



INFORMATIK I

Tutorium 10.01.2017

BESPRECHUNG

Blatt 9



WIEDERHOLUNG

.....
Vorlesung & Für Blatt 10



WIEDERHOLUNG: HIGHER ORDER PROCEDURES

- Prozeduren höherer Ordnung (Higher-order-procedures)
- 1. Akzeptieren Prozeduren als Parameter oder/und
- 2. liefern eine Prozedur als Ergebnis
- Führen zu: kompakteren Programmen, verbesserter Les- und Wartbarkeit

WIEDERHOLUNG: FILTER

- H.O.P filter
- Akzeptiert ein Prädikat $p?$ und Liste xs . Liefert alle Elemente von xs die Prädikat $p?$ erfüllen

```
(: filter ((%a -> boolean) (list-of %a) -> (list-of %a)))  
(define filter  
  (lambda (p? xs)  
    (cond ((empty? xs) empty)  
          ((pair? xs) (if (p? (first xs))  
                          (make-pair (first xs)  
                                      (filter p? (rest xs)))  
                          (filter p? (rest xs)))))))
```

WIEDERHOLUNG: FILTER

- Anwendungsbeispiel: Gebe nur Elemente der Liste zurück, die ≥ 5 sind

```
(filter  
  (lambda (x)  
    ( $\geq$  x 5))  
  (list 1 2 3 4 5 6))
```

WIEDERHOLUNG: MAP

- H.O.P: map
- Akzeptiert Prozedur f und Liste xs und wendet f auf alle Elemente von xs an

```
; Wende f auf alle Elemente von xs an
(: map ((%a -> %b) (list %a) -> (list %b)))
(define map
  (lambda (f xs)
    (cond ((empty? xs) empty)
          ((pair? xs) (make-pair (f (first xs))
                                  (map f (rest xs)))))))
```

WIEDERHOLUNG: MAP

- Anwendungsbeispiel: Multipliziere alle Elemente der Liste xs mit 3

```
(map  
  (lambda (x)  
    (* x 3))  
  (list 1 2 3 4 5))
```


WIEDERHOLUNG: LIST FOLDING

- Idee: Ersetze die Listenkonstruktoren make-pair und empty systematisch
- Ersetze empty durch z
- Ersetze make-pair durch c

```
; Falte Liste xs bzgl. z und c
(: foldr (%b (%a %b -> %b) (list-of %a) -> %b))
(define foldr
  (lambda (z c xs)
    (cond ((empty? xs) z)
          ((pair? xs) (c (first xs) (foldr z c (rest xs)))))))
```

WIEDERHOLUNG: LIST FOLDING

- Anwendungsbeispiel: Summe einer Liste
- Nutze foldr um empty mit 0 und make-pair mit + zu ersetzen

```
(define sum  
  (lambda (xs)  
    (foldr 0 + xs)))
```

WIEDERHOLUNG: CURRYING

- Idee: Wandle eine Funktion mit mehreren Argumenten in eine Funktion mit einem Argument um
- Sinn: Anwendung einer Prozedur auf ihr erstes Argument liefert eine Prozedur der restlichen Argumente

```
; Zweistellige Funktion f in ihre curried Variante überführen  
(: curry ((%a %b -> %c) -> (%a -> (%b -> %c))))  
(define curry  
  (lambda (f)  
    (lambda (x)  
      (lambda (y)  
        (f x y))))))
```

WIEDERHOLUNG: CURRYING

- Beispielanwendung: In Kombination mit Filter
- Gib alle Elemente zurück die größer 2 sind
- Normal:

```
(filter  
  (lambda (x) (> x 2))  
  (list 1 2 3 4 5))
```

- Mit Currying:

```
(filter  
  ((curry <) 2)  
  (list 1 2 3 4 5))
```

WIEDERHOLUNG: STREAMS

- Problem: Unendliche Mengen von Elementen nicht darstellbar (z.B. Liste aller natürlichen Zahlen)
- Lösung: Stream
- Idee: Erst Ausführung des tails (FORCE) erzeugt nächstes Stream-Element (Lazy list)
- Wie? —> Durch verzögerte Auswertungen, liefere stattdessen das „Versprechen“ bei Bedarf später auszuwerten (Promise)
- Promises sind vor allem in asynchronen Sprachen (z.B. JavaScript) weit verbreitet

WIEDERHOLUNG: STREAMS

➤ Was wird benötigt?

; Promise, ein Wert des Vertrags t zu liefern (0-stellig Prozedur)

```
(define promise  
  (lambda (t)  
    (signature (-> t))))
```

; Erzwungene Auswertung

```
(: force ((promise %a) -> %a))  
(define force  
  (lambda (p)  
    (p)))
```

WIEDERHOLUNG: STREAMS

➤ Beispiel:

```
; Beispiel:  
; Promise (werde 41+1 berechnen, falls gefordert)  
(: will-evaluate-to-42 (promise natural))  
(define will-evaluate-to-42  
  (lambda ()  
    (+ 41 1)))  
  
; Im Stepper (keine Addition ausgeführt):  
will-evaluate-to-42  
  
(force will-evaluate-to-42)
```

WIEDERHOLUNG: STREAMS

- Noch mehr Definitionen zu Streams
- Ein Stream ist ein polymorphes Paar

```
; Polymorphe Paare (isomorph zu `pair')
(: make-cons (%a %b -> (cons-of %a %b)))
(: head ((cons-of %a %b) -> %a))
(: tail ((cons-of %a %b) -> %b))
(define-record-procedures-parametric cons cons-of
  make-cons
  cons?
  (head
   tail))
```

WIEDERHOLUNG: STREAMS

➤ Ein Stream besteht aus:

```
; Ein Stream besteht aus  
; - einem ersten Element (head)  
; - einem Promise, den Rest des Streams generieren zu können (tail)  
(define stream-of  
  (lambda (t)  
    (signature (cons-of t (promise (stream-of t))))))
```

WIEDERHOLUNG: STREAMS

➤ Beispiel: Liste von Zahlen ab n

; Beispiel:

; Stream mit Zahlen ab n erzeugen

(: from (number -> (stream-of number)))

(define from

 (lambda (n)

 (make-cons n (lambda () (from (+ n 1)))))

 └──────────┘
 Auswertung erst durch (force ...)!

ÜBUNGSAUFGABE

- Schreibe eine Prozedur, die die kleinste positive Zahl zurück gibt die durch alle Zahlen von 1 bis n ohne Rest teilbar sind
- (Hinweis: Durch Streams, map / curry / fold lässt sich der Code deutlich reduzieren)
- Nehme an, dass Streams bereits implementiert sind (stream-take, stream-of, promise, force etc)