe
(define Mantelflaeche
  (lambda (radius hoehe)
    (* (Kreisumfang radius) hoehe)))
(= (Mantelflaeche 1 1) 6.28)
(= (Mantelflaeche 2 1) 12.56)

;Berechnet die Oberfläche eines Zylinders in Abhängigkeit von Radius und Höhe
(define Zylinderoberflaeche
  (lambda [radius hoehe]
    (+ (* 2 (Kreisflaeche radius)) (Mantelflaeche radius hoehe))))
(= (Zylinderoberflaeche 1 1) 12.56)

;Aufgabe 8c
;Berechnet die Fläche eines Kreisrings in Abhängigkeit von innenradius und
;Breite
(define Kreisringflaeche
  (lambda [innenradius breite]
    (- (Kreisflaeche (+ innenradius breite)) (Kreisflaeche innenradius))))
(= (Kreisringflaeche 1 1) 9.42)
;Berechnet die Oberfläche eines Rohrs in Abhängigkeit von dessen Inneradius,
;Wandstärke und Länge.
(define Rohroberflaeche
  (lambda [innenradius wandstaerke laenge]
    (+ (Mantelflaeche (+ innenradius wandstaerke) laenge)
       (Mantelflaeche innenradius laenge)
       (* 2 (Kreisringflaeche innenradius <dstaerke)))))
(= (Rohroberflaeche 1 1 1) 37.68)
```

9 Ersetzungsmodell

Gegeben sei die folgende Funktionsdefinition:

```
(define f
   (lambda [x y]
       (+ (* 3 x) (* y y))))
```

Werten Sie die folgenden Ausdrücke Schritt für Schritt unter Anwendung des Ersetzungsmodells aus:

a. `(f 1 (* 2 3))`

```
(f 1 (* 2 3))
= ((lambda [x y] (+ (* 3 x) (* y y))) 1 (* 2 3))
= ((lambda [x y] (+ (* 3 x) (* y y))) 1 6)
= (+ (* 3 1) (* 6 6))
= (+ 3 36)
= 39
```

b. `(+ (f 1 2) (f 2 1))`

```
(+ (f 1 2) (f 2 1))
= (+ ((lambda [x y] (+ (* 3 x) (* y y))) 1 2)
     ((lambda [x y] (+ (* 3 x) (* y y))) 2 1))
= (+ (+ (* 3 1) (* 2 2))
     (+ (* 3 2) (* 1 1)))
```

```
= (+ (+ 3 4) (+ 6 1))
= (+ 7 7)
= 14
```

Ausführlich:

```
(+ (f 1 2) (f 2 1))
= (+ ((lambda [x y] (+ (* 3 x) (* y y))) 1 2)
     (f 2 1))
= (+ ((lambda [x y] (+ (* 3 x) (* y y))) 1 2)
     ((lambda [x y] (+ (* 3 x) (* y y))) 2 1))
= (+ (+ (* 3 1) (* 2 2))
     (+ ((lambda [x y] (+ (* 3 x) (* y y))) 2 1))
= (+ (+ (* 3 1) (* 2 2))
     (+ (* 3 2) (* 1 1)))
= (+ (* 3 4) (+ (* 3 2) (* 1 1)))
= (+ (+ 3 4) (+ 6 1))
= (+ 7 (+ 6 1))
= (+ 7 7)
= 14
```

c. `(f (f 1 (* 2 3)) 19)`

```
(f (f 1 (* 2 3)) 19)
= ((lambda [x y] (+ (* 3 x) (* y y))) (f 1 (* 2 3)) 19)
= ((lambda [x y] (+ (* 3 x) (* y y)))
     ((lambda [x y] (+ (* 3 x) (* y y))) 1 (* 2 3)) 19)
= ((lambda [x y] (+ (* 3 x) (* y y)))
     ((lambda [x y] (+ (* 3 x) (* y y))) 1 6) 19)
= ((lambda [x y] (+ (* 3 x) (* y y)))
     (+ (* 3 1) (* 6 6)) 19)
= ((lambda [x y] (+ (* 3 x) (* y y))) (+ 3 36) 19)
= ((lambda [x y] (+ (* 3 x) (* y y))) 39 19)
= (+ (* 3 39) (* 19 19))
= (+ 117 361)
= 478
```

10 Bedingte Funktion

Schreiben Sie ein Programm, das aus dem Bruttoeinkommen eines Arbeitnehmers, das sich aus der Anzahl der Arbeitsstunden und seinem Bruttostundenlohn ergibt, sein Nettoeinkommen durch Abzug der Einkommensteuer berechnet. Die Einkommensteuer wird dabei nach einem steuererklärungaufbierdeckelgeeigneten Tarif ermittelt, der folgendermaßen definiert ist:

Einkommen	**Steuersatz** [%]
$<= 5000$	0
> 5000 *und* ≤ 10000	15
> 10000 *und* ≤ 100000	29
> 100000	64

Der Steuersatz gilt immer nur für die Einkommensanteile in dem jeweiligen Intervall.
Die Funktion `nettoeinkommen` soll nach folgendem Schema aufrufbar sein:

```
(nettoeinkommen anzahl-arbeitsStunden stundenLohn)
```

Hier noch ein paar Testvorgaben:

```
;; Beispielanwendungen
(= (nettoeinkommen 1 5001) 5000.85)
(= (nettoeinkommen 1 10001) 9250.71)
(= (nettoeinkommen 1 100001) 73150.36)
```

Hinweise:

a. Lesen Sie den Aufgabentext aufmerksam durch. Jeder Satz bedeutet etwas.

b. Entwickeln Sie die Funktion gemäß den Regel 1 bis 6. Benutzen Sie Hilfsfunktionen und machen von Variablendefinitionen (benannte Konstanten, Regel 5) Gebrauch.

```
;; Steuertabelle gemäß Aufgabenstellung
(define steuergrenzeI 5000)
(define steuergrenzeII 10000)
(define steuergrenzeIII 100000)

(define steuersatz1 0)
(define steuersatz2 15/100)
(define steuersatz3 29/100)
(define steuersatz4 64/100)

;; feste Steuerbeträge gemäß obiger Tabelle
(define steuern-fuer-steuergrenzeI (* steuersatz1 steuergrenzeI))
(define steuern-fuer-steuergrenzeII
  (+ steuern-fuer-steuergrenzeI
     (* steuersatz2 (- steuergrenzeII steuergrenzeI))))
(define steuern-fuer-steuergrenzeIII
  (+ steuern-fuer-steuergrenzeII
     (* steuersatz3 (- steuergrenzeIII steuergrenzeII))))

;; Ermittlung des Einkommenssteuersatzes aus dem Einkommen gemäß obiger Tabelle
(define steuersatz
  (lambda [einkommen]
    (cond
      [(and (<= einkommen steuergrenzeI) (>= einkommen 0)) steuersatz1]
      [(and (> einkommen steuergrenzeI) (<= einkommen steuergrenzeII)) steuersatz2]
      [(and (> einkommen steuergrenzeII) (<= einkommen steuergrenzeIII)) steuersatz3]
      [(> einkommen steuergrenzeIII) steuersatz4])))
;; Beispielanwendungen
"steuersatz"
(= (steuersatz 2000) 0)
(= (steuersatz steuergrenzeI) 0)
(= (steuersatz 7500) 15/100)
(= (steuersatz steuergrenzeII) 15/100)
(= (steuersatz 50000) 29/100)
(= (steuersatz steuergrenzeIII) 29/100)
(= (steuersatz 1000000) 64/100)

;; Berechnung des Bruttoeinkommens eines Arbeitnehmers
;; der Anzahl der Arbeitsstunden und dem Bruttostundenlohn
(define bruttoeinkommen
  (lambda [arbeitsstunden stundenlohn]
    (* arbeitsstunden stundenlohn)))
;; Beispielanwendungen
"bruttoeinkommen"
(= (bruttoeinkommen 10 5) 50)

;; Ermittlung der Einkommensteuer aus dem einkommen
;; gemäß Steuertarif in Aufgabenstellung
(define einkommensteuer
  (lambda [einkommen]
    (cond
      [(= (steuersatz einkommen) steuersatz1)
       (* steuersatz1 einkommen)]
```

```
      [(= (steuersatz einkommen) steuersatz2)
       (+ steuern-fuer-steuergrenzeI
          (* steuersatz2 (- einkommen steuergrenzeI)))]
      [(= (steuersatz einkommen) steuersatz3)
       (+ steuern-fuer-steuergrenzeII
          (* steuersatz3 (- einkommen steuergrenzeII)))]
      [(= (steuersatz einkommen) steuersatz4)
       (+ steuern-fuer-steuergrenzeIII
          (* steuersatz4 (- einkommen steuergrenzeIII)))])))
;; Beispielanwendungen
"einkommensteuer"
(= (einkommensteuer 2000) 0)
(= (einkommensteuer steuergrenzeI) 0)
(= (einkommensteuer 8000) 450)
(= (einkommensteuer steuergrenzeII) 750)
(= (einkommensteuer 50000) 12350)
(= (einkommensteuer steuergrenzeIII) 26850)
(= (einkommensteuer 100100) (+ 26850 64))

;; Berechnung des Nettolohns eines Arbeitnehmers aus
;; Anzahl Arbeisstunden und Bruttostundenlohn (in Euro)
(define nettoeinkommen
  (lambda [arbeitsstunden stundenlohn]
    (- (bruttoeinkommen arbeitsstunden stundenlohn)
       (einkommensteuer (bruttoeinkommen arbeitsstunden stundenlohn)))))
;; Beispielanwendungen
"nettoeinkommen"
(= (nettoeinkommen 1 5001) (/ 500085 100))
(= (nettoeinkommen 1 10001) (/ 925071 100))
(= (nettoeinkommen 1 100001) (/ 7315036 100))
```

11 Zusatzaufgaben zu bedingten Funktionen

a. Eine Kreditkartengesellschaft gewährt ihren Kunden nach Jahresumsatz gestaffelte Rückerstattung von Kreditkartenbelastungen. Die Rückerstattungen könnten z. B. wie folgt aussehen:

- ein viertel Prozent für die ersten 500€ des Jahresumsatzes (nur Belastungen keine Gutschriften werden gezählt),
- ein halbes Prozent für die nächsten 1000€, d. h. für den Umsatzanteil zwischen 500€ und 1500€,
- ein dreiviertel Prozent für die nächsten 1000€, d. h. für den Umsatzanteil zwischen 1500€ und 2500€ und
- ein Prozent für die Umsatzanteile oberhalb von 2500€.

Ein Kunde mit einem Umsatz von 400€ erhält demnach eine Gutschrift von 1€ ($= \frac{1}{4} \cdot \frac{1}{100} \cdot 400$). Ein Kunde mit einem Umsatz von 1400€ erhält eine Gutschrift von 5,75€:

- 1,25€ ($= \frac{1}{4} \cdot \frac{1}{100} \cdot 500$) für die ersten 500€ plus
- 4,50€ ($= \frac{1}{2} \cdot \frac{1}{100} \cdot 900$) für die nächsten 900€

Lösen Sie die folgenden Teilaufgaben

(a) Bestimmen Sie manuell die Gutschriften für Umsätze 2000€ und 2600€.

(b) Schreiben Sie eine Funktion `rueckerstattung`, die einen Umsatz als Argument akzeptiert und den Rückerstattungsbetrag ermittelt.

```
;; berechnet die Rückerstattung für Umsätze zwischen 0 and 500
(define rueckerstattung-0-500
  (lambda [a]
    (* a (* .25 1/100))))
```

```
;; Beispielanwendungen
(= (rueckerstattung-0-500 400) 1)

;; berechnet die Rückerstattung für Umsätze zwischen 500 and 1500
(define rueckerstattung-500-1500
  (lambda [a]
    (+ (rueckerstattung-0-500 500)
       (* (- a 500) (* .50 1/100)))))
;; Beispielanwendungen
(= (rueckerstattung-500-1500 1400) 5.75)

;; berechnet die Rückerstattung für Umsätze zwischen 1500 and 2500
(define rueckerstattung-1500-2500
  (lambda [a]
    (+ (rueckerstattung-500-1500 1500)
       (* (- a 1500) (* .75 1/100)))))
;; Beispielanwendungen
(= (rueckerstattung-1500-2500 2000) 10.00)

;; berechnet die Rückerstattung für Umsätze von 2500 und höher
(define rueckerstattung-2500+
  (lambda [a]
    (+ (rueckerstattung-1500-2500 2500)
       (* (- a 2500) (* 1 1/100)))))
;; Beispielanwendungen
(= (rueckerstattung-2500+ 2600) 14.75)

;; berechnet Rückerstattungsbetrag für Kreditkarteninhaber
;; bei einem bestimmten Jahresumsatz
(define rueckerstattung
  (lambda [umsatz]
    (cond
      [(<= umsatz 500)
       (rueckerstattung-0-500 umsatz)]
      [(and (> umsatz 500) (<= umsatz 1500))
       (rueckerstattung-500-1500 umsatz)]
      [(and (> umsatz 1500) (<= umsatz 2500))
       (rueckerstattung-1500-2500 umsatz)]
      [else
       (rueckerstattung-2500+ umsatz)])))
;; Beispielanwendungen
(= (rueckerstattung 400) 1)
(= (rueckerstattung 1400) 5.75)
(= (rueckerstattung 2000) 10.00)
(= (rueckerstattung 2600) 14.75)
```

b. Wieviele reelle Lösungen besitzt eine quadratische Gleichung

$$ax^2 + bx + c = 0$$

für beliebige Koeffizienten a, b und c?

(a) Betrachten Sie zunächst nur *echte* quadratische Gleichungen, d. h. es gilt $a \neq 0$

(b) Erweitern Sie die Lösung so, dass auch der Fall $a = 0$ korrekt behandelt wird.

Lösung: Die Lösungen der quadratischen Gleichung

$$ax^2 + bx + c = 0$$

können für den Fall $a \neq 0$ mit der Formel

$$L_q : x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

berechnet werden. Für den Fall $a = 0$ gibt es eine Lösung $-c/b$, vorausgesetzt $b \neq 0$.

```
;; berechnet der Radikanden der Lösungsformel
(define radikand
  (lambda [a b c]
    (- (sqr b) (* 4 a c))))
;: Beispielanwendungen
(= (radikand 1 2 1) 0)
(= (radikand 1 1 1) -3)
(= (radikand 1 3 1) 5)

;; berechnet die Anzahl der Lösungen einer quadratischen Gleichung mit
;; den Koeffizienten a, b und c
(define anzahl-loesungen
  (lambda [a b c]
    (cond
      [(= 0 a) (cond
                 [(= 0 b c) "unendlich viele Lösungen"]
                 [(and (= 0 b) (not (= 0 c))) "keine Lösung"]
                 [else "eine Lösung"])]
      [else (cond [(> (radikand a b c) 0) "zwei Lösungen"]
                  [(= (radikand a b c) 0) "eine Lösung"]
                  [else "keine Lösung"])])))
;; Beispielanwendugen
(string=? (anzahl-loesungen 0 0 0) "unendlich viele Lösungen")
(string=? (anzahl-loesungen 0 0 1) "keine Lösung")
(string=? (anzahl-loesungen 0 1 2) "eine Lösung")
(string=? (anzahl-loesungen 1 4 1) "zwei Lösungen")
(string=? (anzahl-loesungen 1 2 1) "eine Lösung")
(string=? (anzahl-loesungen 4 1 1) "keine Lösung")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Funktion zur Berechnung der Lösungen einer quadratischen Gleichung;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Eine quadgl-loesung ist ein Wert
;;  (make-quadgl-loesung art x1 x2)
;; wobei art die Anzahl der Lösungen durch eine Zeichenkette
;; ("trivial" "keine" "eine" "zwei") repräsentiert wird und
;; x1, x2 ggf. die Lösungen enthalten
(define-struct quadgl-loesung [art x1 x2])

;; berechnet die Lösungen einer quadratischen Gleichung mit
;; den Koeffizienten a, b und c
(define loesungen
  (lambda [a b c]
    (cond
      [(= 0 a) (cond
                 [(= 0 b c)
                    (make-quadgl-loesung "trivial" 0 0)]
                 [(and (= 0 b) (not (= 0 c)))
                    (make-quadgl-loesung "keine" 0 0)]
                 [else (make-quadgl-loesung "eine" (/ (- c) b) 0)])]
      [else (cond [(> (radikand a b c) 0)
                   (make-quadgl-loesung
                       "zwei"
                       (/ (+ (- b) (sqrt (radikand a b c)))
                          (* 2 a))
                       (/ (- (- b) (sqrt (radikand a b c)))
```

```
                        (* 2 a)))]
                  [(= (radikand a b c) 0)
                   (make-quadgl-loesung "eine" (/ (- b) (* 2 a)) 0)]
                  [else (make-quadgl-loesung "keine" 0 0)])]))))
;; Beispielanwendugen
(string=? (quadgl-loesung-art (loesungen 0 0 0)) "trivial")
(string=? (quadgl-loesung-art (loesungen 0 0 1)) "keine")
(and (string=? (quadgl-loesung-art (loesungen 0 1 2)) "eine")
     (= (quadgl-loesung-x1 (loesungen 0 1 2)) -2))
(and (string=? (quadgl-loesung-art (loesungen 1 4 3)) "zwei")
     (= (quadgl-loesung-x1 (loesungen 1 4 3)) -1)
     (= (quadgl-loesung-x2 (loesungen 1 4 3)) -3))
(and (string=? (quadgl-loesung-art (loesungen 1 2 1)) "eine")
     (= (quadgl-loesung-x1 (loesungen 1 2 1)) -1))
(string=? (quadgl-loesung-art (loesungen 4 1 1)) "keine")
```

12 Datenabstraktion

Befolgen Sie für die Lösung der Aufgabe die Regeln 7 und 8!

a. Definieren Sie eine Datenstruktur für „Zeitpunkte seit Mitternacht", die aus den Komponenten `stunden`, `minuten` und `sekunden` besteht.

   Entwickeln Sie eine Funktion `zeit->sekunden`, die eine Zeitpunkt-seit-Mittnacht-Struktur verarbeitet und die seit Mitternacht vergangenen Sekunden berechnet.

b. Definieren Sie geeignete **Datenstrukturen** für Kreise, die durch

   - die Koordinaten des Mittelpunkts und
   - den Radius

   gegeben sind.

   Schreiben Sie eine Funktion, die prüft, ob ein Punkt innerhalb eines Kreises liegt.

```
; Autor der Lösungen zu Aufgabe 12: David Lüder (I16b)
; Aufgabe 12a

; Eine Tageszeit ist ein Wert
; (make-daytime h m s)
; wobei h die Stunden, m die Minuten und s die Sekunden angeben,
; um die Uhrzeit zu definieren.
(define-struct daytime [h m s])

; gibt die absolute Zahl der vergangenen Sekunden seit Mitternacht
; bei gegebeher Tageszeit zurück.
(check-expect (zeit->sekunden (make-daytime 0 0 0)) 0)
(check-expect (zeit->sekunden (make-daytime 0 2 36)) 156)
(check-expect (zeit->sekunden (make-daytime 2 1 50)) 7310)
(check-expect (zeit->sekunden (make-daytime 23 59 59)) 86399)
(define zeit->sekunden
  (lambda [uhrzeit]
    (+ (* (daytime-h uhrzeit) 3600)
       (* (daytime-m uhrzeit) 60)
       (daytime-s uhrzeit))))

; Aufgabe 12b

; Ein Punkt ist ein Wert
; (make-point x y)
; wobei x und y Zahlen sind, welche die Position im Zweidimensionalen
```

```
; angeben.
(define-struct point [x y])

; Ein Kreis ist ein Wert
; (make-circle center radius)
; wobei center ein Punkt ist, welcher den Mittelpunkt darstellt und radius
; eine Zahl ist, welche den Radius des Kreises angibt.
(define-struct circle [center radius])

; berechnet den Abstand zwischen zwei übergebenen Punkten im Zweidimensionalen
(check-expect (point-distance (make-point 1 1) (make-point 1 2)) 1)
(check-expect (point-distance (make-point 7 2) (make-point 2 2)) 5)
(check-within (point-distance (make-point 5 5) (make-point 2 4)) 3.16 0.01)
(define point-distance
  (lambda [p1 p2]
    (sqrt (+ (sqr (- (point-x p1) (point-x p2)))
             (sqr (- (point-y p1) (point-y p2)))))))

; prüft, ob ein Punkt innerhalb eines Kreises liegt, dabei werden der Kreis
; und der Punkt übergeben
(check-expect (in-circle? (make-circle (make-point 10 10) 2)
                          (make-point 10 9)) #true)
(check-expect (in-circle? (make-circle (make-point 7 3) 5)
                          (make-point 2 2)) #false)
(check-expect (in-circle? (make-circle (make-point 9 2) 7)
                          (make-point 8 4)) #true)
(define in-circle?
  (lambda [circle point]
    (<= (point-distance (circle-center circle) point) (circle-radius circle))))
```

c. Definieren Sie eine Datenstruktur für Gäste einer Veranstaltung. Ein Gast besteht aus einer Zeichenkette für den Namen, einem boolschen Wert, der angibt ob es sich um eine Frau handelt, und einem boolschen Wert, der angibt ob es sich um einen Vegetarier handelt.

   Schreiben Sie eine Funktion, die prüft, ob ein Gast ein nicht weiblicher Vegetarier ist.

```
; Ein Gast besteht aus
; - einer Zeichenkette für den Namen
; - einem boolschen Wert, der angibt ob es sich um eine Frau handelt
; - einem boolschen Wert, der angibt ob es sich um einen Vegetarier handelt

(define-struct gast [name weiblich? vegetarier?])

;; prüft, ob ein Gast ein nicht weiblicher Vegetarier ist
(define nicht-weiblicher-vegetarier?
  (lambda [gast]
    (and (not (gast-weiblich? gast)) (gast-vegetarier? gast))))

;; Beispielanwendungen
(nicht-weiblicher-vegetarier? (make-gast "Karl" #false #true))
(not (nicht-weiblicher-vegetarier? (make-gast "Klara" #true #true)))
(not (nicht-weiblicher-vegetarier? (make-gast "Karl" #false #false)))
```

13 Datenabstraktion – Zusatzaufgabe

13.1 Vorbemerkungen

In den Lehrsprachen von DrRacket gibt es ein vordefinierte Strukturdefinition namens `posn` für Punkte in der Ebene, die genauso aufgebaut ist wie die Strukturdefinition `point` aus der Vorlesung. Die damit vordefinierten Funktionen sind

- die Konstruktionsfunktion `make-posn`,
- die Selektionsfunktionen `posn-x` und `posn-y` und
- das Typprädikat `posn?`.

Wenn man über den Menüpunkt `Sprache->Teachpack hinzufügen...` das Teachpack `draw.rkt` auswählt und anschließend den `Start`-Knopf drückt, steht ein Grafikpaket mit den folgenden Funktionen bereit:

`draw-solid-line` erwartet zwei Punkte (`posn`-Strukturen), die den Anfang und das Ende einer Strecke definieren sowie eine Farbe als Argumente

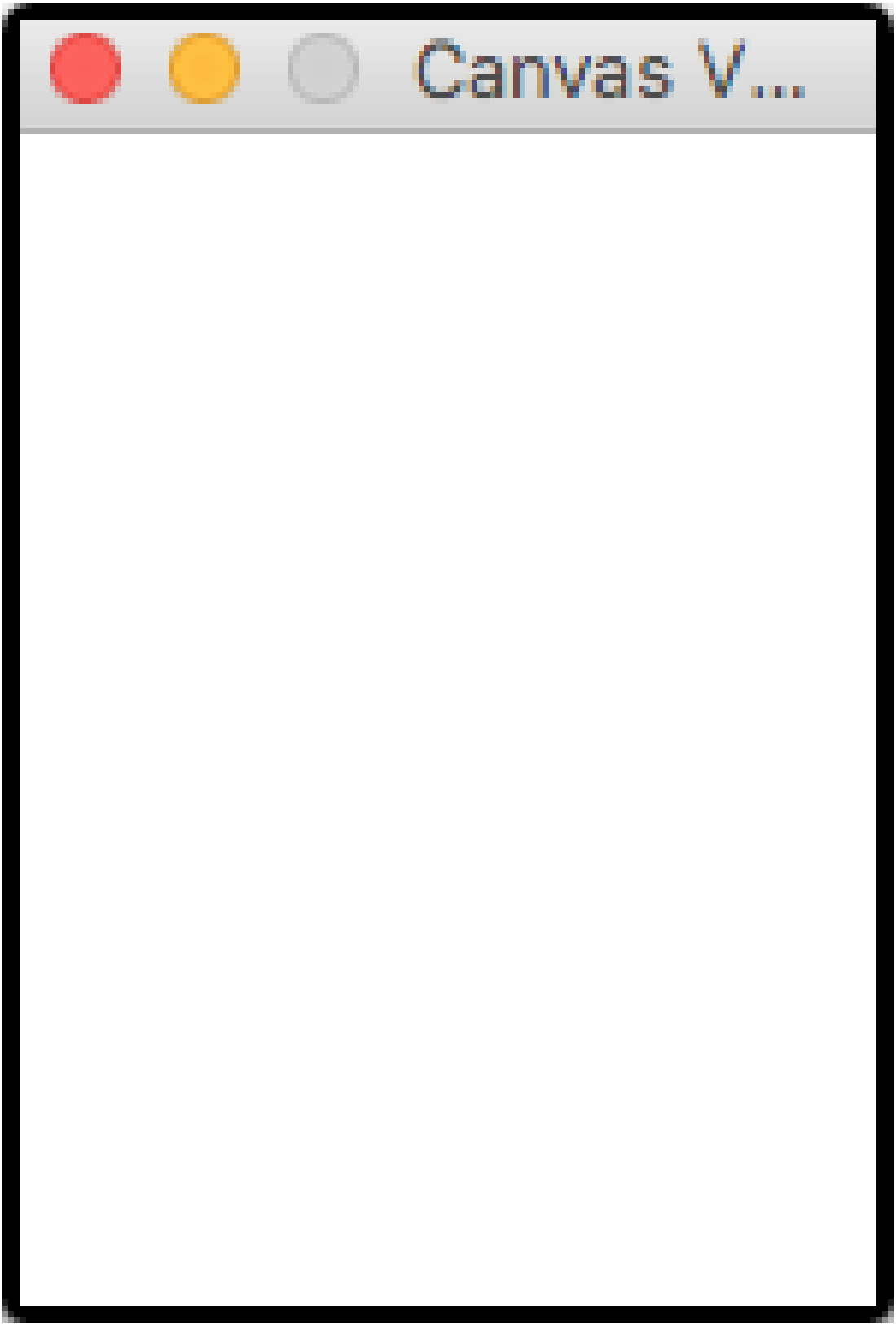
`draw-solid-rect` erwartet vier Argumente: ein Punkt für die linke obere Ecke des Rechtecks, zwei Zahlen für Breite und Höhe des Rechecks und eine Farbe

`draw-solid-disk` erwartet drei Argumente: ein Punkt für den Mittelpunkt, eine Zahl für den Radius der Scheibe und eine Farbe

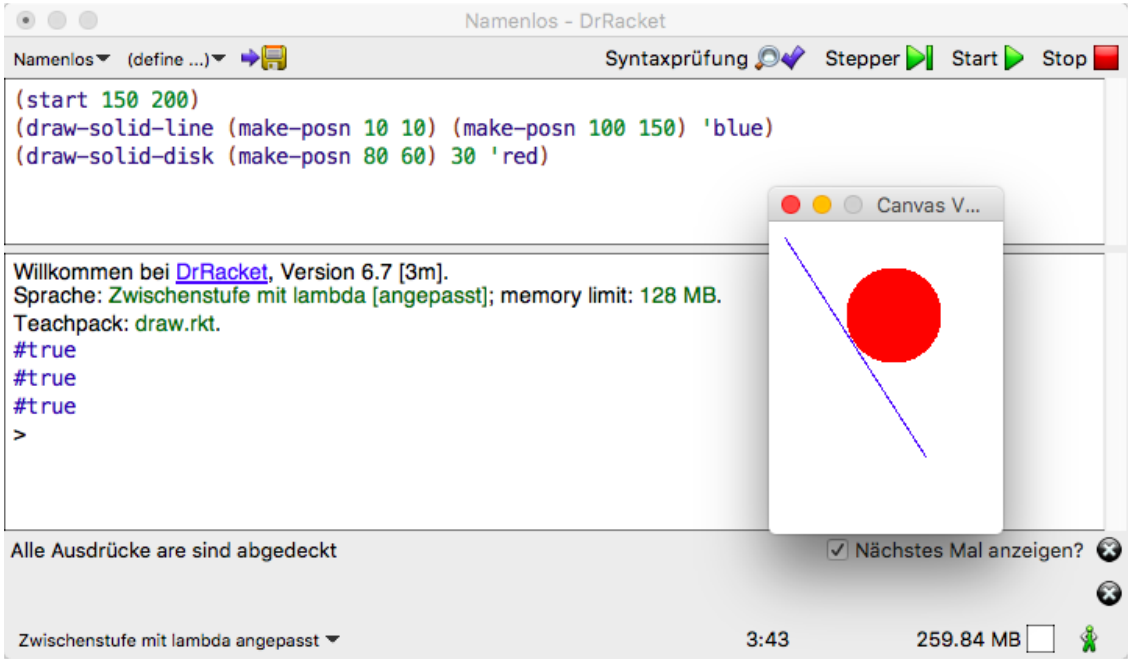
`draw-circle` erwartet drei Argumente: ein Punkt für den Mittelpunkt, eine Zahl für den Radius des Kreises und eine Farbe

Alle Funktionen liefern als Funktionswert `true`, wobei wir in diesem Fall an den Funktionswerten weniger interessiert sind als an ihrem **Effekt**. Der besteht nämlich darin, dass die Prozeduren jeweils eine Strecke, ein Rechteck, eine Scheibe und einen Kreis auf eine zuvor definierte Zeichenfläche zeichnen.

Eine Zeichenfläche kann mit einem Ausdruck `(start x y)` erzeugt werden, wobei `x` und `y` die Breite und die Höhe in Pixeln der Zeichenfläche angeben. Zum Beispiel erzeugt ein Aufruf `(start 150 200)` die folgende Zeichenfläche:



Ein Anwendungsbeispiel zeigt die folgende Abbildung:



Der **Ursprung des Koordinatensystems** liegt in der linken oberen Ecke der Zeichenfläche. Die x-Koordinate zählt nach rechts, die y-Koordinate nach unten positiv:
`http://www.htdp.org/2003-09-26/Book/curriculum1aa-Z-G-17.gif`

Die **Farben** werden durch Symbole wie '`yellow`, '`red`, '`green` angegeben. Bitte beachten Sie das vorangestellte Hochkomma.

Für jede Zeichenoperation gibt es eine korrespondierende **Löschoperation**: `clear-solid-line`, `clear-solid-rect`, `clear-solid-disk` und `clear-circle`. Wenn diese Funktionen mit den gleichen Argumenten wie zuvor die zugehörigen `draw`-Operationen aufgerufen werden, werden die entsprechenden Figuren von der Zeichenfläche entfernt.

Um **mehrere Zeichenoperationen hintereinander** ausführen zu können, d. h. **mehrere Effekte** zu **kombininieren**, macht man davon Gebrauch, dass die einzelnen Zeichenoperationen `#true` als Resultat liefern. Dadurch ist es möglich mehrere Zeichenoperationen hintereinander auszuführen, indem man sie in eine Und-Verknüpfung einschließt:

```
(and exp1 exp2)
```

Dieser Ausdruck bewirkt, dass man zuerst den Effekt von `exp1` und anschließend den von `exp2` zu sehen bekommt.

Mit `(stop)` wird die Zeichenfläche geschlossen.

13.2 Aufgabenstellungen

a. Experimentieren Sie mit den o. g. Funktionen.

b. Kreise und Rechtecke bewegen

   (a) Entwickeln Sie eine Datenstruktur `circle` für farbige Kreise. Diese sollen durch drei Komponenten definiert sein: den Mittelpunkt, den Radius und die Farbe des Umfangs.

   (b) Schreiben Sie die Datendefinition für Kreise und die Funktionsschablone (gemäß Regeln 7 und 8) für Kreise verarbeitende Funktionen.

   (c) Benutzen Sie die Schablone für die Entwicklung einer Funktion `draw-a-circle`. Die Funktion erwartet einen Kreis (`circle`) als Argument und zeichnet ihn auf einer Zeichenfläche. Der Funktionswert sollte `#true` sein.

   (d) Entwickeln Sie eine Funktion `translate-circle` mit einer `circle`-Struktur `c` und einer `posn`-Struktur `delta` als Parameter. Die Funktion liefert einen `circle` als Resultat, dessen Mittelpunkt gegenüber dem von `c` um den x-Wert von `delta` nach rechts und um den y-Wert von `delta` nach unten verschoben ist. Die Funktion hat keinen Effekt auf der Zeichenfläche.

   (e) Schreiben Sie eine Funktion `clear-a-circle`, die einen Kreis von der Zeichenfläche entfernt.

(f) Schreiben Sie eine Funktion `draw-and-clear-circle`, die eine `circle`-Struktur zeichnet, eine kurze Zeit wartet und sie anschließend wieder entfernt. Für die Implementierung der Wartezeit steht die Prozedur `sleep-for-a-while` zur Verfügung. Der Aufruf
`(sleep-for-a-while 3)` erzeugt eine Wartezeit von drei Sekunden. Der Funktionswert ist `#true`.
Die Funktion

```
;; zeichnet und löscht einen Kreis a-circle und
;; bewegt ihn um a-posn
(define move-circle
   (lambda [a-posn a-circle]
      (cond
        [(draw-and-clear-circle a-circle)
           (translate-circle a-circle a-posn)]
        [else a-circle])))
```

zeichnet und löscht einen Kreis auf der Zeichenfläche und erzeugt anschließend einen (verschobenen) Kreis, so dass eine erneute Zeichenoperation den Kreis an einer neuen Position erscheinen lässt.

Zum Beispiel bewegt der Ausdruck

```
(draw-a-circle
 (move-circle
  (make-posn 10 0)
  (move-circle
   (make-posn 10 0)
    (move-circle
     (make-posn 10 0)
     (make-circle (make-posn 10 50) 10 'green)))))
```

einen grünen Kreis dreimal um 10 Pixel nach rechts. Der äußere Aufruf von `draw-a-circle` sorgt dafür, dass auch die letzte Position des Kreises angezeigt wird.

Das sei nur ein kleine Anregung für eigene „Animationen“.

```
;; A circle is a structure:
;;   (make-circle P R C)
;; where P is a posn describing the center of the circle,
;;       R is a number describing the radius of the circle,
;;   and C is a color.
(define-struct circle [center radius color])

#|
;; Funktionsschablone für circle-verarbeitende Funktionen

(define fun-for-circles
  (lambda [a-circle]
    ... (circle-center a-circle) ...
    ... (circle-radius a-circle) ...
    ... (circle-color a-circle) ...)(
    |#

;; draws the disk on the screen
;; no tests, because function always returns true
(define draw-a-circle
  (lambda [c]
    (draw-circle (circle-center c)
                 (circle-radius c)
                 (circle-color c))))

;; EXAMPLES

(start 300 300)
```

```
(draw-a-circle (make-circle (make-posn 50 50) 50 'red))

;; determines if p is inside the circle c.
(check-expect (in-circle? (make-circle (make-posn 6 5) 1 'blue)
                          (make-posn 6 5)) #true)
(check-expect (in-circle? (make-circle (make-posn 6 5) 1 'green)
                          (make-posn 5.5 5)) #true)
(check-expect (in-circle? (make-circle (make-posn 6 5) 1 'yellow)
                          (make-posn 1 5)) #false)
(define in-circle?
  (lambda [c p]
    (<= (sqrt (+ (sqr (- (posn-x (circle-center c))
                         (posn-x p)))
                 (sqr (- (posn-y (circle-center c))
                         (posn-y p)))))
        (circle-radius c))))

;; to translate a-circle by delta (posn)
(check-expect  (translate-circle
                (make-circle (make-posn 0 0) 5 'blue)
                (make-posn 10 5))
               (make-circle (make-posn 10 5) 5 'blue))
(define translate-circle
  (lambda [a-circle delta]
    (make-circle (make-posn
                  (+ (posn-x delta) (posn-x (circle-center a-circle)))
                  (+ (posn-y delta) (posn-y (circle-center a-circle))))
                 (circle-radius a-circle)
                 (circle-color a-circle))))

;; to clear a-circle
;; no tests, because function always returns true
(define clear-a-circle
  (lambda [a-circle]
    (clear-circle
     (circle-center a-circle)
     (circle-radius a-circle))))

;; EXAMPLES
(start 100 100)
(draw-a-circle (make-circle (make-posn 50 50) 25 'red))
(clear-a-circle (make-circle (make-posn 50 50) 25 'red))

;; draw-and-clear : circle -> true
;; to draw a circle, wait 1/2 second, and clear it
(define draw-and-clear-circle
  (lambda [a-circle]
    (and (draw-a-circle a-circle)
         (sleep-for-a-while 1/2)
         (clear-a-circle a-circle))))

;; EXAMPLES
(start 100 100)
(draw-and-clear-circle (make-circle (make-posn 50 50) 50 'blue))

;; zeichnet und löscht einen Kreis a-circle und bewegt ihn um a-posn
;; no tests, because function always returns true
(define move-circle
  (lambda [a-posn a-circle]
```

```
    (cond
      [(draw-and-clear-circle a-circle) (translate-circle a-circle a-posn)]
      [else a-circle])))

(start 60 100)
(draw-a-circle
 (move-circle
  (make-posn 10 0)
  (move-circle
   (make-posn 10 0)
   (move-circle
    (make-posn 10 0)
    (make-circle (make-posn 10 50) 10 'green)))))
```

14 Datenabstraktion – gemischte Daten

Lösen Sie die Aufgabe unter Anwendung der passenden Regeln!

Ein *Mitarbeiter* ist entweder

- ein *Festangestellter* oder
- ein *Werkstudent*

Ein *Festangestellter* wird definiert durch

- seinen Namen,
- sein Grundgehalt,
- die im letzten Monat geleisteten Arbeitsstunden.

Ein *Werkstudent* wird definiert durch

- seinen Namen,
- seinen Stundenlohn,
- die im letzten Monat geleisteten Arbeitsstunden.

Definieren Sie

- geeignete Datenstrukturen für *Mitarbeiter*,
- eine Funktionsschablone für Funktionen, die *Mitarbeiter* verarbeiten.

Entwickeln auf der Grundlage dieser Schablone eine Funktion, die den Bruttomonatslohn eines Mitarbeiters berechnet. Bei *Festangestellten* berechnet sich der Monatslohn aus dem Grundgehalt zuzüglich Überstundenentgelt. Überstunden sind die über die monatliche Sollarbeitszeit (die als globale Konstante definiert wird) hinausgehenden Arbeitsstunden. Der Stundenlohn pro Überstunde berechnet sich aus dem Grundgehalt und der monatlichen Sollarbeitszeit plus 25%. Minderstunden bleiben unberücksichtigt.

```
; Autor der Lösung zu Aufgabe 14: Marlon Tobaben (I16b)

;;Die Konstante für die monatliche sollarbeitszeit eines angestellten
(define sollarbeitszeit 120)

;; Ein mitarbeiter ist entweder
;; - ein festangestellter
;; - ein werksstudent
;; name: mitarbeiter

;; Ein festangestellter ist ein Wert
```

```
;;   (make-festangestellter name grundgehalt arbeitsstunden)
;;  wobei der name ein string und
;;       das grundgehalt und arbeitsstunden Zahlen sind.
(define-struct festangestellter [name grundgehalt arbeitsstunden])

;; Ein werksstudent ist ein Wert
;;   (make-werksstudent name stundenlohn arbeitsstunden)
;;  wobei der name ein string und
;;       das stundenlohn  und arbeitsstunden Zahlen sind.
(define-struct werksstudent [name stundenlohn arbeitsstunden])

;; Die Funktion ueberstunden-festangestellter berechnet die Überstunden eines
;; festangestellten anhand der sollarbeitszeit
(check-expect (ueberstunden-festangestellter 120)  0)
(check-expect (ueberstunden-festangestellter 130) 10)
(check-expect (ueberstunden-festangestellter 150) 30)

(define ueberstunden-festangestellter
  (lambda [arbeitsstunden]
    (cond [(> (- arbeitsstunden sollarbeitszeit) 0)
           (- arbeitsstunden sollarbeitszeit)]
          [else 0])))

;; Funktion bruttolohn-festangesteller berechnet den Bruttolohn eines Festanstellten
;;unter Berücksichtung der sollarbeitszeit und mit 25% zuschlag bei ueberstunden
(check-expect (bruttolohn-festangestellter
               (make-festangestellter "Peter" 3000 120))  3000)
(check-expect (bruttolohn-festangestellter
               (make-festangestellter "Heinz" 3000   0))  3000)
(check-expect (bruttolohn-festangestellter
               (make-festangestellter "Heinz" 3000 240))  6750)
(define bruttolohn-festangestellter
  (lambda [f]
    (+ (festangestellter-grundgehalt f)
       (* (ueberstunden-festangestellter (festangestellter-arbeitsstunden f))
          (* (/ (festangestellter-grundgehalt f) sollarbeitszeit) 1.25)))))

;; Funktion bruttolohn-werkstundent berechnet den Bruttolohn eines werksstundenten
(check-expect (bruttolohn-werksstudent (make-werksstudent "Peter" 10 120))  1200)
(check-expect (bruttolohn-werksstudent (make-werksstudent "Heinz"  0  20))     0)
(check-expect (bruttolohn-werksstudent (make-werksstudent "Heinz" 20 120))  2400)
(define bruttolohn-werksstudent
  (lambda [w]
    (* (werksstudent-stundenlohn w) (werksstudent-arbeitsstunden w))))

;;Funktion bruttolohn-mitarbeiter berechnet den bruttolohn eines mitarbeiters
(check-expect (bruttolohn-mitarbeiter (make-werksstudent "Heinz"  0  20)) 0)
(check-expect (bruttolohn-mitarbeiter (make-festangestellter "Heinz" 3000  0)) 3000)
(define bruttolohn-mitarbeiter
  (lambda [mitarbeiter]
    (cond
      [(festangestellter? mitarbeiter) (bruttolohn-festangestellter mitarbeiter)]
      [(werksstudent?     mitarbeiter) (bruttolohn-werksstudent     mitarbeiter)])))
```

15 Datenabstraktion – gemischte Daten – Zusatzaufgabe

15.1 Vorbemerkung

Diese Aufgabe stellt eine Fortführung von Aufgabe 13 dar. Falls Sie diese noch nicht bearbeitet haben, sollten Sie zunächst damit beginnen.

15.2 Aufgabenstellungen

a. Enwtickeln Sie eine Datenstrukturdefinition für farbige Rechtecke. Ein Rechteck sei durch einen Punkt (`posn`-Struktur), der die linke obere Ecke des Rechtecks bildet, zwei Zahlen für die Höhe und die Breite des Rechecks und ein Symbol für seine Farbe charakterisiert.

b. Entwickeln sie die folgenden Funktionen

`draw-a-rectangle` zeichnet das Rechteck auf die Zeichenfläche. Im Gegensatz zu den Kreisen aus Aufgabe 13 sollen die Rechtecke immer mit der Farbe gefüllt gezeichnet werden.

`in-rectangle?` akzeptiert ein Rechteck und einen Punkt als Argumente und prüft, ob der Punkt innerhalb des Rechecks liegt.

`translate-rectangle` verschiebt ein Rechteck auf die gleiche Art, wie es die Funktion `translate-circle` aus Aufgabe 13 mit Kreisen tut. Diese Funktion hat keinen Effekt auf der Zeichenfläche.

`clear-a-rectangle` entfernt ein Rechteck von der Zeichenfläche.

`move-rectangle` soll analog zur Funktion `move-circle` aus Aufgabe 13 ein Rechteck zeichnen, löschen und mit einem verschobenen Rechteck antworten.

c. Definieren Sie eine gemischte Datenstruktur für Figuren (shapes), die als Generalisierung mindestens Kreise und Rechtecke umfassen sollte. Schreiben Sie die Funktionsschablone für Funktionen, die Figuren verarbeiten, auf.

d. Programmieren Sie nun die oben genannten Funktionen für Figuren (z. B. `draw-a-shape`)

```
;; a shape is either:
;;   - a circle
;;   - a rectangle

;; A circle is a structure:
;;   (make-circle P R C)
;; where P is a posn describing the center of the circle,
;;       R is a number describing the radius of the circle,
;;   and C is a color.
(define-struct circle (center radius color))

;; draws the disk on the screen
;; no tests, because function always returns true
(define draw-a-circle
  (lambda [c]
    (draw-circle (circle-center c)
                 (circle-radius c)
                 (circle-color c))))

;; determines if p is inside the circle c.
(check-expect (in-circle? (make-circle (make-posn 6 5) 1 'blue)
                          (make-posn 6 5)) #true)
(check-expect (in-circle? (make-circle (make-posn 6 5) 1 'green)
                          (make-posn 5.5 5)) #true)
(check-expect (in-circle? (make-circle (make-posn 6 5) 1 'yellow)
                          (make-posn 1 5)) #false)
(define in-circle?
  (lambda [c p]
    (<= (sqrt (+ (sqr (- (posn-x (circle-center c))
```

```
                        (posn-x p)))
                 (sqr (- (posn-y (circle-center c))
                         (posn-y p)))))
        (circle-radius c))))

;; to translate a-circle by delta (posn)
(check-expect  (translate-circle
                (make-circle (make-posn 0 0) 5 'blue)
                (make-posn 10 5))
               (make-circle (make-posn 10 5) 5 'blue))
(define translate-circle
  (lambda [a-circle delta]
    (make-circle (make-posn
                  (+ (posn-x delta) (posn-x (circle-center a-circle)))
                  (+ (posn-y delta) (posn-y (circle-center a-circle))))
                 (circle-radius a-circle)
                 (circle-color a-circle))))

;; to clear a-circle
;; no tests, because function always returns true
(define clear-a-circle
  (lambda [a-circle]
    (clear-circle
     (circle-center a-circle)
     (circle-radius a-circle))))

;; draw-and-clear : circle -> true
;; to draw a circle, wait 1/2 second, and clear it
(define draw-and-clear-circle
  (lambda [a-circle]
    (and (draw-a-circle a-circle)
         (sleep-for-a-while 1/2)
         (clear-a-circle a-circle))))

;; to draw, to clear, and to move a-circle by a-posn
;; no tests, because function always returns true
(define move-circle
  (lambda [a-posn a-circle]
    (cond
      [(draw-and-clear-circle a-circle) (translate-circle a-circle a-posn)]
      [else a-circle])))

;; A rectangle is a structure:
;;   (make-rectangle P W H)
;; where P is a posn, W is a number and H is a number.
(define-struct rectangle (nw-corner width height color))

;; DATA EXAMPLES
(define example-rectangle1 (make-rectangle (make-posn 20 20) 260 260 'red))
(define example-rectangle2 (make-rectangle (make-posn 60 60) 180 180 'blue))

;; to draw a-rect
;; no tests, because function always returns true
(define draw-a-rectangle
  (lambda [a-rectangle]
    (draw-solid-rect
     (rectangle-nw-corner a-rectangle)
     (rectangle-width a-rectangle)
     (rectangle-height a-rectangle)
```

```
     (rectangle-color a-rectangle))))

;; to determine if a-posn is in a-rectangle, or not
(check-expect (in-rectangle? example-rectangle1 (make-posn 0 0)) #false)
(check-expect (in-rectangle? example-rectangle1 (make-posn 25 0)) #false)
(check-expect (in-rectangle? example-rectangle1 (make-posn 0 25)) #false)
(check-expect (in-rectangle? example-rectangle1 (make-posn 25 25)) #true)
(define in-rectangle?
  (lambda [a-rectangle a-posn]
    (and (<= (posn-x (rectangle-nw-corner a-rectangle))
             (posn-x a-posn)
             (+ (posn-x (rectangle-nw-corner a-rectangle))
                (rectangle-width a-rectangle)))
         (<= (posn-y (rectangle-nw-corner a-rectangle))
             (posn-y a-posn)
             (+ (posn-y (rectangle-nw-corner a-rectangle))
                (rectangle-height a-rectangle))))))

;; to translate a-rectangle by delta (posn)
(check-expect (translate-rectangle example-rectangle1 (make-posn 30 0))
              (make-rectangle (make-posn 50 20) 260 260 'red))
(define translate-rectangle
  (lambda [a-rectangle delta]
    (make-rectangle (make-posn
                     (+ (posn-x delta) (posn-x (rectangle-nw-corner a-rectangle)))
                     (+ (posn-y delta) (posn-y (rectangle-nw-corner a-rectangle))))
                    (rectangle-width a-rectangle)
                    (rectangle-height a-rectangle)
                    (rectangle-color a-rectangle))))

;; to erase a rectangle
;; no tests, because function always returns true
(define clear-a-rectangle
  (lambda [a-rectangle]
    (clear-solid-rect
     (rectangle-nw-corner a-rectangle)
     (rectangle-width a-rectangle)
     (rectangle-height a-rectangle))))

;; to draw a circle, wait 1/2 second, and clear it
;; no tests, because function always returns true
(define draw-and-clear-rectangle
  (lambda [a-rectangle]
    (and (draw-a-rectangle a-rectangle)
         (sleep-for-a-while 1/2)
         (clear-a-rectangle a-rectangle))))

;; to draw and clear a rectangle, translate it by delta pixels
;; no tests, because function always returns true
(define move-rectangle
  (lambda [delta a-rectangle]
    (cond
      [(draw-and-clear-rectangle a-rectangle)
       (translate-rectangle a-rectangle delta)]
      [else a-rectangle])))

;;;;;;;;

;; draws a-shape
```

```
(define draw-shape
  (lambda [a-shape]
    (cond
      [(circle? a-shape)
       (draw-a-circle a-shape)]
      [(rectangle? a-shape)
       (draw-a-rectangle a-shape)])))
;; EXAMPLES AS TESTS
(start 200 200)
(draw-shape (make-circle (make-posn 30 30) 20 'red))
(draw-shape (make-rectangle  (make-posn 30 60) 20 50 'blue))

; ---------------------------------------------------------------------------

;; translates a-shape by delta (posn)
(check-expect (translate-shape
                  (make-circle (make-posn 30 30) 20 'red) (make-posn 10 0))
              (make-circle (make-posn 40 30) 20 'red))
(check-expect (translate-shape (make-rectangle (make-posn 30 60) 20 50 'blue)
                               (make-posn 20 0))
              (make-rectangle  (make-posn 50 60) 20 50 'blue))
(define translate-shape
  (lambda [a-shape delta]
    (cond
      [(circle? a-shape)
       (translate-circle a-shape delta)]
      [(rectangle? a-shape)
       (translate-rectangle a-shape delta)])))

;; erases a-shape
;; no tests, because function always returns true
(define clear-shape
  (lambda [a-shape]
    (cond
      [(circle? a-shape)
       (draw-a-circle a-shape)]
      [(rectangle? a-shape)
       (draw-a-rectangle a-shape)])))

;; EXAMPLES
(start 200 200)
(draw-shape (make-circle (make-posn 30 30) 20 'red))
(draw-shape (make-rectangle  (make-posn 30 60) 20 50 'blue))
(clear-shape (make-circle (make-posn 30 30) 20 'red))
(clear-shape (make-rectangle (make-posn 30 60) 20 50 'blue))

; ---------------------------------------------------------------------------

;; draws a-shape, waits one second, clears a-shape,
;; returns a-shape translated by delta pixels.
(define move-shape
  (lambda [a-shape delta]
    (cond
      [(and (draw-shape a-shape)
            (sleep-for-a-while 1)
            (clear-shape a-shape))
       (translate-shape a-shape delta)]
      [else
       (translate-shape a-shape delta)])))
```

```
;; EXAMPLES AS TESTS

(define my-shape (make-circle  (make-posn 30 30) 20 'red))
(start 100 100)
(move-shape
 (move-shape
  (move-shape
   (move-shape my-shape (make-posn 10 0))
   (make-posn 10 0))
  (make-posn 10 0))
 (make-posn 10 0))
"should be"
(make-circle (make-posn 70 30) 20 'red)
```

16 Listenkonstruktion und -zerlegung

Werten Sie die folgenden Funktionsaufrufe aus:

Nr.	Ausdruck	Lösung
a)	`(first '((A) B C D))`	
b)	`(rest '((A)(B C D)))`	
c)	`(cons '(A B) '(A B))`	
d)	`(cons 'A '())`	
e)	`(first '(((A))))`	
f)	`(rest '(((A))))`	
g)	`(cons '((A)) empty)`	
h)	`(equal? 'X1 'X2)`	
i)	`(equal? '(X1) 'X2)`	
j)	`(equal? '(X1) '(X2))`	
k)	`(list? 'X1)`	
l)	`(list? '(X1))`	
m)	`(empty? '())`	
n)	`(empty? '(()))`	

Nr.	Ausdruck	Lösung
a)	`(first '((A) B C D))`	(A)
b)	`(rest '((A)(B C D)))`	((B C D))
c)	`(cons '(A B) '(A B))`	((A B) A B)
d)	`(cons 'A '())`	(A)
e)	`(first '(((A))))`	((A))
f)	`(rest '(((A))))`	()
g)	`(cons '((A)) empty)`	(((A)))
h)	`(equal? 'X1 'X2)`	#false
i)	`(equal? '(X1) 'X2)`	#false
j)	`(equal? '(X1) '(X2))`	#false
k)	`(list? 'X1)`	#false
l)	`(list? '(X1))`	#true
m)	`(empty? '())`	#true
n)	`(empty? '(()))`	#false

17 Listenverarbeitung

a. Die Funktion `enthaelt?` beantworte, angewendet auf ein Symbol und eine Liste von Symbolen, die Frage, ob das Symbol in der Liste enthalten ist oder nicht

```
;; Zweckbestimmung s. Aufgabenstellung
;;  enthaelt? ; (list-of symbol) symbol -> boolean
(check-expect (enthaelt? empty 's) #false)
```

```
(check-expect (enthaelt? '(x y z) 's) #false)
(check-expect (enthaelt? '(s y z) 's) #true)
(check-expect (enthaelt? '(x y s z) 's) #true)

(define enthaelt?
  (lambda [liste symbol]
    (cond
      [(empty? liste) #false]
      [(equal? (first liste) symbol) #true]
      [else (enthaelt? (rest liste) symbol)])))
```

b. Die Funktion `sum` liefere, angewendet auf eine Liste von Zahlen `x`, die Summe der Elemente. Die folgende Lösung erlaubt nur nicht leere Listen von Zahlen:

```
;; Zweckbestimmung s. Aufgabenstellung
;;
;; Datendefinition:
;; Eine Liste-von-Zahlen ist entweder
;;  - (cons n empty), wobei n eine Zahl ist oder
;;  - (cons n lvz), wobei n eine Zahl und lvz eine Liste-von-Zahlen ist
;;
;; sum : (list-of number) -> number
     (check-expect (sum '(1)) 1)
     (check-expect (sum '(1 2 3 4)) 10)

     (define sum
       (lambda [lvz]
         (cond
               [(empty? (rest lvz)) (first lvz)]
               [else (+ (first lvz)
                        (sum (rest lvz)))])))
```

Die folgende Lösung liefert für leere Listen das neutrale Element bzgl. der Addition:

```
;; Zweckbestimmung s. Aufgabenstellung
;;
;; Datendefinition:
;; Eine Liste-von-Zahlen ist entweder
;;  - enpty
;;  - (cons n lvz), wobei n eine Zahl und lvz eine Liste-von-Zahlen ist
;;
;; sum : (list-of number) -> number
     (check-expect (sum '()) 0)
     (check-expect (sum '(1 2 3 4)) 10)

     (define sum
       (lambda [lvz]
         (cond
               [(empty? lvz) 0]
               [else (+ (first lvz)
                        (sum (rest lvz)))])))
```

c. Die Funktion `prod` liefere, angewendet auf eine Liste von Zahlen `x`, das Produkt der Elemente. Die folgende Lösung erlaubt nur nicht leere Listen von Zahlen:

```
;; Zweckbestimmung s. Aufgabenstellung
;;
;; Datendefinition:
;; Eine Liste-von-Zahlen ist entweder
;;  - (cons n empty), wobei n eine Zahl ist oder
;;  - (cons n lvz), wobei n eine Zahl und lvz eine Liste-von-Zahlen ist
```

```
;;
;; sum : (list-of number) -> number
     (check-expect (prod '(1)) 1)
     (check-expect (prod '(1 2 3 4)) 24)

     (define prod
       (lambda [lvz]
         (cond
               [(empty? (rest lvz)) (first lvz)]
               [else (* (first lvz)
                        (prod (rest lvz)))])))
```

Die folgende Lösung liefert für leere Listen das neutrale Element bzgl. der Multiplikation:

```
;; Zweckbestimmung s. Aufgabenstellung
;;
;; Datendefinition:
;; Eine Liste-von-Zahlen ist entweder
;;  - enpty
;;  - (cons n lvz), wobei n eine Zahl und lvz eine Liste-von-Zahlen ist
;;
;; sum : (list-of number) -> number
     (check-expect (prod '()) 1)
     (check-expect (prod '(1 2 3 4)) 24)

     (define prod
       (lambda [lvz]
         (cond
               [(empty? lvz) 1]
               [else (* (first lvz)
                        (prod (rest lvz)))])))
```

d. Die Funktion `maximum` liefere, angewendet auf eine Liste von Zahlen `x`, das Maximum der Elemente. Die folgende Lösung erlaubt nur nicht leere Listen von Zahlen:

```
;; Zweckbestimmung s. Aufgabenstellung
;;
;; Datendefinition:
;; Eine Liste-von-Zahlen ist entweder
;;  - (cons n empty), wobei n eine Zahl ist oder
;;  - (cons n lvz), wobei n eine Zahl und lvz eine Liste-von-Zahlen ist
;;
;; sum : (list-of number) -> number
     (check-expect (maximum '(1)) 1)
     (check-expect (maximum '(1 2 3 0)) 3)

     (define maximum
       (lambda [lvz]
         (cond
               [(empty? (rest lvz)) (first lvz)]
               [else (max (first lvz)
                          (maximum (rest lvz)))])))
```

e. Schreiben Sie eine Funktion `(declist x)`, die aus einer Liste `x` von Zahlen eine neue Liste berechnet, deren Elemente um 1 kleiner sind, als die der ursprünglichen Liste:

x	(declist x)
(2 5 7)	(1 4 6)
()	()

```
;; Zweckbestimmung s. Aufgabenstellung
;; declist : (list-of number) -> (list-of number)
(check-expect (declist empty) empty)
(check-expect (declist '(2 5 7)) '(1 4 6))

(define declist
  (lambda [lvz]
        (cond
          [(empty? lvz) empty]
              [else (cons (- (first lvz) 1)
                          (declist (rest lvz)))]
        )))
```

f. Definieren Sie eine Funktion `(flatten x)`, die als Argument eine Liste `x` mit beliebig tief geschachtelten Unterlisten hat und als Ergebnis eine Liste von Atomen liefern soll mit der Eigenschaft, dass alle Atome, die in x vorkommen auch in `(flatten x)` in derselben Reihenfolge vorkommen:

x	(flatten x)
(A (B C) D)	(A B C D)
(((A B) C)(D E))	(A B C D E)
((((A))))	(A)

Hinweis: Definieren Sie zuerst in der bekannten Art und Weise eine rekursive Datenstruktur für geschachtelte Listen. Leiten Sie daraus eine passende **Funktionsschablone** ab.

```
;; Definition einer beliebig tief geschachtelten Liste (btgl) von Atomen
;; Eine btgl ist
;; - empty
;; - (cons a l) mit a ist ein Atom, l ist eine btgl
;; - (cons l1 l2) mit l1, l2 sind btgl
;; Funktionsschablone:
;(define f
;  (lambda [l]
;    (cond
;      [(empty? l) ...]
;      [(not (list? (first l)))
;       ... (first l) ... (f (rest l)) ...]
;      [else  ... (f (first l)) ... (f (rest l)) ...])))
;;
;; Zweckbestimmung s. Aufgabenstellung
;; flatten : (list-of any) -> (list-of atom)
(check-expect (flatten empty) empty)
(check-expect (flatten '(a (b (c (d))))) '(a b c d))
(check-expect (flatten '((((a) b) c) d)) '(a b c d))
(define flatten
  (lambda [l]
    (cond
      [(empty? l) empty]
      [(not (list? (first l)))
       (cons (first l) (flatten (rest l)))]
      [else  (append (flatten (first l)) (flatten (rest l)))])))

     ;; ;; Konkatiniert die zwei Listen l1 und l2
     ;; ;; append : (list-of any) (list-of any) -> (list-of any)
     ;; (check-expect (append empty empty) empty)
     ;; (check-expect (append empty '(A B C)) '(A B C))
     ;; (check-expect (append '(A B C) empty) '(A B C))
     ;; (check-expect (append '(A B C) '(C D E)) '(A B C C D E))
```

```
;; (define append
;;   (lambda [l1 l2]
;;     (cond
;;           [(empty? l1) l2]
;;           [(empty? l2) l1]
;;           [else (cons (first l1) (append (rest l1) l2))])))
```

g. **Zusatzaufgabe**: Schreiben Sie eine Funktion `(frequencies x)`, die aus einer Liste `x` von Atomen eine Liste von zwei-elementigen Listen erzeugt: Dabei ist das erste Element das Atom aus `x`, das zweite Element die Häufigkeit des Auftretens in `x`. Die Reihenfolge der Strukturen in der Ergebnisliste ist belanglos.

x	(frequencies x)
(A B A B A C A)	((A 4) (B 2) (C 1))
()	()

```
;; Datenstrukturdefinitionen:
;; Eine Atom-Number-Pair (anp) ist ein
;; - (cons a (cons n empty)) mit a ist Atom und n ist Number
;;
;; Eine Liste von Atomen (loa) ist
;; - empty oder
;; - (cons a l) mit a ist Atom und l ist loa
;;
;; Eine Liste von Atom-Number-Pairs (loanp) ist
;; - empty oder
;; - (cons a l) mit a ist anp und l ist loanp

;; erhoeht die number im ersten atom-mumber-pair um 1
;; incr-first-number : (list-of atom-number-pair)
;;                      -> (list-of atom-number-pair)
(check-expect (incr-first-number '((a  5) (b  3) (c  4)))
              '((a  6) (b  3) (c  4)))
(define incr-first-number
  (lambda [aloanp]
    (let
        [(firstatom (first (first aloanp)))
         (firstnumber (first (rest (first aloanp))))
         (rest-anps (rest aloanp))]
      (cons (list firstatom (+ 1 firstnumber)) rest-anps))))

;;; Version von incr-first-number ohne lokale Definitionen
;(define incr-first-number
;  (lambda [aloanp]
;    (cons (list (first (first aloanp))
;                (+ 1 (first (rest (first aloanp)))))) (rest aloanp))))

;; prueft, ob das Atom in der Liste vorkommt:
;; - falls ja ist die number im entsprechenden pair zu inkrementieren
;; - falls nein ist der Liste ein neues pair "(atom  1)" hinzuzufuegen
;; add-atom : atom (list-of atom-number-pair)
;;                 -> (list-of atom-number-pair)))
(check-expect (add-atom 'a empty)  '((a  1)))
(check-expect (add-atom 'a '((a  1) (b  1))) '((a  2) (b  1)))
(check-expect (add-atom 'a '((b  1))) '((b  1) (a  1)))
(define add-atom
  (lambda [atom aloanp]
    (cond
      [(empty? aloanp) (cons (list atom 1) empty)]
      [(equal? atom (first (first aloanp)))
```

```
       (incr-first-number aloanp)]
      [else (cons (first aloanp) (add-atom atom (rest aloanp)))]))))

;; Zweckbestimmung s. Aufgabenblatt
;; frequencies : (list-of atom) ->  (list-of atom-number-pair)
(check-expect (frequencies empty) empty)
(check-expect (frequencies '(a)) '((a  1)))
(check-expect (frequencies '(A B A B A C A)) '((A  4) (C  1) (B  2)))
(define frequencies
  (lambda [aloa]
    (cond
      [(empty? aloa) empty]
      [else
       (add-atom (first aloa) (frequencies (rest aloa)))])))
```

h. Schreiben Sie eine Funktion `anzahl-bevor-summe-erreicht`, die eine positive ganze Zahl, genannt `summe` und eine Liste von positiven ganzen Zahlen, genannt `lvz` als Argumente akzeptiert. Sie gibt eine ganze Zahl n zurück, so dass die Summe der ersten n Elemente von `lvz` kleiner als `sum`, die Summe der ersten $n+1$ Elemente hingegen größer oder gleich `sum` ist. Es ist ein Fehler, wenn die Summe der Elemente der Liste insgesamt kleiner als sum ist.
Beispiele:

sum	lvz	(anzahl-bevor-summe-erreicht sum lvz)
2	(2 5 7)	0
4	(2 5 7)	1
8	(2 5 7)	2
15	(2 5 7)	Fehler

Hinweise:

(a) Für den Fehlerfall darf die Standardfunktion `error` benutzt werden. Sie erwartet zwei Argumente, ein Symbol und eine Zeichenkette und könnte z. B. so benutzt werden:

```
(error 'anzahl-bevor-summe-erreicht "Summe der Listenelemente zu klein")
```

(b) Lösen Sie die Aufgabe, ohne die Summe der Listenelemente zu berechnen.

```
;; Zweckbestimmung s. Aufgabenstellung
; anzahl-bevor-summe-erreicht: number (list-of number) -> number
(check-expect (anzahl-bevor-summe-erreicht 2 '(2 5 7)) 0)
(check-expect (anzahl-bevor-summe-erreicht 4 '(2 5 7)) 1)
(check-expect (anzahl-bevor-summe-erreicht 8 '(2 5 7)) 2)
(check-error
    (anzahl-bevor-summe-erreicht 15 '(2 5 7))
    "anzahl-bevor-summe-erreicht: Summe der Listenelemente zu klein")
(define anzahl-bevor-summe-erreicht
  (lambda [sum lvz]
    (cond
      [(empty? lvz) (error
                      'anzahl-bevor-summe-erreicht
                      "Summe der Listenelemente zu klein")]
      [(<= sum (first lvz)) 0]
      [else (+ 1 (anzahl-bevor-summe-erreicht
                   (- sum (first lvz))
                   (rest lvz)))])))
```

i. Gegeben sei die Datenstrukturdefinition `gast` aus Aufgabe 12 und z. B. die folgende Gästeliste:

```
(define party
  (list
```

```
    (make-gast "Karl" #false #false)
    (make-gast "Rosa" #true #true)
    (make-gast "Klara" #true #false)
    (make-gast "Egon" #false #true)))
```

(a) Schreiben Sie eine Funktion `vegetarier`, die aus einer Gästeliste eine Liste der Vegetarier bildet. Für die Liste `party` müsste sie dieses Ergebnis liefern:

```
(list (make-gast "Rosa" #true #true)
      (make-gast "Egon" #false #true))
```

(b) Schreiben Sie eine Funktion `frauen`, die aus einer Gästeliste eine Liste mit den weiblichen Gästen bildet.

```
; Ein Gast besteht aus
; - einer Zeichenkette für den Namen
; - einem boolschen Wert, der angibt ob es sich um eine Frau handelt
; - einem boolschen Wert, der angibt ob es sich um einen Vegetarier handelt

(define-struct gast [name weiblich? vegetarier?])

; aus einer Gästeliste eine Liste der Vegetarier bilden
;; vegetarier: (list-of gast) -> (list-of gast)
(check-expect (vegetarier party) (list (make-gast "Rosa" #true #true)
                                       (make-gast "Egon" #false #true)))

(define vegetarier
  (lambda [gaeste]
    (cond
      [(empty? gaeste) empty]
      [(gast-vegetarier? (first gaeste))
       (cons (first gaeste)
             (vegetarier (rest gaeste)))]
      [else (vegetarier (rest gaeste))])))

; aus einer Gästeliste eine Liste der Frauen bilden
;; frauen: (list-of gast) -> (list-of gast)
(check-expect (frauen party) (list (make-gast "Rosa" #true #true)
                                   (make-gast "Klara" #true #false)))
(define frauen
  (lambda [gaeste]
    (cond
      [(empty? gaeste) empty]
      [(gast-weiblich? (first gaeste))
       (cons (first gaeste)
             (frauen (rest gaeste)))]
      [else (frauen (rest gaeste))])))
```

18 Funktionen über zwei Listen

Die in den folgenden Aufgaben zu entwickelnden Funktionen haben alle 2 Listen-Parameter. Lösen Sie diese Aufgaben unter Anwendung der Regeln 11 bis 13. Überlegen Sie dabei, ob für die Erstellung der Funktionsschablone der Zugriff auf das erste Element und die Restliste hinsichtlich des ersten, des zweiten oder beider Parameter vorgenommen werden muss.

a. Schreiben Sie ein Funktion `concatenate`, die zwei Listen von Symbolen aneinander hängt. Beispiel:

```
(concatenate '(a b c) '(d e f)) => '(a b c d e f)

   ;; Zweckbestimmung s. Aufgabenstellung
   ;;
```

```
;; Datendefinition
;; Eine Liste-von-Symbolen ist entweder
;;  1. empty oder
;;  2. (cons s lvs), wobei s ein Symbol
;;                   und lvs eine Liste-von-Symbolen ist
;;
;; concatenate : (list-of symbol) (list-of symbol) -> (list-of symbol)

(check-expect (concatenate '(a b c) '(d e f)) '(a b c d e f))
(check-expect (concatenate '() '(d e f)) '(d e f))
(check-expect (concatenate '(a b c) '()) '(a b c))

(define concatenate
       (lambda [l1 l2]
         (cond
               [(empty? l1) l2]
               [(empty? l2) l1]
               [else (cons (first l1)
                           (concatenate (rest l1) l2))])))
```

b. Schreiben Sie eine Funktion `mult-2-num-lists`, die zwei gleich lange Listen mit Zahlen zu einer Liste verarbeitet, die die Produkte der korrespondierenden Elemente der Argumentlisten enthält. Beispiel:

```
(mult-2-num-lists '(2 3 4) '(7 8 9)) => '(14 24 36)
```

```
;; Zweckbestimmung s. Aufgabenstellung
;;
;; Datendefinition
;; Eine Liste-von-Zahlen ist entweder
;;  1. empty oder
;;  2. (cons z lvz), wobei z eine Zahl
;;                   und lvz eine Liste-von-Zahlen ist
;;
;; mult-2-num-lists : (list-of number) (list-of number) -> (list-of number)

(check-expect (mult-2-num-lists '(2 3 4) '(7 8 9)) '(14 24 36))
(check-expect (mult-2-num-lists '() '()) '())

(define mult-2-num-lists
       (lambda [l1 l2]
         (cond
               [(empty? l1) empty]
               [else (cons (* (first l1) (first l2))
                           (mult-2-num-lists (rest l1) (rest l2)))])))
```

c. Entwickeln Sie eine Funktion `merge`, die 2 Listen von Zahlen verarbeitet, die aufsteigend sortiert sind. Sie liefert eine sortierte Liste von Zahlen, die alle Zahlen aus den beiden Argumentlisten enthält. Wenn Zahlen in den Argumentliste mehrfach vorkommen, sollen Sie auch in der Ergebnisliste entsprechend oft auftauchen. Beispiel:

```
(merge '(2 5 7) '(1 3 5 9)) => '(1 2 3 5 5 7 9)
```

```
;; Zweckbestimmung s. Aufgabenstellung
;;
;; Datendefinition
;; Eine Liste-von-Zahlen ist entweder
;;  1. empty oder
;;  2. (cons z lvz), wobei z eine Zahl
;;                   und lvz eine Liste-von-Zahlen ist
```

```
;;
;; merge : (list-of number) (list-of number) -> (list-of number)

(check-expect (merge '(2 3 4 11) '(1 4 7 8 9)) '(1 2 3 4 4 7 8 9 11))
(check-expect (merge '() '()) '())

(define merge
       (lambda [l1 l2]
         (cond
               [(empty? l1) l2]
               [(empty? l2) l1]
               [(< (first l1) (first l2)) (cons (first l1)
                                                (merge (rest l1) l2))]
               [else (cons (first l2) (merge l1 (rest l2)))])))
```

19 Formale Aspekte

a. Werten Sie die folgenden Ausdrücke Schritt für Schritt aus:

Ausdruck 1

```
(cond
           [(= 0 0 ) false]
           [(> 0 1 ) (equal? 'a 'a)]
           [else ( = (/ 1 0 ) 9)])
```

```
(cond
  [(= 0 0 ) false]
  [(> 0 1 ) (equal? 'a 'a)]
  [else ( = (/ 1 0 ) 9)])
    = (cond
        [true false]
            ...)
    = false
```

Ausdruck 2

```
(cond
           [(= 2 0) false]
           [(> 2 1) (equal? 'a 'a )]
           [else (= ( / 1 2) 9)])
```

```
(cond
   [(= 2 0) false]
   [(> 2 1) (equal? 'a 'a )]
   [else (= ( / 1 2) 9)])
     = (cond
         [false false]
             ...)
     = (cond
         [(> 2 1) (equal? 'a 'a)]
             ...)
     = (cond
         [true (equal? 'a 'a)]
             ...)
     = (equal? 'a 'a)
     = true
```

b. Gegeben sei folgende Racket-Funktion

```
(define f
  (lambda [n]
     (cond
       [(= n 0) 0]
```

```
        [else (+ (f (- n 1))
                 (/ 1 (* n (+ n 1))))])))
```

Zeigen Sie, dass der Aufruf `(f n)` für alle natürlichen Zahlen

$$n >= 0$$

die Zahl

$$f(n) = \frac{n}{n+1}$$

berechnet. Behauptung: Der Aufruf (f n) liefert für jede natürliche Zahl $n >= 0$ den Wert $\frac{n}{n+1}$ als Resultat.

Verankerung Der Aufruf (f 0) besitzt Rekursionstiefe 0 und liefert nach dem Ersetzungsmodell:

```
(f 0)
     = ((lambda [n]
     (cond
       [(= n 0) 0]
       [else (+ (f (- n 1))
                (/ 1 (* n (+ n 1))))]))
       0)
     = (cond
     [(= 0 0) 0]
     [else (+ (f (- 0 1))
              (/ 1 (* 0 (+ 0 1))))])
     = (cond
         [true 0]
             ...)
     = 0
```

$= \frac{0}{0+1}$

Induktionsannahme: Die Behauptung gilt für eine Rekursionstiefe $k = m$, d. h. $(fm) = \frac{m}{m+1}$.

Induktionsschluss: Es ist zu zeigen, dass der Aufruf `(f (+ m 1))` den Wert $\frac{m+1}{m+2}$ als Resultat liefert.

Sei x = (+ m 1)

```
     (f x)
     = ((lambda [n]
          (cond
        [(= n 0) 0]
        [else (+ (f (- n 1))
                 (/ 1 (* n (+ n 1))))])) x)
     = (cond
     [(= x 0) 0]
     [else (+ (f (- x 1))
              (/ 1 (* x (+ x 1))))])
     = (cond
         [#false 0]                          x > 0, da
                                                 x = m+1 und m >= 0
             ...)
     = (cond
     [else (+ (f (- x 1))
              (/ 1 (* x (+ x 1))))])
     = (+ (f (- x 1)) (/ 1 (* x (+ x 1))))

     Setze für x wieder (+ m 1) ein:
     = (+ (f (- (+ m 1) 1)) (/ 1 (* (+ m 1) (+ (+ m 1) 1))))

     = (+ (f m) (/ 1 (* (+ m 1) (+ (+ m 1) 1))))
```

Wechsel auf die mathematische Notation:
$= f(m) + \frac{1}{(m+1)*(m+2)}$
Anwendung der Induktionsannahme:
$= \frac{m}{m+1} + \frac{1}{(m+1)*(m+2)}$
$= \frac{m \cdot (m+2)+1}{(m+1)\cdot(m+2)}$
$= \frac{m^2+2m+1}{(m+1)\cdot(m+2)}$
$= \frac{(m+1)^2}{(m+1)\cdot(m+2)}$
$= \frac{m+1}{m+2}$

20 Lokale Definitionen

a. Schreiben Sie unter Anwendung der Regeln 11 bis 13 eine Funktion `exchange`, die eine Liste von Symbolen `l` und zwei Symbole `s1` und `s2` als Argumente erwartet und eine Liste als Resultat liefert, bei der jedes Auftreten von `s1` in `l` durch `s2` ersetzt ist. So soll z. B. der Aufruf

```
(exchange '(a b c a c) 'a 'f)
```

das Resultat

```
(list 'f 'b 'c 'f 'c)
```

liefern.

Modifizieren Sie anschließend die Funktion so, dass durch Verwendung von lokalen Variablen Mehrfachberechnungen des gleichen Ausdrucks vermieden werden.

```
;; Zweckbestimmung s. Aufgabestellung
;; exchange: (list-of symbol) symbol symbol -> symbol
(check-expect (exchange '() 'a 'f) '())
(check-expect (exchange '(a) 'a 'f) '(f))
(check-expect (exchange '(b) 'a 'f) '(b))
(check-expect (exchange '(a b c a c) 'a 'f) '(f b c f c))
; (define exchange
;   (lambda (lvs s1 s2)
;     (cond
;       [(empty? lvs) empty]
;       [else
;        (cond
;          [(equal? (first lvs) s1) (cons s2 (exchange (rest lvs) s1 s2))]
;          [else (cons (first lvs) (exchange (rest lvs) s1 s2))])])))

(define exchange
  (lambda [lvs s1 s2]
    (cond
      [(empty? lvs) '()]
      [else
       (let [(fst (first lvs))
             (exc-rest (exchange (rest lvs) s1 s2))]
         (cond [(equal? fst s1)
                (cons s2 exc-rest)]
               [else (cons fst
                           exc-rest)]))])))
```

b. Bildung der ersten Ableitung mathematischer Formeln (symbolische Differentiation)

Es sollen Ausdrücke abgeleitet werden, die nur aus Konstanten, Variablen und den Operationen + und · bestehen.

Sei D_x die partielle Ableitung einer Funktion f nach x, dann gelten folgende Regeln:

- $D_x(x) = 1$
- $D_x(y) = 0$, $y \neq x$, sei y eine Konstante oder Variable
- $D_x(e_1 + e_2) = D_x(e_1) + D_x(e_2)$ (Summenregel)
- $D_x(e_1 \cdot e_2) = e_1 \cdot D_x(e_2) + e_2 \cdot D_x(e_1)$ (Produktregel)

Repräsentation der Formeln:

- Konstante: numerisches Atom
- Variable: symbolisches Atom
- $e_1 + e_2$: (ADD e_1 e_2)
- $e_1 \cdot e_2$: (MUL e_1 e_2)

Anwendungsbeispiele:

(a) Der Ausdruck

```
(diff '(add x x) x)
```

liefere '`(add 1 1)`

(b) Der Ausdruck

```
(diff '(mul x x) 'x)
```

liefere '`(add (mul x 1) (mul 1 x))`.

Hinweise:

- Definieren Sie zur Erzeugung von Formeln geeignete Hilfsfunktionen!
- Wenn eine Formel nicht korrekt aufgebaut ist, kann das Symbol 'ERROR zurückgeliefert werden, das möglicherweise in einem korrekten Teil der Formel eingeschachtelt erscheint.
- Machen Sie ausgiebig von lokalen Definitionen Gebrauch.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;; Datenstrukturdefinition fuer Formeln: Eine formel ist
;; - numerisches Atom oder
;; - symbolisches Atom oder
;; - (add e1 e2) mit e1, e2 sind Formeln, oder
;; - (mul e1 e2) mit e1, e2 sind Formeln

;Hilfsdefinitionen
(define addop 'add)
(define mulop 'mul)

;ein paar Variablen
(define x 'x)
(define y 'y)
(define z 'z)

;; baut aus den beiden Argumenten eine Summenformel
;; add: formel formel -> formel
(check-expect (add x y) '(add x y))
(define add
  (lambda [u v]
    (list addop u v)))

;; baut aus den beiden Argumenten eine Produktformel
```

```
;; mul: formel formel -> formel
(check-expect (mul x y) '(mul x y))
(define mul
  (lambda [u v]
    (list mulop u v)))

;; bildet die erste Ableitung der Formel exp nach der Variablen var.
;; Regeln s. Aufgabenblatt
;; diff: formel symbol -> formel
(check-expect (diff 1 x) 0)
(check-expect (diff x x) 1)
(check-expect (diff y x) 0)
(check-expect (diff '(add x x) x) '(add 1 1))
(check-expect (diff '(mul x x) 'x) '(add (mul x 1) (mul 1 x)))
(define diff
  (lambda [exp var]
    (let*
        [(operand-1
          (lambda [s] (first (rest s))))
         (operand-2
          (lambda [s] (first (rest (rest s)))))
         (variable? symbol?)
         (same-variable? equal?)
         (add? ;; prueft, ob formula eine Summe ist
          (lambda [formula]
            (and (cons? formula) (equal? (first formula) addop))))
         (mul? ;; prueft, ob formula ein Produkt ist
          (lambda [formula]
            (and (cons? formula) (equal? (first formula) mulop))))]
      (cond [(number? exp) 0]
            [(variable? exp)
             (if (same-variable? exp var) 1 0)]
            [(add? exp)
             (add (diff (operand-1 exp) var)
                  (diff (operand-2 exp) var))]
            [(mul? exp)
             (add
              (mul (operand-1 exp)
                   (diff (operand-2 exp) var))
              (mul (diff (operand-1 exp) var)
                   (operand-2 exp)))]
         [else 'Error]))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Lösung mit Vereinfachung der abgeleiteten Ausdrücke        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Ersetze die obigen Funktionen add und mul durch die folgenden:
;; nimmt primitive Vereinfachungen einer Summenformel vor; eine Summenformel
;; vereinfacht sich quasi selbst
;; add: formel formel -> formel
(check-expect (add x 0) x)
(check-expect (add 0 x) x)
(check-expect (add 2 3) 5)
(check-expect (add x y) '(add x y))
(define add
  (lambda [x y]
    (cond
      [(equal? x 0) y]
```

```
      [(equal? y 0) x]
      [(and (number? x) (number? y)) (+ x y)]
      [else (list 'add x y)])))

;; mul: exp exp -> exp
;; nimmt primitive Vereinfachungen einer Produktformel vor; eine Produktformel
;; vereinfacht sich quasi selbst
(check-expect (mul x 0) 0)
(check-expect (mul x 1) x)
(check-expect (mul 1 x) x)
(check-expect (mul 2 3) 6)
(check-expect (mul x y) '(mul x y))
(define mul
  (lambda [x y]
    (cond
      [(or (equal? x 0) (equal? y 0)) 0]
      [(equal? y 1) x]
      [(equal? x 1) y]
      [(and (number? x) (number? y)) (* x y)]
      [else (list 'mul x y)])))
```

21 Listen über gemischten Daten (Zusatzaufgabe)

Diese Aufgabe ist eine Fortführung der Aufgaben 13 und 15. Dort haben Sie Funktionen zum Bewegen einzelner Figuren (shapes) geschrieben. Ein Bild hingegen besteht aus einer Sammlung von Figuren. Um Bilder zeichnen, verschieben und löschen zu können ist es sinnvoll, alle Bildteile (Figuren) in einer Datenstruktur zusammen zu fassen. Da die Anzahl der Figuren eines Bildes von Bild zu Bild variieren kann, bietet es sich an, die Figuren eines Bildes in einer Liste zu verwalten.

a. Beschreiben Sie eine Datendefinition für Listen von Figuren.

b. Erzeugen Sie eine Beispielliste (genannt face), die aus den Figuren der folgenden Tabelle besteht:

Figur	Position	Größen	Farbe
circle	(50,50)	40	red
rectangle	(30,20)	5×5	blue
rectangle	(65,20)	5×5	blue
rectangle	(40,75)	20×10	red
rectangle	(45,35)	10×30	blue

Die Tabelle setzt eine Zeichenfläche der Größe 300 x 100 voraus.

c. Entwickeln Sie eine Funktionsschablone für figurenverarbeitende Funktionen.

d. Entwickeln Sie auf der Grundlage dieser Funktionsschablone eine Funktion `draw-losh`, die eine Liste von Figuren als Argument erwartet und alle Elemente der Liste zeichnet und `#true` zurückliefert.

e. Entwickeln Sie eine Funktion `translate-losh`, die eine Liste von Figuren sowie eine `posn`-Struktur `delta` als Argumente akzeptiert und eine Liste von Figuren als Resultat liefert, in der alle Figuren der Argumentliste um `delta` verschoben sind. Die Funktion hat keinen Effekt auf der Zeichenfläche.

f. Entwicklen Sie die Funktion `clear-losh`, die die eine Liste von Figuren als Argument erwartet und alle Elemente der Liste von der Zeichenfläche entfernt und `#true` zurückliefert.

g. Entwicklen Sie die Funktion `draw-and-clear-picture`. Sie zeichnet ein Bild, wartet eine Weile und löscht anschließend das Bild von der Zeichenfläche.

h. Entwicklen Sie die Funktion `move-picture`, die eine `posn`-Struktur `delta` und ein Bild als Argumente akzeptiert. Sie zeichnet ein Bild, wartet eine Weile und löscht anschließend das Bild von der Zeichenfläche. Sie liefert als Resultat das um `delta` verschobene Bild.

Testen Sie die Funktion z. B. so

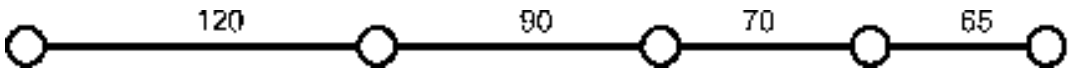
```
(start 500 100)

(draw-losh
  (move-picture (make-posn -5 0)
    (move-picture (make-pos 23 0)
      (move-picture (make-pos 10 0) face))))

(stop)
```

22 Hilfsfunktionen mit akkumulierenden Parametern

a. Schreiben Sie die Funktion **sum**, die die Summe der Elemente einer Liste von Zahlen berechnet, unter Benutzung einer Hilfsfunktion mit akkumulierendem Parameter. Verwenden Sie die Funktionsschablone aus der Vorlesung. Formulieren Sie die Akkumulatorinvariante.

```
;; berechnet die Summe einer Liste von Zahlen
;; sum: (list-of number) -> number
(check-expect (sum '(1 2 3)) 6)
(define sum
  (lambda [alon]
    (letrec
        [(;; Akkumulatorinvariante:
          ;; accu enthält die Summe der Elemente aus alon,
          ;; die denen aus alon1 vorangehen
          sum-acc
          (lambda [alon1 accu]
            (cond
              [(empty? alon1) accu]
              [else
               (sum-acc (rest alon1) (+ (first alon1) accu))])))]
      (sum-acc alon 0))))
```

b. Gegeben ist ein Weg in einem ungerichteten Graphen, dessen Knoten Orte repräsentieren und dessen Kanten mit den Entfernungen zwischen den Orten attributiert sind, z. B. so:



Entwickeln Sie eine Funktion, die aus einer Liste mit relativen Entfernungen eine Liste mit den absoluten Entfernungen der Orte vom Ursprungsort berechnet. Für den obigen Graphen soll also aus der Liste (120 90 70 65) die Liste (120 210 280 345) werden.

(a) Entwickeln Sie zunächst eine Funktion (ggf. mit Hilfsfunktion) nach den bekannten Regeln (ohne akkumulierende Parameter).

(b) Diskutieren Sie, warum eine Hilfsfunktion mit akkumulierendem Parameter sinnvoll ist.

(c) Entwickeln Sie eine solche.

```
;; Lösung ohne akkumulierenden Parameter
; (define relativ->absolut
;   (lambda [lvz]
;     (cond
;       [(empty? lvz) empty]
;       [else
;        (cons (first lvz)
;              (addiere-auf-jede (first lvz)
;                                (relativ->absolut (rest lvz))))])))
;
; (define addiere-auf-jede
;   (lambda [zahl lvz]
```

```
;     (cond
;       [(empty? lvz) empty]
;       [else (cons (+ (first lvz) zahl) (addiere-auf-jede zahl
;                                                    (rest lvz)))])))
;


;; Warum akkumulierender Parameter? Weil die Hilfsfunktion wie die
;; Hauptfunktion die Liste rekursiv abarbeitet!

;; Was ist zu akkumulieren?
;; Das Problem besteht darin, dass der rekursive Aufruf
;; (relativ->absolut (rest lvz)) nichts von dem Wert des
;; ersten Elements "weiß". Dieses Problem kann gelöst werden,
;; indem man in einem Parameter die Distanz vom Ursprung
;; akkumuliert. Das führt zu folgender Lösung:

; ;; relativ->absolut: (list-of number) -> (list-of number)
; (check-expect (relativ->absolut '(50 40 70 30 30))
;               (list 50 90 160 190 220))
; (define relativ->absolut
;   (lambda [lvz]
;     (letrec
;         [(;; Akkumulatorinvariante: akku-distanz ist die akkumulierte Distanz
;           ;; derjenigen Elemente von lvz, die denen aus lvz1 vorangehen.
;           relabs (lambda [lvz1 akku-distanz]
;                    (cond
;                      [(empty? lvz1) empty]
;                      [else
;                       (cons (+ akku-distanz (first lvz1))
;                             (relabs (rest lvz1)
;                                     (+ akku-distanz (first lvz1))))]))])
;       (relabs lvz 0))))
;


;; relabs ist aber nicht endrekursiv. Das kann erreicht werden,
;; indem in einem weiteren akkumulierenden Parameter akku-entfrnngn
;; die Ergebnisliste aufgebaut wird.  Man beachte, dass die Liste
;; dabei in umgekehrter Reihenfolge entsteht.

;; relativ->absolut: (list-of number) -> (list-of number)
(check-expect (relativ->absolut '(50 40 70 30 30))
              (list 50 90 160 190 220))
(define relativ->absolut
  (lambda [lvz]
    (letrec
        [(;; Akkumulatorinvariante: akku-distanz ist die akkumulierte Distanz
          ;; derjenigen Elemente von lvz, die denen aus lvz1 vorangehen.
          relabs (lambda [lvz1 akku-distanz akku-entfrnngn]
                   (cond
                     [(empty? lvz1) (reverse akku-entfrnngn)]
                                    ;;;;;;;
                     [else
                      (relabs (rest lvz1)
                              (+ akku-distanz (first lvz1))
                              (cons (+ akku-distanz (first lvz1))
                                    akku-entfrnngn))]))])
      (relabs lvz 0 '()))))
```

c. Definieren Sie eine Funktion `(singletons x)`, die als Argument eine Liste von den Atomen `x` hat und als Ergebnis eine Liste von den Atomen liefern soll, die in `x` genau einmal auftreten.

```
;; "klassische" Loesung mit natuerlicher Rekursion nach den Regeln
;; für listenverarbeitende Funktionen

;;Test-Hilfsprozedur
;;erzeugt Liste mit anzahl-maligem Auftreten von x
(define vervielfache
  (lambda [x anzahl]
    (cond [(<= anzahl 0) '()]
          [else (cons x (vervielfache x (- anzahl 1)))])))

;; entfernt jedes Auftreten von elem in list
;; eliminate: X (listof X) -> (listof X)
(check-expect (eliminate 2 '()) '())
(check-expect (eliminate 2 '(1 2 3 3 2 1 4 5 4)) '(1 3 3 1 4 5 4))
(check-expect (eliminate 6 '(1 2 3 3 2 1 4 5 4)) '(1 2 3 3 2 1 4 5 4))
(define eliminate
  (lambda [elem list]
    (cond
      [(empty? list) empty]
      [(equal? elem (first list))
       (eliminate elem (rest list))]
      [else (cons (first list)
                  (eliminate elem (rest list)))])))

;; entfernt jedes Duplikat aus list
;; singletons: (listof X) -> (listof X)
(check-expect (singletons empty) empty)
(check-expect (singletons '(a)) '(a))
(check-expect (singletons '(1 2 3 3 2 1 4 5 4)) '(5))
(check-expect (singletons '(1 1 5 2 3 3 2 1 4 5 4)) '())
(check-expect (singletons (vervielfache 'a 10001)) '())
(define singletons
  (lambda [list]
    (cond
      [(empty? list) empty]
      [else
       (let [(singlerest (singletons (rest list)))]
         (cond
           [(member? (first list) (rest list))
            (eliminate (first list) singlerest)]
           [else (cons (first list) singlerest)]))])))
```

d. Modifizieren die Funktion `(singletons x)` so, dass zwei akkumulierende Parameter verwendet werden. Der eine soll zum Akkumulieren der Atome, die genau einmal in `x` auftreten, dienen, der andere zum Akkumulieren der Atome, die mehrmals in `x` auftreten.

```
;; entfernt jedes Duplikat aus list
;; singletons: (list-of atom) -> (list-of atom)
(check-expect (singletons empty) empty)
(check-expect (singletons '(1 2 3 3 2 1 3 4 5 4)) '(5))
(check-expect (singletons (vervielfache 1 100000)) empty)
(define singletons
  (lambda [list]
    (letrec
        [(;; Fuer den Listenkopf ueberpruefe zuerst, ob er schon in
          ;; multis vorhanden ist.
          ;; Wenn ja, braucht er nicht weiter verarbeitet zu werden.
```

```
                ;; Wenn nein, muss er zu multis hinzugefuegt werden,
                ;; wenn er im Listenrumpf noch einmal auftritt, sonst zu
                ;; singles.
                singletonsAccu
                (lambda [list0 singles multis]
                  (cond
                    [(empty? list0) singles]
                    [(member? (first list0) multis)
                     (singletonsAccu (rest list0) singles multis)]
                    [(member? (first list0) (rest list0) )
                     (singletonsAccu (rest list0)
                                     singles
                                     (cons (first list0) multis))]
                    [else
                     (singletonsAccu (rest list0)
                                     (cons (first list0) singles)
                                     multis)]))))]
          (singletonsAccu list empty empty))))

    (define vervielfache
      (lambda (x anzahl)
        (cond [(<= anzahl 0) '()]
              [else (cons x (vervielfache x (- anzahl 1)))])))
```

23 Abstraktion von Funktionen

Wenn Sie die Funktionen `vegetarier` und `frauen` aus Aufgabe 17i regelkonform entwickelt haben, dürften sie sich nur an einer Stelle *wesentlich* unterscheiden. Überlegen Sie, wie diese „Verschiedenheit“ in Form eines Parameters in eine verallgemeinerte Funktion `filter-gaeste` übernommen werden kann, so dass z. B. die Funktion `frauen` dann so implementiert werden könnte:

```
(define frauen
  (lambda [gaeste]
      (filter-gaeste gast-weiblich? gaeste)))
```

a. Schreiben Sie die Funktion `filter-gaeste` regelkonform auf.

b. Schreiben Sie die Funktion `vegetarier` unter Nutzung der Funktion `filter-gaeste` auf.

```
; Ein Gast besteht aus
; - einer Zeichenkette für den Namen
; - einem boolschen Wert, der angibt ob es sich um eine Frau handelt
; - einem boolschen Wert, der angibt ob es sich um einen Vegetarier handelt

(define-struct gast [name weiblich? vegetarier?])

(define party
  (list
   (make-gast "Karl" #false #false)
   (make-gast "Rosa" #true #true)
   (make-gast "Klara" #true #false)
   (make-gast "Egon" #false #true)))

; filtert aus einer Gästeliste diejenigen mit einer bestimmten Eigenschaft
; filter-gaeste : (gast -> boolean) (list-of gast) -> (list-of gast)
(define filter-gaeste
  (lambda [praedikat? gaeste]
    (cond
      [(empty? gaeste) '()]
```

```
      [(praedikat? (first gaeste)) (cons (first gaeste)
                                         (filter-gaeste praedikat? (rest gaeste)))]
      [else (filter-gaeste praedikat? (rest gaeste))]))

; aus einer Gästeliste eine Liste der Vegetarier bilden
;; vegetarier : (list-of gast) -> (list-of gast)
(check-expect (vegetarier party) (list (make-gast "Rosa" #true #true)
                                       (make-gast "Egon" #false #true)))
(define vegetarier
  (lambda [gaeste]
    (filter-gaeste gast-vegetarier? gaeste)))
```

24 Anwendung von `map`, `filter` und `foldr`

Implementieren Sie folgende Funktionen unter Nutzung der Funktionen `filter`, `foldr` und `map` aus dem Skript:

a. Die Funktion `frauen` aus der vorangegangen Aufgabe unter Verwendung der Funktion `filter`.

b. Eine Funktion, die zu allen Zahlen einer Liste jeweils 42 addiert.

c. Eine Funktion, die sich wie b) verhält, aber nur die geraden Zahlen zurückgibt.

d. Eine Funktion, die sich wie c) verhält, aber das Produkt aller Zahlen zurückgibt.

e. Eine Funktion, die aus einer Liste von Zahlen alle Zahlen streicht, die nicht durch 4 oder 5 teilbar sind.

f. Eine Funktion, die die Summe der Quadrate der natürlichen Zahlen in einer Liste berechnet.

g. Eine Funktion `und`, die genau dann `true` zurück liefert, wenn alle Elemente einer Liste von Booleans `true` sind.

h. Eine Funktion `partitioniere`, die ein Prädikat als Argument nimmt und, angewandt auf eine Liste, zwei Listen zurückgibt, wobei erstere alle Elemente enthält, die das Prädikat erfüllen, und die andere die restlichen Elemente enthält.

i. Eine Funktion `sort`, die, angewendet auf eine Liste von Zahlen, diese Liste absteigend sortiert.

j. Modifizieren Sie die Funktion `sort` aus 8. so, dass durch einen zusätzlichen Parameter die Sortierreihenfolge bestimmt werden kann.

k. Gegeben sei eine Liste von Zahlen, z. B. diese:

```
(define zahlen '(1 3 7 5 12 8))
```

Schreiben Sie einen Racket-Ausdruck, der alle Zahlen kleiner als 7 quadriert und aufsummiert. Für die o. g. `zahlen` lautete das Ergebnis 35.

„Racket-Ausdruck"bedeutet hier: Sie brauchen keine Funktion zu definieren. Z. B. sähe der Racket-Ausdruck, der die o. g. Zahlen verdoppelt, so aus:

```
(map (lambda [x] (* p 2)) zahlen)
```

Betrachten Sie zum Schluss Ihren Racket-Audruck hinsichtlich der Lesbarkeit.

```
;;; Lösungen zu Aufgabe 24 ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; b) Eine Funktion, die zu allen Zahlen einer Liste jeweils 42 addiert.
;; add42: (list-of number) -> (list-of number)
(check-expect (add42 '()) '())
(check-expect (add42 '( 3 )) '(45 ))
(check-expect (add42 '( 3 4 5 )) '(45 46 47))
(define add42
```

```
  (lambda [lvz]
    (map (lambda [x] (+ x 42)) lvz)))

;; c) Eine Funktion, die sich wie a) verhält, aber nur die geraden Zahlen zurückgibt.
;; add42even: (list-of number) -> (list-of number)
(check-expect (add42even '()) '())
(check-expect (add42even '( 3 )) '())
(check-expect (add42even '( 3 4 5 6)) '( 46 48 ))
(define add42even
  (lambda [lvz]
    (filter even? (add42 lvz))))

;; d) Eine Funktion, die sich wie b) verhält, aber das Produkt aller Zahlen zurückgibt.
;; prod-of-add42even (list-of number) -> number
(check-expect (prod-of-add42even '()) 1)
(check-expect (prod-of-add42even '( 3 )) 1)
(check-expect (prod-of-add42even '( 3 4 5 6 )) (* 46 48))
(define prod-of-add42even
  (lambda [lvz]
    (foldr * 1 (add42even lvz))))

;; e) Eine Funktion, die aus einer Liste von Zahlen alle Zahlen streicht,
;;    die nicht durch 4 oder 5 teilbar sind.
;; nur-durch-4-und-5-teilbare: (list-of number) -> (list-of number)
(check-expect (nur-durch-4-und-5-teilbare '()) '())
(check-expect (nur-durch-4-und-5-teilbare '( 4 )) '(4))
(check-expect (nur-durch-4-und-5-teilbare '( 3 4 5 6 8 15 40)) '(4 5 8 15 40))
(define nur-durch-4-und-5-teilbare
  (lambda [lvz]
    (filter (lambda [x] (or (= (modulo x 4) 0)
                            (= (modulo x 5) 0)))
            lvz)))

;; f) Eine Funktion, die die Summe der Quadrate der natürlichen Zahlen in einer Liste berechnet.

;; liefert true für natürliche Zahlen
;; nat-number?: number -> boolean
(check-expect (nat-number? -1) #false)
(check-expect (nat-number? 2.5) #false)
(check-expect (nat-number? 11) #true)
(define nat-number?
  (lambda [x] (and (> x 0) (integer? x))))

;; sum-quad-nat: (list-of number) -> number
(check-expect (sum-quad-nat '()) 0)
(check-expect (sum-quad-nat '(1)) 1)
(check-expect (sum-quad-nat '(-2 3.5 2 3 4)) 29)
(define sum-quad-nat
  (lambda [lvz]
    (foldr + 0 (map (lambda [x] (* x x)) (filter nat-number? lvz)))))

;; g) Eine Funktion und, die genau dann true zurück liefert, wenn alle Elemente einer
;;    Liste von Booleans true sind.
;; und: (list-of boolean) -> boolean
(check-expect (und '()) #true)
(check-expect (und '(#true)) #true)
(check-expect (und '(#false)) #false)
(check-expect (und '(#true #true #true)) #true)
(check-expect (und '(#true #true #false)) #false)
```

```
(define und
  (lambda [lvb]
    (foldr (lambda [x y] (and x y)) #true lvb)))
;;; Hinweis: der Aufruf (foldr and #true lvb) ist nicht möglich, da and keine Funktion
;;;  sondern eine Pseudofunktion ist

;; h) Eine Funktion partitioniere, die ein Prädikat als Argument nimmt und, angewandt
;;  auf eine Liste, zwei Listen zurückgibt, wobei erstere alle Elemente enthält, die das
;;  Prädikat erfüllen, und die andere die restlichen Elemente enthält.
;; partitioniere: (number-> boolean) (list-of number) -> (list-of (list-of number))
(check-expect (partitioniere even? '()) '(() ()))
(check-expect (partitioniere even? '(2)) '((2) ()))
(check-expect (partitioniere even? '(3)) '(() (3)))
(check-expect (partitioniere even? '(-2 7 5)) '((-2) (7 5)))
(define partitioniere
  (lambda [praed? lvz]
    (list (filter praed? lvz)
          (filter (lambda [x] (not (praed? x))) lvz))))

;; i) Eine Funktion sort, die, angewendet auf eine Liste von Zahlen, diese Liste absteigend sort
;; sortiere: (list-of number) -> (list-of number)
(check-expect (sortiere '()) '())
(check-expect (sortiere '(1)) '(1))
(check-expect (sortiere '(3 4 9 2 1 2)) '(9 4 3 2 2 1))
(define sortiere
  (lambda [lst]
    (letrec
        [(insert (lambda [an alon]
                   (cond
                    [(empty? alon) (list an)]
                    [(> an (first alon)) (cons an alon)]
                    [else (cons (first alon) (insert an (rest alon)))])))]
      (foldr insert '() lst))))

;; j) Modifizieren Sie die Funktion sort aus i). so, dass durch einen zusätzlichen
;;  Parameter die Sortierreihenfolge bestimmt werden kann.
;; sortiere: (list-of number) -> (list-of number)
(check-expect (sortiere2 '() <) '())
(check-expect (sortiere2 '(1) >) '(1))
(check-expect (sortiere2 '(3 4 9 2 1 2) >) '(9 4 3 2 2 1))
(check-expect (sortiere2 '(3 4 9 2 1 2) <) '(1 2 2 3 4 9))
(define sortiere2
  (lambda [lst auf/ab]
    (letrec
        [(insert (lambda [an alon]
                   (cond
                    [(empty? alon) (list an)]
                    [(auf/ab an (first alon)) (cons an alon)]
                    [else (cons (first alon) (insert an (rest alon)))])))]
      (foldr insert '() lst))))

;; k) Schreiben Sie einen Racket-Ausdruck, der alle Zahlen einer Liste kleiner
;; als 20  quadriert und aufsummiert.
(define zahlen '(1 3 7 5 12 8))

(foldr + 0
     (map (lambda [x] (* x x))
          (filter (lambda [x] (< x 7)) zahlen)))
```

```
;; Der Ausdruck ist insofern unschön, als man ihn eigentlich von hinten nach vorn
;; oder von innen nach außen lesen muss, um ihn zu verstehen. Das ist typisch für
;; geschachtelte Funktionsaufrufe und damit für funktionale Programmierspachen.
;; Manche Sprachen bieten eine alternative Ausdrucksweise mithilfe des sog. threadings.
;; In DrRacket muss man dazu das threading-Package installieren und kann es dann in
;; die Datei mit require einbinden. Der Ausdruck kann dann wie folgt geschrieben
;; werden:
(require threading)

(define zahlen '(10 30 17 20 15 18 45 12))

(~>> zahlen
    (filter (lambda [x] (< x 7)))
    (map (lambda [x] (* x x)))
    (foldr + 0))

;; Das threading-Macro ~>> sorgt dafür, dass sein erstes Argument (hier preise) dem
;; nachfolgenden Funktionsaufruf (hier filter) als letztes Argument hinzugefügt wird.
;; Das Ergebnis des filter-Aufrufs wird dann dem nachfolgenden map-Aufruf als letztes
;; Argument hinzugefügt usw.
```

25 Definition von Funktionen höherer Ordnung

Definieren Sie die folgenden Funktionen höherer Ordnung rekursiv oder durch Verwendung von anderen Funktionen höherer Ordnung:

a. Eine Funktion

```
(Number -> Number) (Number-> Number) list-of Number
  -> list-of Number,
```

die zwei Funktionen und eine Liste von Zahlen als Argument erhält und auf jede Zahl zuerst die erste und dann die zweite Funktion anwendet.

b. Eine Funktion

```
(Number -> Number) (Number -> Number) -> Number,
```

die zwei Funktionen als Argument erhält und die kleinste natürliche Zahl sucht, für die diese beiden Funktionen dasselbe Ergebnis liefern. Begrenzen Sie die Suche auf Zahlen bis 1000 und geben Sie -1 zurück, wenn keine passende Zahl gefunden wurde.

```
;;; Lösungen zu Aufgabe 25 ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; a) Eine Funktion, die zwei Funktionen und eine Liste von Zahlen als Argument erhält
;;    und auf jede Zahl zuerst die erste und dann die zweite Funktion anwendet.
;; hintereinander: (Number -> Number) (Number-> Number) (list-of Number) -> (list-of Number)
(check-expect (hintereinander (lambda [x] (* x x))
                              (lambda [x] (- x 2))
                              '())
              '())
(check-expect (hintereinander (lambda [x] (* x x)) (lambda [x] (- x 2)) '(3)) '(7))
(check-expect (hintereinander (lambda [x] (* x x)) (lambda [x] (- x 2)) '(-2 7 5)) '(2 47 23))
(define hintereinander
  (lambda [f g lvz]
    (map (lambda [x] (g (f x))) lvz)))

;; b) Eine Funktion, die zwei Funktionen als Argument erhält und die kleinste natürliche
;;    Zahl sucht, für die diese beiden Funktionen dasselbe Ergebnis liefern. Begrenzen
;;    Sie die Suche auf Zahlen bis 1000 und geben Sie -1 zurück, wenn keine passende Zahl
```

```
;;    gefunden wurde.
;; gleiches-ergebnis: (Number -> Number) (Number -> Number) -> Number
(check-expect (gleiches-ergebnis (lambda [x] (* x x))
                                 (lambda [x] (- x 2)))
              -1)
(check-expect (gleiches-ergebnis + *) 0)
(check-expect (gleiches-ergebnis (lambda [x] (* x x))
                                 (lambda [x] (- 2 x)))
              1)
(check-expect (gleiches-ergebnis (lambda [x] (- 20 x))
                                 (lambda [x] (+ 2 x)))
              9)
(define gleiches-ergebnis
  (lambda [f g]
    (letrec [(ge2 (lambda [f g n]
                    (cond
                      [(= (f n) (g n)) n]
                      [(= n 1000) -1]
                      [else (ge2 f g (+ n 1))])))]
      (ge2 f g 0))))
```