
Informatik I

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

Übungsblatt 6

 Abgabe: **29.11.2001**
Aufgabe 1:

- (a) Definiere eine einfache Datenstruktur `pair` für Paare (a, b) von Elementen mit Selektoren `fst` und `snd` für die erste und die zweite Komponente.
- (b) Implementiere eine Funktion `zip`, welche zwei Listen in eine Liste von Paaren konvertiert. Beispielsweise ist

```
> (define x (zip (list 1 2 3 4 5) (list 'a 'b 'c)))
> x
(list (make-pair 1 'a) (make-pair 2 'b) (make-pair 3 'c))
```

Lösung.

(ad a)

```
(define-struct pair (fst snd))
```

(ad b)

```
(define zip
  (lambda (l1 l2)
    (cond
      ((empty? l1) empty)
      ((cons? l1)
       (cond
         ((empty? l2) empty)
         ((cons? l2)
          (cons (make-pair (first l1) (first l2))
                (zip (rest l1) (rest l2))))))))))
```

Aufgabe 2: Betrachte die Menge T_Σ der Σ -Terme mit $\Sigma = \{l^{(0)}, n^{(2)}\}$.

- (a) Gebe Σ -Algebren \mathcal{L} und \mathcal{N} an, so daß $\hat{f}_L(t)$ die Anzahl der l in t und $\hat{f}_N(t)$ die Anzahl der n in t ist.
- (b) Zeige $\hat{f}_N(t) + 1 = \hat{f}_L(t)$ für alle $t \in T_\Sigma$.

Lösung.

(ad a) Setze $\mathcal{L} = (\mathbb{N}, l_L, n_L)$ mit $l_L = 1$ und $n_L(x, y) = x + y$. Dann ist \hat{f}_L die induzierte Interpretationsfunktion, um die Anzahl der Vorkommen von l in einem Term zu messen.

Setze $\mathcal{N} = (\mathbb{N}, l_N, n_N)$ mit $l_N = 0$ und $n_N(x, y) = x + y + 1$. Dann ist \hat{f}_N die induzierte Interpretationsfunktion, um die Anzahl der Vorkommen von n in einem Term zu messen.

(ad b) Zu zeigen ist $\hat{f}_N(t) + 1 = \hat{f}_L(t)$. Beweis durch Terminduktion.

Basis: $\hat{f}_N(l) + 1 = l_N + 1 = 0 + 1 = l_L = \hat{f}_L(l)$.

Schritt:

$$\begin{aligned}
 & \hat{f}_N(n t_1 t_2) + 1 \\
 = & \quad \{\text{Def. } \hat{f}_N\} \\
 & n_N(\hat{f}_N(t_1), \hat{f}_N(t_2)) + 1 \\
 = & \quad \{\text{Def. } n_N\} \\
 & (\hat{f}_N(t_1) + \hat{f}_N(t_2) + 1) + 1 \\
 = & \quad \{\text{Arithmetik}\} \\
 & (\hat{f}_N(t_1) + 1) + (\hat{f}_N(t_2) + 1) \\
 = & \quad \{\text{Induktionsannahme}\} \\
 & \hat{f}_L(t_1) + \hat{f}_L(t_2) \\
 = & \quad \{\text{Def. } n_L\} \\
 & n_L(\hat{f}_L(t_1), \hat{f}_L(t_2)) \\
 = & \quad \{\text{Def. } \hat{f}_L\} \\
 & \hat{f}_L(n t_1 t_2).
 \end{aligned}$$

Aufgabe 3:

- Implementiere eine Scheme-Prozedur `number*-min : number* number* -> number*`, welche das Minimum zweier Zahlen aus $\mathbb{Q} \cup \{\pm\infty\}$ berechnet.
- Implementiere eine Scheme-Prozedur `btree-min : btree(number) -> number*`, welche das minimale Element eines Suchbaums über Zahlen berechnet (oder `+infty` für den leeren Baum).

Lösung.

(ad a)

```

(define number*-min
  (lambda (a b)
    (cond
      ((equal? a '-infty)
       (not (equal? a '-infty)))
      ((number? a)
       (or (equal? b '+infty)
           (and (number? b) (< a b))))
      ((equal? a '+infty)
       #f))))

```

```

(define number*-min
  (lambda (a b)
    (if (number*-< a b)
        a
        b)))

(ad b)

(define-struct branch (left elem right))

(define number*-min-3
  (lambda (a b c)
    (number*-min a (number*-min b c))))

(define btree-min
  (lambda (t)
    (cond
      ((empty? t) '+infty)
      ((branch? t)
       (number*-min-3 (btree-min (branch-left t))
                      (branch-elem t)
                      (btree-min (branch-right t)))))))

```

Aufgabe 4: Betrachte die Scheme-Prozeduren

```

set-elem? : number set-number -> boolean
set-insert : number set-number -> set-number
set-union : set-number set-number -> set-number

```

`set-elem?` und `set-insert` wurden bereits in der Vorlesung mit Hilfe von Listen implementiert. `set-union` berechnet die Vereinigung zweier Mengen.

- (a) Implementiere `set-union` mit Hilfe von Listen.
- (b) Implementiere die drei Mengenoperationen mit Hilfe aufsteigend sortierter Listen.
- (c) Implementiere die drei Mengenoperationen mit Hilfe von Suchbäumen.
- (d *) Betrachte Mengen als Funktionen vom Typ `number -> boolean`. Diese Funktion ordnet einer Zahl `#t` zu, falls diese in der Menge ist und sonst `#f`. Implementiere die drei Mengenoperationen mit Hilfe solcher Funktionen.
- (e *) Diskutiere Vor- und Nachteile der Repräsentationen.

Hinweis: Einige der zu implementierenden Prozeduren können aus der Vorlesung übernommen werden.

Lösung.

(ad a)

```
;; Implementierung von Mengen als unsortierte Listen
;; (mit Wiederholungen)
```

```
(define set-union
  (lambda (s1 s2)
    (cond
      ((empty? s1) s2)
      ((cons? s1)
       (set-insert (first s1)
                   (set-union (rest s1) s2))))))
```

(ad b)

```
;; Implementierung von Mengen als aufsteigend sortierte Listen
;; (ohne Wiederholungen)
```

```
(define set-elem?
  (lambda (x s)
    (cond
      ((empty? s) #f)
      ((< x (first s)) #f)
      ((= x (first s)) #t)
      ((> x (first s)) (set-elem? x (rest s)))))
```

```
(define set-insert
  (lambda (x s)
    (cond
      ((empty? s) (list x))
      ((< x (first s)) (cons x s))
      ((= x (first s)) s)
      ((> x (first s)) (cons (first s) (set-insert x (rest s)))))
```

```
(define set-union
  (lambda (s1 s2)
    (cond
      ((empty? s1) s2)
      ((empty? s2) s1)
      (else
       (let ((x1 (first s1)) (x2 (first s2)))
         (cond
           ((= x1 x2) (cons x1 (set-union (rest s1) (rest s2))))
           ((< x1 x2) (cons x1 (set-union (rest s1) s2)))
           ((> x1 x2) (cons x2 (set-union s1 (rest s2))))))))))
```

(ad c)

```
;; Implementierung von Mengen als Suchbäume
```

```

(define set-elem? btree-member)

(define set-insert btree-insert)

(define set-union
  (lambda (s1 s2)
    (cond
      ((empty? s1) s2)
      ((branch? s1)
       (set-union (branch-right s1)
                  (set-union (branch-left s1)
                            (set-insert (branch-elem s1) s2)))))))

(ad d)

;; Implementierung von Mengen als
;; charakteristische Funktion des Types number -> boolean

(define set-empty
  (lambda (x)
    #f))

(define set-elem?
  (lambda (x s)
    (s x)))

(define set-insert
  (lambda (x s)
    (lambda (n)
      (or (= x n)
          (s n)))))

(define set-union
  (lambda (s1 s2)
    (lambda (n)
      (or (s1 n)
          (s2 n)))))

```

(ad e) Implementiert man Mengen als Listen, so benötigt `set-elem?` im schlimmsten Fall so viele rekursive Aufrufe, wie die Liste Elemente hat. `Set-insert` benötigt einen Berechnungsschritt und `set-union` benötigt im schlimmsten Fall so viele rekursive Aufrufe, wie die zweite Liste Elemente hat. `Set-insert` ist allerdings so implementiert, daß ein Element mehrfach in die Liste eingefügt werden kann.

Implementiert man Mengen als geordnete Listen, so benötigt `set-elem?` im schlimmsten Fall so viele Schritte, wie bei der Listenrepräsentation, allerdings kann man in der Regel auch dann vor Ende der Liste aufhören, falls das Element nicht in der Liste ist. Dies verbessert die mittlere Laufzeit. `Set-insert`


```

      ((mult? term) (let ((rand1 (mult-rand1 term))
                          (rand2 (mult-rand2 term)))
                      (make-add (make-mult rand1 (derive rand2 var))
                                (make-mult rand2 (derive rand1 var))))))

(ad b)

(define-struct pair (fst snd))

(define lookup
  (lambda (fun arg)
    (cond
      ((empty? fun) (error "undefined" arg))
      ((cons? fun)
       (let ((p (first fun)))
         (if (equal? (pair-fst p) arg)
             (pair-snd p)
             (lookup (rest fun) arg)))))))

(define kata
  (lambda (t algebra env)
    (cond
      ((var? t) (lookup env (var-name t)))
      ((null? t) (lookup algebra 'null))
      ((one? t) (lookup algebra 'one))
      ((add? t) (let ((op (lookup algebra 'add))
                      (v1 (kata (add-rand1 t) algebra env))
                      (v2 (kata (add-rand2 t) algebra env)))
                  (op v1 v2)))
      ((mult? t) (let ((op (lookup algebra 'mult))
                      (v1 (kata (mult-rand1 t) algebra env))
                      (v2 (kata (mult-rand2 t) algebra env)))
                  (op v1 v2))))))

(define algebra-numbers
  (list (make-pair 'null 0)
        (make-pair 'one 1)
        (make-pair 'add +)
        (make-pair 'mult *)))

(define eval
  (lambda (term env)
    (kata term algebra-numbers env)))

;; pretty-printer

(define algebra-pretty-print
  (list (make-pair 'null "0")
        (make-pair 'one "1")
        (make-pair 'add "+")
        (make-pair 'mult "*")))

```

```

      (make-pair 'one "1")
      (make-pair 'add (lambda (r1 r2) (string-append "(" r1 "+" r2 " ")))
      (make-pair 'mult (lambda (r1 r2) (string-append "(" r1 "*" r2 " "))))))

(define pretty-print
  (lambda (term env)
    (kata term algebra-pretty-print env)))

;; small example

(define x (make-var 'x))
(define y (make-var 'y))

(define term (make-add (make-mult x y)
                       (make-mult (make-one) x)))

(define env-1 (list (make-pair 'x 10)
                   (make-pair 'y 3)))

(define env-2 (list (make-pair 'x "x")
                   (make-pair 'y "y")))

(pretty-print term env-2)
(pretty-print (derive term x) env-2)
(pretty-print (derive term y) env-2)
(eval term env-1)

```

Aufgabe 6:

- Benutze `list-map`, um eine Scheme-Prozedur `even-odd` zu implementieren, welche Zahlen einer Liste jeweils nach `'even` oder `'odd` abbildet.
- Benutze `list-map`, um eine Scheme-Prozedur `apply-all` zu implementieren, welche eine Liste von Prozeduren und einen Wert nimmt und eine Liste ausgibt, deren Elemente die Anwendungen der Prozeduren auf den Wert sind. Beispiel

```

> (apply-all (list sqrt sqare cube factorial fib) 4)
(2 16 64 24 3)
> (apply-all (list length first last reverse) (list 1 2 3 4))
(4 1 (2 3 4) (4 3 2 1))

```

- Benutze die Funktion `filter2`, um eine Scheme-Prozedur `count-zeroes` zu implementieren, welche alle in einer Liste von Zahlen vorkommenden Nullen zählt.
- Benutze die Funktion `list-fold`, um eine Scheme-Prozedur `list-length` zu implementieren. welche die Länge einer Liste berechnet.

- (e) Benutze die Funktion `list-fold`, um eine Scheme-Prozedur `apply-composition` zu implementieren, welche eine List von Prozeduren und ein Argument nimmt und den Wert der Komposition der Funktionen angewandt auf das Argument berechnet.

```
> (apply-composition (list square factorial) 3)
36
> (apply-composition (list first rest reverse) (list 1 2 3 4))
3
```

Lösung.

(ad a)

```
(define even-odd
  (lambda (l)
    (list-map (lambda (x) (if (even? x) 'even 'odd))
              l)))
```

(ad b)

```
(define apply-all
  (lambda (list-of-procs value)
    (list-map (lambda (f) (f value))
              list-of-procs)))
```

(ad c)

```
(define count-zeroes
  (compose list-length
           (filter2 (lambda (x) (= x 0)))))
```

(ad d)

```
(define list-length
  (lambda (l)
    (list-fold (lambda (x a) (+ a 1))
              0
              l)))
```

(ad e)

```
(define apply-composition
  (lambda (list-of-procs value)
    (list-fold (lambda (proc a) (proc a))
              value
              list-of-procs)))
```

;; oder auch ...

```
(define id (lambda (x) x))
```

```
(define apply-composition-2
  (lambda (list-of-procs)
    (list-fold compose id list-of-procs)))

(define apply-composition
  (lambda (list-of-procs v)
    ((apply-composition-2 list-of-procs) v)))
```