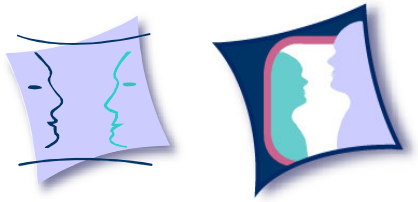
Technische Universität Darmstadt



Telecooperation/RBG

Grundlagen der Informatik I
Thema 3: Rekursive Datentypen und Strukturelle Rekursion

Prof. Dr. Max Mühlhäuser
Dr. Guido Rößling

Copyrighted material; for TUD student use only

Listen

- Mit Strukturen können Datenobjekte mit einer festen Zahl von Daten gespeichert werden
- Häufig wissen wir jedoch nicht, aus wie vielen Datenelementen eine Datenstruktur besteht
  - Oder die Struktur der Daten ist rekursiv
- Mit rekursiven Datentypen können auch beliebig große Datenobjekte strukturiert abgespeichert werden
- Idee: Ein Element der Datenstruktur speichert (direkt oder indirekt) ein Exemplar der Datenstruktur
  - Das nennt man eine **rekursive Datenstruktur**
  - Der Rekursionsanker wird benötigt, um eine endliche Datenstruktur zu bekommen
  - Diesen Rekursionsanker modellieren wir mit der Technik zu heterogenen Daten aus der letzten Vorlesung

Modellierung eines rekursiven Datentypen

- Eine Liste **`lst`** ist entweder
  - Die leere Liste, **`the-emptylst`**, oder
  - **`(make-lst s r)`**, wobei **`s`** ein Wert ist und **`r`** eine Liste

Listen

- Modellierung von Listen mit Strukturen

```
(define-struct lst (first rest))
(define-struct emptylst ())
(define the-emptylst (make-emptylst))

;; a list with 0 elements
(define list0 the-emptylst)

;; a list with 1 element
(define list1 (make-lst 'a the-emptylst))

;; a list with 2 elements
(define list2 (make-lst 'a
                (make-lst 'b
                 the-emptylst)))

;; get the 2nd element from list2
(lst-first (lst-rest list2)) → 'b
```

Listen

- Listen sind ein wichtiger Datentyp, weshalb es einen eingebauten Datentyp für Listen gibt
  - (und aus historischen Gründen)
  - Konstruktor mit 2 Argumenten: **`cons`** (für **`cons`**truct)
    - entspricht **`make-lst`** in unserem "Eigenbau"
  - Die leere Liste: **`empty`**
    - entspricht **`the-emptylst`**
  - Selektoren: **`first`** für das erste Element, **`rest`** für das zweite Element
    - entspricht **`lst-first`**, **`lst-rest`**
    - "historische" Namen für **`first`** und **`rest`**: **`car`** und **`cdr`**
  - Prädikate: **`list?`**, **`cons?`** und **`empty?`**
    - entspricht **`lst?`**, **`emptylst?`**

Listen

- Beispiel

```
;; a list with 0 elements
;; (define list0 the-emptylst)
(define list0 empty)

;; a list with 1 element
;; (define list1 (make-lst 'a the-emptylst))
(define list1 (cons 'a empty)

;; a list with 2 elements
;; (define list2 (make-lst 'a
;;                (make-lst 'b the-emptylst)))
(define list2 (cons 'a (cons 'b empty)))

;; get the 2nd element from list2
;; (lst-first (lst-rest list2)) → 'b
(first (rest list2)) → 'b
```

Listen

- Einziger Unterschied zwischen **`make-lst`** und **`cons`**
  - **`cons`** erwartet als zweites Argument **`empty`** oder **`(cons …)`**
  - z.B. **`(cons 1 2)`** ist ein Fehler, **`(make-lst 1 2)`** nicht
  - cons verhindert also inkorrekte Benutzung
    - In anderen Scheme-Versionen fehlt dieser Check allerdings
- Eine bessere Emulation sähe wie folgt aus:

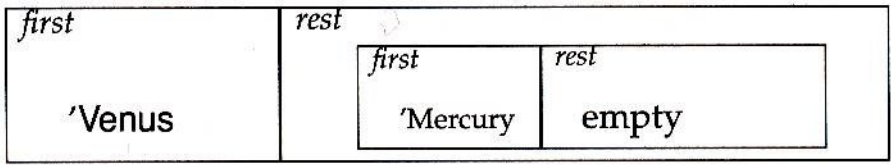
```
(define-struct lst (first rest))
(define-struct emptylst ())
(define the-emptylst (make-emptylst))

(define (our-cons a-value a-list)
  (cond
    [(emptylst? a-list) (make-lst a-value a-list)]
    [(lst? a-list) (make-lst a-value a-list)]
    [else (error 'our-cons "list as second argument expected")]))
```

- Kann allerdings nicht verhindern, dass man **`make-lst`** direkt verwendet

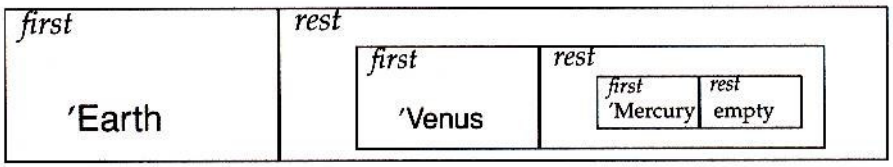
Listen

(cons 'Mercury empty)

(cons 'Venus
  (cons 'Mercury empty))



(cons 'Earth
  (cons 'Venus
    (cons 'Mercury empty)))



Listen

- Listen können jeden beliebigen Datentyp speichern, auch gemischte Daten

```
(cons 0
  (cons 1
    (cons 2
      (cons 3
        (cons 4
          (cons 5
            (cons 6
              (cons 7
                (cons 8
                  (cons 9 empty)))))))))
```

```
(cons 'RobbyRound
  (cons 3
    (cons true
      empty)))
```

Die Abgeschlossenheitseigenschaft

- Eine Operation zum Kombinieren von Daten besitzt die Abgeschlossenheitseigenschaft, wenn die Ergebnisse der Kombination von Datenobjekten wiederum mit der Operation kombiniert werden können
  - Beispiel: **`cons`**

- Solche Kombinationsoperatoren können verwendet werden, um hierarchische Daten aufzubauen.

Die Abgeschlossenheitseigenschaft

Frage:

Sind wir der Abgeschlossenheitseigenschaft bereits begegnet?

Die Abgeschlossenheitseigenschaft

- Der Ursprung des Begriffs „Abgeschlossenheit“ (engl. closure)
  - Abstrakte Algebra: Eine Menge von Elementen ist abgeschlossen bezüglich einer Operation, wenn die Anwendung der Operation an Elementen der Menge wieder ein Element der Menge produziert

Die Abgeschlossenheitseigenschaft

- Der Gedanke, dass ein Mittel der Kombination die Abgeschlossenheitseigenschaft besitzen soll, ist intuitiv
- Leider erfüllen Kombinationsoperatoren vieler Sprachen diese nicht
  - So kann man Elemente kombinieren, indem man diese in Arrays speichert
  - Man kann aber u.U. keine Arrays aus Arrays zusammenstellen bzw. als Rückgabewerte von Prozeduren verwenden
  - Es fehlt ein eingebauter „Universalkleber“, der es einfacher macht, zusammengesetzte Daten in einer uniformen Art und Weise zu manipulieren.

Der `list` Konstruktor

- Längere Listen mit **`cons`** zu erzeugen ist unhandlich

- Daher gibt es in Scheme den Konstruktor **`list`**
  - Er bekommt eine beliebige Menge von Argumenten
  - Er erzeugt eine Liste mit allen Argumenten als Elementen
  - z.B. **`(list 1 2 3)`** statt
    `(cons 1 (cons 2 (cons 3 empty)))`
  - z.B. **`(list (list 'a 1) (list 'b 2))`**
- Allgemein ist
  `(list exp-1 … exp-n)` äquivalent zu
  `(cons exp-1 (cons … (cons exp-n empty ) … ))`

Der ‘ / `quote` Konstruktor

- Listen und Symbole können mit Hilfe des Quote-Konstruktors ‘ noch weiter abgekürzt werden
  - `'(1 2 3)` ist die Kurzform für `(list 1 2 3)`
  - `'((1 2) (3 4) (5 6))` steht für `(list (list 1 2) (list 3 4) (list 5 6))`
  - `'(a b c)` steht für `(list 'a 'b 'c)`
    - nicht zu verwechseln mit `(list a b c) - list` wertet alle Argumente aus, quote nicht
  - Im Allgemeinen bedeutet `'(`*`exp-1 … exp-n`*`)` `(list '`*`exp-1 … 'exp-n`*`)`, wobei `'`*`exp-i = exp-i`* für alle selbstauswertenden Werte (Zahlen, Booleans)
    - Beachten Sie, dass diese Regel rekursiv ist!
- Um `list` und `quote` zu verwenden,
  - benutzen Sie bitte ab jetzt das “Beginning Student with List Abbreviations” Level in DrScheme!

Verarbeitung von rekursiven Datentypen

- Wie verarbeiten wir Exemplare rekursiver Datentypen?
  - Antwort: Wir benutzen unser Designrezept für heterogene Daten
  - Beispiel: Ist ein bestimmtes Symbol in einer Liste von Symbolen enthalten?
- 1. Schritt: Definition des Vertrags, Header etc.

```
;; contains-doll? : list-of-symbols  ->  boolean
;; to determine whether the symbol 'doll occurs
;; in a-list-of-symbols
(define (contains-doll? a-list-of-symbols) ...)
```

  - Beachten Sie die Konvention **`list-of-XXX`**, um im Vertrag zu dokumentieren, was für Daten als Listenelemente erwartet werden

Verarbeitung von rekursiven Datentypen

- Template erstellen
  - Für jeden Datentyp ein cond-Zweig, Selektoren andeuten

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) ...]
    [(cons? a-list-of-symbols)
       ... (first a-list-of-symbols) ...
       ... (rest a-list-of-symbols) ...]))
```

- Erster Fall (leere Liste) ist trivial

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) false]
    [(cons? a-list-of-symbols)
       ... (first a-list-of-symbols) ...
       ... (rest a-list-of-symbols) ...]))
```

Verarbeitung von rekursiven Datentypen

- Mit den verfügbaren Daten können wir direkt das erste Element der Liste überprüfen

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) false]
    [(cons? a-list-of-symbols)
       (cond
         [(symbol=? (first a-list-of-symbols) 'doll) true]
          ... (rest a-list-of-symbols) ...]))
```

- Was machen wir mit dem Rest der Liste?
  - Wir brauchen eine Hilfs-Prozedur, die überprüft, ob die (Rest)-Liste das Symbol enthält
  - Diese Hilfsprozedur ist **`contains-doll?`** selbst!

Verarbeitung von rekursiven Datentypen

- Lösung:

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) false]
    [(cons? a-list-of-symbols)
       (cond
         [(symbol=? (first a-list-of-symbols) 'doll) true]
         [else (contains-doll? (rest a-list-of-symbols))])
    ]))
```

Verarbeitung von rekursiven Datentypen

- Wieso ist die Lösung wohldefiniert?
  - Wohldefiniert heißt hier: die Auswertung der Prozedur terminiert
- Nicht jede rekursive Definition ist wohldefiniert!
  - z.B. `(define (f a-bool) (f (not a-bool)))`
- Unsere rekursive Definition ist ein Beispiel für **strukturelle Rekursion**
  - Die Struktur der Prozedur folgt der (rekursiven) Struktur der Daten
  - Solche rekursiven Definitionen sind stets wohldefiniert, weil die Schachtelungstiefe der Daten in jedem rekursiven Aufruf strikt abnimmt

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) false]
    [(cons? a-list-of-symbols)
       (cond
         [(symbol=? (first a-list-of-symbols) 'doll) true]
         [else (contains-doll? (rest a-list-of-symbols))])
    ]))
```

Design von Prozeduren für rekursive Daten

- Wie ändert sich unser Designrezept?
  - Datenanalyse und Design
    - Wenn die Problembeschreibung Informationen beliebiger Größe beschreibt, benötigen wir eine rekursive Datenstruktur
    - Diese Datenstruktur benötigt mindestens zwei Fälle, von denen mindestens einer nicht (direkt oder indirekt) auf die Definition zurückverweist
  - Vertrag und Prozedurkopf: keine Änderung
  - Beispiele: keine Änderung
  - Template: In den rekursiven Zweigen den rekursiven Aufruf andeuten
  - Prozedurkörper
    - Erst die Basisfälle (nicht-rekursiv) implementieren, dann die rekursiven Zweige. Für die rekursiven Aufrufe davon ausgehen, dass die Prozedur bereits wie gewünscht funktioniert
  - Test: keine Änderung

Erzeugen von rekursiven Daten

- Prozeduren, die rekursive Daten nach unserem Designrezept verarbeiten, kombinieren üblicherweise die Lösung aus den rekursiven Aufrufen mit den nicht-rekursiven Daten.

```
(define (sum a-list-of-nums)
  (cond
    [(empty? a-list-of-nums) 0]
    [else (+
            (first a-list-of-nums)
            (sum (rest a-list-of-nums)))]))
```

- Diese Kombination besteht häufig in der Konstruktion neuer Daten, deren Struktur analog zu denen der Eingabe ist.

Erzeugen von rekursiven Daten

- Berechnung des Arbeitslohns (wage)
  - Für eine Person oder mehrere Personen

```
;; wage : number  ->  number
;; to compute the total wage (at $12 per hour)
;; of someone who worked for h hours
(define (wage h)
  (* 12 h))

;; hours->wages : list-of-numbers  ->  list-of-numbers
;; to create a list of weekly wages from
;; a list of weekly hours (alon)
(define (hours->wages alon)
  (cond
    [(empty? alon) empty]
    [else (cons
            (wage (first alon))
            (hours->wages (rest alon)))]))
```

Strukturen, die Strukturen enthalten

- Strukturen (und Listen) müssen nicht notwendigerweise atomare Daten enthalten
  - Sie können z.B. Exemplare ihrer eigenen Struktur als Elemente enthalten (wie bei der Listenstruktur)
  - Sie können beliebige andere Strukturen/Listen enthalten
    - z.B. Liste von Punkten, Liste von Listen von Punkten

- Beispiel:
  - Ein Inventareintrag ist eine Struktur **`(make-ir s n)`**, wobei **`s`** ein Symbol ist und **`n`** eine (positive) Zahl:
    `(define-struct ir (name price))`
  - Eine *Inventarliste* ist entweder
    1. **`empty`** (leer), oder
    2. **`(cons ir inv)`** wobei **`ir`** ein Inventareintrag ist und **`inv`** ein *Inventarliste*

Die Struktur natürlicher Zahlen

- Natürliche Zahlen werden häufig als Aufzählungen eingeführt:
  - 0,1,2, etc.
  - “etc.” ist für Programmierung nicht besonders hilfreich
- Mit Hilfe einer rekursiven Definition kann man das “etc.” los werden:
  - 0 ist eine natürliche Zahl
  - Falls n eine natürliche Zahl ist, dann auch der Nachfolger (succ n)
  - Diese Definition generiert: 0, (succ 0), (succ (succ 0)) etc.
  - (Vgl. Peano-Axiome in der Mathematik)
- Diese Konstruktion ist analog zu der Konstruktion von Listen
  - cons entspricht succ
  - rest entspricht pred
  - empty? entspricht zero?
- Funktionen auf natürlichen Zahlen können daher analog zu Funktionen auf Listen strukturiert werden!

Die Struktur natürlicher Zahlen

- Hier die Definition der Prozeduren *pred*, *succ* und *zero?*:

```
;; pred: N  ->  N
;; returns the predecessor of n; if n is 0, return 0
(define (pred n)
  (if (> n 0)
     (- n 1)
     0))

;; succ: N -> N
;; returns the successor of n; if negative, return 0
(define (succ n)
  (if (>= n 0)
     (+ n 1)
     0))

;; zero?: N -> boolean
;; returns true if N is 0, else false
;; Already defined in Scheme, no need to do it again!
```

Die Struktur natürlicher Zahlen

- Beispiel 1

```
;; hellos : N  ->  list-of-symbols
;; to create a list of n copies of 'hello
(define (hellos n)
  (cond
    [(zero? n) empty]
    [else (cons 'hello (hellos (pred n)))]))
```

- Beispiel 2

```
;; ! : N  ->  N
;; computes the faculty function
(define (! n)
  (cond
    [(zero? n) 1]
    [else (* n (! (pred n)))]))
```

Intermezzo

Nochmal: Design von Hilfsprozeduren

Design von Hilfsprozeduren: Wunschlisten

- Größere Programme bestehen aus vielen verschiedenen (Hilfs-)Prozeduren
- Sinnvolles Vorgehen: Anlegen und Pflegen einer **Wunschliste** von Hilfsprozeduren
  - Eine Liste der Prozeduren, die noch entwickelt werden müssen, um ein Programm zu beenden
- Die Prozeduren auf der Wunschliste können wir dann schrittweise mit Hilfe der Designrezepte implementieren
- Diese Wunschliste ändert sich häufig im Laufe der Implementierung
  - Etwa weil man entdeckt, dass bestimmte neue Hilfsprozeduren erforderlich sind

Design von Hilfsprozeduren: Wunschlisten

- Die Reihenfolge der Abarbeitung der Wunschliste ist wichtig
  - "Bottom-Up"-Ansatz: Erst die Prozeduren implementieren, die von keiner anderen Prozedur auf der Wunschliste abhängen
    - Vorteil: Wir können sofort und jederzeit testen
    - Nachteil: Am Anfang ist oft noch nicht klar, welche Prozeduren auf der untersten Ebene benötigt werden; es kann der Gesamtüberblick verloren gehen
  - "Top-Down"-Ansatz: Erst die Hauptprozedur implementieren, dann die dort aufgerufenen Funktionen usw.
    - Vorteil: Inkrementelle Verfeinerung der Programmstruktur
    - Nachteil: Man kann erst sehr spät testen; manchmal stellt man erst "unten" fest, dass man "oben" einen konzeptuellen Fehler gemacht hat
  - Oft ist eine Mischung aus Top-Down und Bottom-Up sinnvoll

Design von Hilfsprozeduren: Wunschlisten

- Beispiel: Sortieren einer Liste von Zahlen
  - Wir folgen erst einmal unserem Designrezept…

```
;; sort : list-of-numbers  ->  list-of-numbers
;; to create a sorted list of numbers from all
;; the numbers in alon

;; Examples:
;; (sort empty) → empty
;; (sort (cons 1297.04 (cons 20000.00 (cons -505.25 empty))))
;; → (cons -505.25 (cons 1297.04 (cons 20000.00 empty)))

(define (sort alon)
  (cond
    [(empty? alon) ...]
    [else ... (first alon) ... (sort (rest alon)) ...]))
```

Design von Hilfsprozeduren: Wunschlisten

- Der erste Fall (leere Liste) ist trivial
- Der Aufruf `(sort (rest alon))` im rekursiven Fall gibt uns eine sortierte Liste zurück
- In diese Liste muss **(first alon)** einsortiert werden
- Die Prozedur, die dies erledigt, kommt auf unsere Wunschliste, die `sort`-Prozedur können wir schon mal komplettieren

```
;; insert : number list-of-numbers  ->  list-of-numbers
;; to create a list of numbers from n and the numbers
;; in alon that is sorted in descending order; alon is
;; already sorted
(define (insert n alon) ...)

(define (sort alon)
  (cond
    [(empty? alon) empty]
    [else (insert (first alon) (sort (rest alon)))]))
```

Design von Hilfsprozeduren: Wunschlisten

- Die **`insert`**-Prozedur kann nun unabhängig von sort implementiert werden
  - Hier wichtig: genaue Analyse der unterschiedlichen Fälle

```
;; insert : number list-of-numbers (sorted)
;;                -> list-of-numbers (sorted)
;; to create a list of numbers from n and the numbers in
;; alon that is sorted in ascending order; alon is sorted

(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [else (cond
            [(<= n (first alon))
             (cons n alon)]
            [(>  n (first alon))
             (cons
              (first alon)
              (insert n (rest alon)))])]))
```

  - Diese Art der Sortierung ist bekannt als Insertion-Sort

Design von Hilfsprozeduren: Verallgemeinerung von Problemen

- Häufig ist ein allgemeineres Problem als das, was man eigentlich lösen möchte, **einfacher zu lösen** als das eigentliche Problem
- Vorgehen: Hilfsprozedur löst allgemeineres Problem, Hauptprozedur spezialisiert die Hilfsprozedur für eigentliches Problem
- Beispiel: Münzwechsel
  - Auf wie viele Arten kann man einen Euro-Betrag **`a`** wechseln (unter Verwendung von Münzen à 1,2,5,10,20,50 Cent)?
  - Verallgemeinerung: Auf wieviele Arten kann man einen Euro-Betrag **`a`** wechseln unter Verwendung der ersten **`n`** Münzarten aus 1,2,5,10,20,50 Cent?
  - Spezialisierung: n = 6

Design von Hilfsprozeduren: Verallgemeinerung von Problemen

- Wechselmöglichkeiten zählen: Einfache Formulierung des allgemeineren Problems als rekursive Prozedur:
  - Die Anzahl der Möglichkeiten, den Betrag **a** unter Verwendung von **n** Sorten von Münzen ist:
    - 1 falls **a** = 0
    - 0 falls **a**<0 oder **n** = 0
    - ansonsten:
      - Die Anzahl von Möglichkeiten **a** zu wechseln unter Verwendung aller außer der letzten Münzsorte, **plus**
      - Die Anzahl der Möglichkeiten **a**-**d** zu wechseln unter Verwendung aller **n** Münzsorten, wobei **d** der Nennwert der letzten Münzsorte ist.

Design von Hilfsprozeduren: Verallgemeinerung von Problemen

```
(define (count-change amount)
  (cc amount 6))

(define (cc amount kinds-of-coins)
  (cond [(= amount 0) 1]
        [(or (< amount 0) (= kinds-of-coins 0)) 0]
        [else
           (+
              (cc amount (- kinds-of-coins 1))
              (cc
                 (- amount (denomination kinds-of-coins))
                    kinds-of-coins))]))

(define (denomination coin-number)
  (cond ((= coin-number 1) 1)
        ((= coin-number 2) 2)
        ((= coin-number 3) 5)
        ((= coin-number 4) 10)
        ((= coin-number 5) 20)
        ((= coin-number 6) 50)))
```

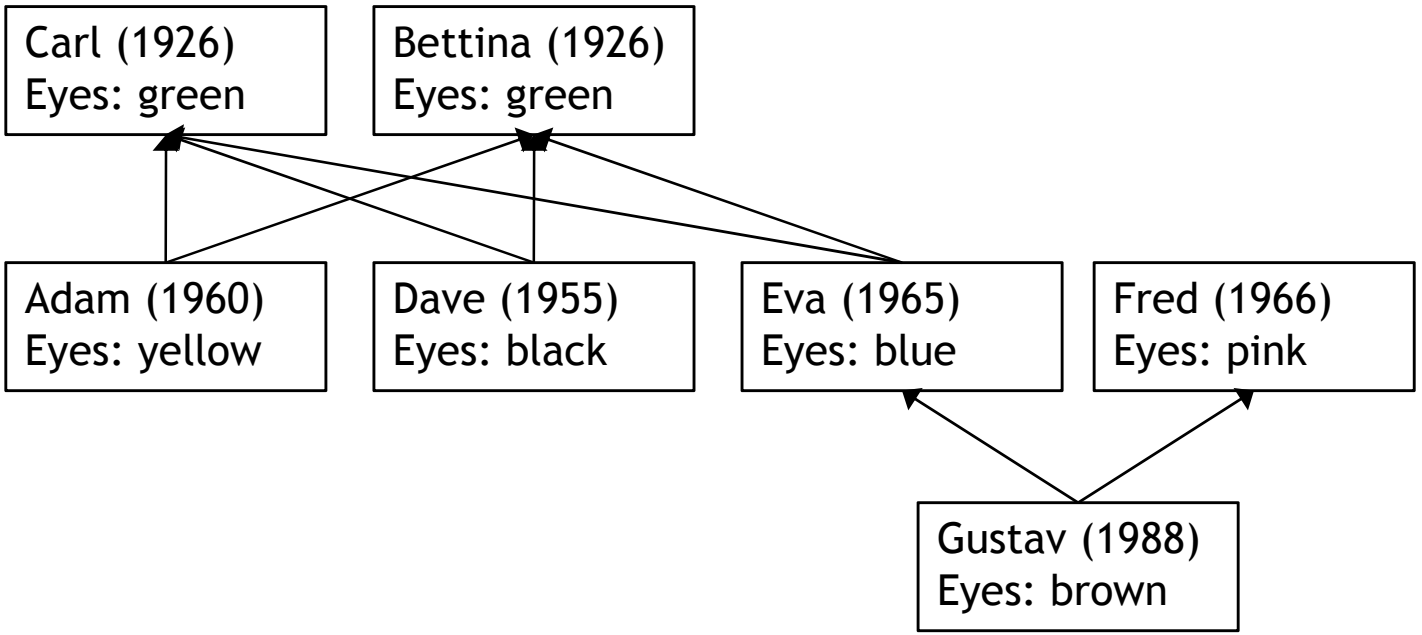
Design von Hilfsprozeduren: Verallgemeinerung von Problemen

- Kommentare zur count-change Prozedur:
  - Die Art der Rekursion bei dieser Lösung ist eine andere Technik als die strukturelle Rekursion, die sie bisher kennengelernt haben
    - Später mehr dazu
  - Die Prozedur ist sehr rechenintensiv
    - Bei Beträgen >100 kann es sehr lange dauern
    - Die benötigte Zeit wächst exponentiell mit der Größe des Betrags
    - Auch dazu später mehr

Bäume

Bäume

- Bäume sind eine der wichtigsten Datenstrukturen in der Informatik
  - Repräsentation von hierarchischen Beziehungen
- Beispiel: Stammbäume



Bäume

- Datenmodellierung, 1. Versuch:

> Ein *Kind* ist eine Struktur
> **`(make-child father mother name year eyes)`**
> wobei **`father`** und **`mother`** *Kind*-Strukturen sind; **`name`** und **`eyes`** sind Symbole, **`year`** ist eine Zahl

- Problem: Es ist unmöglich, ein Exemplar dieser Datenstruktur zu erzeugen
  - Es fehlt der Basisfall (Ausstieg aus der Rekursion)
  - Wir haben uns nicht an unsere Regel für das Design rekursiver Datenstrukturen gehalten
    - Rekursive Datenstrukturen müssen mindestens zwei Alternativen haben, von denen mindestens ein Fall nicht rekursiv ist.

Bäume

- Datenmodellierung, 2. Versuch:

Ein *Stammbaumknoten* (family tree node *ftn*) ist entweder

1. empty, oder
2. `(make-child father mother name year eyes)`

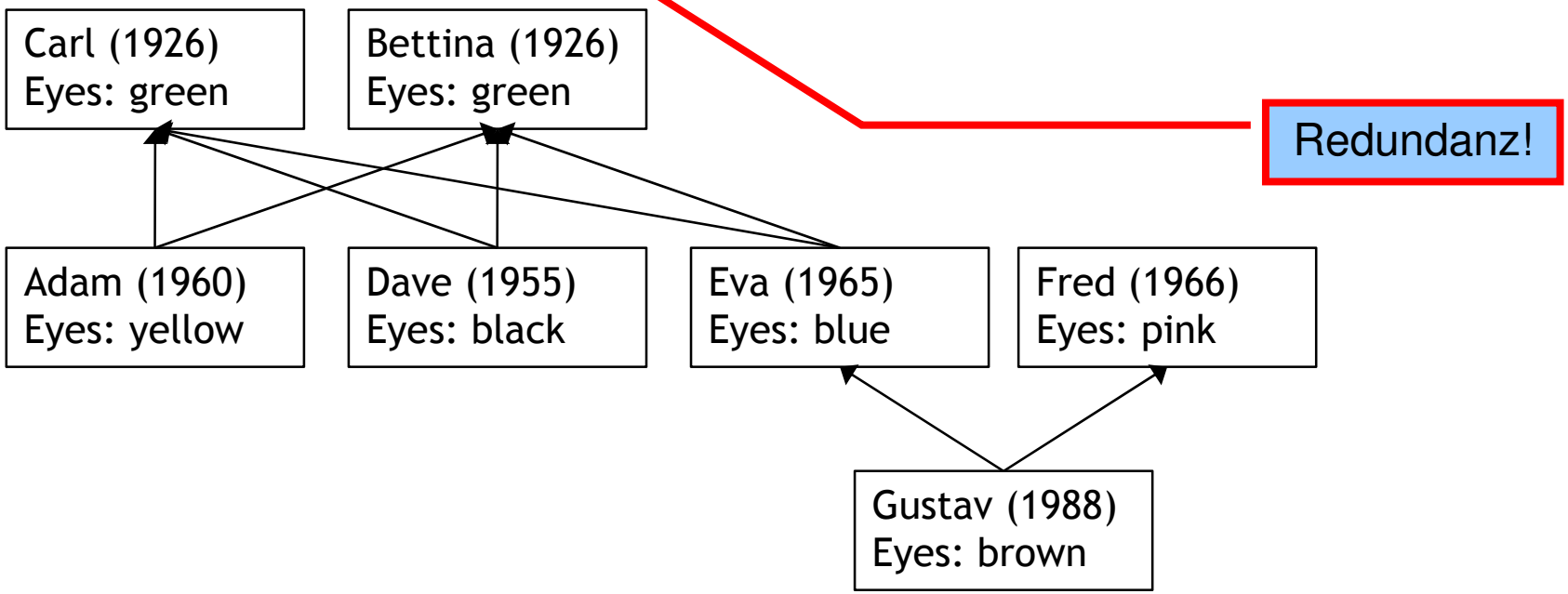
wobei **`father`** und **`mother`** *ftn* sind; **`name`** und **`eyes`** sind Symbole, **`year`** ist eine Zahl

- Konform zu unserer Design-Regel, Stammbaum kann damit repräsentiert werden

Bäume

- Carl: `(make-child empty empty 'Carl 1926 'green)`
- Adam:

```
(make-child
   (make-child empty empty 'Carl 1926 'green)
   (make-child empty empty 'Bettina 1926 'green)
  'Adam 1950 yellow)
```



Bäume

- Vermeidung der Redundanz durch Binden an Namen

```
;; Oldest Generation:
(define Carl (make-child empty empty 'Carl 1926 'green))
(define Bettina (make-child empty empty 'Bettina 1926
'green))

;; Middle Generation:
(define Adam (make-child Carl Bettina 'Adam 1950 'yellow))
(define Dave (make-child Carl Bettina 'Dave 1955 'black))
(define Eva (make-child Carl Bettina 'Eva 1965 'blue))
(define Fred (make-child empty empty 'Fred 1966 'pink))

;; Youngest Generation:
(define Gustav (make-child Fred Eva 'Gustav 1988 'brown))
```

Bäume

- Design von Funktionen auf Bäumen
  - Das Vorgehen bleibt das Gleiche, führt nun aber zu mehreren rekursiven Aufrufen
- Allgemeines Template

```
;; fun-for-ftn : ftn  ->  ???
(define (fun-for-ftn a-ftree)
  (cond
    [(empty? a-ftree) ...]
    [else
      ... (fun-for-ftn (child-father a-ftree)) ...
      ... (fun-for-ftn (child-mother a-ftree)) ...
      ... (child-name a-ftree) ...
      ... (child-year a-ftree) ...
      ... (child-eyes a-ftree) ...]))
```

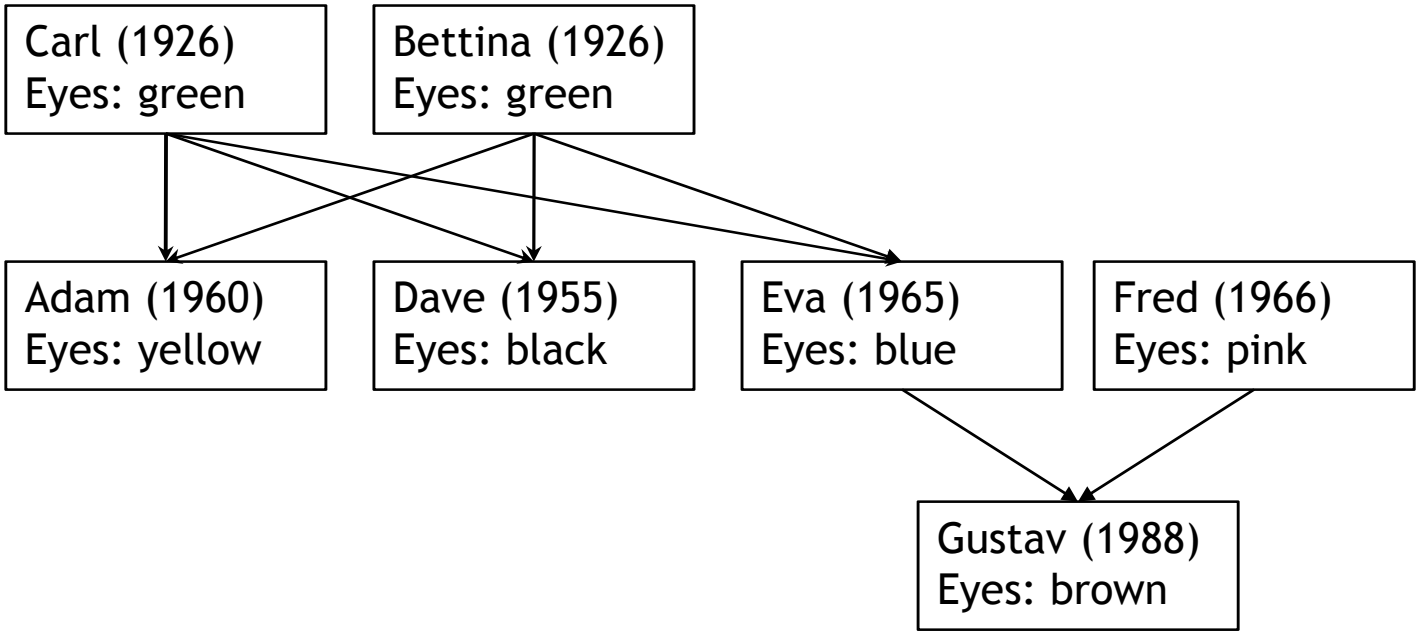
Bäume

- Beispiel: blue-eyed-ancestor?

```
;; blue-eyed-ancestor? : ftn  ->  boolean
;; to determine whether a-ftree contains a
;; structure with 'blue in the eyes field

(define (blue-eyed-ancestor? a-ftree)
  (cond
    [(empty? a-ftree) false]
    [else (or (symbol=? (child-eyes a-ftree) 'blue)
              (or (blue-eyed-ancestor? (child-father a-ftree))
                  (blue-eyed-ancestor? (child-mother a-ftree))))]))
```

Wechselseitig rekursive Strukturen

- Das Beispiel umgedreht: Kinder statt Vorfahren



Wichtiger Unterschied für die Modellierung: Eine Person kann beliebig viele Kinder haben, aber nur zwei Vorfahren (Eltern)
→ Wir benötigen eine Liste zur Speicherung der Kinder

Wechselseitig rekursive Strukturen

- Wir benötigen zwei separate Datenstrukturen:
  - Ein ***parent*** ist eine Struktur **`(make-parent children name date eyes)`**, wobei **`loc`** eine ***Liste von Kindern***, **`name`** und **`eyes`** Symbole, und **`year`** eine Zahl ist.
  - Eine ***Liste von Kindern*** ist entweder
    1. **`empty`**, oder
    2. **`(cons p children)`**, wobei **`p`** ein ***parent*** ist und **`children`** eine ***Liste von Kindern***
- Eine dieser Definitionen alleine macht keinen Sinn, da sie zu einer anderen Datenstruktur verweist, die noch nicht definiert ist
  - Man sagt, die Datenstrukturen sind **wechselseitig rekursiv**
  - Wechselseitig rekursive Datenstrukturen sollten immer zusammen definiert werden

Wechselseitig rekursive Strukturen

- Repräsentation des Stammbaums

```
;; Youngest Generation:
(define Gustav (make-parent empty 'Gustav 1988 'brown))

(define Fred&Eva (list Gustav))

;; Middle Generation:
(define Adam (make-parent empty 'Adam 1950 'yellow))
(define Dave (make-parent empty 'Dave 1955 'black))
(define Eva (make-parent Fred&Eva 'Eva 1965 'blue))
(define Fred (make-parent Fred&Eva 'Fred 1966 'pink))

(define Carl&Bettina (list Adam Dave Eva))

;; Oldest Generation:
(define Carl (make-parent Carl&Bettina 'Carl 1926 'green))
(define Bettina (make-parent Carl&Bettina 'Bettina 1926
'green))
```

Wechselseitig rekursive Strukturen

- Wie erstellt man die Prozedur **`blue-eyed-descendant?`**
- Vertrag, Beschreibung, Header, Beispiele:

```
;; blue-eyed-descendant? : parent  ->  boolean
;; to determine whether a-parent or any of its descendants
;; (children, grandchildren, and so on) have 'blue
;; in the eyes field
(define (blue-eyed-descendant? a-parent) ...)

;; Here are three simple examples, formulated as tests:

(boolean=? (blue-eyed-descendant? Gustav) false)
(boolean=? (blue-eyed-descendant? Eva) true)
(boolean=? (blue-eyed-descendant? Bettina) true)
```

Wechselseitig rekursive Strukturen

- Template:

```
(define (blue-eyed-descendant? a-parent)
  ... (parent-children a-parent) ...
  ... (parent-name a-parent) ...
  ... (parent-date a-parent) ...
  ... (parent-eyes a-parent) ... )
```

- Aktuelle Person kann direkt überprüft werden

```
(define (blue-eyed-descendant? a-parent)
  (cond
    [(symbol=? (parent-eyes a-parent) 'blue) true]
    [else
      ... (parent-children a-parent) ...
      ... (parent-name a-parent) ...
      ... (parent-date a-parent) ...]))
```

Name und Geburtsdatum sind für diese Prozedur unwichtig

Wechselseitig rekursive Strukturen

- Wir benötigen eine Prozedur, die eine Liste von Kindern auf blauäugige Nachfolger untersucht
- Unserer Richtlinie für komplexe Prozeduren folgend, fügen wir eine solche Prozedur unserer Wunschliste hinzu:

```
;; blue-eyed-children? : list-of-children  ->  boolean
;; to determine whether any of the structures
;; in aloc is blue-eyed or has any blue-eyed descendant
(define (blue-eyed-children? aloc) ...)
```

- Damit können wir **`blue-eyed-descendant?`** fertigstellen:

```
(define (blue-eyed-descendant? a-parent)
 (cond
  [(symbol=? (parent-eyes a-parent) 'blue) true]
  [else (blue-eyed-children? (parent-children a-parent))]))
```

Wechselseitig rekursive Strukturen

- Abarbeiten der Wunschliste:
  - Erstellen von **`blue-eyed-children?`**
- Standard Template für Listen:

```
(define (blue-eyed-children? aloc)
  (cond
    [(empty? aloc) ...]
    [else
      ... (first aloc) ...
      ... (blue-eyed-children? (rest aloc)) ...]))
```

- Liste ist empty → trivial
- Resultat des rekursiven Aufrufs muss nur mit “or” verknüpft werden, um zu prüfen, ob das erste Element der Liste einen blauäugigen Nachfahren hat
- Wir brauchen also eine Prozedur, die zu einer parent Struktur berechnet, ob ein Nachfahre blaue Augen hat
  - Diese Prozedur ist **`blue-eyed-descendant?`**

Wechselseitig rekursive Strukturen

- Finale Lösung

```
;; blue-eyed-children? : list-of-children  ->  boolean
;; to determine whether any of the structures in aloc
;; is blue-eyed or has any blue-eyed descendant

(define (blue-eyed-children? aloc)
  (cond
    [(empty? aloc) false]
    [else (or (blue-eyed-descendant? (first aloc))
              (blue-eyed-children? (rest aloc)))]))
```

Wechselseitig rekursive Strukturen

- Lektion:

> ***Prozeduren auf wechselseitig rekursiven Datenstrukturen sind selbst wechselseitig rekursiv!***

  - Dies liegt daran, dass die Struktur der Prozeduren immer der Struktur der Daten folgen sollte!
  - Wechselseitig rekursive Daten sollten immer als Einheit betrachtet werden, da sie nur zusammen Sinn machen
  - Prozeduren auf wechselseitig rekursiven Daten mit dem Wunschlisten-Ansatz implementieren
    - Strenges bottom-up oder top-down Vorgehen ist aufgrund der Abhängigkeiten nicht möglich!

Intermezzo

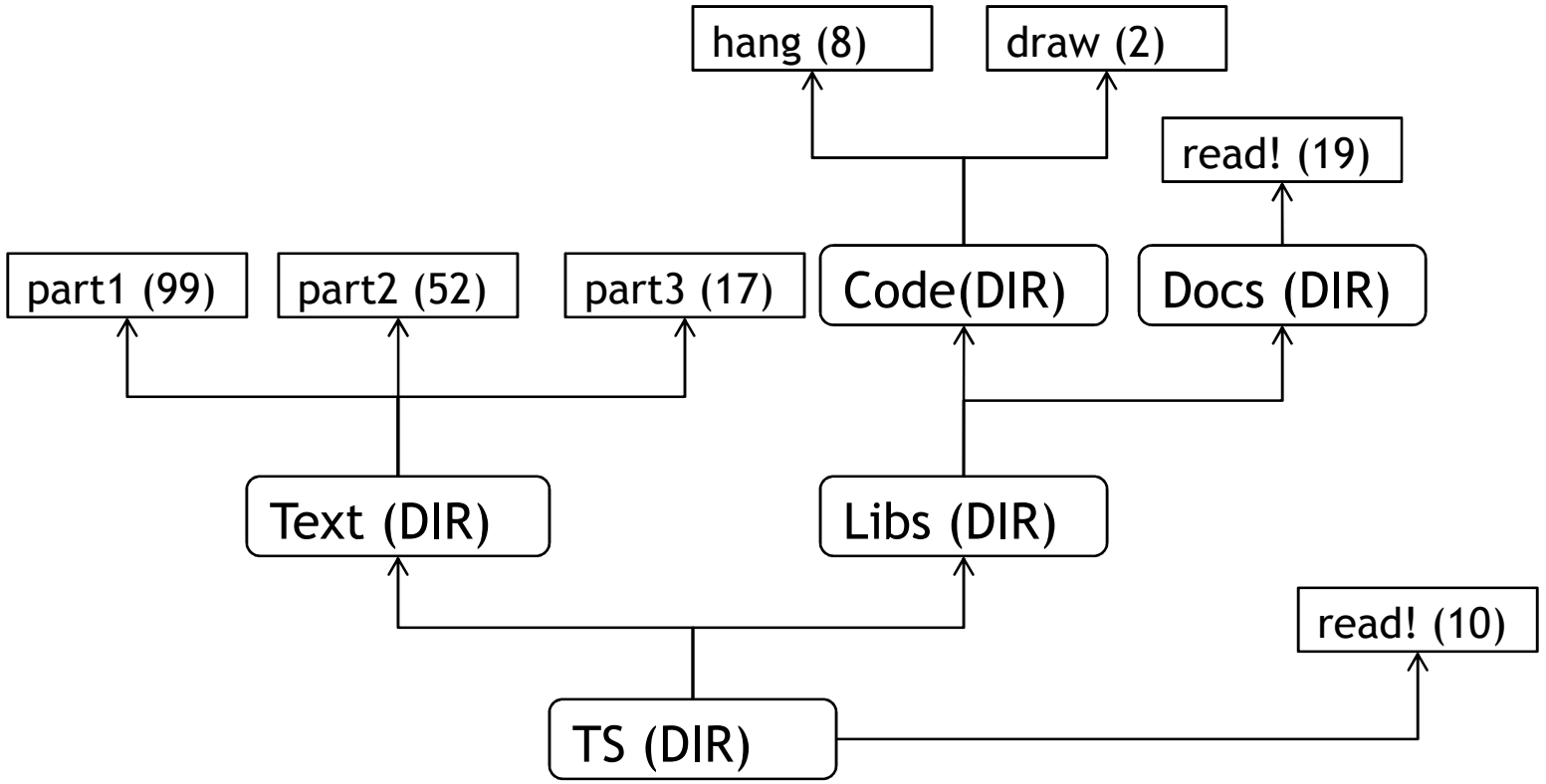
Iterative Verfeinerung von Programmen

Iterative Verfeinerung von Programmen

- Beim Programmieren erstellen wir ein Modell eines kleinen Teils der realen Welt
  - Welche Aspekte/Informationen in das Modell aufgenommen werden sollten, ist am Anfang häufig nicht klar
- Aus diesem Grund sollte man iterativ vorgehen, also das Programm und den Datenentwurf schrittweise verfeinern
  - Starten mit (sehr) einfachem Modell
  - Nach und nach präzisieren / verfeinern
- Unser Beispiel: Iterative Verfeinerung eines Datenmodells für Dateisysteme

Iterative Verfeinerung von Programmen

- Beispiel für Dateisystemstruktur



Iterative Verfeinerung von Programmen

- Modell 1:
  - Dateien werden durch ihren Dateinamen (Symbol) repräsentiert;
  - Ein Verzeichnis ist eine Liste bestehend aus Dateien und Verzeichnissen
- Datenmodell:

Ein ***file*** ist ein Symbol

Ein ***directory*** (kurz: ***dir***) ist entweder
`empty`, oder
`(cons f d)` wobei **`f`** ein ***file*** und **`d`** ein ***directory*** ist; oder
`(cons d1 d2)` wobei **`d1`** und **`d2`** ***dirs*** sind.

Iterative Verfeinerung von Programmen

- Modell 2:
  - Ein Verzeichnis hat einen Namen und andere Attribute, die wir modellieren möchten
- Verfeinertes Datenmodell:
  - `(define-struct dir (name content)`

Ein ***directory*** (kurz: ***dir***) ist eine Struktur `(make-dir n c)`, wobei `n` ein Symbol und `c` eine ***Liste von Files und Directories*** ist

Eine ***Liste von Files und Directories (LOFD)*** ist entweder
1. `empty`
2. `(cons f d)` wobei `f` ein File und `d` ein ***LOFD*** ist
3. `(cons d1 d2)` wobei `d1` ein ***dir*** und `d2` ein ***LOFD*** ist

  - Die beiden Datenstrukturen sind wechselseitig rekursiv!

Iterative Verfeinerung von Programmen

- Modell 3:
  - Wir möchten auch die Attribute von Dateien (Größe, Inhalt) modellieren
  - Wir wollen zwischen Dateien und Verzeichnissen unterscheiden
- Verfeinertes Datenmodell:
  - `(define-struct file (name size content))`

Ein ***file*** ist eine Struktur `(make-file name size content)`, wobei `name` ein Symbol, `size` eine Zahl und `content` ein String ist

Eine ***Liste von Files (LOF)*** ist entweder
1. `empty`
2. `(cons f lof)` wobei `f` ein File und `lof` eine ***LOF*** ist

Iterative Verfeinerung von Programmen

- Modell 3:…

- Verfeinertes Datenmodell, Fortsetzung:
  - `(define-struct dir (name dirs files))`

Ein ***directory*** (kurz: ***dir***) ist eine Struktur `(make-dir name dirs files)`, wobei `name` ein Symbol, `dirs` eine ***LOD*** und `files` eine ***LOF*** ist

Eine ***Liste von Directories (LOD)*** ist entweder
1. `empty`
2. `(cons d lod)` wobei `d` ein ***dir*** und `lod` eine ***LOD*** ist

Prozeduren mit mehreren komplexen Argumenten

Prozeduren mit mehreren komplexen Argumenten

- Bis jetzt haben wir nur Prozeduren gesehen, die ein komplexes Argument bekommen
  - Komplex bedeutet: Der Datentyp des Arguments ist rekursiv definiert
- Bei zwei oder mehr komplexen Argumenten gibt es im wesentlichen drei Fälle:
  1. Eines der Argumente kann für den Zweck der Prozedur als atomar angesehen werden
  2. Alle Argumente werden synchron zerlegt
  3. Es müssen alle möglichen Fälle der Eingabeargumente getrennt betrachtet werden
- Wir werden nun jeweils ein Beispiel für jeden dieser Fälle betrachten

Prozeduren mit mehreren komplexen Argumenten

- 1. Beispiel: ein Argument wird als atomar behandelt
  - Rekursiv nur im ersten Argument

```
;; append : list list  ->  list
;; to construct a new list by replacing empty
;; in alon1 with alon2
(define (append alon1 alon2)
  (cond
    ((empty? alon1) alon2)
    (else (cons (first alon1)
                (append (rest alon1) alon2)))))
```

Prozeduren mit mehreren komplexen Argumenten

- 2. Beispiel: alle Argumente werden synchron zerlegt

```
;; addlist :
;; list-of-numbers list-of-numbers -> list-of-numbers
;;
;; to construct a new list of numbers where the i-th number
;; is the sum of the i-th number of alon1 and the i-th
;; number of alon2
;;
;; Example: (addlist (list 1 2) (list 7 8 9)) → (list 8 10)

(define (addlist alon1 alon2)
  (cond
    [(or (empty? alon1) (empty? alon2)) empty]
    [else (cons
            (+ (first alon1) (first alon2))
            (addlist (rest alon1) (rest alon2)))]))
```

Prozeduren mit mehreren komplexen Argumenten

- 3. Beispiel: alle Fälle müssen einzeln betrachtet werden

```
;; list-pick : list-of-symbols N[>= 1]  ->  symbol
;; to determine the nth symbol from alos, counting from 1;
;; signals an error if there is no n-th item
(define (list-pick alos n) ...)
```

- Beispiele:

```
(list-pick empty 1)
;; expected behavior:
(error 'list-pick "...")

(list-pick (cons 'a empty) 1)
;; expected value:
'a
```

```
(list-pick empty 3)
;; expected behavior:
(error 'list-pick "...")

(list-pick (cons 'a empty) 3)
;; expected behavior:
(error 'list-pick "...")
```

Prozeduren mit mehreren komplexen Argumenten

- 3. Beispiel: alle Fälle müssen einzeln betrachtet werden
- Offensichtlich gibt es vier interessante Fälle

	`(empty? alos)`	`(cons? alos)`
`(= n 1)`	`(and (= n 1)` `  (empty? alos))`	`(and (= n 1)` `  (cons? alos))`
`(> n 1)`	`(and (> n 1)` `  (empty? alos))`	`(and (> n 1)` `  (cons? alos))`

Prozeduren mit mehreren komplexen Argumenten

- Übertragen der unterschiedlichen Fälle auf das Prozedurdesign

```
define (list-pick alos n)
  (cond
    [(and (= n 1) (empty? alos))
     ...]
    [(and (> n 1) (empty? alos))
     ... (pred n) ...]
    [(and (= n 1) (cons? alos))
     ... (first alos) ... (rest alos)...]
    [(and (> n 1) (cons? alos))
     ... (pred n) ... (first alos) ... (rest alos) ...]))
```

- Nun kann jeder Fall einzeln implementiert werden

Prozeduren mit mehreren komplexen Argumenten

- Vollständige Implementierung

```
;; list-pick : list-of-symbols N[>= 1]  ->  symbol
;; to determine the nth symbol from alos, counting from 1;
;; signals an error if there is no nth item
(define (list-pick alos n)
 (cond
   [(and (= n 1) (empty? alos)) (error 'list-pick "list too short")]
   [(and (> n 1) (empty? alos)) (error 'list-pick "list too short")]
   [(and (= n 1) (cons? alos)) (first alos)]
   [(and (> n 1) (cons? alos)) (list-pick (rest alos) (pred n))]))
```

- Funktioniert, aber die Prozedur kann wesentlich vereinfacht werden!
- Nächster Schritt bei solchen Prozeduren
  - Analyse der Fälle in Bezug auf Vereinfachungen

Prozeduren mit mehreren komplexen Argumenten

- Fall 1 und Fall 2 haben dieselbe Implementierung

```
(define (list-pick alos n)
 (cond
   [(and (= n 1) (empty? alos)) (error 'list-pick "list too short")]
   [(and (> n 1) (empty? alos)) (error 'list-pick "list too short")]
   [(and (= n 1) (cons? alos)) (first alos)]
   [(and (> n 1) (cons? alos)) (list-pick (rest alos) (pred n))]))
```

- Vereinfachte Implementierung:

```
(define (list-pick alos n)
  (cond
    [(or (and (= n 1) (empty? alos))
         (and (> n 1) (empty? alos)))
               (error 'list-pick "list too short")]
    [(and (= n 1) (cons? alos)) (first alos)]
    [(and (> n 1) (cons? alos)) (list-pick (rest alos) (pred n))]))
```

Prozeduren mit mehreren komplexen Argumenten

```
(define (list-pick alos n)
  (cond
    [(or (and (= n 1) (empty? alos))
         (and (> n 1) (empty? alos)))
                (error 'list-pick "list too short")]
    [(and (= n 1) (cons? alos)) (first alos)]
    [(and (> n 1) (cons? alos)) (list-pick (rest alos) (pred n))]))
```

- Ein “scharfer Blick” (Gesetze der Logik) genügt, um festzustellen, dass die erste Bedingung zu **`(empty? alos)`** vereinfacht werden kann
  - Damit kann die cons-Bedingung aus Fall 2+3 wegfallen
- Endversion:

```
(define (list-pick alos n)
  (cond
    [(empty? alos) (error 'list-pick "list too short")]
    [(= n 1) (first alos)]
    [(> n 1) (list-pick (rest alos) (pred n))]))
```

Auswerten arithmetischer Ausdrücke

- Nun folgt ein etwas größeres Beispiel, welches die in dieser Vorlesung vorgestellten Techniken illustriert
- Gleichzeitig ist dieses Programm der erste Schritt hin zu einem Scheme-Interpreter in Scheme
  - Konkretisierung der Auswertungsregeln!
- **Aufgabenstellung:**
  - Repräsentation und Berechnung beliebig geschachtelter arithmetischer Ausdrücke mit Multiplikation und Addition
- **Eingabe:** “Tiefe” Liste von Symbolen und Zahlen
  - Beispiel: '(+ 3 (* (+ 3 5) (* 1 2))))
    = (list '+ 3 (list '* (list '+ 3 5) (list '* 1 2)))
    = (cons '+ (cons 3 (cons (cons '* (cons (cons '+ (cons 3 (cons 5 empty)))
    (cons (cons '* (cons 1 (cons 2 empty))) empty))) empty))) empty)))
- **Ausgabe:** Wert, den der arithmetische Ausdruck repräsentiert

Auswerten arithmetischer Ausdrücke

- 1. Schritt: Datenanalyse
- Design einer geeigneten Datenstruktur für arithmetische Ausdrücke
  - Im Prinzip könnte man direkt mit den tiefen Listen aus Symbolen/Zahlen (sogenannten “s-expressions”) arbeiten
  - Dies wird jedoch schnell unübersichtlich, weil s-expressions eine zu allgemeine Datenstruktur sind
    - So wollen wir nur 2-stellige Operatoren berücksichtigen, bei s-expressions gibt es jedoch immer nur Listen
  - Außerdem müssen wir stets mit fehlerhaften Eingaben rechnen
- Daher werden wir eine eigene spezielle Datenstrukturen für das Speichern arithmetischer Datenstrukturen definieren und die s-expressions darin umwandeln
  - Sogenanntes “Parsing”

Auswerten arithmetischer Ausdrücke

- Definition des Datentypen “aexp”:

```
;; Data definition:
;; An arithmetic expression aexp is either
;;  1. a number
;;  2. (make-add l r) where l and r are aexp
;;  3. (make-mul l r) where l and r are aexp

(define-struct add (left right))
(define-struct mul (left right))
```

Auswerten arithmetischer Ausdrücke

- Definition des Parsers

```
;; parse :: deep-list-of symbol -> aexp
;; converts a deep list of symbols into a corresponding
;; aexp, if possible
;;
;; Example:
;; (parse '(+ 3 (* (+ 3 5) (* 1 2))))
;; = (make-add 3 (make-mul (make-add 3 5) (make-mul 1 2)))
(define (parse sexp)
 (cond
   [(number? sexp) sexp]
   [(cons? sexp)
     (cond
       [(symbol=? (first sexp) '+)
         (make-add (parse (second sexp))
                   (parse (third sexp)))]
       [(symbol=? (first sexp) '*)
         (make-mul (parse (second sexp))
                   (parse (third sexp)))])]))
```

Auswerten arithmetischer Ausdrücke

- Definition der Auswertungsprozedur

```
;; calc :: aexp -> number
;; calculates the number represented by
;; the arithmetic expression exp
;;
;; Example:
;; (calc (make-add 3 (make-mul (make-add 3 5) (make-mul 1 2))))
;;  = 19
(define (calc exp)
  (cond
     [(number? exp) exp)
     [(add? exp) (+
                    (calc (add-left exp))
                    (calc (add-right exp)))]
     [(mul? exp) (*
                    (calc (mul-left exp))
                    (calc (mul-right exp)))]))
```

Auswerten arithmetischer Ausdrücke

- Benutzung des Interpreters:
  - z.B. (calc (parse '(+ 5 (* 7 3)))) → 26
- Wozu dient dieses Programm?
  - Gutes Beispiel für rekursive Datenstrukturen und rekursive Funktionen darauf
  - Wir haben gerade einen (primitiven) Interpreter geschrieben
    - Er interpretiert die *Sprache* der arithmetischen Ausdrücke
  - Er modelliert, wie auch in Scheme selbst arithmetische Ausdrücke ausgewertet werden
  - Die Basisstruktur dieses Mini-Interpreters ist bereits dieselbe wie die eines "ausgewachsenen" Interpreters für eine Sprache wie Scheme
    - Parser + rekursiv definierte Datenstrukturen + rekursiv definierte Auswertungsprozedur
  - Wir können den Interpreter später so erweitern, dass er wesentliche Bestandteile der Sprache Scheme interpretieren kann!

Auswerten arithmetischer Ausdrücke

- Wie gut ist der Code aus Software-Engineering Sicht:
  - Wie gut ist er erweiterbar?
- Zwei wichtige Erweiterungsdimensionen:
  - Hinzufügen neuer Datentypen (z.B. Division, Subtraktion)
  - Hinzufügen neuer Operationen (z.B. scale)
- Neue Operationen hinzuzufügen ist leicht
  - Der bestehende Code muss nicht verändert werden!

Auswerten arithmetischer Ausdrücke

- Hinzufügen einer neuen Operation

```
;; swap+* :: aexp -> aexp
;;
;; generates new aexp within which
;; every addition is replaced by multiplication
;; and vice versa
;;
;; Example:
;; (swap+* (make-add 3 (make-mul 5 7)))
;;  = (make-mul 3 (make-add 5 7))
(define (swap+* exp)
  (cond
     [(number? exp) exp)]
     [(add? exp)  (make-mul
                     (swap+* (add-left exp))
                     (swap+* (add-right exp)))]
     [(mul? exp)  (make-add
                     (swap+* (mul-left exp))
                     (swap+* (mul-right exp)))]))
```

Auswerten arithmetischer Ausdrücke

- Wie gut ist der Code aus Software-Engineering Sicht:
  - Wie gut ist er erweiterbar?
- Zwei wichtige Erweiterungsdimensionen:
  - Hinzufügen neuer Datentypen (z.B. Division, Subtraktion)
  - Hinzufügen neuer Operationen (z.B. scale)
- Neue Operationen hinzuzufügen ist leicht
  - Der bestehende Code muss nicht verändert werden!
- Neue Datentypen hinzuzufügen ist jedoch schwierig
  - Die cond-Statements aller Operationen müssen um einen Fall für den neuen Datentyp erweitert werden

Auswerten arithmetischer Ausdrücke

- Hinzufügen eines neuen Datentypen

```
;; Data definition:
;; An arithmetic expression aexp is either
;;  1. a number
;;  2. (make-add l r) where l and r are aexp
;;  3. (make-mul l r) where l and r are aexp
;;  4. (make-sub l r) where l and r are aexp

(define-struct add (left right))
(define-struct mul (left right))
(define-struct sub (left right))
```

Auswerten arithmetischer Ausdrücke

- Hinzufügen eines neuen Datentypen

```
;; calc :: aexp -> number
;; …
(define (calc exp)
  (cond
     [(number? exp) exp]
     [(add? exp) (+
                   (calc (add-left exp))
                   (calc (add-right exp)))]
     [(sub? exp) (-
                   (calc (add-left exp))
                   (calc (add-right exp)))]
     [(mul? exp) (*
                   (calc (mul-left exp))
                   (calc (mul-right exp)))]))
```

Bestehender Code muss geändert werden
Analog andere Operationen wie swap+*

Auswerten arithmetischer Ausdrücke

- Wie gut ist der Code aus Software-Engineering Sicht:
  - Wie gut ist er erweiterbar?
- Zwei wichtige Erweiterungsdimensionen:
  - Hinzufügen neuer Datentypen (z.B. Division, Subtraktion)
  - Hinzufügen neuer Operationen (z.B. scale)
- Neue Operationen hinzuzufügen ist leicht:
  - Der bestehende Code muss nicht verändert werden!
- Neue Datentypen hinzuzufügen ist jedoch schwierig
  - Die cond-Statements aller Operationen müssen um einen Fall für den neuen Datentyp erweitert werden
  - Bestehender Code muss geändert werden; das ist besonders schlecht, wenn die Codeteile unabhängig voneinander entwickelt werden sollen
- Wir werden später feststellen, dass dies bei objektorientierten Sprachen genau umgekehrt ist.