Informatik I

5. Listen und Rekursion

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

4. November 2010

Informatik I

4. November 2010 — 5. Listen und Rekursion

5.1 Listen

5.2 Rekursion auf Listen

5.3 Polymorphismus

5.4 Eingebaute Listen

5.1 Listen

- Einführung
- Sortendefinition
- Beispiele

Listen

- Eine Sorte bezeichnet man auch als Datenstruktur, insbesondere, wenn sie nicht ganz trivial ist und wenn man eher auf die „konkreten“ Aspekte abzielt. Die verschiedenen Kekssorten oder die Sorte für kartesische Punkte sind Datenstrukturen.
- Wir betrachten jetzt unsere erste sehr wichtige und sehr allgemeine Datenstruktur: Listen.
- Eine Liste ist eine Aneinanderreihung von beliebig vielen Elementen. Beispiele in einer ad-hoc Syntax:
  - Liste der Jahreszeiten:
    `["Frühling", "Sommer", "Herbst", "Winter"]`
  - Liste der Primzahlen < 20:
    `[2,3,5,7,11,13,17,19]`.

Listen selbst gemacht!

- Da Listen so wichtig sind, sind sie natürlich in Scheme eingebaut!
- Es ist aber sehr lehrreich, sie noch einmal selbst zu bauen:
  - Besseres Verständnis von Listen;
  - Übung, wie man grundsätzlich Datenstrukturen baut;
  - Scheme-Listen leiden unter gewissen historischen Altlasten.
- Einstweilen betrachten wir Listen von Zahlen.

Die leere Liste

Was die Null für die Zahlen, das ist die leere Liste für die Listen; es ist also die Liste mit 0 Elementen.

Wir brauchen ein Symbol für die leere Liste. Saubere Lösung: Wir definieren einen Record. Um Kollisionen mit existierenden Namen auszuschließen, benutzen wir deutsche Namen:

```
(define-record-procedures leere-liste
  make-leere-liste leer?
  (...))
```

```
(define-record-procedures leere-liste
  make-leere-liste leer?
  ())
```

Das Sortenprädikat heißt `leer?` statt `leere-liste?` da kurz und üblich.

```
(: make-leere-liste (-> leere-liste))
(: leer? (%value -> boolean))
```

Die leere Liste

Verwendung

```
   make-leere-liste
=> #<procedure:make-leere-liste>
   (make-leere-liste)
=> #<record:leere-liste>
```

Um eine leere Liste zu konstruieren, müssen wir `make-leere-liste` anwenden, und zwar auf 0 Argumente! Daher die Klammern.

Wir führen einen Namen für die leere Liste ein:

```
(define leer (make-leere-liste))
```

Das Sortenprädikat `leer?`:

```
   (leer? leer)
=> #t
   (leer? "Banane")
=> #f
```

Allgemeine Listen

Wir werden gleich eine Sorte für nichtleere (Zahlen-)Listen definieren.
Diese Sorte wird `nll` heißen (weil `nichtleere-liste` zu lang ist).

Die Sorte für die leere Liste heißt also `leere-liste`, und für nichtleere Listen `nll`. Wie ist demnach die Sorte für allgemeine (leere oder nichtleere) (Zahlen-)Listen zu definieren?
Als gemischte Sorte!

```
(define zahlenliste
  (signature
   (mixed leere-liste nll)))
```

Nichtleere Listen: `nll`

Erinnerung: eine (Zahlen-)Liste ist eine Aneinanderreihung von beliebig vielen Zahlen. Man kann aber nicht beliebig viele Zahlen „auf einen Schlag“ aneinanderreihen. Deshalb sind nichtleere Listen folgendermaßen definiert:

- Verknüpfe eine Zahl mit der leeren Liste, um eine Liste der Länge 1 zu erhalten.
- Verknüpfe eine Zahl mit einer Liste der Länge 1, um eine Liste der Länge 2 zu erhalten.
- Verknüpfe eine Zahl mit einer Liste der Länge 2, um eine Liste der Länge 3 zu erhalten.
- ...

Allgemein setzt sich eine nichtleere Liste der Länge $n+1$ (wobei $n \geq 0$) also zusammen aus einer Zahl und einer Liste der Länge n.

Der Record `nll`

Wahl der Namen

```
(define-record-procedures nll
  kons nichtleer?
  (kopf rumpf))
```

Beachte die Abweichungen von den Namenskonventionen, um kurze bzw. übliche Namen zu haben:

- `kons` statt `make-nll`;
- `nichtleer?` statt `nll?`;
- `kopf` statt `nll-kopf`;
- `rumpf` statt `nll-rumpf`.

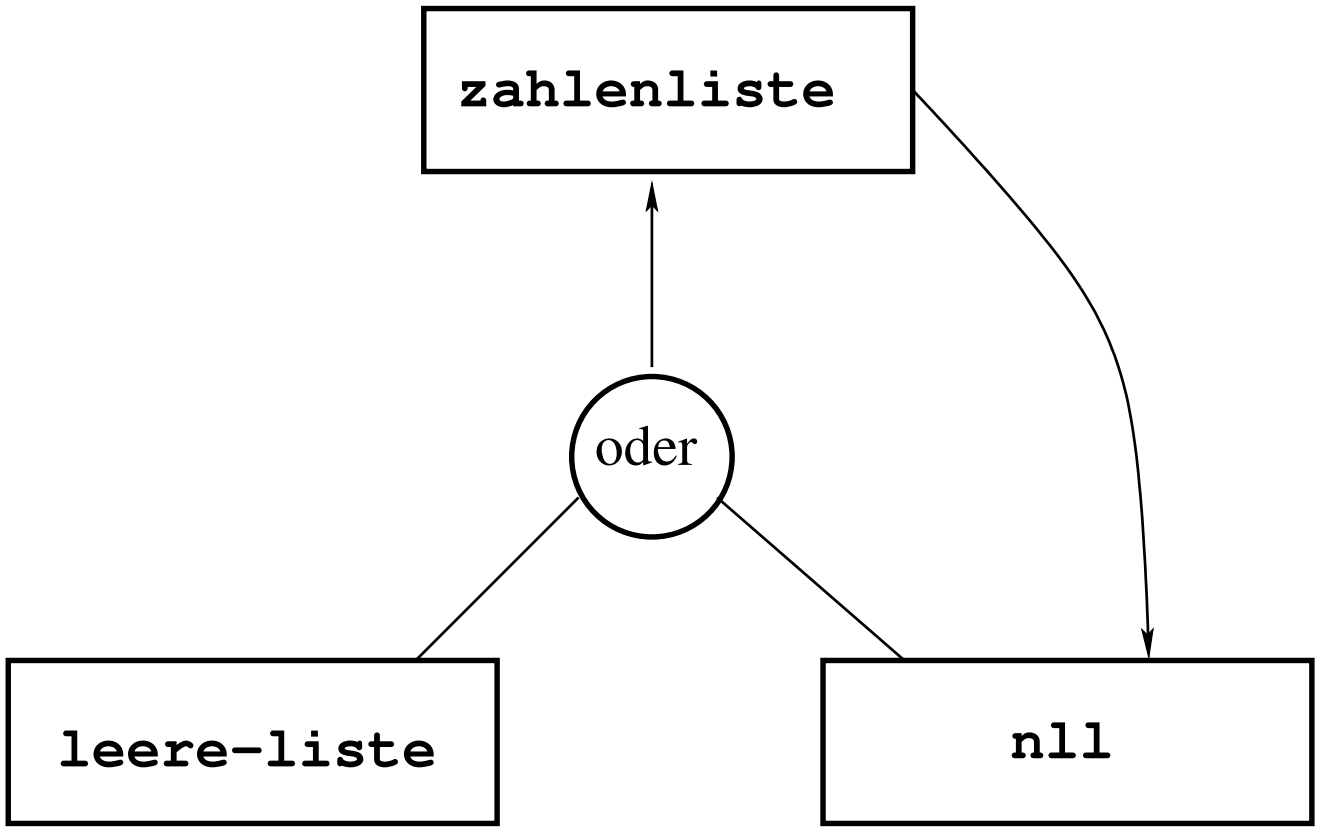
Der Record nll

Signaturen

```
(define-record-procedures nll
  kons nichtleer?
  (kopf rumpf))
(: kons (number zahlenliste -> nll))
(: nichtleer? (%value -> boolean))
(: kopf (nll -> number))
(: rumpf (nll -> zahlenliste))
```

Die Definition von nichtleeren Listen verwendet allgemeine Listen. Allgemeine Listen haben aber nichtleere Listen verwendet: `zahlenliste` ist eine rekursive Sorte definiert durch eine rekursive Definition.

Graphische Darstellung



Die Sortendefinitionen sagen aus:

- um eine Liste zu konstruieren, benötigen wir eine leere Liste oder eine nichtleere Liste;
- um eine nichtleere Liste zu konstruieren, benötigen wir eine Liste.

Beispiele für Listen

```
leer
(kons 2 leer)
(kons 3 (kons 2 leer))
(kons 5 (kons 3 (kons 2 leer)))
```

liefern die Ausgabe

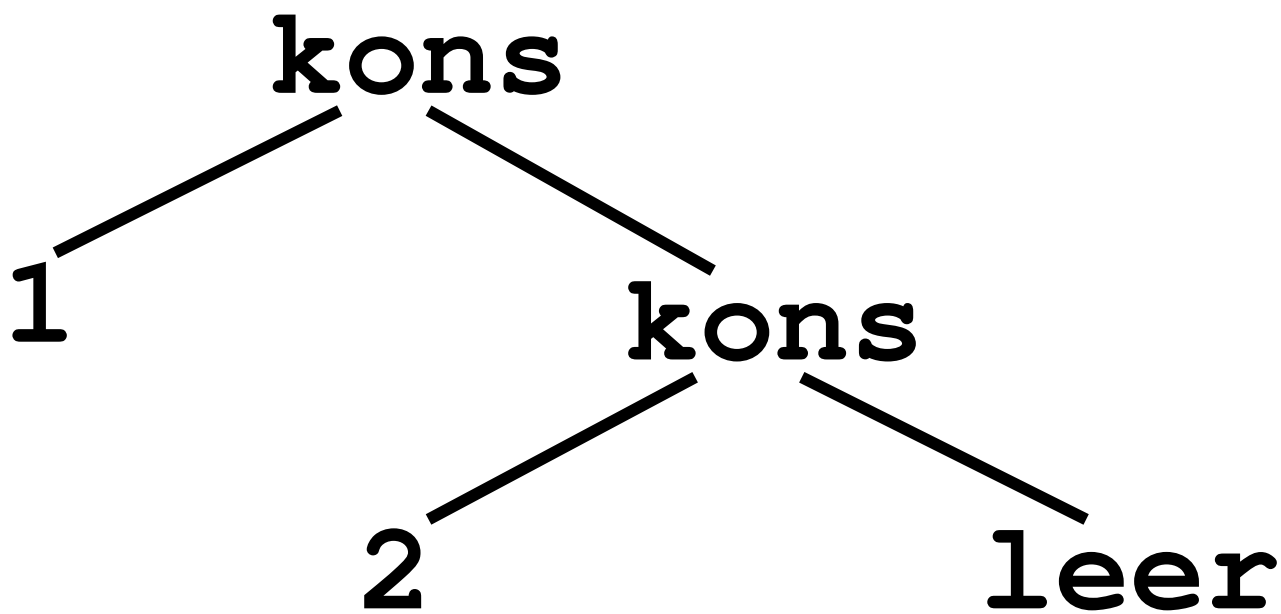
```
#<record:leere-liste>
#<record:nll 2 #<record:leere-liste>>
#<record:nll 3 #<record:nll 2 #<record:leere-liste>>>
#<record:nll 5
     #<record:nll 3
          #<record:nll 2 #<record:leere-liste>>>>
```

Visualisierung

Die Liste

```
(kons 1 (kons 2 leer))
```

kann wie folgt dargestellt werden:



Operationen auf Listen

Die Bedeutung der Operationen auf Listen ergibt sich direkt aus den Record-Definitionen, trotzdem nochmal zur Klarheit:

Sortenprädikate

`leer?` und `nichtleer?`, um eine leere Liste bzw. eine nichtleere Liste zu erkennen.

Konstruktoren

`make-leere-liste` liefert die leere Liste.
`kons` erweitert eine Liste um ein neues Kopfelement.

Selektoren

`kopf` liefert das erste Element.
`rumpf` liefert den Rest der Liste, d.h. die Liste ohne das erste Element.

Beispiele für die Grundoperationen auf Listen

```
   (define liste-1 leer)
   (define liste-2 (kons 1 liste-1))
   (define liste-3 (kons 2 liste-2))
   (define liste-4 (kons 4 liste-3))

   (kopf liste-2)
=> 1
   (rumpf liste-2)
=> #<record:leere-liste>
   (kopf liste-3)
=> 2
   (rumpf liste-3)
=> #<record:nll 1 #<record:leere-liste>>
   (rumpf liste-4)
=> #<record:nll 2 #<record:nll 1 #<record:leere-liste>>>
```

zahlenlisten.rkt

Die bisher definierten Listen finden Sie unter dem Namen `zahlenlisten.rkt` auf
http://www.informatik.uni-freiburg.de/~ki/teaching/ws1011/infoI/

5.2 Rekursion auf Listen

- Summe der Listenelemente
- Länge einer Liste
- Konstruktionsanleitung 7

Prozeduren, die Listen konsumieren

Summe der Listenelemente

```
; Elemente einer Liste addieren
(: list-sum (zahlenliste -> number))
(define list-sum
  (lambda (xs)
    (cond
     ((leer? xs) ...)
     ((nichtleer? xs) (... (kopf xs) ...
                           (rumpf xs) ...)))))
(check-expect (list-sum liste-1) 0)
(check-expect (list-sum liste-3) 3)
```

- Konstruktionsanleitung für Prozeduren
- Konstruktionsanleitung für gemischte Daten (Liste ist gemischte Sorte)
- Konstruktionsanleitung für zusammengesetzte Daten

Neuer Schritt: Komponente besitzt Listensorte

Aus `(: xs zahlenliste)`
und `(: rumpf (nll -> zahlenliste))` ergibt sich
`(: (rumpf xs) zahlenliste)`.

Standardansatz: verwende das Ergebnis von
`(list-sum (rumpf xs))`.
Mit anderen Worten: rufe `list-sum` rekursiv auf.

```
; Elemente einer Liste addieren
(: list-sum (zahlenliste -> number))
(define list-sum
  (lambda (xs)
    (cond
     ((leer? xs) ...)
     ((nichtleer? xs)  (... (kopf xs) ...
                       (list-sum (rumpf xs)) ...)))))
```

Ausfüllen der Schablone

```
; Elemente einer Liste addieren
(: list-sum (zahlenliste -> number))
(define list-sum
  (lambda (xs)
    (cond
     ((leer? xs) 0)
     ((nichtleer? xs) (+ (kopf xs)
                         (list-sum (rumpf xs)))))))
```

Länge einer Liste

Gerüst und Schablone genau wie bei `list-sum`:

```
; Anzahl der Elemente einer Liste bestimmen
(: list-length (zahlenliste -> number))
(define list-length
  (lambda (xs)
    (cond
     ((leer? xs)
      ...)
     ((nichtleer? xs)
      (... (kopf xs) ...
           (list-length (rumpf xs)) ...)))))
(check-expect (list-length leer) 0)
(check-expect (list-length liste-3) 2)
```

Länge einer Liste

Ausfüllen der Schablone

```
(define list-length
  (lambda (xs)
    (cond
     ((leer? xs)
      0)
     ((nichtleer? xs)
      (+ 1 (list-length (rumpf xs)))))))
```

Berechnung von `list-length` (nach Schritt 0)

```
   (list-length (kons 1 (kons 2 leer))))
=>1 (cond
      ((leer? (kons 1 (kons 2 leer))) ; #f
       0)
      ((nichtleer? (kons 1 (kons 2 leer)))  ; #t
       (+ 1 (list-length
              (rumpf (kons 1 (kons 2 leer)))))))
=>4 (+ 1 (list-length
           (rumpf (kons 1 (kons 2 leer)))))
=>1 (+ 1 (list-length (kons 2 leer)))
=>5 (+ 1 (+ 1 (list-length leer)))
=>3 (+ 1 (+ 1 0))
=>2 2
```

Bemerkungen

- Die Berechnung der Länge einer Liste mit zwei Elementen hat 21 Schritte!
- Jeden Schritt nachzuvollziehen, schaffen wir!
- Wenn eine Liste *l* die Länge *n* hat und *a* eine beliebige Zahl ist, dann ist die durch

  `(kons` *a l*`)`

  konstruierte Liste um genau ein Element länger als *l*, sie hat also die Länge $n + 1$. Dies zu verstehen, schaffen wir (vielleicht)!
- Damit, den gesamten Berechnungsprozess auf einmal zu überblicken, sind wir mental überfordert!

Konstruktionsanleitung 7 (Listen)

Eine Prozedur, die eine (Zahlen-)Liste konsumiert, hat folgende Schablone:

```
(: p ((zahlenliste) -> τ))
(define p
  (lambda (l)
    (cond
      ((leer? l) ...)
      ((nichtleer?  l)
       ... (kopf l)
       ... (p (rumpf l)) ...))))
```

- Fülle zuerst den `leer?`-Zweig aus.
- Fülle dann den `nichtleer?`-Zweig aus unter der Annahme, dass der rekursive Aufruf `(p (rumpf l))` das gewünschte Ergebnis für den Rest der Liste liefert.

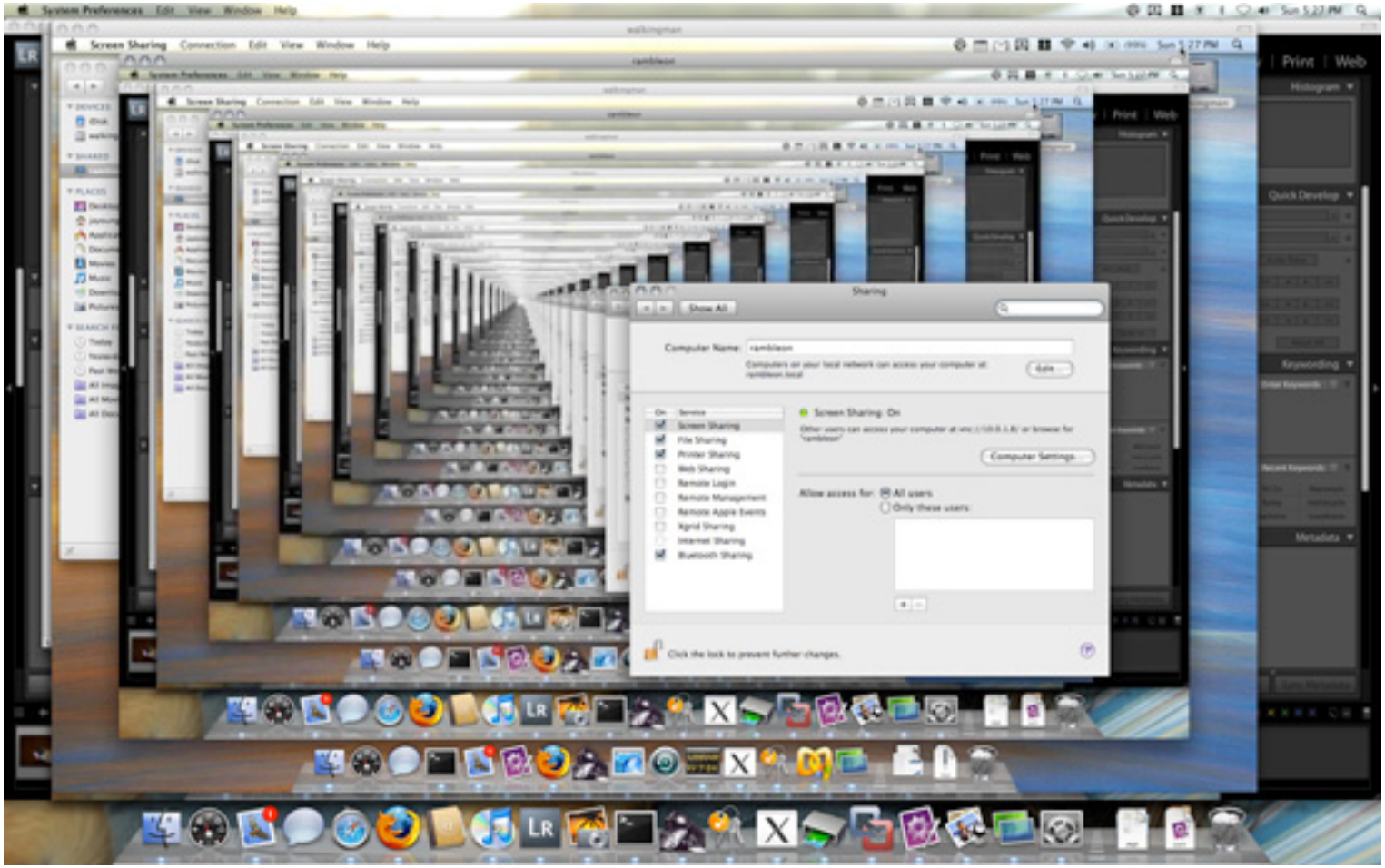
Mantra

Mantra #7 (Prozeduren über Listen)

Befolge für Prozeduren, die Listen konsumieren, zuerst die Konstruktionsanleitung und schreibe Signatur, Gerüst und Schablone auf, vor tieferem Nachdenken über die Aufgabenstellung!

Mantra #8 (Flaches Denken)

Denke niemals rekursiv über einen rekursiven Prozess nach! [KS07] (Versuche nicht, einen rekursiven Prozess in seiner Gesamtheit zu überblicken!)



Quelle http://rambleon.org/wp-content/uploads/2007/11/recursion.jpg

ANTI-Mantra

Um Rekursion zu verstehen, muss man zunächst Rekursion verstehen.

5.3 Polymorphismus

- Listen mit beliebiger Elementsorte
- Polymorphe Listen
- Sonstiger Polymorphismus

Definition von Listen mit beliebiger Elementsorte

Bisher haben wir Zahlenlisten betrachtet. Nun wollen wir Listen mit beliebiger Elementsorte definieren.
Für die leere Liste ändert sich nichts:

```
(define-record-procedures leere-liste
  make-leere-liste leer?
  ())
(: make-leere-liste (-> leere-liste))
(: leer? (%value -> boolean))

(define leer (make-leere-liste))
```

`nll` bei beliebiger Elementsorte

```
(define-record-procedures nll
  kons nichtleer?
  (kopf rumpf))
(: kons (%a liste -> liste))
(: nichtleer? (%value -> boolean))
(: kopf (nll -> %a))
(: rumpf (nll -> liste))
```

Was hat sich gegenüber Zahlenlisten geändert?
Statt `number` steht jetzt die Sortenvariable `%a`, statt `zahlenliste` steht jetzt `liste` (damit der Name passend ist).

Die gemischte Sorte `liste`

```
(define liste
  (signature
   (mixed leere-liste nll)))
```

Was hat sich gegenüber Zahlenlisten geändert?
Nur der Name: `liste` statt `zahlenliste`

Listenelemente beliebiger Sorte

Die Listenelemente können jetzt von beliebiger Sorte sein:

```
   (: l liste)
   (define l (kons 1 (kons "Ende" leer)))
   l
=> #<record:nll 1 #<record:nll "Ende"
                        #<record:leere-liste>>>
```

Zum Vergleich: Gemäß der früheren Definition von `zahlenliste` ist

```
   (: l zahlenliste)
   (define l (kons 1 (kons "Ende" leer)))
   l
=> #<record:nll 1 #<record:nll "Ende"
                        #<record:leere-liste>>>
```

eine Signaturverletzung, die u.U. von DrRacket gemeldet wird.

Eine Prozedur

Erinnern wir uns an `list-length` auf Zahlenlisten:

```
(: list-length (zahlenliste -> number))
(define list-length
  (lambda (xs)
    (cond
     ((leer? xs)
      0)
     ((nichtleer? xs)
      (+ 1 (list-length (rumpf xs)))))))
```

Was ändert sich, wenn die Sorte der Elemente beliebig ist?
Außer der Signatur nichts! Für `list-length` ist die Sorte der Listenelemente völlig gleichgültig.

Noch eine Prozedur

Erinnern wir uns an `list-sum` auf Zahlenlisten:

```
(: list-sum (zahlenliste -> number))
(define list-sum
  (lambda (xs)
    (cond
      ((leer? xs) 0)
      ((nichtleer? xs) (+ (kopf xs)
                          (list-sum (rumpf xs)))))))
```

Was ändert sich, wenn die Sorte der Elemente beliebig ist?
Die Prozedur funktioniert im Allgemeinen nicht mehr! Das erste Argument von + muss eine Zahl sein. Für Listen, die eine Nichtzahl enthalten, kommt es zu einem Laufzeitfehler.

anylisten.rkt

Die soeben definierten Listen finden Sie unter dem Namen `anylisten.rkt` auf
http://www.informatik.uni-freiburg.de/~ki/teaching/ws1011/infoI/

Polymorphismus

- Um die Länge einer Liste zu berechnen, können wir unabhängig von der Sorte der Listenelemente immer die selbe Prozedur verwenden. Das ist schön, und es ist eine Idee, die wir noch sehr viel weiterentwickeln werden.
- Dafür haben wir jegliche Information über die Sorte der Listenelemente gänzlich weggeworfen, obwohl sie vielleicht an anderer Stelle nützlich sein könnte. Schön wäre, wenn wir ausdrücken könnten:
  Liste von Zahlen, Liste von Strings, Liste von Booleans, Liste von Schokokeksen, Liste von Personen, Liste von kartesischen Punkten, Liste von Listen von Zahlen, Liste von Listen von Strings, . . . Liste von Listen von Listen von Zahlen, . . . Liste von Listen von Listen von Listen von Listen von Listen von Listen von Listen von Zahlen, . . .

Polymorphe Listen

- Bisher hatten wir die Sorten `number`, `boolean`, `string`, `real`, `natural`, `chocolate-cookie`, `cream-jelly-cookie`, `person`, ...`zahlenliste`, `liste`, d.h., eine Sorte wurde durch einen Namen benannt.
- Wir werden jetzt eine Syntax `(liste` σ`)` einführen, wobei σ wiederum eine Sorte ist. Somit erhalten wir Sorten `(liste number)`, `(liste string)`, `(liste boolean)`, `(liste chocolate-cookie)`, `(liste person)`, `(liste cartesian)`, ..., `(liste (liste number))`, ..., `(liste (liste (liste (liste number))))`, ...
- Wir nennen dies parametrisch polymorphe Listen.
  - `liste` nimmt eine Sorte als Parameter;
  - `liste` ist vielgestalt, altgriechisch πολύμορφος.

Definition von parametrisch polymorphen Listen

Für die leere Liste ändert sich nichts:

```
(define-record-procedures leere-liste
  make-leere-liste leer?
  ())
(: make-leere-liste (-> leere-liste))
(: leer? (%value -> boolean))

(define leer (make-leere-liste))
```

`nll` für parametrisch polymorphe Listen

Die Sorte für nichtleere Listen wird wie gehabt `nll` heißen. Aber wir verwenden nun eine besondere Form von `define-record-procedures`:

```
(define-record-procedures-parametric-2 nll nll-of
  kons nichtleer?
  (kopf rumpf))
```

Was tut dieser Code, über `define-record-procedures` hinaus?

nll für parametrisch polymorphe Listen II

```
(define-record-procedures-parametric-2 nll nll-of
  kons nichtleer?
  (kopf rumpf))
```

Der Name `nll-of` wird an einen Signaturkonstruktor gebunden: `(nll-of` σ τ`)` ist nun eine zusammengesetzte Sorte. Der Ausdruck `(kons` s t`)` konstruiert einen Record dieser Sorte, sofern s die Sorte σ und t die Sorte τ hat.
Anders gesagt: Konstruktor und Selektoren haben kraft `define-record-procedures-parametric-2` folgende Signaturen:

```
(: kons (%a %b -> (nll-of %a %b)))
(: kopf ((nll-of %a %b) -> %a))
(: rumpf ((nll-of %a %b) -> %b))
```

nll für parametrisch polymorphe Listen III

```
(define-record-procedures-parametric-2 nll nll-of
  kons nichtleer?
  (kopf rumpf))
```

```
(: kons (%a %b -> (nll-of %a %b)))
(: kopf ((nll-of %a %b) -> %a))
(: rumpf ((nll-of %a %b) -> %b))
```

(Unsinniges!) Beispiel: `(kons 4711 "Banane")` hat die Sorte `(nll-of number string)`.
Wir brauchen: `(nll-of %a (liste %a))` ...

Signaturen für `nll`

Wir erlauben uns, die Signaturen von Konstruktor und Selektoren zu spezialisieren: statt %b setzen wir `(liste %a)` ein:

```
(: kons (%a (liste %a) -> (nll-of %a (liste %a))))
(: kopf ((nll-of %a (liste %a)) -> %a))
(: rumpf ((nll-of %a (liste %a)) -> (liste %a)))
```

Die gemischte Sorte (liste σ)

Erinnern wir uns nochmal an zahlenliste:

```
(define zahlenliste
  (signature
   (mixed leere-liste nll)))
```

Für jede beliebige Sorte σ soll die Sorte (liste σ) definiert sein als gemischte Sorte bestehend aus leere-liste und (nll-of σ (liste σ)). Versuch:

```
(define (liste sort) ;falsche Syntax!

    (signature
     (mixed leere-liste
            (nll-of sort (liste sort)))))

(define liste
  (lambda (sort)
```

Erläuterungen zu (liste σ)

```
(define liste
  (lambda (sort)
    (signature
     (mixed leere-liste
            (nll-of sort (liste sort))))))
```

- Die Sorte (liste %a) ist parametrisch, daher muss sie durch eine Lambda-Abstraktion definiert werden:
- Mit anderen Worten: liste ist eine „Sortenprozedur“: sie nimmt eine Sorte als Argument und liefert eine Sorte als Ergebnis.
- Die Verwendung von liste im Rumpf braucht nicht zu schockieren: dies ist Rekursion, siehe z.B. list-length.

Erklärung von Listensignaturen

Die Liste `xs` ist von der Sorte `(liste` σ`)` genau dann, wenn

- `xs` ist von der Sorte `leere-liste` oder
- `xs` ist von der Sorte `(nll-of` σ `(liste` σ`))`, so dass
  - `(kopf xs)` ist von der Sorte σ und
  - `(rumpf xs)` ist von der Sorte `(liste` σ`)`.

Beispiele für korrekte Signaturen

```
   (: l1 (liste number))
   (define l1 (kons 1 (kons 4 leer)))
=> #<record:nll 1 #<record:nll 4 #<record:leere-liste>>>
   (: l2 (liste string))
   (define l2 (kons "A" (kons "B" leer)))
=> #<record:nll "A"
        #<record:nll "B" #<record:leere-liste>>>
```

Bemerkung: `define-record-procedures-parametric-2` erfordert Sprachlevel „Die Macht der Abstraktion mit Zuweisungen“.

Signaturverletzung

```
   (: l3 (liste number))
   (define l3 (kons "B" (kons 1 leer)))
=> #<record:nll "B" #<record:nll 1 #<record:leere-liste>>>
```

Die Signaturverletzung wird von DrRacket gemeldet.

Keine Signaturverletzung

```
   (: l4 (liste %a))
   (define l4 (kons "B" (kons 1 leer)))
=> #<record:nll "B" #<record:nll 1 #<record:leere-liste>>>
```

Hier meldet DrRacket keine Signaturverletzung. Wie kann man das verstehen?

- Die Sortenprüfung von Scheme ist nicht mächtig genug, um zu erkennen, dass die beiden Listenelemente von der selben Sorte sein müssten.
- %a entspricht hier der Sorte `any` (neue Scheme-Version Racket v5.0.2).
- Man könnte %a auch als `(mixed number string)` instanziieren.

Signaturverletzungen in anderen Sprachen

```
   (: l4 (liste %a))
   (define l4 (kons "B" (kons 1 leer)))
=> #<record:nll "B" #<record:nll 1 #<record:leere-liste>>>
```

Es gibt Programmiersprachen, in denen die Sortenprüfung mächtig genug ist, um zu erkennen, dass die beiden Listenelemente von der selben Sorte sein müssten, und in denen es zudem die gemischten Sorten so nicht gibt. Was bedeutet dass für `l3` bzw. `l4`?
Diese Liste wäre verboten!

Eine Prozedur

An `list-length` ändert sich außer der Signatur nichts:

```
(: list-length ((liste %a) -> number))
(define list-length
  (lambda (xs)
    (cond
     ((leer? xs)
      0)
     ((nichtleer? xs)
      (+ 1 (list-length (rumpf xs)))))))
```

Noch eine Prozedur

Wir erinnern uns: `list-sum` funktioniert nur auf Zahlenlisten. Müssen wir deshalb auf `zahlenlisten.rkt` zurückgreifen und die Sorte `zahlenliste` verwenden?
Nein! Eine Zahlenliste wird durch `(liste number)` bezeichnet:

```
(: list-sum ((liste number) -> number))
(define list-sum
  (lambda (xs)
    (cond
      ((leer? xs) 0)
      ((nichtleer? xs) (+ (kopf xs)
                          (list-sum (rumpf xs)))))))
```

polymorphe_listen.rkt

Die soeben definierten Listen finden Sie unter dem Namen `polymorphe_listen.rkt` auf
http://www.informatik.uni-freiburg.de/~ki/teaching/ws1011/infoI/

Weitere Beispiele für polymorphe Prozeduren

Polymorphismus gibt es nicht nur im Zusammenhang mit Listen.

```
; Die Identität: Argument zurückgeben
(: identity (%a -> %a))
(define identity
  (lambda (x) x))
; 1. Projektion: Das erste Argument zurückgeben
(: proj_one (%a %b -> %a))
(define proj_one
  (lambda (x y) x))
; 2. Projektion: Das zweite Argument zurückgeben
(: proj_two (%a %b -> %b))
(define proj_two
  (lambda (x y) y))
```

Parametrisch polymorphe Signaturen

- Eine Prozedur, die einen Teil ihrer Argumente immer gleichartig behandelt (ohne sie „anzuschauen“ nur „herumschiebt“), kann eine parametrisch polymorphe Signatur erhalten. Diese verwendet eine Sortenvariable (wie z.B. %a, %b, ...) um über eine konkrete Sorte zu abstrahieren.
- Die parametrisch polymorphe Signatur fasst alle Signaturen zusammen, wobei Sorten konsistent für die Sortenvariablen eingesetzt werden (Signaturinstanzen).
- Beispiel: Wenn p die Signatur (`number %a %a -> %a`) erfüllt, dann erfüllt p alle folgenden Signaturen
  - `number number number -> number`
  - `number string string -> string`
  - `number boolean boolean -> boolean`
  - `number cookie cookie -> cookie`

Noch ein Beispiel: Paare

```
; Ein Paar von A und B ist ein Wert
;  (make-paar a b)
; wobei a und b jeweils Werte aus A bzw. B sind.
(define-record-procedures-parametric-2 paar paar-of
  make-paar paar?
  (erstes zweites))
```

Kraft `define-record-procedures-parametric-2` lauten die Signaturen:

```
(: make-paar (%a %b -> (paar-of %a %b)))
(: paar? (%value -> boolean))
(: erstes ((paar-of %a %b) -> %a))
(: zweites ((paar-of %a %b) -> %b))
```

Beachte: deutsche Namen zum Ausschließen von Kollisionen!

5.4 Eingebaute Listen

Vordefinierte Listen

Ab Sprachlevel „Die Macht der Abstraktion“ sind Listen in Scheme eingebaut. Die Namen lauten folgendermaßen:

	Hier definiert:	Eingebaut:
Sorte „leer“	`leere-liste`	`empty-list`
Konstrukt „leer“	`leer`	`empty`
Prädikat „leer“	`leer?`	`empty?`
Sorte „nichtleer“	`nll`	$\not\exists$
Konstr. „nichtleer“	`kons`	`make-pair` / `cons`
Präd. „nichtleer“	`nichtleer?`	`pair?` / `cons?`
Selektor	`kopf`	`first`
Selektor	`rumpf`	`rest`
Sorte	`(liste` σ`)`	`(list` σ`)` / `(list-of` σ`)`

Ab Scheme-Version Racket v5.0.2

Die Namen `make-pair` und `pair?` sind etwas überraschend . . .

Historische Altlasten

Wir haben oben eine Recordsorte `paar` definiert:

```
(: make-paar (%a %b -> (paar-of %a %b)))
(: paar? (%value -> boolean))
```

Man würde erwarten, dass `make-pair` genau `make-paar` und `pair?` genau `paar?` entspricht.
Das war auch einmal so. Die Verwendung von `make-pair` zur Konstruktion von Listen war eine mögliche Verwendung von `make-pair`. Irgendwann hat sich die Verwendung von `make-pair` für Listen so durchgesetzt und die Zulassung anderer Verwendungen hat so viel Verwirrung gestiftet, dass in unserer Scheme-Version alle anderen Verwendungen ausgeschlossen wurden.

Erklärung von Listensignaturen

Wir haben die Bedeutung von Listensignaturen schon für unsere selbst definierten polymorphen Listen erklärt. Hier noch einmal leicht umformuliert für eingebaute Listen:

Die Liste `xs` ist von der Sorte `(list` σ`)` genau dann, wenn

- `xs` das Prädikat `empty?` erfüllt oder
- `xs` das Prädikat `pair?` erfüllt, so dass
  - `(first xs)` von der Sorte σ ist und
  - `(rest xs)` von der Sorte `(list` σ`)` ist.

Länge einer Liste

Die Länge einer Liste ist vordefiniert als `length` und hat die parametrisch polymorphe Signatur

```
(: length ((list %a) -> number))
```

Wie gesagt: Die Länge einer Liste ist unabhängig von Sorte der Listenelemente.

Ausgabe von Listen

Da Listen sehr häufig vorkommen, werden sie auf eine besonders kurze und lesbare Weise ausgegeben: Statt wie bei unseren selbstgebauten Listen

```
#<record:nll 5
     #<record:nll 2
          #<record:nll 1
               #<record:leere-liste>>>>
```

bekommen wir

```
#<list 5 2 1>
```

Konstruktionsanleitung 7 (eingebaute Listen)

Hier Konstruktionsanleitung 7 noch einmal leicht umformuliert für eingebaute Listen:

```
(: p ((list σ) -> τ))
(define p
  (lambda (l)
    (cond
      ((empty? l) ...)
      ((pair?  l)
       ... (first l)
       ... (p (rest l)) ...))))
```

- Fülle zuerst den `empty?`-Zweig aus.
- Fülle dann den `pair?`-Zweig aus unter der Annahme, dass der **rekursive Aufruf** `(p (rest l))` das gewünschte Ergebnis für den Rest der Liste liefert.

Zusammenfassung

- Zahlenlisten
- Rekursion auf Listen: Listen als Argumente
- Polymorphismus
  - Listen mit beliebiger Elementsorte
  - Polymorphe Listen
  - Sonstiger Polymorphismus
- Eingebaute Listen

Literatur


Herbert Klaeren and Michael Sperber.
Die Macht der Abstraktion.
Teubner Verlag, 2007.