



Institut für Informatik  
Prof. Dr. Peter Thiemann  
Jochen Walter

Georges-Köhler-Allee 79  
D-79110 Freiburg i. Br.

Freiburg, den 9. November 2000

## Informatik 1, WiSe 2000/2001

### Übungsblatt 3

Die Aufgaben werden in den Übungs- und Programmiergruppen  
vom 16.11 bis zum 22.11. besprochen.

Die Lösungen müssen nicht abgegeben werden!

*Die Aufgaben auf diesem Blatt können in Teams bearbeitet werden. Bevor drscheme gestartet werden kann, muß setup lang eingegeben werden. Der Sprachumfang sollte auf „Advanced Student“ eingestellt werden. Die Aufgaben sind mit einem bis drei Sternen versehen, wobei die Aufgaben mit einem Stern am einfachsten, die mit dreien am schwierigsten sind.*

#### **Aufgabe 1 (\*\*):**

Seien  $\rho_1, \rho_2 \subseteq A \times A$  symmetrische Relationen. Beweisen Sie, daß auch  $\rho_1 \cap \rho_2$  symmetrisch ist.

#### **Lösung zu Aufgabe 1:**

Für alle  $(x, y) \in A \times A$  gilt:

$$\begin{aligned}(x, y) \in \rho_1 \cap \rho_2 &\Leftrightarrow (x, y) \in \rho_1 \wedge (x, y) \in \rho_2 \\ &\Leftrightarrow (y, x) \in \rho_1 \wedge (y, x) \in \rho_2 \\ &\Leftrightarrow (y, x) \in \rho_1 \cap \rho_2\end{aligned}$$

Die erste Äquivalenzaussage folgt aus der Definition des Schnittes von Relationen, die zweite aus der Definition von Symmetrie bei Relationen und die dritte wieder aus der Definition des Schnittes.

#### **Aufgabe 2 (\*\*):**

Beweisen Sie: Wenn  $\rho$  eine Relation auf  $A$  ist, dann gilt:

$$\rho \text{ ist symmetrisch} \Leftrightarrow \rho^{-1} \subseteq \rho.$$

**Lösung zu Aufgabe 2:** Für alle  $\rho \subseteq A \times A$  gilt:

$$\begin{aligned}\rho^{-1} \subseteq \rho &\Leftrightarrow \{(y, x) \mid (x, y) \in \rho\} \subseteq \rho \\ &\Leftrightarrow (\forall (x, y) \in \rho) (y, x) \in \rho\end{aligned}$$

Die erste Äquivalenzaussage folgt aus der Definition der Umkehrrelation, die zweite aus der Definition der Teilmenge.

### Aufgabe 3 (Programmierung, \*\*\*):

Ein  $n$ -Eck ( $n > 2$  läßt sich durch eine Liste seiner Eckpunkte  $((x_1, y_1), (x_2, y_2), \dots, (x_n, y_n))$  darstellen, wobei die Seiten des Polygons für alle  $i \in \{1, 2, \dots, n - 1\}$  von  $(x_i, y_i)$  nach  $(x_{i+1}, y_{i+1})$  und von  $(x_n, y_n)$  nach  $(x_1, y_1)$  gerichtet sind. Dabei dürfen zwei Ecken nicht zusammenfallen und zwei Seiten dürfen sich nicht schneiden.

Erweitern Sie die Datei von der Aufgabe 7 des vorigen Blattes um eine Datenstruktur für Polygone. Schreiben Sie eine Scheme-Funktion `polygon-move` mit der Signatur `polygon posn → polygon`, die eine um `posn` verschobene Kopie des Polygons zurückliefert.

### Lösung zu Aufgabe 3:

```
(load "ub2-7.scm")

;;; Datentyp "polygon"
;;; ein "polygon" ist eine Struktur (make-polygon vertices), wobei
;;; vertices eine Liste von "posn"s ist, die die Ecken des Polygons
;;; beschreiben. Diese Liste muß mindestens zwei Ecken enthalten!
;;; Zwei Ecken dürfen nicht zusammenfallen und zwei Seiten dürfen sich
;;; nicht schneiden.
(define-struct polygon (vertices))

;;; SIGNATUR
;;; polygon-move: polygon posn -> polygon
;;; ERKLÄRUNG
;;; (polygon-move poly p) erzeugt eine um p verschobene Kopie des Polygons
;;; poly.
;;; BEISPIEL
;;; (polygon-move (make-polygon (list (make-posn 0 0) (make-posn 1 0)
;;;                                   (make-posn 0.5 1))) (make-posn 5 1))
;;; => (make-polygon (list (make-posn 5 1) (make-posn 6 1) (make-posn 5.5 2)))
;;; DEFINITION
(define polygon-move
  (lambda (poly p)
    (make-polygon
     (list-posn-move (polygon-vertices poly) p))))

;;; SIGNATUR
;;; list-posn-move: list(posn) posn -> list(posn)
;;; ERKLÄRUNG
;;; (list-posn-move v p) erzeugt eine Liste, die aus den um p verschobenen
;;; Kopien der Komponenten von v besteht.
;;; BEISPIEL
;;; (list-posn-move (list (make-posn 0 1) (make-posn 1 1)) (make-posn 5 1))
;;; => (list (make-posn 5 2) (make-posn 6 2))
;;; DEFINITION
```

```
(define list-posn-move
  (lambda (v p)
    (if (null? v)
        '()
        (cons (posn-move (car v) p) (list-posn-move (cdr v) p))))))
```

#### Aufgabe 4 (Programmierung, \*\*):

- (a) Erweitern Sie die Datei aus der vorigen Aufgabe um eine Scheme-Funktion `polygon-area` mit der Signatur `polygon → number`. Verwenden Sie dazu folgendes Verfahren:

Für jede von  $(x_i, y_i)$  nach  $(x_j, y_j)$  gerichtete Seite  $(x_i, y_i) \rightarrow (x_j, y_j)$  betrachtet man den Flächeninhalt  $T_i$  des Trapezes  $(x_i, y_i) \rightarrow (x_j, y_j) \rightarrow (x_j, 0) \rightarrow (x_i, 0) \rightarrow (x_i, y_i)$ . Dieser ist durch

$$T_i = 1/2(x_j - x_i)(y_i + y_j)$$

gegeben. (Die Fläche von Trapezen ist dabei negativ, falls  $x_j < x_i$ ).

Die Fläche des Polygons  $A$  ist dann die Summe der Flächeninhalte der Trapeze:

$$A = \sum_{i=1}^n T_i.$$

(Falls die Ecken des Polygons im Uhrzeigersinn angegeben wurden, ist  $A > 0$ , andernfalls  $A < 0$ .)

- (b) Erweitern Sie nun die Scheme-Funktion `shape-area` aus `constr-04.ss` so, daß sie zu jedem Kreis, Rechteck, Dreieck, jeder Strecke und jedem Polygon den Flächeninhalt zurückliefert.

#### Lösung zu Aufgabe 4:

```
(load "ub3-3.scm")

;;; SIGNATUR
;;; polygon-area: polygon -> number
;;; ERKLÄRUNG
;;; (polygon-area poly) berechnet den Flächeninhalt von Polygon poly. Der
;;; Flächeninhalt ist dabei positiv, falls die Ecken des Polygons im
;;; Uhrzeigersinn angegeben wurden, negativ sonst.
;;; BEISPIELE
;;; (polygon-area (make-polygon (list (make-posn 0 1) (make-posn 1 1)
;;;                                   (make-posn 1 0))))
;;; => 0.5
;;; (polygon-area (make-polygon (list (make-posn 0 1) (make-posn 1 0)
;;;                                   (make-posn 1 1))))
;;; => -0.5
;;; DEFINITION
(define polygon-area
  (lambda (poly)
```

```

(sum-trapezium-areas (polygon-vertices poly))))

;;; SIGNATUR
;;; sum-trapezium-areas: list(posn) -> number
;;; ERKLÄRUNG
;;; (sum-trapezium-areas t) berechnet die Summe der Flächeninhalte der n
;;; Trapeze, die durch die Liste t = (p1, p2, ..., pn) wie folgt beschrieben
;;; werden:
;;; * (pn -> p1 -> P p1 -> P pn)
;;; * Für alle i in {1,2, n-1} (pi -> p{i+1} -> P p{i+1} -> P pi)
;;; Dabei ist mit P die Projektion auf die X-Achse gemeint.
;;; BEISPIEL
;;; (sum-trapezium-areas (list (make-posn 0 0) (make-posn 1 1) (make-posn 1 0)))
;;; => 0.5
;;; DEFINITION
(define sum-trapezium-areas
  (lambda (t)
    (+ (trapezium-area (car (reverse t)) (car t))
       (sum-trapezium-areas-1 t))))

;;; SIGNATUR
;;; sum-trapezium-areas-1: list(posn) -> number
;;; ERKLÄRUNG
;;; (sum-trapezium-areas-1 t) berechnet die Summe der Flächeninhalte der n-1
;;; Trapeze, die durch die Liste t = (p1, p2, ..., pn) wie folgt beschrieben
;;; werden:
;;; * Für alle i in {1,2, n-1} (pi -> p{i+1} -> P p{i+1} -> P pi)
;;; Dabei ist mit P die Projektion auf die X-Achse gemeint.
;;; BEISPIEL
;;; (sum-trapezium-areas-1 (list (make-posn 0 0) (make-posn 1 1)
;;;                               (make-posn 1 0)))
;;; => 0.5
;;; DEFINITION
(define sum-trapezium-areas-1
  (lambda (t)
    (cond
      ((null? t) ; Dieser Fall kann bei einem korrekten Aufruf nicht eintreten
       0)
      ((null? (cdr t))
       0)
      (#t
       (+ (trapezium-area (car t) (cadr t))
          (sum-trapezium-areas-1 (cdr t)))))))

;;; SIGNATUR

```

```

;;; trapezium-area: posn posn -> number
;;; ERKLÄRUNG
;;; (trapezium-area p1 p2) berechnet den Flächeninhalt des
;;; Trapezes, das durch p1 und p2 wie folgt beschrieben wird:
;;; * (p1 -> p1 -> P p2 -> P p1)
;;; Dabei ist mit P die Projektion auf die X-Achse gemeint.
;;; BEISPIEL
;;; (trapezium-area (make-posn 0 0) (make-posn 1 1))
;;; => 0.5
;;; DEFINITION
(define trapezium-area
  (lambda (p1 p2)
    (/ (* (- (posn-x p2) (posn-x p1))
          (+ (posn-y p1) (posn-y p2)))
       2)))

;;; SIGNATUR
;;; shape-area : shape -> number
;;; ERKLÄRUNG
;;; (shape-area s) berechnet den Flächeninhalt von s
;;; BEISPIELE
;;; siehe circle-area, rectangle-area, line-area
;;; DEFINITION
(define shape-area
  (lambda (s)
    (cond
      ((circle? s)
       (circle-area s))
      ((rectangle? s)
       (rectangle-area s))
      ((line? s)
       (line-area s))
      ; Bis hierhin war die Funktion vorgegeben
      ((triangle? s)
       (triangle-area s))
      ((polygon? s)
       (polygon-area s))))))

```

### Aufgabe 5 (Programmierung, \*\*):

Entwickeln Sie Datenstrukturen für eine Menge von Zootiere. Diese Menge beinhaltet

**Spinnen:** Die wichtigen Attribute sind die Anzahl der Beine (wir nehmen an, daß Spinnen bei Unfällen Beine verlieren können), Gefährlichkeit des Giftes und der Platzbedarf, der beim einem Transport nötig ist.

**Elefanten:** Einziges Attribut ist ihr Platzbedarf bei Transporten.

**Schlangen:** Attribute sind Gefährlichkeit des Giftes und Platzbedarf bei Transporten.

**Affen:** Attribute sind Intelligenz und Platzbedarf bei einem Transport.

Wir nehmen an, daß es nur würfelförmige Transportkisten gibt, so daß der Platzbedarf durch die Kantenlänge der Transportkisten angegeben werden kann. Wir nehmen weiter an, daß die Intelligenz und Gefährlichkeit des Giftes durch eine Zahl ausgedrückt werden können. Für Schlangen und Spinnen, die kein Gift absondern, ist diese Zahl gleich Null. Diese Datentypen bilden zusammen `animal`, einen Datentyp mit Varianten.

- (a) Programmieren Sie eine Scheme-Funktion `animal?` mit der Signatur `scheme-value` → `boolean`, die `#t` zurückgibt, falls ihr Argument ein `animal` ist, `#f` sonst.
- (b) Schreiben Sie eine Scheme-Funktion `animal-fits?` mit der Signatur `animal number` → `boolean`, die zurückliefert, ob das betreffende Tier in eine Transportkiste mit der gegebenen Kantenlänge paßt.
- (c) Schreiben Sie eine Scheme-Funktion `animal-poisonousness` mit der Signatur `animal` → `number`, die angibt, wie stark das Gift des Tieres wirkt. Für Tiere, die wie Elefanten kein Gift absondern, soll dabei 0 zurückgeliefert werden.

### Lösung zu Aufgabe 5:

```
;;; Datentyp "spider"
;;; Datentyp für die Darstellung von Spinnen
(define-struct spider (legs poisonousness space))

;;; Datentyp "elephant"
;;; Datentyp für die Darstellung von Elefanten
(define-struct elephant (space))

;;; Datentyp "snake"
;;; Datentyp für die Darstellung von Schlangen
(define-struct snake (poisonousness space))

;;; Datentyp "monkey"
;;; Datentyp für die Darstellung von Affen
(define-struct monkey (intelligence space))

;;; Datentyp "animal"
;;; Ein Wert vom Typ animal ist entweder ein
;;; 1. spider oder
;;; 2. elephant oder
;;; 3. snake oder
;;; 4. monkey

;;; SIGNATUR
```

```

;;; animal?: scheme-value -> boolean
;;; ERKLÄRUNG
;;; BEISPIEL
;;; DEFINITION
(define animal?
  (lambda (a)
    (or
     (spider? a)
     (elephant? a)
     (snake? a)
     (monkey? a))))

;;; SIGNATUR
;;; animal-fits?: animal number -> boolean
;;; ERKLÄRUNG
;;; (animal-fits? animal space) gibt #t zurück, falls das Zootier animal
;;; in einer Transportkiste mit der Seitenlänge space transportiert werden kann.
;;; BEISPIEL
;;; (animal-fits? (make-elephant 3) 2)
;;; => false
;;; DEFINITION
(define animal-fits?
  (lambda (animal space)
    (cond
     ((spider? animal)
      (<= (spider-space animal) space))
     ((elephant? animal)
      (<= (elephant-space animal) space))
     ((snake? animal)
      (<= (snake-space animal)))
     ((monkey? animal)
      (<= (monkey-space animal) space)))))

;;; SIGNATUR
;;; animal-poisonousness: animal -> number
;;; ERKLÄRUNG
;;; (animal-poisonousness animal) gibt an, wie gefährlich das von dem Tier
;;; abgesonderte Gift ist. Wenn das Tier kein Gift absondert, wird 0
;;; zurückgegeben;
;;; BEISPIELE
;;; (animal-poisonousness (make-elephant 3))
;;; => 0
;;; (animal-poisonousness (make-snake 3 0.5))
;;; => 3

```

```

;;; DEFINITION
(define animal-poisonousness
  (lambda (animal)
    (cond
      ((spider? animal)
       (spider-poisonousness animal))
      ((snake? animal)
       (snake-poisonousness animal))
      ((elephant? animal)
       0)
      ((monkey? animal)
       0))))

```

### Aufgabe 6 (Programmierung, \*\*):

Programmieren Sie eine Scheme-Funktion `list-member` mit der Signatur

$$\text{number list}(\text{number}) \rightarrow \text{boolean},$$

die `#t` zurückgibt, falls ihr erstes Argument gleich einer der Komponenten des zweiten Arguments ist, `#f` sonst. Die Gleichheit soll dabei mit `equal?` überprüft werden. Eine Standard-Scheme-Funktion mit ähnlicher Aufgabe ist `member`. Schreiben Sie Ihre Funktion, ohne auf `member` zurückzugreifen.

### Lösung zu Aufgabe 6:

```

;;; SIGNATUR
;;; list-member: number list(number) -> boolean
;;; ERKLÄRUNG
;;; (list-member n lst) liefert #t zurück, falls n gleich einer der
;;; Komponenten von lst ist, #f sonst. Die Gleichheit wird mit equal?
;;; überprüft.
;;; BEISPIEL
;;; (list-member 2 (list 1 2 3))
;;; => #t
;;; DEFINITION
(define list-member
  (lambda (n lst)
    (cond
      ((null? lst)
       #f)
      ((equal? (car lst) n)
       #t)
      (#t
       (list-member n (cdr lst))))))

```