Aufgabe 1

ca. 5 Minuten

Aussage	wahr	falsch
Das Gebiet Informatik lässt sich in Theoretische, Angewandte, Praktische und Technische Informatik gliedern.	X	
Die Wirtschaftsinformatik zählt zur Praktischen Informatik.		X
Die Syntax einer Programmiersprache besteht aus einer Menge von Regeln, die die Struktur ihrer Programme bestimmt.	X	
Die Semantik einer Programmiersprache bestimmt, welche Intention ein Programmierer mit einem Programm verfolgt.		X
Informationen werden als Daten repräsentiert und verarbeitet.	X	

Aufgabe 2

ca. 15 Minuten

Entwerfen Sie Datenstrukturen für verschiedene Kontentypen:

Girokonto

hat die Merkmale *Kontostand* und *Zahl der Transaktionen*

Sparkonto

hat die Merkmale *Kontostand* und *Zahl der Transaktionen*

Kreditkonto

hat die Merkmale *Kontostand*, *Ausgabelimit* und *Zahl der Transaktionen*

Entwickeln Sie eine Funktion `abhebung`, die ein Konto und einen Geldbetrag als Argumente akzeptiert und als Resultat entweder ein neues Konto oder das Symbol `'Fehler` liefert. Falls die Abhebung den Kontostand eines Sparkontos unter 0 sinken ließe, wird `'Fehler` zurück gegeben. Andernfalls wird eine neues Sparkonto mit dem neuen Kontostand und der um 1 erhöhten Zahl der Transaktionen zurück gegeben. Ein Girokonto darf bis zu einem Betrag von 1000 € überzogen werden. Eine Abhebung von einen Kreditkonto erhöht den Kontostand, der aber das Ausgabelimit nicht übersteigen darf.

Lösung:

```
(define fehler 'Fehler)
;; Ein Girokonto ist ein Wert
;;    (make-girokonto kontostand anzahl-Transaktionen)
;; wobei kontostand und anzahl-Transaktionen Zahlen sind
(define-struct girokonto [kontostand anzahl-Transaktionen])

;; Ein Sparkonto ist ein Wert
;;    (make-sparkonto kontostand anzahl-Transaktionen)
;; wobei kontostand und anzahl-Transaktionen Zahlen sind
```

```
(define-struct sparkonto [kontostand anzahl-Transaktionen])


;; Ein Kreditkonto ist ein Wert
;;    (make-kreditkonto kontostand limit anzahl-Transaktionen)
;; wobei kontostand, limit und anzahl-Transaktionen Zahlen sind
(define-struct kreditkonto [kontostand limit anzahl-Transaktionen])

;; ...
;; abhebung-girokonto : number girokonto -> girokonto | 'Fehler
(check-expect (abhebung-girokonto 1000 (make-girokonto 2000 5))
              (make-girokonto 1000 6))
(check-expect (abhebung-girokonto 1000 (make-girokonto 0 0))
              (make-girokonto -1000 1))
(check-expect (abhebung-girokonto 1001 (make-girokonto 0 0)) fehler)

(define abhebung-girokonto
   (lambda [betrag konto]
      (let [ (neuer-Kontostand (- (girokonto-kontostand konto) betrag))
             (inc-anzahl-Transaktionen (+ (girokonto-anzahl-Transaktionen konto) 1))]
           (cond
              [(>= neuer-Kontostand -1000)
               (make-girokonto neuer-Kontostand
                               inc-anzahl-Transaktionen)]
              [else fehler]))))

;; ...
;; abhebung-sparkonto : number sparkonto -> sparkonto | 'Fehler
(check-expect (abhebung-sparkonto 1000 (make-sparkonto 2000 5))
              (make-sparkonto 1000 6))
(check-expect (abhebung-sparkonto 100 (make-sparkonto 100 0))
              (make-sparkonto 0 1))
(check-expect (abhebung-sparkonto 1000 (make-sparkonto 999 0))
              fehler)

(define abhebung-sparkonto
   (lambda [betrag konto]
      (let    [(neuer-Kontostand (- (sparkonto-kontostand konto) betrag))
               (inc-anzahl-Transaktionen (+ (sparkonto-anzahl-Transaktionen konto) 1))]
           (cond
              [(>= neuer-Kontostand 0)
               (make-sparkonto neuer-Kontostand
                               inc-anzahl-Transaktionen)]
              [else fehler]))))

;; ...
;; abhebung-kreditkonto : number kreditkonto -> kreditkonto | 'Fehler
(check-expect (abhebung-kreditkonto 1000 (make-kreditkonto 0 1000 5))
              (make-kreditkonto 1000 1000 6))
(check-expect (abhebung-kreditkonto 100 (make-kreditkonto 900 1000 2))
              (make-kreditkonto 1000 1000 3))
(check-expect (abhebung-kreditkonto 1001 (make-kreditkonto 0 1000 5))
```

```
62                fehler)
63 (check-expect (abhebung-kreditkonto 101 (make-kreditkonto 1900 2000 2))
64                fehler)
65
66 (define abhebung-kreditkonto
67    (lambda [betrag konto]
68       (let [(neuer-Kontostand (+ (kreditkonto-kontostand konto)
69                                  betrag))
70             (inc-anzahl-Transaktionen (+ (kreditkonto-anzahl-Transaktionen konto) 1))
71             (limit (kreditkonto-limit konto))]
72          (cond
73             [(<= neuer-Kontostand limit)
74              (make-kreditkonto neuer-Kontostand
75                                limit
76                                inc-anzahl-Transaktionen)]
77             [else fehler]))))
78
79 ;; Ein Konto ist entweder
80 ;;  -  ein girokonto oder
81 ;;  -  ein sparkonto oder
82 ;;  -  ein Kreditkonto
83 ;; name: konto
84 ;; Zweckbestimmung s. Aufgabenstellung
85 ;; abhebung : number konto -> kreditkonto | 'Fehler
86 ;; kreditkonto
87 (check-expect (abhebung 1000 (make-kreditkonto 0 1000 5))
88               (make-kreditkonto 1000 1000 6))
89 (check-expect (abhebung 100 (make-kreditkonto 900 1000 2))
90               (make-kreditkonto 1000 1000 3))
91 (check-expect (abhebung 1001 (make-kreditkonto 0 1000 5)) fehler)
92 (check-expect (abhebung 101 (make-kreditkonto 1900 2000 2)) fehler)
93 ;; sparkonto
94 (check-expect (abhebung 1000 (make-sparkonto 2000 5)) (make-sparkonto 1000 6))
95 (check-expect (abhebung 100 (make-sparkonto 100 0)) (make-sparkonto 0 1))
96 (check-expect (abhebung 1000 (make-sparkonto 999 0)) fehler)
97 ;; girokonto
98 (check-expect (abhebung 1000 (make-girokonto 2000 5))
99                (make-girokonto 1000 6))
100 (check-expect (abhebung 1000 (make-girokonto 0 0))
101               (make-girokonto -1000 1))
102 (check-expect (abhebung 1001 (make-girokonto 0 0)) fehler)
103
104 (define abhebung
105    (lambda [betrag konto]
106       (cond
107          [(girokonto? konto) (abhebung-girokonto betrag konto)]
108          [(sparkonto? konto) (abhebung-sparkonto betrag konto)]
109          [(kreditkonto? konto) (abhebung-kreditkonto betrag konto)]
110          [else fehler])))
```

Aufgabe 3

Wenden Sie die passenden Regeln für die Entwicklung der folgenden Funktionen an:

(a) CA. 4-5 MINUTEN

Die Funktion quadsum liefere, angewendet auf eine Liste von Zahlen, die Summe Quadrate der Elemente.

Lösung:

```
;; a)
;; Zweckbestimmung s. Aufgabenstellung
;; eine Liste-von-Zahlen ist entweder
;; 1. empty oder
;; 2. (cons n l), wobei n eine number und l eine Liste-von-Zahlen ist.
;; quadsum : (list-of number) -> number
(check-expect (quadsum '(1 2 3)) 14)
(check-expect (quadsum '(1)) 1)
(check-expect (quadsum '()) 0)
(define quadsum
  (lambda [lon]
    (cond
      [(empty? lon) 0]
      [else (+ (sqr (first lon))
               (quadsum (rest lon)))])))

```

(b) CA. 5-6 MINUTEN

Die Funktion oddelements liefere, angewendet auf eine Liste von Zahlen lvz, eine Liste mit allen ungeraden Zahlen aus lvz.

Lösung:

```
;; b)
;; Zweckbestimmung s. Aufgabenstellung
;; eine Liste-von-Zahlen ist entweder
;; 1. empty oder
;; 2. (cons n l), wobei n eine number und l eine Liste-von-Zahlen ist.
;; oddelements : (list-of number) -> number
(check-expect (oddelements '(1 2 3 7)) '(1 3 7))
(check-expect (oddelements '(4)) empty)
(check-expect (oddelements empty) empty)
(define oddelements
  (lambda [lon]
    (cond
      [(empty? lon) empty]
      [(odd? (first lon)) (cons (first lon)
                                (oddelements (rest lon)))]
      [else (oddelements (rest lon))])))
```

Aufgabe 4

Die hier zu entwickelnden Funktionen haben 2 Listen-Parameter. Lösen Sie diese Aufgaben unter Anwendung der Regeln. 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.

Lösung:

(a) CA. 5-6 MINUTEN

Die Funktion `cross` verarbeite eine Liste von Symbolen `lvs` und eine gleich lange Liste von Zahlen `lvz` und produziere eine Liste mit zweielementigen Listen von korrespondierenden Symbolen und Zahlen.

Beispiel:

```
(cross '(a b c) '(3 5 9)) => '((a 3) (b 5) (c 9))
```

Lösung:

```
;; a)
;; Zweckbestimmung s. Aufgabenstellung
;;
;; Datendefinition:
;; Ein Symbol-Number-Pair ist (list s n),
;; wobei s ein Symbol und n eine Number ist.
;;
;; Eine Liste-von-SNPs ist entweder
;; 1. empty oder
;; 2. (cons p l), wobei p ein Symbol-Number-Pair und l eine Liste-von-SNPs ist
;;
;; cross : (list-of symbol) (list-of number) -> (list-of symbol-number-pair)
(check-expect (cross empty empty) empty)
(check-expect (cross '(a) '(3)) '((a 3)))
(check-expect (cross '(a b c) '(3 5 9)) '((a 3) (b 5) (c 9)))

(define cross
  (lambda [los lon]
    (cond
      [(empty? los) empty]
      [else (cons (list (first los) (first lon))
                  (cross (rest los) (rest lon)))])))

```

(b) CA. 5-6 MINUTEN

Die Funktion `shuffle` verarbeite 2 Listen `l1` und `l2` von Symbolen. Sie liefere eine Liste von Symbolen als Resultat mit abwechselnd je einem Element aus `l1` und `l2`.

Beispiel:

```
(shuffle '(a b c) '(u v w x y )) => '(a u b v c w x y)
```

Lösung:

```
;; b)
;; Zweckbestimmung s. Aufgabenstellung
;;
;; Datendefinition:
;; Eine Liste-von-Symbolen ist entweder
;; 1. empty oder
;; 2. (cons s l), wobei s ein Symbol und l eine Liste-von-Symbolen ist
;;
;; shuffle : (list-of symbol) (list-of symbol) -> (list-of symbol)
(check-expect (shuffle empty empty) empty)
(check-expect (shuffle empty '(b)) '(b))
(check-expect (shuffle '(a) empty) '(a))
(check-expect (shuffle '(a) '(b)) '(a b))
(check-expect (shuffle '(a b c) '(u v w x y )) '(a u b v c w x y))

(define shuffle
   (lambda [l1 l2]
      (cond
         [(and(empty? l1) (empty? l2)) empty]
         [(and(empty? l1) (cons? l2)) l2]
         [(and(cons? l1) (empty? l2)) l1]
         [else (cons (first l1)
                     (cons (first l2)
                           (shuffle (rest l1) (rest l2))))])))

```

Aufgabe 5

Gegeben sei die folgende Funktionsdefinition:

```
;; f: number number -> number
(define f
   (lambda [x y]
      (cond
         [(< x y) (- y x)]
         [(= x y) (+ x y)]
         [else (- x y)])))
```

Wenden Sie das Ersetzungsmodell für Funktionsanwendungen sowie die Auswertungsregeln für `cond` an, um die folgenden Ausdrücke Schritt für Schritt auszuwerten:

(a) CA. 8-10 MINUTEN

```
(+ (f 1 2) (f 2 2))
```

(b) CA. 6-7 MINUTEN

```
(f 12 (* 2 3))
```

Aufgabe 6

CA. 8 MINUTEN

Schreiben Sie eine Funktion, die eine Liste von Zahlen aufsteigend sortiert. Nutzen Sie eine Hilfsfunktion, die die Aufgabe mithilfe eines akkumulierenden Parameters löst. Falls sie weitere Hilfsfunktionen benötigen, definieren Sie diese.

Lösung:

```
;; Zweckbestimmung s. Aufgabenstellung
;; Eine Liste-von-Zahlen ist entweder
;; 1. empty oder
;; 2. (cons n l), wobei n eine Zahl und l eine Liste-von Zahlen ist
;; sort : (list-of number) -> (list-of number)
(check-expect (sortsj '(3 1 6 7 2 1)) '(1 1 2 3 6 7))
(check-expect (sortsj empty) empty)

(define sortsj
   (lambda [lon]
      (letrec [(sort-akku (lambda [sorted to-insert]
                            (cond
                               [(empty? to-insert) sorted]
                             [else (sort-akku
                                      (insert (first to-insert) sorted)
                                      (rest to-insert))])))]
               (sort-akku empty lon))))

(define insert
   (lambda [e lon]
      (cond
         [(empty? lon) (cons e empty)]
         [(<= e (first lon)) (cons e lon)]
         [else (cons (first lon)
                     (insert e (rest lon)))])))
```

Aufgabe 7

(a) CA. 3 MINUTEN

Abstrahieren Sie die beiden folgenden Funktionen in eine einzige Funktion:

```
;; mini: (list-of number) -> number
;; to determine the smallest number
;; on a non-empty list of numbers
(define mini
   (lambda [alon]
      (cond
         [(empty? (rest alon)) (first alon)]
         [else
            (cond
               [(< (first alon) (mini (rest alon))) (first alon)]
               [else (mini (rest alon))])])))
```

```
maxi: (list-of number) -> number
;; to determine the largest number
;; on a non-empty list of numbers
(define maxi (lambda [alon]
   (cond
      [(empty? (rest alon)) (first alon)]
      [else
         (cond
            [(> (first alon) (maxi (rest alon))) (first alon)]
            [else (maxi (rest alon))])])))
```

Lösung:

```
;; extremum : (list-of number) (number number -> number) -> number
(check-expect (extremum '(3 1 5 -2 5 7 4) <) -2)
(check-expect (extremum '(3 1 5 -2 5 7 4) >) 7)

(define extremum
   (lambda [alon f]
      (cond
         [(empty? (rest alon)) (first alon)]
         [else
            (cond
               [(> (first alon) (extremum (rest alon) f)) (first alon)]
               [else (extremum (rest alon) f)])])))
```

(b) CA. 1-2 MINUTEN

Warum sind die beiden Funktionen langsam?

Verbessern Sie die Effizienz der abstrahierten Funktion durch Benutzung von lokalen Definitionen!

Lösung:

```
;; extremum : (list-of number) (number number -> boolean) -> number
(check-expect (extremum '(3 1 5 -2 5 7 4) <) -2)
(check-expect (extremum '(3 1 5 -2 5 7 4) >) 7)

(define extremum
   (lambda [alon f]
      (let [(rest-ergebnis (cond
                             [(empty? (rest alon)) -1000]
                             [else (extremum (rest alon) f)]))]
         (cond
            [(empty? (rest alon)) (first alon)]
            [(f (first alon) rest-ergebnis) (first alon)]
            [else rest-ergebnis]))))

```

oder:

```
;; extremum: (list-of number) (number number -> boolean) -> number
(define extremum2
   (lambda [alon </>]
      (cond
         [(empty? (rest alon)) (first alon)]
         [else
          (let [(rest-ergebnis (extremum (rest alon) </>))]
               (cond
                  [(</> (first alon) rest-ergebnis) (first alon)]
                  [else rest-ergebnis]))])))
```

Aufgabe 8

CA. 6 MINUTEN

Schreiben Sie die beiden Funktion `quadsum` und `oddelements` aus Aufgabe 2 neu unter Verwendung von geeigneten Funktionen höherer Ordnung. Verwenden Sie dabei die aus der Vorlesung bekannten.

Lösung:

```
;; quadsum : (list-of number) -> number
(check-expect (quadsum '(1 2 3 4)) 30)

(define quadsum
   (lambda [lon]
      (foldl + 0 (map sqr lon))))


```

```
;; oddelements : (list-of number) -> (list-of number)
(check-expect (oddelements '(1 2 3 4 9 7 10 12)) '(1 3 9 7))

(define oddelements
   (lambda [lon]
      (filter odd? lon)))
```

Aufgabe 9

CA. 10 MINUTEN

Gegeben sei folgende Racket-Funktion

```
(define f
   (lambda [n]
      (cond
         [(= n 0) 0]
         [else (+ n (f (- n 1)))])))
```

Zeigen Sie, dass der Aufruf `(f n)` für alle natürlichen Zahlen $n \geq 0$ die Zahl $f(n) = \frac{n \cdot (n+1)}{2}$ berechnet.

Lösung:

Behauptung:

Der Aufruf `(f n)` liefert für jede natürliche Zahl ($n \geq 0$) den Wert $f(n) = \frac{n \cdot (n+1)}{2}$

Verankerung:

Der Aufruf (f 0) liefert nach dem Ersetzungsmodell:

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

$= \frac{0 \cdot (0+1)}{2}$

Für die Rekursionstiefe $k = 0$ liefert die Funktion 0.

Lösung:

Induktionsannahme:

Die Behauptung gilt für die Rekursionstiefe $k = m$.

Induktionsschluss:

Es ist zu zeigen, dass der Aufruf $(f\ (+m1))$ den Wert $\frac{(m+1)(m+2)}{2}$ als Resultat liefert.

Sei $x = (+\ m\ 1)$

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

Wechsel auf math. Notation:

$= (m\ +\ 1) + f(m\ +\ 1\ -\ 1)$

$= (m\ +\ 1) + f(m)$

$= (m\ +\ 1) + \frac{m \cdot (m\ +\ 1)}{2}$

$= \frac{2m\ +\ 2}{2} + \frac{m \cdot (m\ +\ 1)}{2}$

$= \frac{2m\ +\ 2\ +\ m^2\ +\ m)}{2}$

$= \frac{(m\ +\ 1)(m\ +\ 2)}{2}$