

Informatik I

<http://www.informatik.uni-freiburg.de/proglang/teaching/aktuell/info1>

Übungsblatt 4

Abgabe: **15.11.2001**

Wir verwenden ab diesem Aufgabenblatt das Symbol * zur Kennzeichnung von optionalen Aufgaben(teilen).

Aufgabe 1: (Diese Aufgabe ist eine Präsenzaufgabe. Sie soll in den Übungsgruppen zusammen mit den Tutoren bearbeitet werden.)

Das *relationale Produkt* \circ zweier binärer Relationen R und S auf einer Menge A ist definiert als

$$R \circ S = \{(x, y) \in A \times A : \exists z. (x, z) \in R \wedge (z, y) \in S\}.$$

- (a) Zeige, daß das relationale Produkt das Distributivgesetz $R \circ (S \cup T) = (R \circ S) \cup (R \circ T)$ erfüllt.
- (b) Zeige, daß das relationale Produkt das folgende Monotoniegesetz erfüllt. Aus $S \subseteq T$ folgt $(R \circ S) \subseteq (R \circ T)$.

Lösung.

(ad a)

$$\begin{aligned} & (a, b) \in R \circ (S \cup T) \\ \Leftrightarrow & \{\text{Definition von } \circ, \text{ Mengenlehre}\} \\ & \exists x. ((a, x) \in R \wedge (x, b) \in S \cup T) \\ \Leftrightarrow & \{\text{Mengenlehre}\} \\ & \exists x. ((a, x) \in R \wedge ((x, b) \in S \vee (x, b) \in T)) \\ \Leftrightarrow & \{\text{Logik}\} \\ & (\exists x. ((a, x) \in R \wedge (x, b) \in S) \vee (\exists x. (a, x) \in R \wedge (x, b) \in T)) \\ \Leftrightarrow & \{\text{Definition von } \circ, \text{ Mengenlehre}\} \\ & (a, b) \in R \circ S \vee (a, b) \in R \circ T \\ \Leftrightarrow & \{\text{Mengenlehre}\} \\ & (a, b) \in (R \circ S) \cup (R \circ T). \end{aligned}$$

(ad b)

$$\begin{aligned} & (a, b) \in R \circ S \\ \Leftrightarrow & \{\text{Definition von } \circ, \text{ Mengenlehre}\} \\ & \exists x. ((a, x) \in R \wedge (x, b) \in S) \\ \Rightarrow & \{\text{Mengenlehre: } S \subseteq T \Leftrightarrow (\forall x. (x \in S \Rightarrow x \in T))\} \\ & \exists x. ((a, x) \in R \wedge (x, b) \in T) \\ \Leftrightarrow & \{\text{Definition von } \circ, \text{ Mengenlehre}\} \\ & (a, b) \in R \circ T. \end{aligned}$$

Aufgabe 2: Implementiere eine Scheme-Prozedur `string-match?`, welche testet, ob ein gegebener String ein Teilstring eines anderen gegebenen String ist. Ein String v ist ein *Teilstring* eines String w , falls es Strings a und b gibt, so daß $w = avb$. Dabei bedeutet ab , daß der String b an das Ende des String a angehängt wird. Beispielsweise liefert

```
> (string-match? "warze" "Schwarzenegger")
#t
> (string-match? "graz" "Schwarzenegger")
#f
```

Lösung.

```
(define string-match?
  (lambda (sub-string string)
    (string-match?-1 sub-string
                     string
                     0
                     (string-length sub-string)
                     (string-length string))))

(define string-match?-1
  (lambda (sub-string string pos sub-string-len string-len)
    (and (<= pos (- string-len sub-string-len))
         (or (string-match?-2 sub-string string pos 0 sub-string-len)
             (string-match?-1 sub-string
                              string
                              (+ pos 1)
                              sub-string-len
                              string-len)))))

(define string-match?-2
  (lambda (sub-string string pos offset sub-string-len)
    (or (>= offset sub-string-len)
        (and (char=? (string-ref string (+ pos offset))
                     (string-ref sub-string offset))
              (string-match?-2 sub-string
                               string
                               pos
                               (+ offset 1)
                               sub-string-len)))))
```

Aufgabe 3: Im Lande Manfredonien gibt es (in der Reihenfolge Ihres Standes) einen Herzog, Ritter und das gemeine Volk.

(a) Implementiere die Scheme-Datenstrukturen `herzog`, `ritter` und `gemeiner`.

- Ein Herzog hat einen Vornamen, eine Zahl und ein Land (z.B. *Luitpold II von Manfredonien*).
- ein Ritter hat einen Vornamen, einen Beinamen und ein Pferd,
- ein Gemeiner hat einen Vornamen, einen Nachnamen, ein Geschlecht und einen Beruf.

Alle Attributwerte sollen als Symbole angegeben werden.

(b) Betrachte eine Variante `person`, welche alle Personen mit ihrem Stand umfasst. Implementiere ein Prädikat `person?`, welches testet, ob ein gegebener Wert zum Variantentyp gehört.

(c *) Implementiere eine Operation `person-titel`, welcher zu jeder Person ihren vollen Titel ausgibt (z.B. *Seine Durchlaucht Herzog Luitpold der Zweite von Manfredonien* oder *Ritter Kunibert der Kühne*). Benutze dafür die eingebaute Scheme-Prozedur `string-append`.

(d) Implementiere die Prädikate `person-ranggleich?` und `person-ranghoeher?`, welcher zu jedem Paar von Personen angibt, ob sie den gleichen Rang haben bzw. ob die erste ranghöher ist als die zweite.

(e) Implementiere die Operation `person-degradiere`, mittels derer der Herzog einen Ritter oder einen Gemeinen zu einem Gemeinen mit Beruf *Bettler* degradieren kann. Dabei erhält der Ritter den Namen seines Pferds zum Nachnamen.

(f *) Gib eine Beispielsession (*Höfisches Treiben in Manfredonien*) in Scheme an, in welcher Personen mit verschiedenen Rängen definiert und die wichtigsten Prozeduren durchgespielt werden.

Lösung.

```
(define-struct herzog (vorname zahl land))
(define-struct ritter (vorname beiname pferd))
(define-struct gemeiner (vorname nachname geschlecht beruf))
```

```
(define person?
  (lambda (object)
    (or (herzog? object)
        (ritter? object)
        (gemeiner? object))))
```

```
(define person-anrede
  (lambda (person)
    (cond
      ((herzog? person) "Seine Durchlaucht")
      ((ritter? person) "Ritter"))
```

```

((and (gemeiner? person)
      (equal? (gemeiner-geschlecht person) 'weiblich))
 "Frau")
((and (gemeiner? person)
      (equal? (gemeiner-geschlecht person) 'maennlich))
 "Herr"))))

(define person-titel
  (lambda (person)
    (cond
      ((herzog? person)
       (string-append (person-anrede person)
                       " Herzog "
                       (symbol->string (herzog-vorname person))
                       " der "
                       (symbol->string (herzog-zahl person))
                       " von "
                       (symbol->string (herzog-land person))))
      ((ritter? person)
       (string-append (person-anrede person)
                       " "
                       (symbol->string (ritter-vorname person))
                       " "
                       (symbol->string (ritter-beiname person))))
      ((gemeiner? person)
       (string-append (person-anrede person)
                       " "
                       (symbol->string (gemeiner-vorname person))
                       " "
                       (symbol->string (gemeiner-nachname person)))))))

(define person-hoeher?
  (lambda (person-1 person-2)
    (cond
      ((herzog? person-1) (not (herzog? person-2)))
      ((ritter? person-1) (not (or (herzog? person-2) (ritter? person-2))))
      ((gemeiner? person-1) #f))))

(define person-ranggleich?
  (lambda (person-1 person-2)
    (and (not (person-hoeher? person-1 person-2))
         (not (person-hoeher? person-2 person-1)))))

(define person-vorname
  (lambda (person)
    (cond
      ((herzog? person) (herzog-vorname person))

```

```

      ((ritter? person) (ritter-vorname person))
      ((gemeiner? person) (gemeiner-vorname person))))))

(define person-neuename
  (lambda (person)
    (cond
      ((herzog? person) (herzog-land person))
      ((ritter? person) (ritter-pferd person))
      ((gemeiner? person) (gemeiner-nachname person))))))

(define person-geschlecht
  (lambda (person)
    (cond
      ((or (herzog? person)
           (ritter? person))
       'maennlich)
      ((gemeiner? person) (gemeiner-geschlecht person))))))

(define person-degradiere
  (lambda (person-1 person-2)
    (and (herzog? person-1)
         (or (ritter? person-2)
             (gemeiner? person-2))
         (make-gemeiner
          (person-vorname person-2)
          (person-neuename person-2)
          (person-geschlecht person-2)
          'bettler))))))

```

Aufgabe 4: *Lindenmayer-Systeme* oder *L-Systeme* dienen zur Modellierung der Morphogenese von Pflanzen.

Anschaulich werden Pflanzen durch die Bewegung einer Schildkröte erzeugt. Der Zustand einer Schildkröte ist ein Paar $t = (p, \alpha)$ aus einer Position $p = (x, y)$ und einem Winkel α , in dessen Richtung der Kopf der Schildkröte zeigt. Bei gegebener Schrittgröße d und Winkeländerung ϕ geht die Schildkröte unter folgenden Befehlen in einen Zustand t' über.

- F Gehe einen Schritt der Länge d vorwärts. Dann ist $t' = (p', \alpha)$ mit $p' = (x + d \cos(\phi), y + d \sin(\phi))$. Eine Linie zwischen p und p' wird gezeichnet.
- f Gehe einen Schritt wie unter F , aber ohne eine Linie zu zeichnen.
- $+$ Drehe nach rechts um den Winkel ϕ . Dann ist $t' = (p, \alpha + \phi)$.
- $-$ Drehe nach links um den Winkel ϕ . Dann ist $t' = (p, \alpha - \phi)$.
- [Markiere den gegenwärtigen Zustand.
-] Springe (!) in den Zustand mit der entsprechenden öffnenden Klammer zurück.

Alle anderen Befehle werden von der Schildkröte ignoriert.

Um eine Pflanze zu zeichnen, startet die Schildkröte in Zustand (p_0, α_0) und bewegt sich mit Hilfe von Regeln, die als Strings kodiert werden. Hier soll die Pflanze von dem L -System

$$X, \tag{1}$$

$$F \rightarrow FF, \tag{2}$$

$$X \rightarrow F[+X]F[-X] + X, \tag{3}$$

erzeugt werden. (1) beschreibt die Pflanze zur Generation 0. Die i -te Generation wird wiederum durch einen String S_i beschrieben, welcher mit Hilfe der Ersetzungsregeln (2) und (3) aus S_i erzeugt wird. Dazu ersetzt man jedes Vorkommen der linken Seite einer Ersetzungsregel in S_i durch die rechte Seite der Ersetzungsregel. So sind zum Beispiel

$$S_0 = X,$$

$$S_1 = F[+X]F[-X] + X,$$

$$S_2 = FF[+F[+X]F[-X] + X]FF[-F[+X]F[-X] + X] + F[+X]F[-X] + X$$

Wie werden nun die Strings S_0 und S_1 von unserer schlaunen Schildkröten gelesen? Im Falle von S_0 bleibt die Schildkröte regungslos; die Pflanze ist noch im Boden. Im Falle von S_1 macht die Schildkröte einen F -Schritt aus dem Anfangszustand. Dann markiert sie ihren Zustand, dreht sich nach rechts, überliest das X und springt in den markierten Zustand zurück. Als nächstes macht sie einen F -Schritt, markiert, dreht sich nach links, überliest ein weiteres X und springt in den zweiten markierten Zustand. Schließlich dreht sie sich nach rechts und überliest ein letztes X .

- (a) Implementiere obiges L -System. Setze zunächst $\phi = 30^\circ$ und $d = 6/10$. Berechne die durch das System beschriebene Pflanze (*Petroselinum crispum*) bis zur siebten Generation. Gehe dazu wie folgt vor.

1. Implementiere die Datenstruktur `turtle`, um den Zustand der Schildkröte als Paar einer Position und eines Winkels zu speichern.
2. Implementiere die Befehle an die Schildkröte als Scheme-Prozeduren

```
f-big : viewport turtle -> turtle
f-small : turtle -> turtle
plus: turtle -> turtle
minus : turtle -> turtle
```

3. (*) Implementiere Scheme-Prozeduren

```
p-X : viewport turtle number -> turtle
p-F : viewport turtle number -> turtle
```

für die Ersetzungsregeln (2) und (3). Das dritte Argument gibt die Generation an. Zum Beispiel ist

```

(define p-F
  (lambda (vp t n)
    (if (zero? n)
        (f-big vp t)
        (let*
            ((n1 (- n 1))
             (t1 (p-F vp t n1))
             (t2 (p-F vp t1 n1)))
          t2))))

```

4. (*) Implementiere die Scheme-Prozedur

```
draw-plant : viewport turtle number -> turtle
```

welche die initiale Ersetzung auf dem Ausgangsstring (1) modelliert.

5. Benutze das Hauptprogramm `gen-plant`, um die Pflanze in ein Fenster zu zeichnen. Das Argument `n` gibt die Generation an.

```

(define gen-plant
  (lambda (n)
    (open-graphics)
    (let*
        ((v (open-viewport "Petroselinum crispum" 500 500))
         (turtle (make-turtle (make-posn 250 450) 270)))
      (draw-plant v turtle n)
      (get-mouse-click v)
      (close-viewport v)
      (close-graphics)))

```

- (b *) Finde ein *L*-System für Deine Lieblingsblume und implementiere dieses in Scheme. Schicke das Ergebnis an Deinen Tutor.

Lösung.

(ad a)

```
(require-library "graphics.ss" "graphics")
```

```
(define-struct turtle (posn phi))
```

```
(define pi 22/7)
```

```
(define omega 30)
```

```
(define step 6/10)
```

```

(define plus
  (lambda (turtle)
    (make-turtle
     (turtle-posn turtle)
     (remainder (+ (turtle-phi turtle) omega) 360))))

```

```

(define minus
  (lambda (turtle)
    (make-turtle
     (turtle-posn turtle)
     (remainder (- (turtle-phi turtle) omega) 360))))

```

```

(define f-small
  (lambda (turtle)
    (let*
      ((pos (turtle-posn turtle))
       (phi (turtle-phi turtle))
       (phi-rad (grad->rad phi))
       (x1 (+ (posn-x pos) (* step (cos phi-rad))))
       (y1 (+ (posn-y pos) (* step (sin phi-rad))))
       (make-turtle (make-posn x1 y1) phi))))

```

```

(define grad->rad
  (lambda (n)
    (/ (* n 2 pi) 360)))

```

```

(define f-big
  (lambda (v turtle)
    (let* ((pos (turtle-posn turtle))
           (turtle1 (f-small turtle))
           (pos1 (turtle-posn turtle1)))
      ((draw-line v) pos pos1 (make-rgb 0 0.8 0))
      turtle1)))

```

```

(define p-X
  (lambda (vp t n)
    (if (zero? n)
        t
        (let*
          ((n1 (- n 1))
           (t1 (p-F vp t n1)) ; F
           (t2 (plus t1)) ; [+
           (t3 (p-X vp t2 n1)) ; X]
           (t4 (p-F vp t1 n1)) ; F
           (t5 (minus t4)) ; [-
           (t6 (p-X vp t5 n1)) ; X]
           (t7 (plus t4)) ; +
           (t8 (p-X vp t7 n1))) ; X
          t8))))

```

```

(define p-F
  (lambda (vp t n)

```

```

(if (zero? n)
    (f-big vp t)
    (let*
        ((n1 (- n 1))
         (t1 (p-F vp t n1)) ; F
         (t2 (p-F vp t1 n1))) ; F
        t2))))

(define draw-plant
  (lambda (vp t n)
    (p-X vp t n)))

(define gen-plant
  (lambda (n)
    (open-graphics)
    (let*
        ((v (open-viewport "Petroselinum crispum" 500 500))
         (turtle (make-turtle (make-posn 250 450) 270)))
        (draw-plant v turtle n)
        (get-mouse-click v)
        (close-viewport v))
      (close-graphics)))

```