

; Repräsentiere Mengen (set-of <t>) von Elementen der Signatur <t>
; durch ihre charakteristische Funktion mit Signatur (<t> -> boolean):

```
(define set-of
  (lambda (t)
    (signature (t -> boolean))))
; S ∪ T
; x ∈ S ∪ T ⇔ x ∈ S ∨ x ∈ T
(: set-union ((set-of %a) (set-of %a) -> (set-of %a)))
(define set-union
  (lambda (S T)
    (lambda (x) (or (S x) (T x)))))
; S ∩ T
; x ∈ S ∩ T ⇔ x ∈ S ∧ x ∈ T
(: set-intersect ((set-of %a) (set-of %a) -> (set-of %a)))
(define set-intersect
  (lambda (S T)
    (lambda (x) (and (S x) (T x)))))
; S \ T
; x ∈ S \ T ⇔ x ∈ S ∧ x ∉ T
(: set-difference ((set-of %a) (set-of %a) -> (set-of %a)))
(define set-difference
  (lambda (S T)
    (lambda (x) (and (S x) (not (T x)))))
```

; Implementiert nun die duale Funktion unfoldr, die aus
; einem einzelnen gegebenen Startwert
; (seed value) x eine ganze Liste xs erzeugen kann
(: id (%a -> %a))
(define id
 (lambda (x) x))
; unfoldr erzeugt aus einem einzelnen gegebenen Startwert
; (seed value) x eine ganze Liste xs
(: unfoldr ((%b -> boolean) (%b -> %a) (%b -> %b) %b -> (list-of %a)))
(define unfoldr
 (lambda (p f g x)
 (if (p x)
 empty
 (cons (f x) (unfoldr p f g (g x))))))

; Baut die Funktion (indexed-foldr z c xs), in der die Funktion c nicht nur
; das aktuell zu verarbeitende Listenelement x und das Ergebnis der
; Faltung auf der Restliste als Parameter bekommt, sondern auch die
; Position i von x in der Liste (i = 0,1,2,...).
(: indexed-foldr (%b (natural %a %b -> %b) (list-of %a) -> %b))
(define indexed-foldr
 (lambda (z c xs)
 (letrec ((indexed-foldr-worker
 (lambda (i z c xs)
 (match xs
 (empty) z
 ((cons y ys) (c i y (indexed-foldr-worker (+ i 1) z c ys))))))
 (indexed-foldr-worker 0 z c xs)))
(: indexed-foldr (%b (natural %a %b -> %b) (list-of %a) -> %b))
(define indexed-foldr*
 (lambda (z c xs)
 ((fold (lambda (i) z)
 (lambda (x fz) (lambda (i)
 (c i x (fz (+ i 1))))
 xs) 0)))

; letztes Element einer Liste aufrufen
(: last ((list-of %a) -> %a))
(define last
 (lambda (lst)
 (match lst
 (empty) empty
 ((cons x empty) x)
 ((cons x xs) (last (rest lst)))))
; letztes Element einer Liste entfernen
(: rem-last ((list-of %a) -> (list-of %a)))
(define rem-last
 (lambda (lst)
 (match lst
 (empty) empty
 ((cons x empty) empty)
 ((cons x xs) (cons (first lst) (rem-last (rest lst))))))
; gegebenes Element an der richtigen Position in der Liste einfügen
(: insert-sorted (real (list-of real) -> (list-of real)))
(define insert-sorted
 (lambda (x lst)
 (match lst
 (empty) (list x)
 ((cons x xs) (if (< x xs)
 (cons x lst)
 (cons xs (insert-sorted x xs))))))

; Liste in aufsteigender Reihenfolge sortieren
(: sort-list ((list-of real) -> (list-of real))) (: reversel ((list-of %a) -> (list-of %a)))
(define sort-list
 (lambda (lst)
 (match lst
 (empty) empty
 ((cons x xs) (insert-sorted x (sort-list xs))))))
(define reversel
 (lambda (lst)
 (match lst
 (empty) empty
 ((cons x xs) (snoc (reverse xs) x))))

; Summenfunktion einer Liste
(: sum ((list-of number) -> number))
(define sum
 (lambda (xs)
 (match xs
 (empty) 0
 ((cons y ys)
 (+ y (sum ys)))))
; Funktion in der jedes Listenelement mit einer gegebenen
; Zahl x1 multipliziert wird
(: mult (number (list-of number) -> (list-of number)))
(define mult
 (lambda (x1 xs)
 (match xs
 (empty) empty
 ((cons y ys)
 (cons (* x1 y)
 (mult x1 ys)))))
; Funktion die, die ersten n Elemente von einer Listeignoriert
; und eine Liste xs ohne n Elemente ausgibt
(: drop (natural (list-of %a) -> (list-of %a)))
(define drop
 (lambda (n xs)
 (match xs
 (empty) empty
 ((cons y ys)
 (cond ((= n 0) (cons y ys))
 ((= n 1) ys)
 (else (drop (- n 1) ys))))))
; Funktion, welche Listenpositionen um n überspringt
(: every-nth (natural (list-of %a) -> (list-of %a)))
(define every-nth
 (lambda (n xs)
 (match xs
 (empty) empty
 ((cons y empty) (cons y empty))
 ((cons y ys)
 (cons y (every-nth n (drop (- n 1) ys))))))

; Greift auf ntes Listenelement zu
(: nth ((combined natural (predicate positive?) (list of%a) -> %a))
(define nth
 (lambda (n xs)
 ((compose first (repeat (- n 1) rest)) xs)))
; Funktion zur Testung von genau 13 Elementen
(: exact-13-pred ((list-of %a) -> boolean))
(define exact-13-pred
 (lambda (xs)
 (= (list-length xs) 13)))
; Funktion zur Testung ob Elemente von Signatur t
(: element-of-t-pred (real -> boolean))
(define element-of-t-pred
 (lambda (x)
 (= x t)))
; Zahl Element von gegebener List von Zahlen
(: elem? (number (list-of number) -> boolean))
(define elem?
 (lambda (zahl lst)
 (match lst
 (empty) #f
 ((cons x xs) (if (= zahl x)
 #t
 (elem? zahl xs)))))

; Größtes Element einer Liste
(: list-max ((list-of natural) -> natural))
(define list-max
 (lambda (lst)
 (match lst
 (empty) (violation "Liste ist leer")
 ((cons x empty) x)
 ((cons x xs) (max x (list-max xs)))))
; Element beliebiger Signatur als eigene Liste an
; gegebene Liste anhängen
(: snoc ((list-of %a) %a -> (list-of %a)))
(define snoc
 (lambda (liste element)
 (match liste
 (empty) (cons element empty)
 ((cons a b) (cons a (snoc b element)))))
; Elemente einer Liste in der Reihenfolge umdrehen
(: reverse ((list-of %a) -> (list-of %a)))
(define reverse
 (lambda (lst)
 (match lst
 (empty) empty
 (empty) (violation "Liste ist leer")
 ((cons x empty) empty)
 ((cons x (cons xs xss)) (cons x (init (cons xs xss)))))

; Sind alle Werte in der Liste gleich?
(: all-equal? ((list-of number) -> boolean))
(define all-equal?
 (lambda (lst)
 (match lst
 (empty) #t
 ((cons x empty) #t)
 ((cons x xs) (if (= x (first xs))
 (all-equal? xs) #f)))))
; Letztes Element einer Liste wird weggelassen
(: init ((list-of %a) -> (list-of %a)))
(define init
 (lambda (lst)
 (match lst
 (empty) (violation "Liste ist leer")
 ((cons x empty) empty)
 ((cons x (cons xs xss)) (cons x (init (cons xs xss)))))

; Überprüft ob eine Zahl in einer Liste von Zahlen enthalten ist
(: elem? (number (list-of number) -> boolean))
(define elem?
 (lambda (element xs)
 (match xs
 (empty) #f
 ((cons y ys)
 (if (= y element) #t (elem? element ys)))))
; Reverse mithilfe von foldl
(: my-reverse ((list-of %a) -> (list-of %a)))
(define my-reverse
 (lambda (xs)
 (foldl empty (lambda (y ys) (cons ys y)) xs)))
; Berücksichtigung der Schaltjahre
(define leap-year?
 (lambda (calendar-date)
 (cond ((= 0 (modulo (calendar-date-year calendar-date) 400)) #t)
 ((= 0 (modulo (calendar-date-year calendar-date) 100)) #f)
 ((= 0 (modulo (calendar-date-year calendar-date) 4)) #t)
 (else #f))))
; filter abstrahiert von einem konkreten Filterprädikat und akzeptiert Funktion
p? als Parameter
(: filter ((%a -> boolean) (list-of %a)))
(define filter
 (lambda (p? xs)
 (match xs
 (empty) empty
 ((cons y ys) (if (p? y)
 (cons y (filter p? ys))
 (filter p? ys)))))
; Überprüfe, welche Zahlen gerade sind und gib diese als Liste zurück
(: even-nums ((list-of number) -> (list-of number)))
(define even-nums
 (lambda (xs)
 (filter even? xs)))
; Überprüft ob Zahlen gerade(#t) oder ungerade(#f) sind, gibt Liste mit booleans
zurück
(: num-even? ((list-of number) -> (list-of boolean)))
(define num-even?
 (lambda (xs)
 (map even? xs)))
; Addiere Liste und gib Ergebnis zurück, nutze foldr
(: simple-list-add ((list-of number) -> number))
(check-property (for-all ((xs (list-of number))) (= (list-add xs) (simple-list-add xs))))
(define simple-list-add
 (lambda (xs)
 (foldr 0 (+ xs))))
; Listenlänge
(: length-list ((list-of %a) -> number))
(define length-list
 (lambda (xs)
 (foldr 0 (lambda (a n) (+ 1 n)) xs)))
; Füge am Ende einer Liste Element ein
(: snoc ((list-of %a) %a -> (list-of %a)))
(define snoc
 (lambda (list element)
 (match list
 (empty) (cons element empty)
 ((cons x xs) (cons x (snoc xs element)))))

; Fakultät
; Wrapper
(: fac (natural -> natural))
(define fac (lambda (n)
 (fac-worker n 1))) :Worker
; Worker
(: fac-worker (natural natural -> natural))
(define fac-worker
 (lambda (n acc)
 (cond ((= n 0) acc)
 ((> n 0) (fac-worker (- n 1) (* n acc)))))
; n-fache Komposition von f mit sich selbst; Exponentiation f^n
(: repeat (natural (%a -> %a) -> (%a -> %a)))
(define repeat (lambda (n f) (cond ((= n 0) (lambda (x) x))
 ((> n 0) (compose (repeat (- n 1) f) f)))))
; Sortierung xs bzgl. Ordnungsprädikat ⊗
(: qsort ((%a %a -> boolean) (list-of %a) -> (list-of %a)))
(define qsort
 (lambda (⊗ xs)
 (let (lgt (curry ⊗)))
 (match xs
 (empty) empty
 ((cons y ys)
 (append
 (qsort ⊗ (filter (lgt y) ys))
 (list y)
 (qsort ⊗ (filter (compose not (lgt y)) ys)))))))
; Differenzquotient von f mit Differenz h
(: diff-quot (real (real -> real) -> (real -> real)))
(define diff-quot
 (lambda (h f)
 (lambda (x)
 (/ (- (f (+ x h)) (f x)) h)))
; Ableitung
(: derive ((real -> real) -> (real -> real)))
(define derive
 ((curry diff-quot) 0.00001))
; Ist Liste xs aufsteigend sortiert?
(: sorted? ((list-of real) -> boolean))
(define sorted?
 (lambda (xs)
 (match xs (empty) #t) ((cons _ empty) #t) ((cons x (cons y ys)) (and (<= x y)
sorted?) (cons y ys))))))

