12 Anwendungen von Funktionen höherer Ordnung

12.1 Erinnerung: Das Suchproblem

Grundmenge M mit Äquivalenzrelation $= \subseteq M \times M$

Gegeben: Suchmenge $S \subseteq M$, $x \in M$

Gewünschte Operationen

- Suche eines Elements: $x \in S$?
- Vergrößern der Suchmenge $S \cup \{y\}$
- Verkleinern der Suchmenge $S \setminus \{y\}$

12.1.1 Gewünschte Operationen

Spezialisiert für M Menge der ganzen Zahlen

```
(: make-empty-set ( -> set-of-integer))
(: set-member? (set-of-integer integer -> boolean))
(: set-insert (set-of-integer integer -> set-of-integer))
(: set-remove (set-of-integer integer -> set-of-integer))
```

Verschiedene Implementierungen (*Repräsentationen*) einer `set-of-integer` möglich:

- Liste der Elemente der Menge (Gleichheit)
- Liste der Elemente ohne Duplikate (Gleichheit)
- aufsteigend sortierte Liste der Elemente ohne Duplikate (totale Ordnung)
- binärer Suchbaum (totale Ordnung)
- charakteristische Funktion (Gleichheit)

12.1.2 Prädikate auf Menge M

Eine n-*stellige Relation* R auf M ist eine Teilmenge von M^n

$$R \subseteq \underbrace{M \times \ldots \times M}_{n}$$

Binäre Relation $\Leftrightarrow$ 2-stellige Relation

Ein n-*stelliges Prädikat* P auf M ist eine totale Funktion von M^n in $\{0,1\}$.

$$P : \underbrace{M \times \ldots \times M}_{n} \to \{0,1\}$$

Fakt: Jede n-stellige Relation bestimmt umkehrbar eindeutig ein n-stelliges Prädikat.

Beweis: Für Prädikat P def. $\mathcal{R}(P)(x_1, \ldots, x_n) = \begin{cases} 1 & (x_1, \ldots, x_n) \in R \\ 0 & \text{sonst} \end{cases}$

und für Relation R def. $\mathcal{P}(R) = \{(x_1, \ldots, x_n) \in M^n \mid P(x_1, \ldots, x_n) = 1\}$

Es gilt $(\forall P)\ P = \mathcal{P}(\mathcal{R}(P))$ und $(\forall R)\ R = \mathcal{R}(\mathcal{P}(R))$.

12.1.3 Charakteristische Funktion

Sei $S \subseteq M$.

Die **charakteristische Funktion** $\chi_S : M \to \{0,1\}$ von S ist definiert durch

$$\chi_S(x) = \begin{cases} 1 & x \in S \\ 0 & x \notin S \end{cases}$$

Es gilt:

- $S \subseteq M$ ist einstellige Relation auf M.
- χ_S ist einstelliges Prädikat auf M.
- Jede Teilmenge $S \subseteq M$ bestimmt eindeutig ihre charakteristische Funktion χ_S.
- Jede Funktion $\chi : M \to \{0,1\}$ bestimmt eindeutig eine Teilmenge $S = \{x \in M \mid \chi(x) = 1\}$.

Idee: repräsentiere eine Menge im Programm durch eine Funktion mit Ergebnistyp `boolean` (anstelle von $\{0,1\}$)! D.h. $f(x) =$ `#t`, wenn $x \in S$.

12.1.4 Implementierung von Mengen durch Funktionen

```
(define set-of-integer
  (contract (integer -> boolean)))

; leere Menge
(: make-empty-set ( -> set-of-integer))
(define make-empty-set
  (lambda ()
    (lambda (i)
      #f)))

; Elementtest
(: set-member? (set-of-integer integer -> boolean))
(define set-member?
  (lambda (set i)
    (set i)))
```

```
; Element einfügen
(: set-insert (set-of-integer integer -> set-of-integer))
(define set-insert
  (lambda (set i)
    (lambda (j)
      (or (= i j) (set j)))))

; Element löschen
(: set-remove (set-of-integer integer -> set-of-integer))
(define set-remove
  (lambda (set i)
    (lambda (j)
      (if (= i j)
          #f
          (set j)))))
```

12.1.5 Eigenschaften

- Effizienz vergleichbar mit Implementierung durch Listen
  linear in der Anzahl der Elemente in der Listenrepräsentation
- (Übung: erzeuge eine Implementierung mit Listen, die exakt das gleiche Laufzeitverhalten hat)
- Nur Gleichheit auf M erforderlich
- Unendliche Mengen repräsentierbar
- Mengenoperationen (Komplement, Vereinigung, Durchschnitt) in konstanter Zeit durchführbar
- Nachteil: Mengenelemente können nicht aufgezählt werden, falls M unendlich ist
- Nachteil: Keine Implementierung für Mengengleichheit, Teilmengenrelation

12.2 Numerische Differentiation

```
; berechne die Ableitung einer Funktion
(: derivative (real -> ((real -> real) -> (real -> real))))

; Tests
(define x->2x ((derivative .001) (lambda (x) (* x x))))
(check-property
 (for-all ((x real))
   (expect-within (x->2x x) (* 2 x) .01)))
(define x->3xx ((derivative .00001) (lambda (x) (* x x x))))
(check-property
 (for-all ((x real))
   (expect-within (x->3xx x) (* 3 x x) .01)))
```

12.2.1 Definition

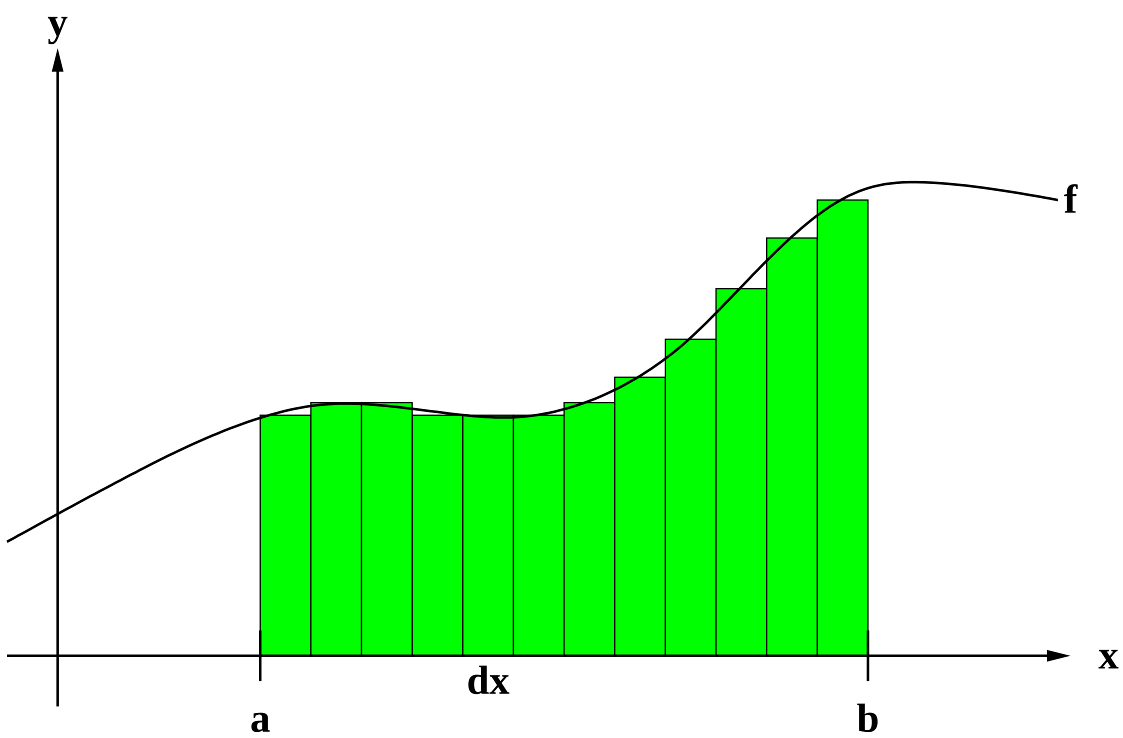
```
(define derivative
  (lambda (h)
    (lambda (f)
      (lambda (x)
        (/ (- (f (+ x h))
              (f x))
           h)))))
```

12.3 Numerische Integration

```
; berechne das Integral einer Funktion zwischen zwei Grenzen
(: integral ((real -> real) real real natural -> real))

; Tests
(check-expect-within
 (integral (lambda (x) (+ x 1)) 0 1 1000)
 1.5 .005)
(check-expect-within
 (integral (lambda (x) (* x x)) 0 1 1000)
 (/ 1 3) .005)
```

Ansatz: summiere die Flächen der Rechtecke (Keplers Regel)



```
(define integral
  (lambda (f a b nr-samples)
    (let ((dx (/ (- b a) nr-samples)))
      (letrec ((sum-samples
                 (lambda (n x0 partial-sum)
                   (if (zero? n)
                       partial-sum
                       (sum-samples (- n 1)
                                    (+ x0 dx)
                                    (+ (f x0) partial-sum))))))
        (* dx (sum-samples nr-samples a 0))))))
```

12.4 Funktionen und Datenstrukturen

- Listen, die mit `make-pair` und `empty` aufgebaut sind, haben endlich viele Elemente
- Mit Hilfe von Funktionen können Listen (*Streams*) mit unendlich vielen Elementen konstruiert werden.
- Natürlich kann ein Programm immer nur endliche viele Elemente davon betrachten.

12.4.1 Der Datentyp Stream

Ein Stream ist eine potentiell unendliche Liste, d.h., der Inhalt des Restes des Stroms wird erst bei Zugriff ausgewertet. Ein Stream besitzt folgende Prädikate und Selektoren:

```
(: stream-empty? (stream -> boolean))
(: stream-head (stream -> %X))
(: stream-tail (stream -> stream))

; Implementierung der Elemente eines Stream
(define-record-procedures stream-cons
  make-stream-cons stream-cons?
  (stream-cons-real-head stream-cons-real-tail))
; Der Rest eines Stroms ist speziell:
(: make-stream-cons (%X ( -> stream) -> stream-cons))

(define stream
  (contract (mixed (one-of empty) stream-cons))
```

Operationen:

```
(define stream-empty?
  empty?)
(define stream-head
  (lambda (s)
    (stream-cons-real-head s)))
(define stream-tail
  (lambda (s)
    ((stream-cons-real-tail s))))
```

12.4.2 Konstruktion von Streams

Signatur: `(: stream-from (natural -> stream))`

Erklärung: `(stream-from n)` liefert den Stream `n`, `(+ n 1)`, `(+ n 2)`, ...

Definition:

```
(define stream-from
  (lambda (n)
    (make-stream-cons
     n
     (lambda () (stream-from (+ n 1))))))
```

12.4.3 Filtern von Streams

Erklärung: `(stream-filter p s)` liefert einen Stream, in dem nur die Elemente von `s` sind, für die `(p s)` gilt.

Definition:

```
(: stream-filter ((%X -> boolean) stream -> stream))
(define stream-filter
  (lambda (p s)
              (letrec ((loop
                (lambda (s)
                 (cond
                  ((stream-empty? s)
                   empty)
                  ((stream-cons? s)
                   (let* ((x  (stream-head s))
                          (ys (lambda () (loop (stream-tail s)))))
                     (if (p x)
                         (make-stream-cons x ys)
                         (ys))))))))
  (loop s))))
```

12.4.4 Das Sieb des Eratosthenes

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	3		5		7		9		11		13		15		17		19		21		23		25
			5		7				11		13				17		19				23		25
					7				11		13				17		19				23		
									11		13				17		19				23		
											13				17		19				23		

1. Erste Zahl (blau unterstrichen) ist Primzahl
2. Alle Vielfachen entfernen; weiter bei 1

Erklärung: `(sieve s)` implementiert das Sieb des Eratosthenes.

Definition:

```
(: sieve (stream -> stream))
(define sieve
  (lambda (s)
    (let ((p (stream-head s)))
      (make-stream-cons
       p
       (lambda ()
         (sieve
          (stream-filter (lambda (x) (not (zero? (remainder x p))))
                         (stream-tail s))))))))
```

12.4.5 Ein Stream von Primzahlen

Erklärung: `primes` ist der Stream der Primzahlen.

Definition:

```
(: primes stream)
(define primes
  (sieve (stream-from 2)))
```

Liefert in der REPL:

```
> (stream-display primes)
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
101, 103, 107, 109, 113, 127, 131, 137, 139,
149, 151, 157, 163, 167, 173, 179, 181, 191,
193, 197, 199, 211, 223, 227, 229, 233, 239,
```

12.4.6 Ausdrucken eines Streams

```
(: stream-display (stream -> boolean))
(define stream-display
  (lambda (s)
    (cond
     ((stream-empty? s)
      #f)
     ((stream-cons? s)
      (let* ((n (stream-head s))
             (xx (write-string (number->string n)))
             (xx (write-string ", "))
             (ns (stream-tail s))
        (stream-display (ns))))))))
```