



Institut für Informatik
Prof. Dr. Peter Thiemann
Jochen Walter

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

Freiburg, den 22. Dezember 2000

Informatik 1, WiSe 2000/2001

Übungsblatt 8

Die Aufgaben werden in den Übungs- und Programmiergruppen
vom 8.1. bis zum 12.1. 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 „Full Scheme“ 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 (*):

Zeigen Sie: Für jedes Rangalphabet Σ mit $\Sigma^{(0)} \neq \emptyset$ gibt es eine Σ -Algebra \mathcal{A} , deren Trägermenge A die Menge der Grundterme $T_\Sigma(\emptyset)$ über Σ ist. Geben Sie dafür ein Zuordnungsschema an, das jedem Operationssymbol $F \in \Sigma$ des Rangalphabetes eine entsprechende Funktion F_A zuordnet.

Lösung zu Aufgabe 1:

Man ordnet jedem n -stelligen Operationssymbol $F \in \Sigma$ eine Funktion

$$F_A : T_\Sigma(\emptyset) \times \cdots \times T_\Sigma(\emptyset) \rightarrow T_\Sigma(\emptyset),$$
$$F_A(t_1, \dots, t_n) = f t_1 \dots t_n$$

zu.

Aufgabe 2 (Programmierung, *):**

Die charakteristische Funktion χ_M einer Menge M ist durch

$$\chi_M(x) = \begin{cases} 1 & ; x \in M \\ 0 & ; \text{sonst} \end{cases}$$

definiert.

- Geben Sie nach dem Vorbild von Teil K10.1 der Vorlesung eine Implementierung `charfun-set` des abstrakten Datentypen `set(X)` an, die eine Menge durch ihre charakteristische Funktion darstellt. Dabei soll der Bildbereich der charakteristischen Funktion (im Gegensatz zu der obigen mathematischen Definition) der Datentyp `boolean` sein.
- Geben Sie (analog zu `list->list-set` aus K10.2.2) eine Funktion `fun->charfun-set` an, so dass `(fun->charfun-set = charfun)` eine generische Implementierung von `set(X)` ist (mit message-passing).

Lösung zu Aufgabe 2:

```
(a) ;;; Ein charfun-set(X) ist eine Struktur
(define-struct charfun-set (= chi))
;;; Dabei ist =: X X -> boolean und chi: X -> boolean

;;; SIGNATUR
;;; make-empty-charfun-set: (X X -> boolean) X -> boolean -> charfun-set(X)
;;; ERKLÄRUNG
;;; (make-empty-charfun-set = <) Erzeugt eine leere Menge in
;;; charfun-set-Darstellung mit Gleichheitsprädikat = und Ordnungsrelation <.
;;; DEFINITION
(define make-empty-charfun-set
  (lambda (= <)
    (make-charfun-set = (lambda (x) #f))))

;;; SIGNATUR
;;; charfun-set-insert: X charfun-set(X) -> charfun-set(X)
;;; ERKLÄRUNG
;;; (charfun-set-insert x s) erzeugt eine neue Mengendarstellung, die aus den
;;; Elementen von s und dem Objekt x besteht.
;;; DEFINITION
(define charfun-set-insert
  (lambda (x s)
    (let ((= (charfun-set-= s)))
      (make-charfun-set =
        (lambda (e)
          (or (= e x)
              ((charfun-set-chi s) e))))))))

;;; SIGNATUR
;;; charfun-set-remove: X charfun-set(X) -> charfun-set(X)
;;; ERKLÄRUNG
;;; (charfun-set-remove x s) erzeugt eine neue Mengendarstellung, die aus den
;;; Elementen von s ohne x besteht.
;;; DEFINITION
(define charfun-set-remove
  (lambda (x s)
    (let ((= (charfun-set-= s)))
      (make-charfun-set =
        (lambda (e)
          (and (not (= e x))
              ((charfun-set-chi s) e))))))))

;;; SIGNATUR
;;; charfun-set-elem: X charfun-set(X) -> charfun-set(X)
```

```

;;; ERKLÄRUNG
;;; (charfun-set-elem x s) liefert #t zurück, falls x in der
;;; Mengendarstellung s enthalten ist, #f sonst.
;;; DEFINITION
(define charfun-set-elem
  (lambda (x s)
    ((charfun-set-chi s) x)))
(b) ;;; SIGNATUR
;;; fun->charfun-set: (X X -> boolean) (X -> boolean) -> charfun-set(X)
;;; charfun-set(X) = symbol ->
;;; (X -> charfun-set(X) + X -> charfun-set(X) + X -> boolean)
;;; ERKLÄRUNG
;;; (fun->charfun-set = f) liefert eine Mengendarstellung, deren
;;; charakteristische Funktion durch f gegeben ist. Als
;;; Gleichheitsoperation wird = verwendet.
;;; DEFINITION
(define fun->charfun-set
  (lambda (= f)
    (lambda (op)
      (case op
        ((insert)
         (lambda (x)
           (fun->charfun-set = (lambda (e)
                                (or (= e x)
                                    (f e)))))))
        ((remove)
         (lambda (x)
           (fun->charfun-set = (lambda (e)
                                (and (not (= e x))
                                     ((charfun-set-chi s) e))))))
        ((elem)
         (lambda (x)
           (f x)))))))

```

Aufgabe 3 (Programmierung, **):

In der Vorlesung haben Sie Datenstrukturen wie `(cons a b)`, `(make-posn x y)`, `(make-rectangle ...)` kennengelernt, bei denen sich unter einem Konstruktor eine feste Anzahl von Elementen befindet. Bei `cons` sind diese durch die Selektoren `car` und `cdr` zu erreichen und durch die Mutatoren `set-car!` und `set-cdr!` zu verändern.

Demgegenüber gibt es Datenstrukturen, bei denen unter *einem* Konstruktor eine *beliebig festlegbare* Anzahl von Elementen befindet. In Scheme ist eine solche Datenstruktur der Vektor (auch Reihung oder Array genannt). Ein Vektor mit n Elementen, die alle den Wert i haben, kann durch den *Konstruktor*

```
(make-vector n i)
```

erzeugt werden. Durch den *Selektor*

```
(vector-ref vector k)
```

kann man das k -te Element des Vektors `vector` bestimmen. Durch den *Mutator*

```
(vector-set! vector k obj)
```

wird das k -te Element des Vektors `vector` auf den Wert `obj` gesetzt.

- (a) Schreiben Sie eine Funktion `histogram` mit der Signatur `list(number) number number → vector(number)`, die bei einem Aufruf `(histogram numbers low up)` die Häufigkeitsverteilung der Zahlen in `numbers` zwischen `low` und `up` berechnet.

Dabei sind die Zahlen in `numbers` sowie die untere und obere Grenze ganze Zahlen und $low \leq up$. Die Länge des zurückgegebenen Vektors beträgt $up - low + 1$. Das erste Element des Vektors ist die Anzahl der Vorkommen der Zahl `low` in `numbers`, das zweite Element ist die Anzahl der Vorkommen der Zahl `low+1` in `numbers` usw. Zahlen, die kleiner als `low` oder größer als `up` sind, werden nicht berücksichtigt.

- (b) Freiwillige Zusatzaufgabe: Schreiben Sie eine Scheme-Funktion `show-histogram`, die ein Histogramm graphisch darstellt.

Lösung zu Aufgabe 3:

```
;;; SIGNATUR
;;; histogram: list(number) number number -> vector(number)
;;; ERKLÄRUNG
;;; (histogram numbers low up) erzeugt einen Vektor mit up-low+1 Elementen,
;;; deren i-tes die Anzahl der Vorkommen der Zahl i+low in numbers angibt.
;;; Zahlen, die <low oder >up sind, werden dabei nicht berücksichtigt.
;;; BEISPIEL
;;; (histogram '(1 2 4 4 4 10) 1 4)
;;; => (vector 1 1 0 3)
;;; DEFINITION
(define histogram
  (lambda (numbers low up)
    (let ((frequency (make-vector (add1 (- up low)) 0)))
      (let loop ((n numbers))
        (if (null? n)
            frequency
            (begin
              (if (<= low (car n) up)
                  (vector-inc! frequency (- (car n) low)))
              (loop (cdr n))))))))))

;;; SIGNATUR
;;; vector-inc!: vector(number) number -> VOID
;;; ERKLÄRUNG
;;; (vector-inc! vector k) erhöht das k-te Element in vector. Dabei muß k
;;; ein gültiger Index von vector sein.
```

```

;;; BEISPIEL
;;; (define v (vector 1 2 3))
;;; (vector-inc! v 0)
;;; v => (vector 2 2 3)
;;; DEFINITION
(define vector-inc!
  (lambda (vector k)
    (vector-set! vector k (add1 (vector-ref vector k)))))

```

Aufgabe 4 (***):

In Teil K11.4.4 der Vorlesung wurde die Funktion `my-cons` eingeführt, mit deren Hilfe man Listen darstellen kann. Die folgenden zwei Scheme-Ausdrücke verwenden diese Definition.

```

(load "constr-10.ss")

(define fl (my-cons 'sr 'mm))

(begin (my-set-car! fl 'ss)
       (my-car fl))

```

Werten Sie diese Ausdrücke schrittweise von Hand mit dem Ausführungsmodell aus Teil K11.3 der Vorlesung aus. Die wichtigen Schritte sind die Funktionsanwendungen.

Lösung zu Aufgabe 4:

Der erste Ausdruck lautet

```
(define fl (my-cons 'sr 'mm))
```

Einsetzen der Definition von `my-cons`:

```

(define fl
  (lambda (x y)
    (make-my-pair
     (lambda (op)
       (case op
         ((car) x)
         ((cdr) y)
         ((set-car!) (lambda (v) (set! x v)))
         ((set-cdr!) (lambda (v) (set! y v)))))) 'sr 'mm))

```

Regel für Funktionsanwendung K11.3:

```

(define x-000000 'sr)
(define y-000000 'mm)
(define fl
  (make-my-pair
   (lambda (op)
     (case op
       ((car) x-000000)

```

```

((cdr) y-000000)
((set-car!) (lambda (v) (set! x-000000 v)))
((set-cdr!) (lambda (v) (set! y-000000 v))))))

```

Das Ergebnis der Auswertung von (make-my-pair ...) ist (make-my-pair ...). Nun zum zweiten Ausdruck:

```

(begin (my-set-car! fl 'ss)
      (my-car fl))

```

Einsetzen der Definition von my-set-car!:

```

(begin ((lambda (p v) ((my-pair-fun p) 'set-car!) v)) fl 'ss)
      (my-car fl))

```

Einsetzen der Definition von fl:

```

(define x-000000 'sr)
(define y-000000 'mm)
(begin ((lambda (p v) ((my-pair-fun p) 'set-car!) v))
      (make-my-pair
        (lambda (op)
          (case op
            ((car) x-000000)
            ((cdr) y-000000)
            ((set-car!) (lambda (v) (set! x-000000 v)))
            ((set-cdr!) (lambda (v) (set! y-000000 v))))))
      'ss)
      (my-car fl))

```

Die Reihenfolge der Auswertung innerhalb eines Funktionsaufrufes ist im R5RS-Scheme-Standard nicht festgelegt. Wir hätten deshalb auch fl vor my-set-car! einsetzen können. DrScheme wertet, wie wir, die Ausdrücke von links nach rechts aus.

Funktionsanwendung (nach der alten Definition):

```

(define x-000000 'sr)
(define y-000000 'mm)
(begin ((my-pair-fun (make-my-pair
  (lambda (op)
    (case op
      ((car) x-000000)
      ((cdr) y-000000)
      ((set-car!) (lambda (v) (set! x-000000 v)))
      ((set-cdr!) (lambda (v) (set! y-000000 v))))))
  'set-car!) 'ss)
      (my-car fl))

```

Anwendung des Selektors my-pair-fun:

```

(define x-000000 'sr)
(define y-000000 'mm)
(begin ((lambda (op)
        (case op
          ((car) x-000000)
          ((cdr) y-000000)
          ((set-car!) (lambda (v) (set! x-000000 v)))
          ((set-cdr!) (lambda (v) (set! y-000000 v))))
       'set-car!) 'ss)
(my-car fl))

```

Funktionsanwendung (nach der alten Definition):

```

(define x-000000 'sr)
(define y-000000 'mm)
(begin ((case 'set-car!
        ((car) x-000000)
        ((cdr) y-000000)
        ((set-car!) (lambda (v) (set! x-000000 v)))
        ((set-cdr!) (lambda (v) (set! y-000000 v))))
       'ss)
(my-car fl))

```

Auswertung des case-Ausdrucks (wir überspringen die Details):

```

(define x-000000 'sr)
(define y-000000 'mm)
(begin ((lambda (v) (set! x-000000 v))
       'ss)
(my-car fl))

```

Funktionsanwendung (nach der alten Definition):

```

(define x-000000 'sr)
(define y-000000 'mm)
(begin (set! x-000000 'ss)
(my-car fl))

```

Auswertung des set!-Ausdrucks:

```

(define x-000000 'ss)
(define y-000000 'mm)
(begin (my-car fl))

```

Einsetzen der Definition von my-car:

```

(define x-000000 'ss)
(define y-000000 'mm)
(begin ((lambda (p) ((my-pair-fun p) 'car)) fl))

```

Funktionsanwendung (nach der alten Definition):

```
(define x-000000 'ss)
(define y-000000 'mm)
(begin ((my-pair-fun fl) 'car))
```

Einsetzen der Definition von fl: (my-pair-fun ist eine spezielle Funktion, deren Definition wir nicht kennen. Wir schreiben für das Ergebnis der Auswertung von my-pair-fun wieder my-pair-fun).

```
(define x-000000 'ss)
(define y-000000 'mm)
(begin ((my-pair-fun (make-my-pair
  (lambda (op)
    (case op
      ((car) x-000000)
      ((cdr) y-000000)
      ((set-car!) (lambda (v) (set! x-000000 v)))
      ((set-cdr!) (lambda (v) (set! y-000000 v)))))) 'car))
```

Auswertung von my-pair-fun:

```
(define x-000000 'ss)
(define y-000000 'mm)
(begin (
  (lambda (op)
    (case op
      ((car) x-000000)
      ((cdr) y-000000)
      ((set-car!) (lambda (v) (set! x-000000 v)))
      ((set-cdr!) (lambda (v) (set! y-000000 v)))))) 'car))
```

Funktionsanwendung (nach der alten Definition):

```
(define x-000000 'ss)
(define y-000000 'mm)
(begin
  (case 'car
    ((car) x-000000)
    ((cdr) y-000000)
    ((set-car!) (lambda (v) (set! x-000000 v)))
    ((set-cdr!) (lambda (v) (set! y-000000 v)))))
```

Auswertung des case-Ausdrucks (wir überspringen die Details):

```
(define x-000000 'ss)
(define y-000000 'mm)
(begin x-000000)
```

Einsetzen der Definition von x-000000:

```
(define x-000000 'ss)
(define y-000000 'mm)
(begin 'ss)
```

Auswertungsregel für begin:

```
(define x-000000 'ss)
(define y-000000 'mm)
'ss
```

Aufgabe 5 (Programmierung, **):

In Teil K11.5.2 der Vorlesung wurden Funktionen zum Rechnen mit Strömen gegeben.

- Definieren Sie eine Funktion `stream-ref` : `stream(X) number -> X`, welche, für $i \geq 0$, das i -te Element aus einem Strom liefert. (So soll `(stream-ref primes 500)` die 500-te Primzahl liefern.)
- Definieren Sie einen Strom `fibonacci` : `stream(number)`, so daß `(stream-ref fibonacci i)` genau die i -te Fibonacci Zahl F_i liefert (mit $F_0 = 0$, $F_1 = 1$, $F_{n+2} = F_n + F_{n+1}$). Hinweis: definieren Sie `fibonacci` rekursiv!
- Definieren Sie nun eine Funktion `fib` : `number -> number`, die unter Verwendung des Stroms `fibonacci` die Fibonacci-Funktion berechnet, d.h. $(\text{fib } i) = F_i$ (für $i > 0$). Lassen Sie sich hintereinander F_{10000} und F_{10001} berechnen. Warum benötigt der Scheme-Interpreter für die zweite Berechnung weniger Zeit, obwohl das Ergebnis größer ist?

Lösung zu Aufgabe 5:

```
(load "constr-10.ss")
```

```
;;; SIGNATUR
;;; stream-ref: stream(X) number -> X
;;; ERKLÄRUNG
;;; (stream-ref s i) liefert das i-te Element des Stromes s, wobei die
;;; Zählung bei 0 anfängt. s muss entweder unendlich sein oder eine Länge
;;; größer als i haben.
;;; DEFINITION
(define stream-ref
  (lambda (s i)
    (if (zero? i)
        (stream-head s)
        (stream-ref (stream-tail s) (sub1 i)))))

;;; SIGNATUR
;;; fibo: number number -> stream(number)
;;; ERKLÄRUNG
;;; (fibo f0 f1) liefert einen Strom zurück, dessen erste zwei Elemente durch
;;; f0 und f1 gegeben sind, und deren weitere Elemente durch das Bildungsgesetz
;;;  $f_{n+2} = f_{n+1} + f_n$  bestimmt sind.
;;; DEFINITION
(define fibo
```

```

(lambda (f0 f1)
  (make-stream-cons
   f0
   (delay (fibo f1 (+ f0 f1))))))

;;; SIGNATUR
;;; fibonacci: stream(number)
;;; ERKLÄRUNG
;;; fibonacci ist der Strom der Fibonacci-Zahlen
;;; DEFINITION
(define fibonacci (fibo 0 1))

;;; SIGNATUR
;;; fib: number -> number
;;; ERKLÄRUNG
;;; (fib i) ist die i-te Fibonacci-Zahl
(define fib
  (lambda (i)
    (stream-ref fibonacci i)))

```

Die zweite Berechnung geht schneller, da der Strom nach der ersten Berechnung bereits alle Fibonacci-Zahlen bis F_{1000} enthält.