

Deklarative Programmierung

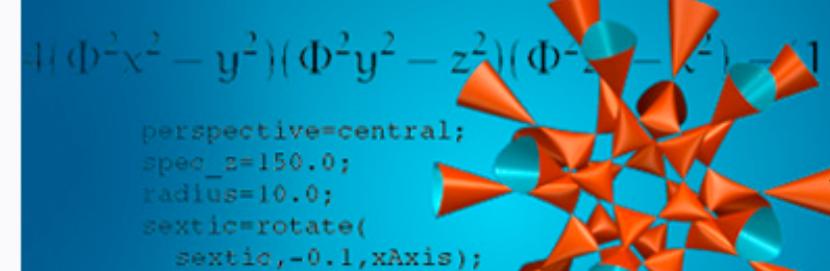
Sommersemester 2018

Prof. Christoph Bockisch

(Programmiersprachen und –werkzeuge)

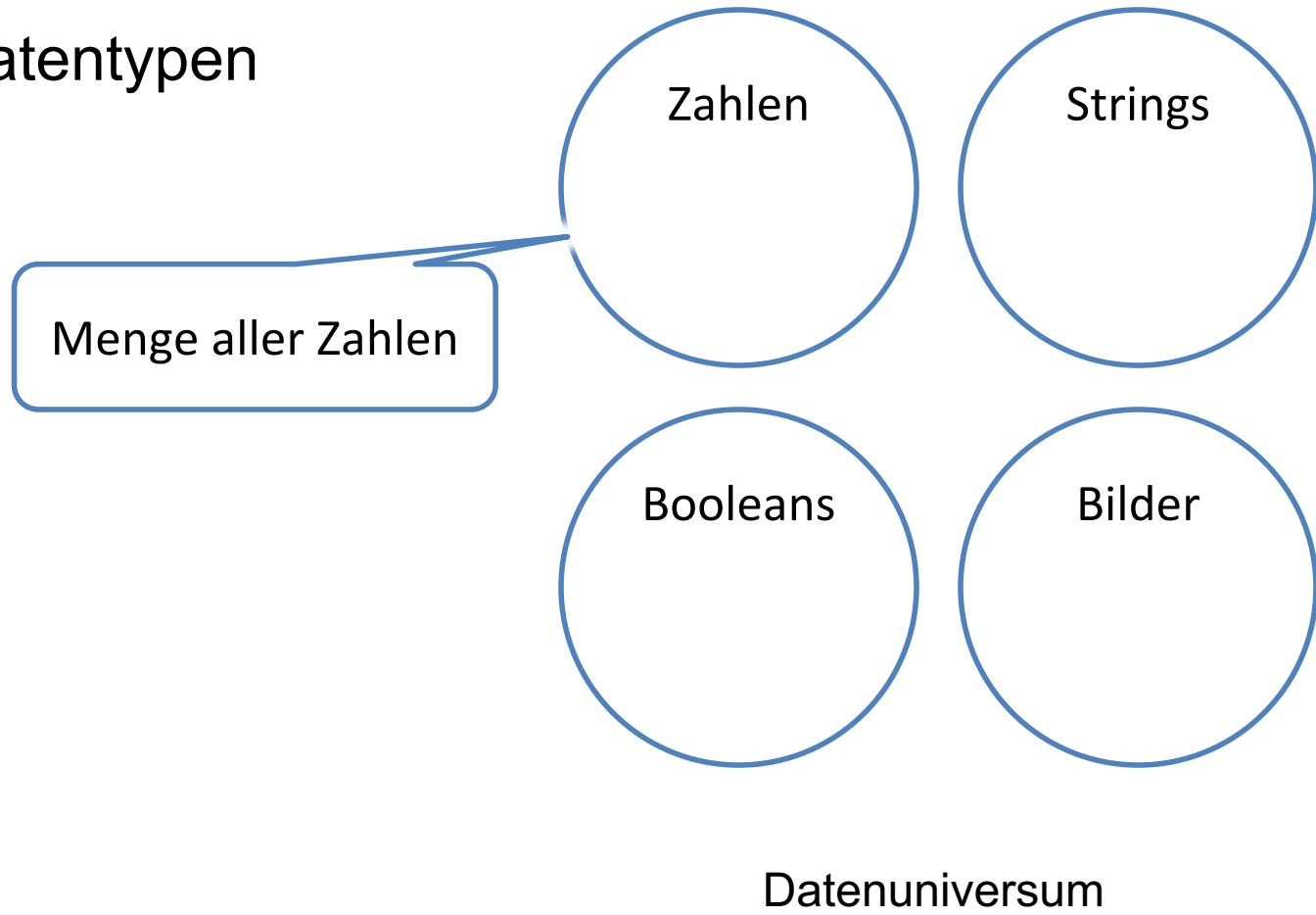
Steffen Dick, Alexander Bille, Johannes Frankenau,
Patrick Frömel, Niclas Schmidt, Jonas Stettin,
Robert Tran, Julian Velten

[Skript 8.11, 9]



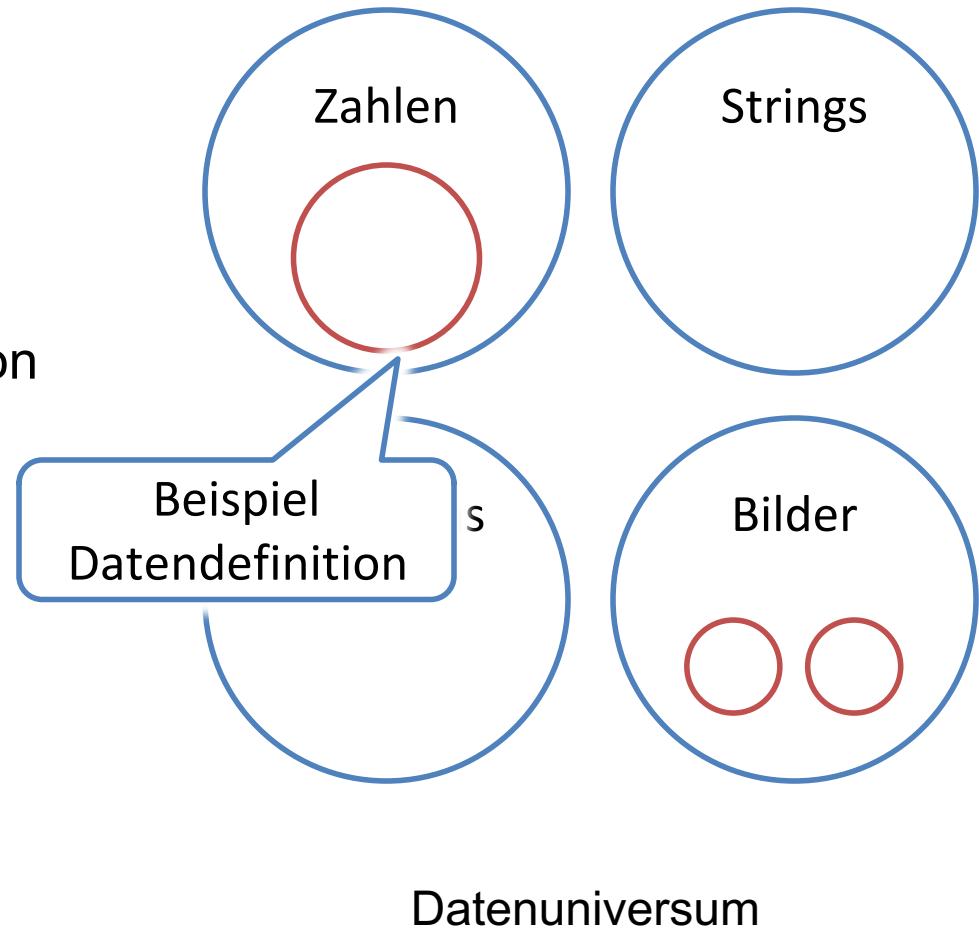
Datenuniversum

- Primitive Datentypen



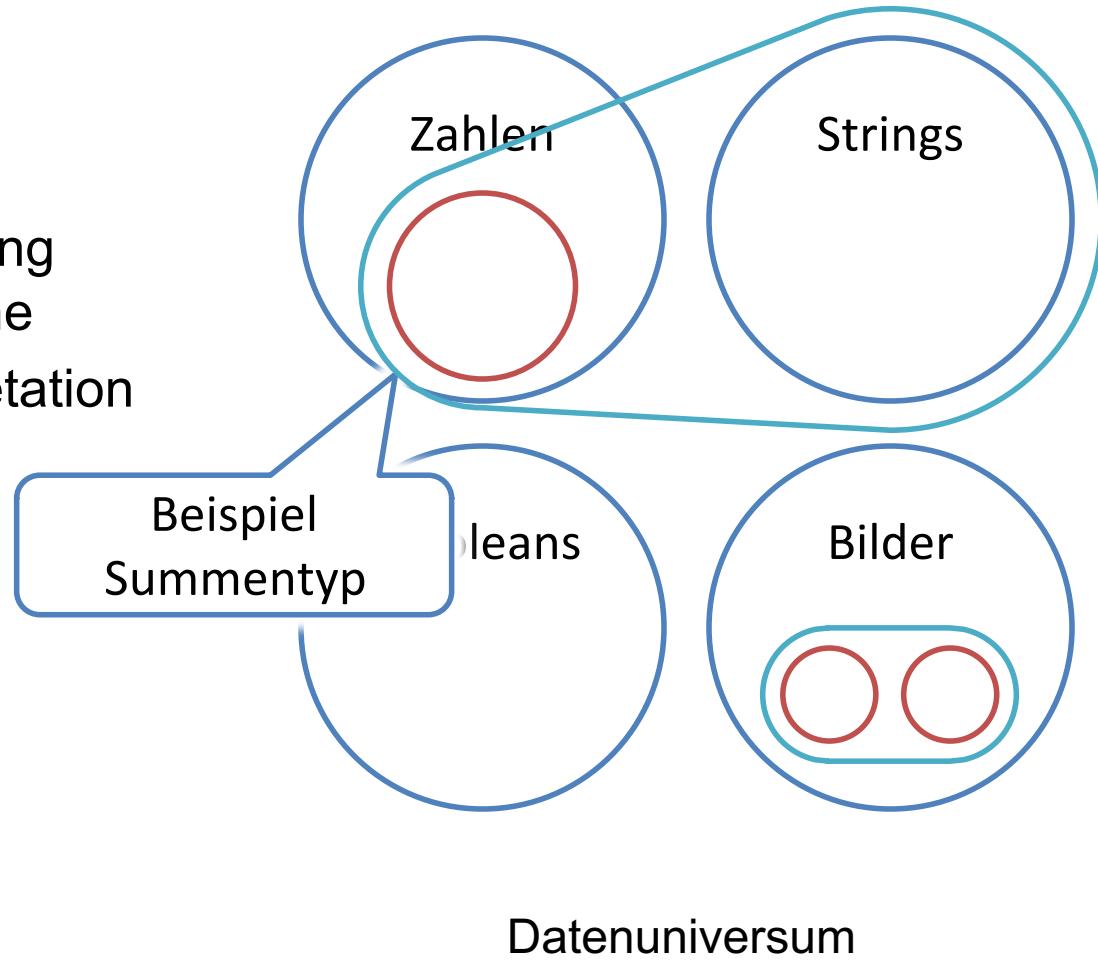
Datenuniversum

- Einfache Datendefinition
 - Als Kommentar
 - Keine präzise Bedeutung für Programmiersprache
 - Dokumentierte Interpretation für Programmierer



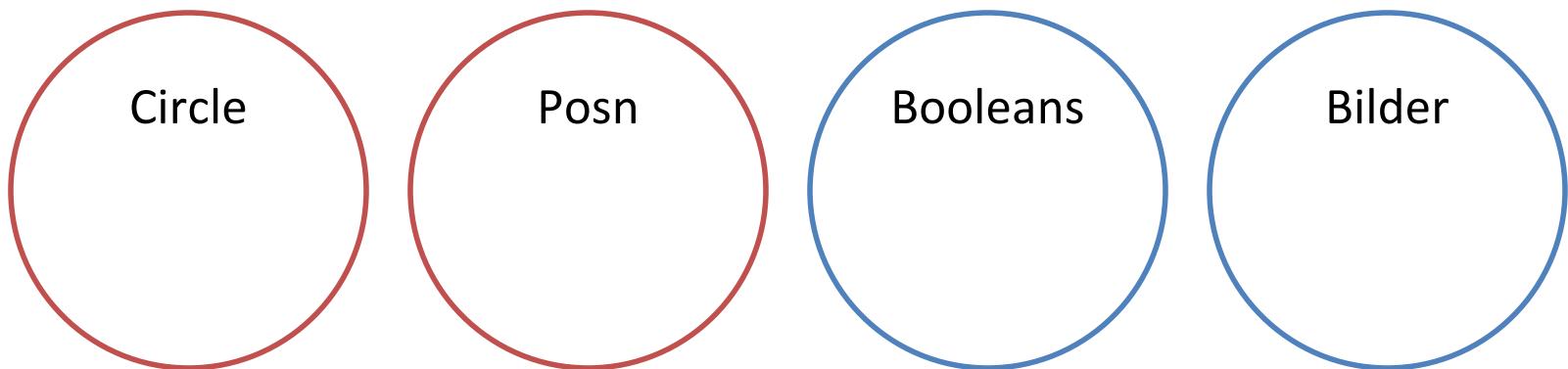
Datenuniversum

- Datendefinition
 - Als Kommentar
 - Keine präzise Bedeutung für Programmiersprache
 - Dokumentierte Interpretation für Programmierer
- Einschränkung des Datenuniversums
 - Teilmengen



Datenuniversum

- Strukturdefinition
 - Mit Sprachmittel
- Erweiterung des Datenuniversums



Syntax vs. Semantik

- Datenuniversum
 - Alle Werte, die sich durch Grammatik <v> bilden lassen
- Einschränkung von Sprache erzwungen:
 - Nur Instanzen von definierten Strukturen
- Einschränkung durch Interpretation
 - Datendefinition gibt vor, welche Werte für Struktur-Felder verwendet werden dürfen

Beispiel

(define-struct posn (x y))

; A Posn is a structure: (make-posn Number Number)

; interp. the number of pixels from left and from top

<make-undefined 3 4 >

Nicht Teil des Datenuniversums:
Struktur undefined nicht definiert
Kein entsprechender Ausdruck (make-
undefined 3 4)

<make-posn true "x" >

Teil des Datenuniversums aber
Widerspricht Interpretation.
Daher nicht Teil der Wertemenge des
Datentyps.

<make-posn 3 4 >

Teil Des Datenuniversums.

Datendefinition

- Eine Datendefinition: kohärente Teilmenge des Datenuniversums
- Funktions-Signatur
 - Welche Werte aus dem Datenuniversum werden als Argumente akzeptiert?
 - Welche Werte aus dem Datenuniversum werden als Resultat produziert?

“Rechnen“ mit Typen

- Aus Sicht von Typisomorphie
 - Namen spielen keine Rolle
 - Nur Typen spielen Rolle
 - Allgemeine Notation
 - Produkttypen: (* String String Number)
 - Summentyp: (+ ...)
 - Isomorphie: =

- Beispiel Summentyp

; A UniversityPerson is either:

; - a Student

; - a Professor

; - a ResearchAssociate

(+ Student Professor ResearchAssociate)

“Rechnen“ mit Typen

- Es gelten bekannte Rechenregeln

; Assoziativität von $*$

$$(* X (* Y Z)) = (* (* X Y) Z) = (* X Y Z)$$

; Kommutativität von $*$

$$(* X Y) = (* Y X)$$

; Assoziativität von $+$

$$(+ X (+ Y Z)) = (+ (+ X Y) Z) = (+ X Y Z)$$

; Kommutativität von $+$

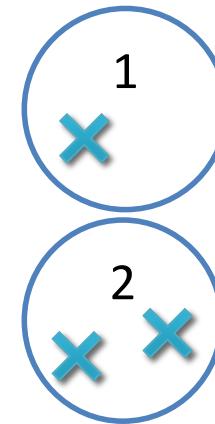
$$(+ X Y) = (+ Y X)$$

; Distributivität von $*$ und $+$

$$(* X (+ Y Z)) = (+ (* X Y) (* X Z))$$

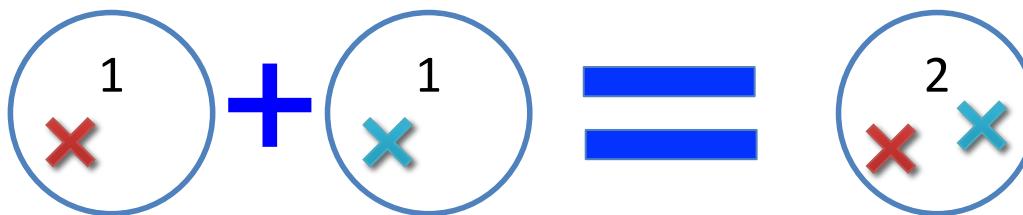
“Rechnen“ mit Typen

- Summentypen als Tagged Unions
 - Alternativen können immer unterschieden werden
- Dann geht Analogie noch weiter
- Wir nennen
 - Einen Datentyp mit einem Wert 1
 - Einen Datentyp mit zwei Werten 2

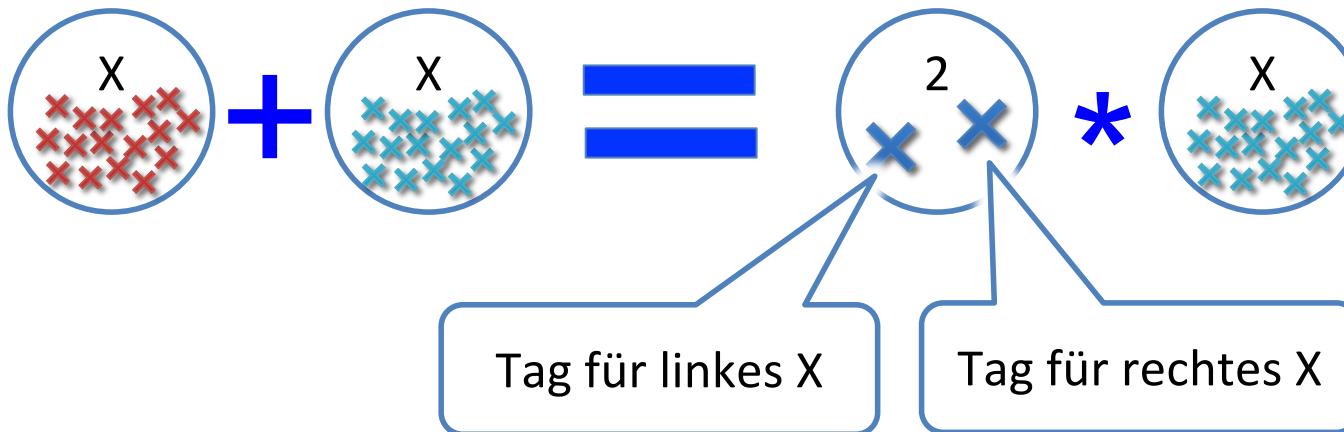


“Rechnen“ mit Typen

- $(+ 1 1) = 2$



- $(+ X X) = (* 2 X)$



“Rechnen“ mit Typen

- Kardinalität von Typen
 - $|X|$ - Kardinalität von Typ X
 - Anzahl der Werte von Typ X
- Rechnen mit Kardinalitäten
 - $|X + Y| = |X| + |Y|$
 - $|X * Y| = |X| * |Y|$

Refactoring von algebraischen Datentypen

- (define-struct student1 (lastname firstname matnr))
- ; a Student1 is: (make-student1 String String Number)
- ; interp. lastname, firstname, and matrikel number of
- ; a student
- (define-struct student2 (matnr lastname firstname))
- ; a Student2 is: (make-student2 Number String String)
- ; interp. matrikel number, lastname, and firstname of
- ; a student
- (define-struct fullname (firstname lastname))
- ; a FullName is: (make-fullname String String)
- ; interp. first name and last name of a person
- (define-struct student3 (fullname matnr))
- ; a Student3 is: (make-student3 FullName Number)
- ; interp. full name and matrikel number of a student

Aufgabe: finden Sie ein Beispiel für einen Studenten, der sich in student1, -2 oder -3 darstellen lässt, aber nicht in den jeweils anderen Strukturen.

Es gibt kein Beispiel!

Refactoring von algebraischen Datentypen

- Im Beispiel:
 - Alle Repräsentation können die gleichen Informationen darstellen
 - Es lässt sich jeweils eine Funktion schreiben, die eine Instanz einer Struktur in eine Instanz einer anderen Struktur abbildet

; Student1 -> Student2

(define (Student1ToStudent2 s)

 (make-student2 (student1-matnr s) (student1-lastname s)
 (student1-firstname s)))

; Student2 -> Student1

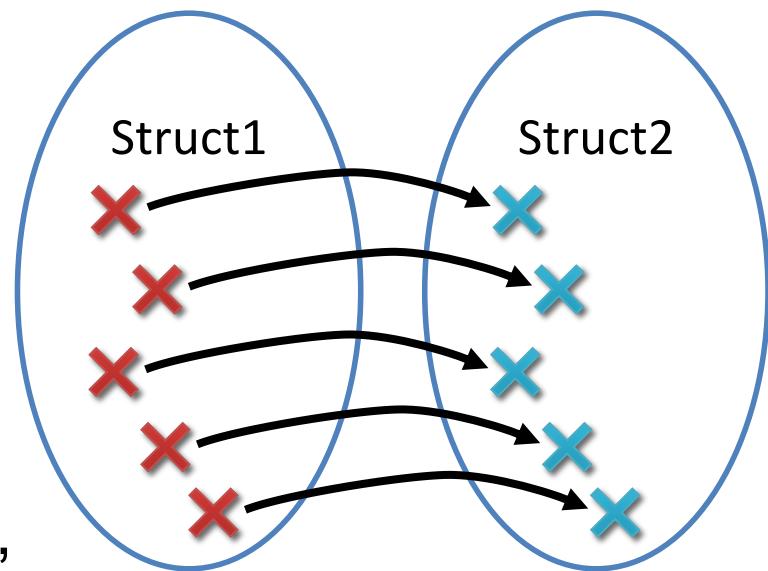
(define (Student2ToStudent1 s)

 (make-student1 (student1-lastname s) (student1-firstname s)
 (student1-matnr s)))

“Bijektive Abbildung”

Isomorphie

- Bijektive Abbildung:
; Struct1 -> Struct2
`(define (Struct1ToStruct2 s) ...)`
 - Jedem Wert aus Struct1 wird ein Wert auf Struct2 zugeordnet
 - Jeder Wert aus Struct2 kommt als Resultat vor
- Zwei Datentypen sind isomorph, wenn es eine bijektive Abbildung zwischen ihnen gibt



Refactoring mit algebraischen Datentypen

- Ein Datentyp kann immer durch einen isomorphen Datentyp ausgetauscht werden
 - Anpassen der Konstruktor-Aufrufe
 - Anpassen der Selektor-Aufrufe
- Isomorphie auch, wenn Daten gruppiert sind

; Student1 -> Student3

```
(define (Student1ToStudent3 s)
  (make-student3 (make-fullname (student1-firstname s)
                                (student1-lastname s)) (student1-matnr s)))
; Student3 -> Student1
(define (Student3ToStudent1 s)
  (make-student1 (fullname-lastname (student3-fullname s))
                (fullname-firstname (student3-fullname s))
                (student3-matnr s)))
```

Refactorings mit algebraischen Datentypen

- Mögliche Refactorings
 - Vertauschung von Reihenfolgen
 - "Inlining" von Datentypen
 - "Outsourcing" von Datentypen
 - "Ausmultiplikation" von Produkten

Größe von Daten

- Bisher:
 - Daten setzen sich zusammen aus fester Anzahl von atomaren Daten

- Beispiel
 - (define-struct gcircle (center radius))
; A GCircle is (make-gcircle Posn Number)
; interp. the geometrical representation of a circle

Posn besteht
aus 2 Zahlen

1 Zahl

Kreis setzt sich
aus 3 Zahlen
zusammen

Daten beliebiger Größe

- Oft ist die Größe von Daten abhängig von der Eingabe
- Beispiel: Stammbaum
 - Repräsentation des Vorfahren einer Person

```
(define-struct person (name father mother))
```

; A Person is: (make-person String Parent Parent)

; interp. the name of a person with his/her parents

```
(define-struct parent (name grandfather grandmother))
```

; A Parent is: (make-parent String String String)

; interp. the name of a person with the names of his/her grandparents

Für jede Generation muss eine neue Struktur definiert werden.

Daten beliebiger Größe

(define-struct person (name father mother))

; A Person is: (make-person String Parent Parent)

; interp. the name of a person with his/her parents

(define-struct parent (name grandfather grandmother))

; A Parent is: (make-parent String String String)

; interp. the name of a person with the names of his/her grandparents

Gibt es ein Problem mit
dieser Definition?

Redundanz: Struktur von Person und
Parent ist annähernd identisch.

Selbstähnlichkeit

- Daten können "selbstähnlich" sein
 - Ein Teil hat die gleiche Struktur wie das Ganze
 - Beispiel: jede Person in jeder Generation hat eine Mutter und einen Vater
- Rekursive Datentypen
 - Möglichkeit, selbstähnliche Daten zu definieren

Rekursive Datentypen

(define-struct person (name father mother))

; A FamilyTree is: (make-person name FamilyTree FamilyTree)

; interp. the name of a person and the tree of his/her parents.

Rekursive Datentypen

(define-struct person (name father mother))

; A FamilyTree is: (make-person name FamilyTree FamilyTree)

; interp. the name of a person and the tree of his/her parents.

In der Definition für den
Datentyp wird auf den
Datentyp selbst verwiesen.

- Bisher: In der Definition eines DatenTypen dürfen vorher definierte Datentypen verwendet werden
- Jetzt: In der Definition eines DatenTypen dürfen vorher definierte Datentypen und der aktuell definierte Datentyp verwendet werden

Rekursive Datentypen

- Wie erzeugt man einen Wert eines rekursiven Datentyps?

(make-person "Heinz")

(make-person "Horst")

(make-person "Joe" ...)

...)

...)

Wie kann diese
Verkettung aufhören?

Rekursive Datentypen

- Rekursion muss terminieren können
- Definition von FamilyTree als Summentyp

(define-struct person (name father mother))

; A FamilyTree is either:

; - (make-person name FamilyTree FamilyTree)

; - false

; interp. either the name of a person and the tree of its parents,

; or false if the person is not known/relevant.

Rekursive Datentypen

```
(define HEINZ  
  (make-person "Heinz"  
    (make-person "Elke" false false)  
    (make-person "Horst"  
      (make-person "Joe"  
        false  
        (make-person "Rita"  
          false  
          false))  
      false)))
```

Werte eines rekursiven Datentyps

- Definition der möglichen Werte "induktiv"
- Erster Schritt: Werte, die nicht rekursiv sind:
 - $ft_1 = \{ \text{false} \}$
- Zweiter Schritt: rekursive Werte, die mit ft_1 konstruiert werden können
 - $ft_2 = ft_1 \cup \{ (\text{make-person} \text{ name } \text{false} \text{ false}) \mid \text{name} \in \text{Strings} \}$
- Folgeschritte: Werte, die aus Werten des vorherigen Schritts konstruiert werden können
 - $ft_n = ft_{n-1} \cup \{ (\text{make-person} \text{ name } p_1 p_n) \mid \text{name} \in \text{Strings} \text{ und } p_1, p_2 \in ft_{n-1} \}$
- Menge der Werte: ft_n für $n \rightarrow \infty$
 - Die Tiefe der Rekursion und damit die Anzahl der Werte ist unbegrenzt

Funktionen über rekursiven Datentypen

- Frage: Hat eine Person p einen Vorfahren mit Namen a?
- Wir sagen, eine Person ist selbst auch Vorfahre von sich
- Vorgehen:
 - Hat die Person selbst den Namen?
 - Hat der Vater oder die Mutter den Namen?
 - Und so weiter für deren Väter und Mütter

Funktionen über rekursiven Datentypen

```
; FamilyTree String -> Boolean
; determines whether person p has an ancestor a
(check-expect (person-has-ancestor HEINZ "Joe") true)
(check-expect (person-has-ancestor HEINZ "Emil") false)
(define (person-has-ancestor p a)
  (cond [(person? p)
         ...(person-name p) ...
         ...(person-father p)...
         ...(person-mother p) ...]
        [else ...]))
```

Entwurfsrezept für Summentyp

Entwurfsrezept für Produkttyp

Funktionen über rekursiven Datentypen

- Laut Entwurfsrezept:
 - Hilfsfunktionen für alle Felder, die selbst komplexen Typ haben

```
(define (person-has-ancestor p a)
  (cond [(person? p)
         ... (person-name p) ...
         ... (father-has-ancestor (person-father p) ...)...
         ... (mother-has-ancestor (person-mother p) ...)...]
        [else ...]))
```

Wie müssen **father-has-ancestor** und **mother-has-ancestor** aussehen?

Funktionen über rekursiven Datentypen

```
; FamilyTree String -> Boolean  
; determines whether father p has an ancestor a  
(define (father-has-ancestor p a)  
  (cond [(person? p)  
          ... (person-name p) ...  
          ... (grand-father-has-ancestor (person-father p) ...) ...  
          ... (grand-mother-has-ancestor (person-mother p) ...) ...]  
        [else ...]))
```

Wir müssen wieder
zwei Hilfsfunktionen
definieren.

Es fällt auf, Signatur und
Bedeutung der
Funktionen sind identisch.

Funktionen über rekursiven Datentypen

- Beim verarbeiten von selbstähnlichen Teildaten kann die Funktion selbst aufgerufen werden

```
(define (person-has-ancestor p a)
  (cond [(person? p)
         ... (person-name p) ...
         ... (person-has-ancestor (person-father p) ...)...
         ... (person-has-ancestor (person-mother p) ...)...]
        [else ...]))
```

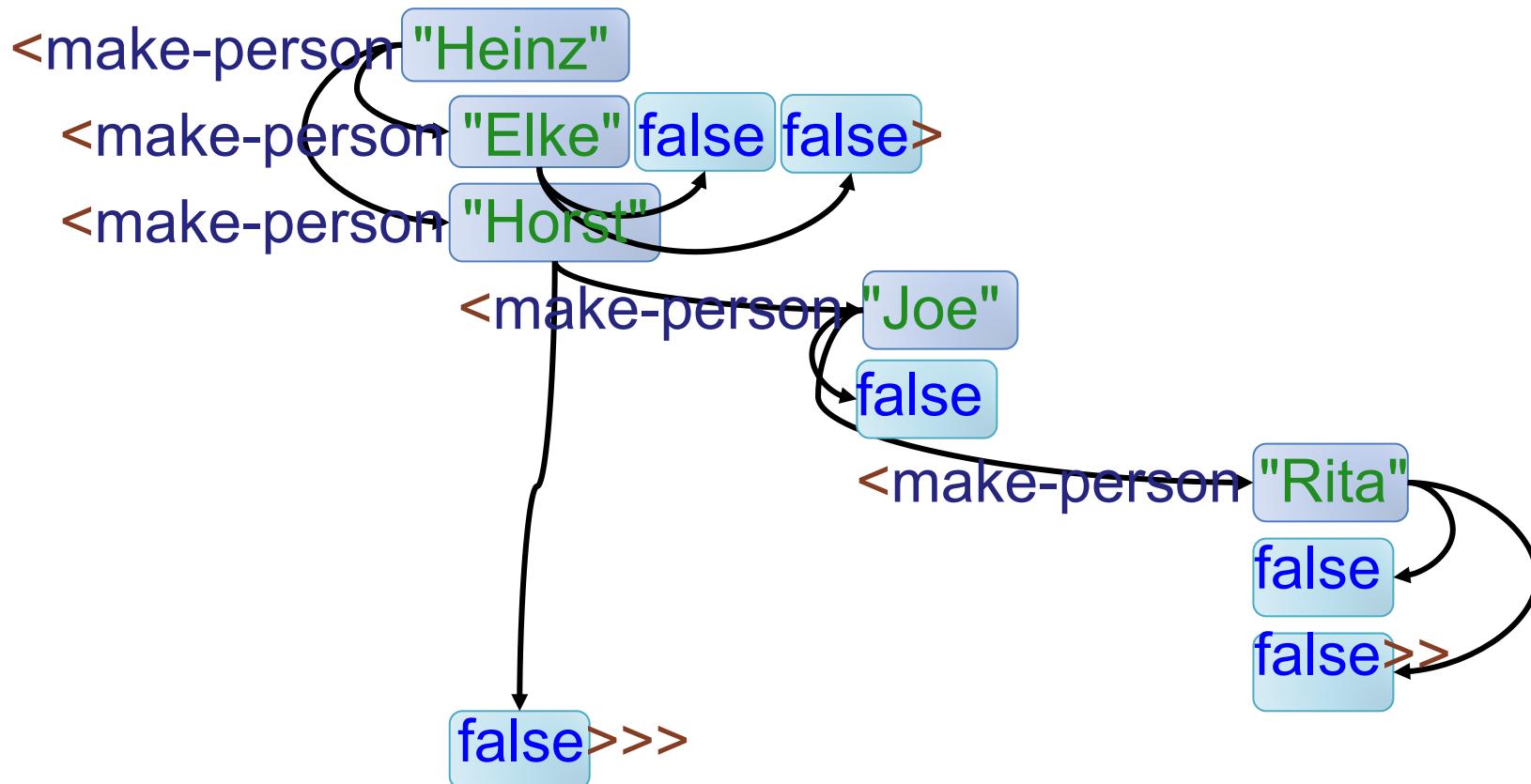
Funktionen über rekursiven Datentypen

- Wie bisher: Die Struktur der Daten bestimmt die Struktur der Funktion
 - Bei rekursiven Daten
 - Sind auch die Funktionen rekursiv

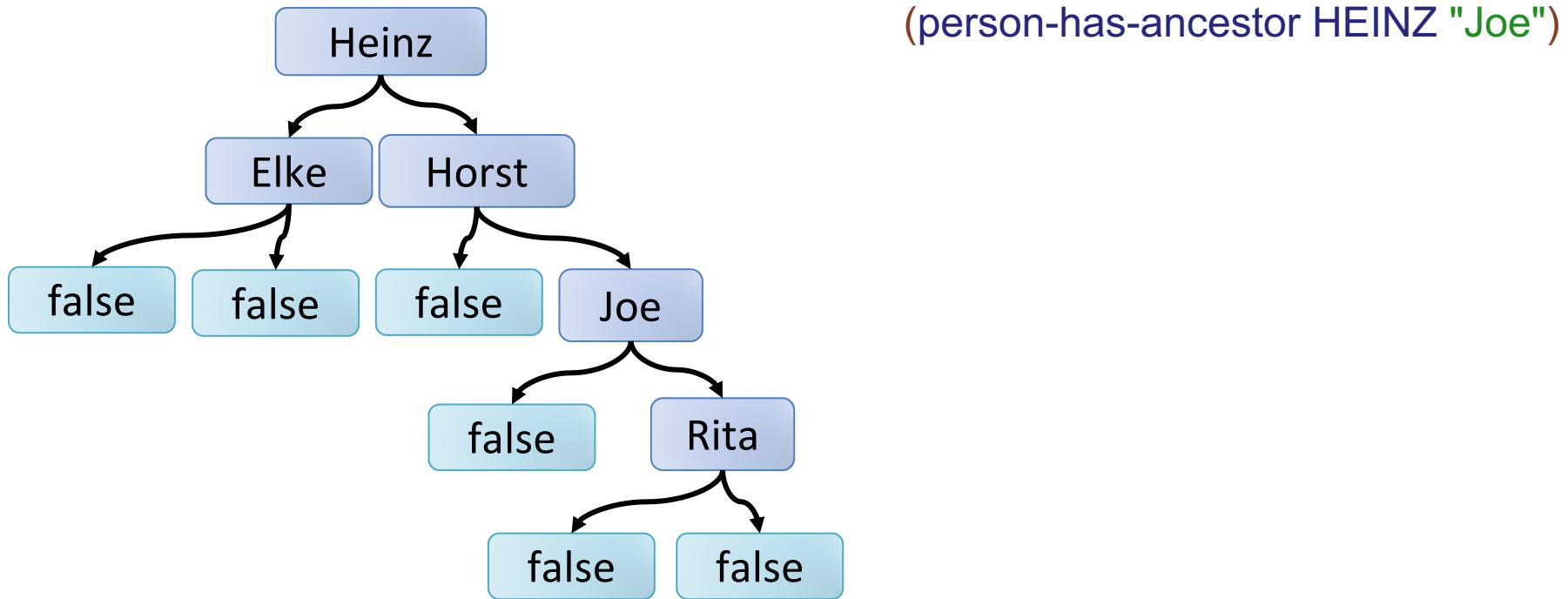
Funktionen über rekursiven Datentypen

```
(define (person-has-ancestor p a)
  (cond [(person? p)
         (or
           (string=? (person-name p) a)
           (person-has-ancestor (person-father p) a)
           (person-has-ancestor (person-mother p) a))]
        [else false]))
```

Rekursive Datentypen



Rekursive Datentypen



Funktionen über rekursiven Datentypen

- Wichtig: rekursive Aufrufe dürfen nur für rekursive Teildaten gemacht werden
- Bei jedem rekursiven Aufruf muss das “Problem“ kleiner werden, sonst bricht die Rekursion nicht ab
- Jede rekursive Funktion muss eine Abbruchbedingung haben, d.h. einen Fall ohne rekursiven Aufruf

Funktionen über rekursiven Datentypen

```
(define (person-has-ancestor p a)
```

```
  (cond [(person? p)
```

```
        (or
```

```
            (string=? (person-name p) a)
```

```
            (person-has-ancestor (person-father p) a))
```

```
            (person-has-ancestor (person-mother p) a)))]
```

```
  [else false]))
```

Rekursiver Aufruf mit Teildaten.

Abbruchbedingung

Abbruch von rekursiven Funktionen

- "Induktionsbeweis"
 - Schrittweises Beweisen einer Eigenschaft
 - Analog zum Schrittweisen Aufbau der Werte eines rekursiven Datentyps
- Am Beispiel FamilyTree
 - Für jede Person p gibt es ein i , so dass $p \in ft_i$ und $p \notin ft_{i-1}$
 - Die Vorfahren von p sind dann $\in ft_{i-1}$
 - Das heißt, das Argument der rekursiven Aufrufe ist $\in ft_{i-1}$
 - Für $p \in ft_0$ terminiert die Funktion offensichtlich: $ft_0 = \{ \text{false} \}$
 - Damit terminiert die Funktion auch für $p \in ft_1, p \in ft_2$, usw.
- Wir haben gezeigt: die Funktion person-has-ancestor ist "wohldefiniert"
 - Für jeden möglichen Argument-Wert liefert sie ein Ergebnis

Abbruch von rekursiven Funktionen

- Gegeben eine andere rekursive Funktion

; FamilyTree -> Boolean

; determines whether person p has an ancestor a

(check-expect (person-has-ancestor-stupid HEINZ "Joe") true)

(check-expect (person-has-ancestor-stupid HEINZ "Emil") false)

(define (person-has-ancestor-stupid p a)

 (person-has-ancestor-stupid p a))

- Terminiert diese auch?

- Für $p \in ft_i$ ist auch für den rekursiven Aufruf $p \in ft_i$

- Sei $p \in ft_0$, also $p = \text{false}$, dann terminiert die Reduktion (also auch die Funktion) nicht

- Wir haben also gezeigt, dass die Funktion nicht terminiert

Strukturelle Rekursion

- Rekursive Daten
 - Unbestimmte Größe
 - Endliche Größe
- Daher
 - Folgt eine rekursive Funktion der rekursiven Struktur der Daten
 - Und sind die Daten wohldefiniert
 - Dann ist die Funktion wohldefiniert

Produzieren von Instanzen rekursiver Datentypen

- Beispielsweise Erzeugen eines modifizierten Stammbaums (Hinzufügen eines Titels)
- Vorgehen
 - Analog zur vorigen Funktion
 - Rekursion geschachtelt
 - Angepasster Abbruch-Fall

```
(define (promote p t)
  (cond [(person? p)
         (make-person
          (string-append t (person-name p)))
          (promote (person-father p) t)
          (promote (person-mother p) t))]
        [else p]))
```

Rekursiver Aufruf als Argument von make-person.

Rückgabe des aktuellen Arguments.

Listen

- Bisher: binäre Bäume als rekursive Datenstruktur
- Binärer Baum: jeder Knoten hat zwei Nachfolger
- Bäume können auch andere Anzahlen von Nachfolgern haben
- Sonderfall:
 - Jeder Knoten hat einen Nachfolger
 - Baum “degeneriert“ zu einer Liste
- D.h.: Listen sind auch rekursive Datenstrukturen

Listen als rekursive Datenstruktur

(define-struct **Ist** (**first rest**))

; A List-of-Numbers is either:

; - (make-Ist Number List-Of-Numbers)

; - false

; interp. the head and rest of a list, or the empty list

(make-Ist 1 (make-Ist 2 (make-Ist 3 false))))

Funktionen über Listen

- Funktionsentwurf analog zu Bäumen

```
; List-Of-Numbers -> Number
```

```
; adds up all numbers in a list
```

```
(check-expect (sum (make-lst 1 (make-lst 2  
                                (make-lst 3 false))))) 6)
```

```
(define (sum l) ...)
```

Wie sieht das
Template aus?

Funktionen über Listen

- Funktionsentwurf analog zu Bäumen

; List-Of-Numbers -> Number

; adds up all numbers in a list

(check-expect (sum (make-lst 1 (make-lst 2

(define (sum l)

Selektor

(cond [(lst? l) ... (lst-first l) ... (sum (lst-rest l)) ...]

(Selektor false)))) 6)

[else ...])

Fall:

Rekursion

Fall: Rekursionsabbruch

Rekursiver Aufruf

Funktionen über Listen

- Funktionsentwurf analog zu Bäumen

; List-Of-Numbers -> Number

; adds up all numbers in a list

(check-expect (sum (2 3) 4 5) 6) (2 3) 4 5) 6)

(define (sum l)
 (cond [(lst? l) ...] (lst-first l) ... (sum (lst-rest l)) ...]
 [else ...]))

Was geschieht mit Resultat
der rekursiven Aufrufe?

Was ist das “Standard-
Resultat”?

Funktionen über Listen

- Funktionsentwurf analog zu Bäumen

```
; List-Of-Numbers -> Number
; adds up all numbers in a list
(check-expect (sum (make-lst 1 (make-lst 2
                                         (make-lst 3 false))))) 6)

(define (sum l)
  (cond [(lst? l) (+ (lst-first l) (sum (lst-rest l)))]
        [else 0]))
```

Listen eingebaut in BSL

- Konstruktor Funktion: `cons`
- Konstante für Leere Liste: `empty`

N.B.: Definition von Konstanten als “Marker“ (einige Bedeutung: Markierung einer bestimmten Absicht)

Beispiel:

```
(define-struct empty-lst ())  
(define EMPTYLIST (make-empty-lst))  
(make-lst 1 (make-lst 2 (make-lst 3 EMPTYLIST))))
```

Listen eingebaut in BSL

- Die Operation `cons` entspricht unserem `make-Ist`
 - Zusatznutzen: BSL überprüft, dass das zweite Argument Liste ist

> `(cons 1 2)`

`cons: second argument must be a list, but received 1 and 2`

- Wir können so eine Überprüfung auch selbst schreiben

`(define (our-cons x l)`

`(if (or (empty-lst? l) (lst? l))`

`(make-lst x l)`

`(error "second argument of our-cons must be a list")))`

Listen eingebaut in BSL

Eingebaute Liste	Selbstdefinierte Liste	Bedeutung
empty	EMPTYLIST	Wert für leere Liste
empty?	empty-lst?	Test, ob ein Wert die leere Liste ist
cons	our-cons	Konstruktor Funktion
first	lst-first	Funktion liefert das erste Element der Liste
rest	lst-rest	Funktion liefert Liste ohne das erste Element
cons?	lst?	Test, ob ein Wert eine Liste ist

Listen eingebaut in BSL

Verstecken von Funktionen ist ein wichtiges Mittel für Information Hiding. Bisher kennen wir noch keine Möglichkeit in BSL, um selbst Definitionen zu verstecken.

- BSL versteckt Definition von Struktur für Listen
 - Listen müssen über **cons** Funktion erzeugt werden
 - Zusätzlicher Test, dass zweites Argument eine Liste ist
 - Direkter Zugriff würde Test umgehen

Listen eingebaut in BSL

- Beispiel für Verwendung eingebauter Listen

; A List-of-Numbers is one of:

; - (cons Number List-Of-Numbers)

; - empty

; List-Of-Numbers -> Number

; adds up all numbers in a list

(check-expect (sum (cons 1 (cons 2 (cons 3 empty))))) 6)

(define (sum l)

 (cond [(cons? l) (+ (first l) (sum (rest l)))]

 [else 0]))

Syntaktischer Zucker für Listen

- Listen erzeugen
 - Geschachtelte Aufrufe von `cons` und `empty`
 - Aufwendig/ komplex
- Syntaktischer Zucker: `list` Funktion
 $(\text{list } \text{exp-1} \dots \text{exp-n})$

Transformation

$(\text{cons } \text{exp-1} (\text{cons} \dots (\text{cons } \text{exp-n} \text{ empty})))$

Syntaktischer Zucker für Listen

- Beispiel
 - (`list 1 2 3`) entspricht:
 - (`(cons 1 (cons 2 (cons 3 empty)))`)
- In Racket:
 - Sprach-Level auf “Anfänger mit Listenabkürzungen“ einstellen
 - Dann werden Listen-Werte auch in der verkürzten Schreibweise dargestellt

Listen für bestimmte Datentypen?

- Im Beispiel: Definition des Datentyps List-Of-Numbers
- Brauchen wir für jeden Datentyp eine Listenstruktur?
- Nein:
 - Die Struktur ist immer dieselbe (Don't Repeat Yourself!)
 - Viele Funktionen sind unabhängig vom Element-Datentyp
 - `second`, `third`, `fourth`, etc.
 - Wiederverwendung dieser Funktionen bei gemeinsamer Listenstruktur

Homogene Listen

- Listen bei denen alle Elemente denselben Typ haben, werden “homogen” genannt
 - Typ kann bspw. Summentyp sein, also Alternativen haben
- Basierend auf allgemeiner Listenstruktur
 - Datendefinition für Liste mit Typ-Einschränkung
 - Einschränkung durch “Typparameter”
 - ; A (List-of X) is one of:
 - ; - (cons X (List-of X))
 - ; - empty
 - Für X kann jeder beliebige Typ eingesetzt werden, bspw: **(List-of String)**, **(List-of Boolean)**, **(List-of (List-of String))** oder **(List-of FamilyTree)**

Listen mit Typvariablen

- “Typvariablen“ für Funktionen
 - Funktionen, die für mehrere Typen anwendbar sind
 - Mit Abhängigkeiten zwischen Typen von Parametern und Resultat
 - Deklaration von Typparameter: [X] am Anfang der Signatur
 - X kann nun in der Signatur verwendet werden

```
; [X] (List-of X) -> X  
(define (second l) ...)
```

- Typvariablen können durch beliebigen Typ ersetzt werden, bspw:
 $(\text{List-of Number}) \rightarrow \text{Number}$
 $(\text{List-of} (\text{List-of String})) \rightarrow (\text{List-of String})$

Warum Listen als rekursiver Datentyp?

- Auch Erweiterung der Sprache zur Unterstützung von Listen denkbar
- Konsequenzen
 - Repräsentation näher an der Hardware
 - Explizite Sprachkonstrukte zum Verarbeiten von Listen
 - Listen haben Sonderstatus
- Spracherweiterung nicht nötig in BSL
 - Definition und Verarbeitung von Listen mit Bordmitteln (rekursiver Datentyp)
 - Keine Einschränkung für die Verwendung von Listen
- Weitere Argumentation, siehe Skript 9.3.5

Natürliche Zahlen als rekursive Datenstruktur

- Auch natürliche Zahlen können als rekursive Datenstruktur dargestellt werden
 - (Das gilt für alle “abzählbaren“ Wertemengen)
 - Intuition:
 - Listen sind rekursive Datenstrukturen
 - Die Zahl n lässt sich durch eine Liste mit n Elementen repräsentieren
 - Jedoch spielt das Element keine Rolle
 - Wir speichern also keine Werte in der List
 - Sondern reihen leere Zellen aneinander

; A Nat (Natural Number) is one of:

; – 0

; – (add1 Nat)

- Beispiel: Repräsentation der Zahl 3 als (add1 (add1 (add1 0)))

Natürliche Zahlen als rekursive Datenstruktur

Liste	Nat
empty	0
empty?	zero?
cons	add1
first	-/-
rest	sub1
cons?	positive?

Natürliche Zahlen als rekursive Datenstruktur

- Warum wollen wir natürliche Zahlen als rekursiven Datentyp sehen?
 - So können wir wohldefinierte rekursive Funktionen über natürlichen Zahlen schreiben
 - Diese können rekursive Datenstrukturen mit gleicher Struktur verarbeiten oder produzieren.
 - Beispiel: Erzeugung einer Liste mit n Einträgen

Natürliche Zahlen als rekursive Datenstruktur

; [X] Nat X -> (List-of X)

; creates a list with n occurrences of x

(check-expect (iterate-value 3 "abc") (list "abc" "abc" "abc"))

(define (iterate-value n x) ...)

Wie muss Schablone aussehen?

Natürliche Zahlen als rekursive Datenstruktur

; [X] Nat X -> (List-of X)

; creates a list with n occurrences of x

(check-expect (iterate-value 3 "abc") (list "abc" "abc" "abc"))

(define (iterate-value n x)

(cond [(zero? n) ...])

Fall: Rekursionsabbruch

[(positive? n) ... (iterate-value (sub1 n) ...) ...])

Fall: Rekursion

Rekursiver Aufruf für Teildaten

Natürliche Zahlen als rekursive Datenstruktur

; [X] Nat X -> (List-of X)

; creates a list with n occurrences of x

(check-expect (iterate-value 3 "abc") (list "abc" "abc" "abc"))

(define (iterate-value n x)

 (cond [(zero? n) empty]

 [(positive? n) (cons x (iterate-value (sub1 n) x))]))

Funktionen über mehreren rekursiven Datentypen

- Beispiel:
 - append
 - Zusammenfügen von zwei Listen

; [X] (list-of X) (list-of X) -> (list-of X)

; concatenates l1 and l2

(check-expect (lst-append (list 1 2) (list 3 4)) (list 1 2 3 4))

(define (lst-append l1 l2) ...)

Welches Argument nutzen wir für
unser Entwurfsrezept?

Funktionen über mehreren rekursiven Datentypen

- Beispiel:
 - append
 - Zusammenfügen von zwei Listen

; [X] (list-of X) (list-of X) -> (list-of X)

; concatenates l1 and l2

(check-expect (lst-append (list 1 2) (list 3 4)) (list 1 2 3 4))

(define (lst-append l1 l2)

 (cond [(empty? l1) ...l2...]

 [(cons? l1) ... (first l1) ... (lst-append (rest l1) ...))])

Bspw.: Bevorzugen des ersten
Arguments.

Funktionen über mehreren rekursiven Datentypen

- Beispiel:

- append
 - Zusammenfügen von zwei Listen

; [X] (list-of X) (list-of X) -> (list-of X)

; concatenates l1 and l2

(check-expect (lst-append (list 1 2) (list 3 4)) (list 1 2 3 4))

(define (lst-append l1 l2)

 (cond [(empty? l1) l2]

 [(cons? l1) (cons (first l1) (lst-append (rest l1) l2))]))

Führt in diesem Fall zum Ziel.

Funktionen über mehreren rekursiven Datentypen

- Im Allgemeinen
 - Auswahl des Arguments für die Rekursion "kreativer Prozess"
 - Abhängig davon wie sich die Information sinnvoll zerlegen lässt

Entwurfsrezept mit rekursiven Datentypen

- 1. Informationsrepräsentation
- Informationen unbegrenzter Größe müssen über rekursiven Datentyp dargestellt werden
- Bedingungen:
 1. Der Datentyp ist ein Summentyp
 2. Es muss mindestens zwei Alternativen geben
 3. Mindestens eine der Alternativen ist nicht rekursiv
- Auf jeden Fall: Angabe von Datenbeispielen
 - Dient u.a. zur Überprüfung des korrekten Rekursionsabbruchs

Entwurfsrezept mit rekursiven Datentypen

- 3. Tests
- Datengröße ist unbegrenzt
- Daher ist ein Test pro Alternative unmöglich
- Testfälle beispielsweise für
 - Alternative mit und ohne Rekursion
 - Weitere Fallunterscheidungen

Entwurfsrezept mit rekursiven Datentypen

- 4. Schablone
- Rekursive Datentypen sind algebraische Datentypen
- Entspricht im Wesentlichen Schablone für algebraische Datentypen
- Ergänzungen:
 - Mindestens eine Alternative muss rekursiv sein
 - In diesem Fall: selbst-Aufruf statt Hilfsfunktion
 - Als Argument für rekursiven Aufruf: Selektor zur Extraktion des Wertes aus der Datenrekursion

Entwurfsrezept mit rekursiven Datentypen

- 5. Funktionsbody implementieren
- Start mit den Basisfällen (Rekursionsabbruch)
- Bei Rekursion: Wir gehen davon aus, dass der rekursive Funktionsaufruf das Ergebnis korrekt berechnet

Entwurfsrezept mit rekursiven Datentypen

- 7. Nachbearbeitung
- Gibt es Datentypen, die nicht rekursiv definiert sind, aber möglicherweise durch einen rekursiven Datentypen vereinfacht werden können?
- Z.B.:
 - Gibt es Datentypen mit derselben Struktur?
 - Hierarchien können typischerweise durch Rekursion modelliert werden