Übungsaufgaben: Informatik I WS 16/17
Musterlösung

Steffen Lindner

30. Januar 2017

1 Records

1. Schreibe eine Daten- und Recorddefinition für 3-dimensionale Vektoren (vector).

   Lösung:

```
; Ein Vektor (vector) besteht aus
; einer x-Koordinate (vector-x)
; einer y-Koordinate (vector-y)
; einer z-Koordinate (vector-z)
(: make-vector (number number number -> vector))
(: vector? (vector -> boolean))
(: vector-x (vector -> number))
(: vector-y (vector -> number))
(: vector-z (vector -> number))
(define-record-procedures vector
  make-vector
  vector?
  (vector-x
   vector-y
   vector-z))
```

2. Schreibe eine Prozedur, die die Länge eines Vectors berechnet.

   Lösung:

```
(: vector-length (vector -> number))
(check-expect (vector-length v1) 1)
(check-within (vector-length v2) (sqrt 14) 0.01)
(define vector-length
  (lambda (v)
    (sqrt
```

```
(+
 (expt (vector-x v) 2)
 (expt (vector-y v) 2)
 (expt (vector-z v) 2)))))
```

2 Prozeduren auf Listen

1. Schreibe eine Prozedur **sorted?**, die überprüft, ob eine Liste sortiert ist. Die Prozedur akzeptiert eine beliebige Liste und einen Vergleichsoperator für eine lexikalische Ordnung.

   (: sorted? ((list-of %a) (%a %a → boolean) → boolean))

   Lösung:

```
(: sorted? ((list-of %a) (%a %a -> boolean) -> boolean))
(define sorted?
  (lambda (xs comp)
    (cond
      ((empty? xs) #t)
      ((empty? (rest xs)) #t)
      (else
       (and (comp (first xs) (first (rest xs)))
            (sorted? (rest xs) comp))))))
```

2. Schreibe eine Prozedur **sumEven**, die zwei Listen von Zahlen akzeptiert und die geraden Zahlen addiert.

   (: sumEven ((list-of number) (list-of number) → number))

   Lösung:

```
(: sumEven ((list-of number) (list-of number) -> number))
(check-expect (sumEven (list 1 2 3) (list 4 5 6)) 12)
(define sumEven
  (lambda (xs ys)
    (letrec
        ((flatten
          (lambda (xs ys)
            (append xs ys))))
         (fold 0 +
```

```
(filter (lambda (x)
          (= (modulo x 2) 0))
        (flatten xs ys))))))
```

3 Streams

1. Schreibe eine Prozedur **stream-merge**, die zwei Streams akzeptiert und den alternierenden Stream der beiden Streams bildet.

   (: stream-merge ((stream-of %a) (stream-of %a) → (stream-of %a)))

   Lösung:

```
(: stream-merge ((stream-of %a) (stream-of %a) -> (stream-of %a)))
(define stream-merge
  (lambda (s1 s2)
    (make-cons (head s1)
               (lambda () (stream-merge s2 (force (tail s1)))))))
```

4 Higher Order Procedures - H.O.P

1. Programmieren Sie die Funktionen **filter** und **map** nur durch die Verwendung von fold.

   Lösung:

```
(define filter
  (lambda (p xs)
    (fold empty (lambda (a b) (if (p a) (make-pair a b) b)) xs)))

(define map
  (lambda (p xs)
    (fold empty (lambda (a b) (make-pair (p a) b)) xs)))
```

5 Ausdrücke vereinfachen

Vereinfache folgende Scheme Ausdrücke.

(a)
```
(define compare
  (lambda (t)
        (cond
          ((> t 0) #t)
          ((< t 0) #f)
          ((= t 0) #t))))
```

Lösung:

```
(define compare
  (lambda (t)
    (>= t 0)))
```

(b)
```
(define f
  (lambda (a b c)
    (not (and a
          (or a c
          (or a b))))))
```

Lösung:

```
(define f
  (lambda (a b c)
    (not a)))
```