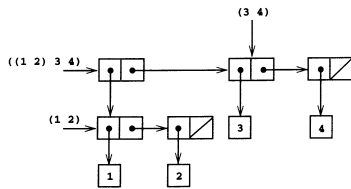


## Bäume und Formelmanipulation

### Darstellung von Bäumen durch Listen

Die Darstellung sequentieller Strukturen durch Listen gestattet in kanonischer Weise die Darstellung von Sequenzen, deren Elemente selbst wieder Sequenzen sind.

Beispiel: `(cons (list 1 2) (list 3 4))`



## Baum-Rekursion

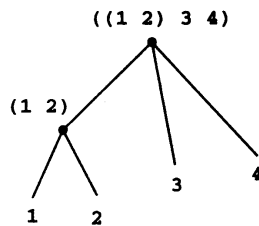
Beispiele:

```
(define (countcells l)
  (cond ((atom? l) 0) ; atom? = not pair?
        (else (+ (countcells (car l))
                  (countcells (cdr l))
                  1))))

(define (countatoms x)
  (cond ((null? x) 0)
        ((atom? x) 1)
        (else (+ (countatoms (car x))
                  (countatoms (cdr x))))))
```

Die Folge der Verarbeitungsschritte bildet strukturell den zu verarbeitenden Baum (allg.: hierarchische Datenstruktur) nach!

`((1 2) 3 4)` in graphischer Baumdarstellung:



*Hinweis:* Binäre Bäume (ohne Beschriftung innerer Knoten) können auch direkt als Paare implementiert werden.

### reverse-all

reverse kehrt eine Liste "auf der obersten Ebene" um:

```
(reverse '(a (b c) (d (e f))))
==> ((d (e f)) (b c) a)
```

REVERSE-ALL soll eine Liste auf allen Ebenen — d.h. beliebig tief eingeschachtelte Sublisten — umkehren:

```
(define (reverse-all l)
  (if (null? l)
      '()
      (append (reverse-all (cdr l))
              (list (if (pair? (car l))
                        (reverse-all (car l))
                        (car l)) ) ) )
```

## Beispiel: Symbolisches Differenzieren

### ▷ Computer-Algebra

Ableitungsregeln für algebraische Ausdrücke:

$$\frac{dc}{dx} = 0 \text{ für Konstante } c \text{ oder Variable } c \neq x$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$
$$\frac{d(uv)}{dx} = u \left( \frac{dv}{dx} \right) + v \left( \frac{du}{dx} \right)$$

## Implementation der Ableitungsregeln

```
(define (deriv exp var)
  (cond ((constant? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                    (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
          (make-product
           (multiplier exp)
           (deriv (multiplicand exp) var))
          (make-product
           (deriv (multiplier exp) var)
           (multiplicand exp))))))
```

## Darstellung algebraischer Ausdrücke

nach dem Prinzip der Datenabstraktion

Prädikate, Selektoren, Konstruktoren

(constant? <e>)	Ist <e> Konstante?
(variable? <e>)	Ist <e> Variable?
(same-variable? <v1> <v2>)	<v1> und <v2> selbe Var.?
(sum? <e>)	Ist <e> ein Summenterm?
(product? <e>)	Ist <e> ein Produktterm?
(addend <e>)	Erster Summand von <e>
(augend <e>)	Zweiter Summand von <e>
(multiplier <e>)	Multiplikator von <e>
(multiplicand <e>)	Multiplikand von <e>
(make-sum <a1> <a2>)	Konstruiere Summenterm aus <a1> und <a2>
(make-product <m1> <m2>)	Konstruiere Produktterm aus <m1> und <m2>

## Darstellung algebraischer Ausdrücke (1)

Es liegt nahe, zur Darstellung algebraischer Ausdrücke dieselbe Repräsentation zu wählen, in der Scheme Terme darstellt, z.B.

$$ax + b \quad \text{durch } (+ (* a x) b)$$

- Konstanten sind Zahlen:  
(define (constant? x) (number? x))
- Variablen sind Symbole:  
(define (variable? x) (symbol? x))
- Gleichheit von Variablen: eq?  
(define (same-variable? v1 v2)
 (and (variable? v1) (variable? v2)
 (eq? v1 v2)))

## Darstellung algebraischer Ausdrücke (2)

- Summen- und Produktterme werden als Listen konstruiert:

```
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
```

- Summenterme sind Listen mit Kopfelement +:

```
(define (sum? x)
  (if (list? x) (eq? (car x) '+) #f))
```

- Analog Produktterme:

```
(define (product? x)
  (if (list? x) (eq? (car x) '*) #f))
```

## Beispiele:

```
> (deriv '(+ x 3) 'x)
(+ 1 0)
```

```
> (deriv '(* x y) 'x)
(+ (* x 0) (* 1 y))
```

```
> (deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0))
  (* (+ (* x 0) (* 1 y))
    (+ x 3)))
```

## Darstellung algebraischer Ausdrücke (3)

- Erster und zweiter Summand eines Summenterms:

```
(define (addend s) (cadr s))
(define (augend s) (caddr s))
```

- Multiplikator und Multiplikand eines Produktterms:

```
(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))
```

Damit: ablauffähiges Ableitungsprogramm

## Symbolische Vereinfachung: optimize-expr

Vereinfachung von Summentermen mit 0 und Produkttermen mit 1

```
(define (optimize-expr expr)
  (cond
    ((sum? expr)
     (optimize-sum
      (optimize-expr (addend expr))
      (optimize-expr (augend expr))))
    ((product? expr)
     (optimize-product
      (optimize-expr (multiplier expr))
      (optimize-expr (multiplicand expr))))
    (else expr)))
```

## Vereinfachen von Summen und Produkten

```
(define (optimize-sum a1 a2)
  (cond
    ((and (constant? a1)
          (constant? a2))
     (+ a1 a2)) ; c1 + c2 = c3
    ((eqv? 0 a1) a2) ; 0 + a2 = a2
    ((eqv? 0 a2) a1) ; a1 + 0 = a1
    (else (make-sum a1 a2))))

(define (optimize-product m1 m2)
  (cond
    ((and (constant? m1)
          (constant? m2))
     (* m1 m2)) ; c1 * c2 = c3
```

G. Görz, FAU, Inf.8

6-13

```
((eqv? 1 m1) m2) ; 1 * m2 = m2
((eqv? 1 m2) m1) ; m1 * 1 = m1
((or (eqv? 0 m1)
      (eqv? 0 m2)) ; 0 * m2 = 0
      (eqv? 0 m2)) ; m1 * 0 = 0
0)
(else (make-product m1 m2))))
```

G. Görz, FAU, Inf.8

6-14

## Datenabstraktion und symbolische Vereinfachung

```
(define (make-sum a1 a2)
  (cond ((and (constant? a1) (constant? a2))
        (+ a1 a2))
        ((constant? a1)
         (if (= a1 0) a2 (list '+ a1 a2)))
        ((constant? a2)
         (if (= a2 0) a1 (list '+ a1 a2)))
        (else (list '+ a1 a2))))

(define (make-product m1 m2)
  (cond ((and (constant? m1) (constant? m2))
        (* m1 m2))
```

G. Görz, FAU, Inf.8

6-15

```
((constant? m1)
  (cond ((= m1 0) 0)
        ((= m1 1) m2)
        (else (list '* m1 m2))))
((constant? m2)
  (cond ((= m2 0) 0)
        ((= m2 1) m1)
        (else (list '* m1 m2))))
(else (list '* m1 m2))))
```

Bei konsequenter Anwendung der Datenabstraktion kann die Vereinfachung arithmetischer Ausdrücke bereits bei deren Aufbau erfolgen!

G. Görz, FAU, Inf.8

6-16

## Hinweis:

### Abstrakte Datentypen und Algebraische Spezifikation

Algebraische Sicht der Datenabstraktion:  $\Sigma$ -Algebra

- Abstrakte Typen sind durch Signaturen (Trägermengen, Operationen und auf ihnen erklärte Gesetze) gegeben
- Isomorphie von Algebren ist die Äquivalenzrelation der Datenabstraktion
- Abstrakter Datentyp ist implementations-unabhängig; wichtig sind nur die Eigenschaften der Operatoren!
- Axiomatische Spezifikation:  
Herstellung von Beziehungen zwischen den Operationen  
Parnas (1974): *V- (value returning) und O- (operate) Funktionen*

- Technisch:  
*Modularisierung* als Mittel zur Beherrschung komplexer Systeme  
Liskov/Zilles (1972): Modul = Datenstruktur + Zugriffsfunktionen  
⇒ Objektorientierte Programmierung

▷ Ergänzungskapitel 6-ADT-Algebra

### Abbildungsfunktionen: map, for-each, apply

Abbildungsfunktionen haben typischerweise Prozeduren und Listen als Argumente und wenden ihre prozeduralen Argumente auf ihre Listenargumente in verschiedenen Varianten an.

Betrachte:

```
(define (add1-to-each-item ls)
  (if (null? ls) '()
      (cons (+ 1 (car ls))
            (add1-to-each-item (cdr ls)))))
```

Gegeben: Standardprozedur max und Liste ls = (2 4)

**Problem:** (max ls) ==> error!

Lösung: Abbildungsfunktionen. Hier: map, for-each, apply (Auswahl)

## MAP (1)

```
( map <proc> <list1> <list2> ... )
```

Semantik:

- *<proc>* soll eine Prozedur mit so vielen formalen Parametern wie die Anzahl der *<list<sub>n</sub>>* sein.
- Wenn mehr als eine *<list<sub>i</sub>>* vorhanden, müssen alle gleiche Länge haben.
- map wendet *<proc>* elementweise auf *<list<sub>i</sub>>* an und gibt eine Liste der Ergebnisse als Wert zurück.
- Die *dynamische* Reihenfolge, in der die Prozedur auf die Elemente der Listen angewendet wird, ist *unspezifiziert*.

Beispiele:

```
> (map + '(1 2 3) '(4 5 6))
(5 7 9)
```

```
> (map (lambda (n) (expt n n)) '(1 2 3 4 5))
(1 4 27 256 3125)
```

## MAP (2)

Eine Prozedur als Argument für map darf keine Seiteneffekte verursachen, da Reihenfolge nicht festgelegt. Nur der Wert (Ergebnisliste) interessiert!

Implementation von map (einstellig)

```
(define (my-map fn l)
  (if (null? l)
      '()
      (cons
        (fn (car l))
        (my-map fn (cdr l)))))
```

## FOR-EACH

```
( for-each <proc> <list1> <list2> ... )
```

Semantik:

- $\langle proc \rangle$  soll eine Prozedur mit so vielen formalen Parametern wie die Anzahl der  $\langle list_n \rangle$  sein.
- Wenn mehr als eine  $\langle list_i \rangle$  vorhanden, müssen alle gleiche Länge haben.
- for-each wendet  $\langle proc \rangle$  elementweise auf  $\langle list_i \rangle$  an, und zwar in der Reihenfolge der Elemente in den Listen.
- Wert von for-each ist *unspezifiziert*.

Beispiel:

```
(define writeln
  (lambda (args)
    (for-each display args)
    (newline)))
```

for-each soll für Prozeduren mit Seiteneffekten verwendet werden, da Auswertungsreihenfolge festgelegt und Wert unspezifiziert ist.

## APPLY

```
( apply <proc> <list>)
```

Semantik: apply wendet eine Prozedur *<proc>* mit *k* formalen Parametern auf die *k*-elementige Liste *<list>* der Argumente an.

Beispiele:

```
(apply max '(2 4)) ==> 4
(max 2 4) ==> 4
(define add
  (lambda args      ; lambda unrestringiert
    (if (null? args) 0
        (+ (car args)
            (apply add (cdr args))))))
(add 1 2 3) ==> 6
```

fold verknüpft die Elemente einer Liste mit einer gegebenen Operation proc

```
(define (fold proc list)
  (cond ((null? list) '())
        ((null? (cdr list)) (car list))
        (else
         (proc (car list)
               (fold proc (cdr list))))))
```

## Übung: filter und fold

filter bildet die Teilliste derjenigen Elemente, die die Bedingung pred erfüllen

```
(define (filter pred list)
  (cond ((null? list) '())
        ((pred (car list))
         (cons (car list) (filter pred (cdr list))))
        (else
         (filter pred (cdr list)))))
```

## “Quantoren” (“Quantifiers”)

Prozeduren, die ein Prädikat als Argument bekommen und eine Prozedur als Wert zurückgeben, welche für jeweils zwei Argumente feststellt, ob sie das Prädikat erfüllen.

```
(define both
  (lambda (pred)
    (lambda (arg1 arg2)
      (and (pred arg1) (pred arg2)))))

((both (compose not null?)) '(a b) '(c))
==> #t
```

```
(define neither
  (lambda (pred)
    (lambda (arg1 arg2)
      (not (or (pred arg1) (pred arg2)))))))
```

```
((neither null?) '(a b) '(c))
=> #t
```

```
(define at-least-one
  (lambda (pred)
    (lambda (arg1 arg2)
      (or (pred arg1) (pred arg2))))))
```

```
((at-least-one even?) 1 3)
=> #f
```

## Vereinfachung von Quantoren (1)

Jeder der Quantoren `both`, `neither` und `at-least-one` ist "minimal vollständig" in dem Sinne, dass man die restlichen zwei damit definieren kann; z.B.:

```
(define both
  (lambda (pred)
    (lambda (arg1 arg2)
      ((neither (lambda (arg) (not (pred arg))))
       arg1
       arg2))))
```

Diese Form ist aber nicht gut lesbar!

## Vereinfachung von Quantoren (2)

Wir wissen aber, dass z.B.:

```
(lambda (arg) (not (pred arg)))
=> (lambda (arg) ((compose not pred) arg))
=> (compose not pred)
```

Eingesetzt:

```
(define both
  (lambda (pred)
    (lambda (arg1 arg2)
      ((neither (compose not pred)) arg1 arg2))))
```

Eine nochmalige Anwendung der gleichen Vereinfachung ergibt schließlich:

```
(define both
  (lambda (pred)
    (neither (compose not pred))))
```

Diese Art von Vereinfachung ist nur möglich, weil wir den Parameter `pred` "auscurriert" haben.

Quantoren ermöglichen eine kompakte Definition von Kombinationen von Prädikaten, die ihrer natürlichsprachlichen Formulierung stark ähnelt; sie stellen somit eine weitere Stufe der sprachlichen Abstraktion dar.

## Mengen als abstrakter Datentyp (ADT) (nach Springer/Friedman)

### Basisoperationen

Es werden die folgenden Grundfunktionen für den ADT Menge festgelegt, mit denen dann *alle* anderen Mengenfunktionen implementiert werden sollen (*Implementationsunabhängigkeit!*)

Operationen mit Mengen wie Durchschnitt und Vereinigung sollen gemäss dem Prinzip der Datenabstraktion *unabhängig* von der gewählten (internen) Darstellung von Mengen definiert werden.

Später werden Alternativen für die Implementation des ADT Menge diskutiert.

```
the-empty-set :           → set
empty-set? :      set → bool
adjoin-set :      obj × set → set
pick :            set → obj
residue :         obj → (set → set)
element :         obj → (set → bool)
```

Funktionale "Schnittstellen":

```
the-empty-set
(empty-set? s)
(adjoin-set e s)
(pick s)
((residue e) s)
((element e) s)
```

## Basisoperationen für den Mengen-ADT

1. *Neutrales Element*: leere Menge

Konstante `the-empty-set`

oder alternativ als nullstellige Funktion ("Thunk"): `(the-empty-set)`

Zugehöriges *Prädikat*: `(empty-set? s)`

2. *Konstruktoren*:

`(make-set)` bzw. `(make-set e1 e2 ...)`

`(adjoin-set e s)`

3. *Selektoren*:

`(pick s)` liefert beliebiges Element von `s`

`((residue e) s)` liefert die Restmenge, d.h. `s` ohne `e`

4. *Prädikat*:

`(set? obj)`

## Quantoren für Mengen (1)

1. `(none <pred>)`

angewandt auf ein Prädikat `<pred>` liefert ein Prädikat, das auf eine Menge `s` angewandt `#t` ergibt, wenn `<pred>` auf kein Element von `s` zutrifft,

z.B. `((none odd) {2,4,6}) ==> #t`

```
(define (none pred)
  (define (test s)
    (or (empty-set? s)
        (let ((elem (pick s)))
          (and (not (pred elem))
               (test ((residue elem) s))))))
  test)
```

## Quantoren für Mengen (2)

2. `(there-exists <pred>)`  
ergibt auf eine Menge `s` angewandt `#t`, wenn mindestens ein Element von `s` `<pred>` erfüllt. Dies kann man mit  
`((compose not (none <pred>)) s)`  
erreichen:

```
(define (there-exists pred)
  (compose not (none pred)))
```

3. `(for-all <pred>)`  
sinngemäss für das Zutreffen von `<pred>` auf alle Elemente einer Menge `s`:

```
(define (for-all pred)
  (none (compose not pred)))
```

## Mengen-Element-Prozeduren (1)

Damit könnte `((element obj) s)` "curried" definiert werden:

```
(define (element obj)
  (lambda (s)
    ((there-exists (set-equal obj)) s)))
; (set-equal obj) prüft, ob sein Arg = obj
```

; Kuerzer:

```
(define (element obj)
  (there-exists (set-equal obj)))
```

; NOCH kuerzer:

```
(define element
  (compose there-exists set-equal))
```

## Gleichheit von Mengen

Zwei Mengen  $S_1$  und  $S_2$  sind gleich gdw.  $S_1 \subset S_2 \wedge S_2 \subset S_1$

Gleichheitsprädikat: Prozedur höherer Ordnung mit

`((set-equal set1) set2) ==> #t`, wenn `set1` und `set2` gleich sind.

```
(define (set-equal s1)
  (lambda (s2)
    (or (and ((neither set?) s1 s2)
             (equal? s1 s2))
        (and ((both set?) s1 s2)
             ((subset s1) s2)
             ((subset s2) s1))))))
```

Ein zweistelliges Prädikat `set-equal?` ist damit trivial zu implementieren:

```
(define (set-equal? set1 set2)
  ((set-equal set1) set2))
```

## Mengen-Element-Prozeduren (2)

Wegen "Currying": Der Aufruf `((element obj) s)` prüft die mengentheoretische Relation `obj ∈ s`.

Wir brauchen noch eine Prozedur für die inverse Relation `s ∋ obj`:  
`((contains s) obj)`

```
(define (contains s)
  (lambda (obj)
    ((element obj) s)))
```

## Obermenge, Teilmenge und Kardinalität

```
(define (superset s1)
  (lambda (s2)
    ((for-all (contains s1)) s2)))

(define (subset s1)
  (lambda (s2)
    ((superset s2) s1)))

(define (cardinal s)
  (if (empty-set? s)
      0
      (let ((elem (pick s)))
        (+ 1 (cardinal ((residue elem) s)) ) ) ) )
```

## Durchschnitt von Mengen

```
(define (intersection s1 s2)
  (define (helper s1)
    ; s2 wird beim rek. Aufruf nicht veraendert
    (if (empty-set? s1)
        THE-EMPTY-SET
        (let ((elem (pick s1)))
          (if ((CONTAINS s2) elem)
              (adjoin-set
               elem
               (helper ((residue elem) s1)))
              (helper ((residue elem) s1))) ) ) )
  (helper s1))
```

## Allgemeines zum Invertieren von Relationen

contains ist die inverse Relation zu element:

$((\text{element } \text{obj}) \text{ s}) \Leftrightarrow ((\text{contains } \text{s}) \text{ obj})$

**Allgemein:** (define contains (inverse element))

rel :  $X \rightarrow (Y \rightarrow \text{bool})$   
invrel :  $Y \rightarrow (X \rightarrow \text{bool})$   
inverse :  $(X \rightarrow (Y \rightarrow \text{bool})) \rightarrow (Y \rightarrow (X \rightarrow \text{bool}))$

```
(define (inverse r)
  (lambda (y)
    (lambda (x) ((r x) y)) ) )
```

Damit:

```
(define superset (inverse subset))
```

## Vereinigung von Mengen

```
(define (union s1 s2)
  (define (helper s1)
    ; s2 wird beim rek. Aufruf nicht veraendert
    (if (empty-set? s1)
        S2
        (let ((elem (pick s1)))
          (if (NOT ((CONTAINS s2) elem))
              (adjoin-set
               elem
               (helper ((residue elem) s1)))
              (helper ((residue elem) s1))) ) ) )
  (helper s1))
```

## Symmetrische Differenz von Mengen

```
(define (difference s1 s2)
  (define (helper s1)
    ; s2 wird beim rek. Aufruf nicht veraendert
    (if (empty-set? s1)
        THE-EMPTY-SET
        (let ((elem (pick s1)))
            (if (NOT ((CONTAINS s2) elem))
                (adjoin-set
                 elem
                 (helper ((residue elem) s1)))
                (helper ((residue elem) s1))))))
  (helper s1))
```

## set-builder

Prozedur	base-set	pred
intersection	the-empty-set	(contains s2)
union	s2	(compose not (contains s2))
difference	the-empty-set	(compose not (contains s2))

Die abstrahierte Prozedur:

```
(define (set-builder pred base-set)
  (define (helper s)
    (if (empty-set? s) base-set
        (let ((elem (pick s)))
            (if (pred elem)
                (adjoin-set elem (helper ((residue elem) s)))
                (helper ((residue elem) s))))))
  helper)
```

## Durchschnitt, Vereinigung und Differenz von Mengen (Forts.)

Wie zu erwarten, weisen diese Prozeduren eine große strukturelle Ähnlichkeit auf.

⇒ *Abstraktionsschritt* liegt nahe:

Die spezifischen Unterschiede sollen als Parameter an *eine* Prozedur übergeben werden, die dann die jeweilige Prozedur liefert.

Unterschiede:

1. Welche Menge wird zurückgegeben, wenn das Abbruchkriterium erfüllt ist?
2. Welches Prädikat wird bei der Konstruktion des Resultats benutzt?

## set-builder: Anwendung

Damit können wir formulieren

```
(define (intersection s1 s2)
  ((set-builder (contains s2) the-empty-set)
   s1))

(define (union s1 s2)
  ((set-builder (compose not (contains s2)) s2)
   s1))

(define (difference s1 s2)
  ((set-builder (compose not (contains s2))
               the-empty-set)
   s1))
```

## Implementation von Mengen

Wir diskutieren drei verschiedene Darstellungen für Mengen:

- durch ungeordnete Listen
- durch geordnete Listen
- durch binäre Bäume (ihrerseits implementiert durch Listen)

Um zu erkennen, dass eine Liste eine Menge darstellt, sollte man als erstes Element einen Typ-Indikator, z.B. `*set*`, einsetzen (hier der Einfachheit halber weggelassen).

## Grundoperationen auf Mengen: Darstellung durch ungeordnete Listen

- `make-set` liefert leere Menge (Liste)  

```
(define (make-set) '())
```
- `empty-set?` mit `null?`
- `adjoin-set` mit `cons`, aber das Element `e` darf nicht bereits in der Liste (Menge `s`) enthalten sein:

```
(define (adjoin-set e s)
  (cond (((element e) s) s)
        (else (cons e s))))
```

- Mengengleichheit  $\neq$  Listengleichheit!

## Darstellung von Mengen durch ungeordnete Listen

- Listen haben Ordnung
- Elemente können mehrfach auftreten

*Bedeutung:* Zahlreiche Anwendungen, z.B. "Information Retrieval"

Datenverwaltung: Darstellung, Zugriff und Verarbeitung von Kollektionen strukturierter Daten (Personaldaten, Konten, Stücklisten, ...).  
Effiziente Methoden zum Zugriff auf Daten über Schlüssel (von Datensätzen)

Die Selektoren `pick` und `residue` können in Listendarstellungen durch `car` und `cdr` implementiert werden — oder komplizierter:

- `pick` mit `list-ref` und zufällig gewähltem Index
- `residue` liefert eine Prozedur, die, auf eine Liste angewandt, diese Liste *ohne* Argument-Objekt von `residue` zurückgibt.

`element` kann mit `member` implementiert werden.

## Mengenoperationen: Komplexitätsbetrachtung

Da die meisten Mengenoperationen von `element` abhängen, hat die Geschwindigkeit dieser Operation einen entscheidenden Einfluss auf die Mengenimplementation im Ganzen.

Test auf Enthaltensein muss im ungünstigsten Fall die gesamte Menge Element für Element überprüfen: Zeitbedarf  $O(n)$  bei Kardinalität  $n$ .  
⇒ Zeitbedarf von `adjoin-set`:  $O(n)$

`intersection` führt für jedes Element von `set1` einen Test mit `element` durch, sodass der Zeitbedarf mit dem Produkt der Kardinalitäten der beiden Mengen wächst, d.h.  $O(n^2)$ .

Dasselbe trifft auch auf `union` zu.

## Komplexität von `intersection`

Effizienzsteigerung noch signifikanter:

Bei Vergleich der ersten Elemente beider Listen,  $x_1$  und  $x_2$ , erhalten wir ein Element der Schnittmenge, falls  $x_1 = x_2$ . Ist  $x_1 < x_2$ , so kann, da  $x_2$  das kleinste Element in `set2` ist,  $x_1$  nicht mehr darin vorkommen und somit auch nicht im Durchschnitt; analog für  $x_2 < x_1$ .

In jedem Schritt wird das Problem der Schnittmengenbildung reduziert auf die Berechnung des Durchschnitts kleinerer Mengen — Elimination des ersten Elements von `set1` oder von `set2` oder von beiden.

Daher ist die Anzahl der benötigten Schritte höchstens gleich der Summe der Größen von `set1` und `set2` — anstelle des Produkts bei Darstellung durch ungeordnete Listen,

d.h.  $O(n)$  statt  $O(n^2)$  !

## Darstellung von Mengen durch geordnete Listen

*Voraussetzung:* Ordnungsrelation über den Elementen, z.B. lexikographische Ordnung; oder allgemein: Zuordnung eines numerischen Ordnungsindex und Vergleich mit  $<$ ,  $=$

*Beispiel:*

Die Menge  $\{1, 3, 6, 10\}$  wird eindeutig durch die Liste  $(1\ 3\ 6\ 10)$  repräsentiert.

In `element` brauchen wir daher nicht mehr **alle** Elemente zu überprüfen.

Zeitersparnis: Im ungünstigsten Fall (wenn  $x \geq$  letztes Element) Vergleich mit allen Elementen.

Abschätzung der Anzahl der Vergleiche durch  $n/2$  — ebenfalls  $O(n)$  —, spart aber (im Mittel) Faktor 2 in der Zeit.

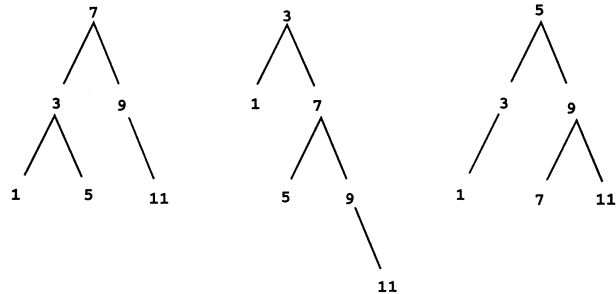
## Darstellung von Mengen durch binäre Bäume

Jeder Knoten des binären Baums enthält ein Element der Menge, den "entry", und zwei Verweise auf — möglicherweise leere — Knoten. Der linke Zweig verweist auf kleinere Elemente, der rechte auf größere. (\*)

⇒ Dieselbe Menge kann durch verschiedene Bäume repräsentiert werden, die bzgl. der Bedingung (\*) äquivalent sind.

Beispiel:

Drei zulässige Bäume zur Darstellung der Menge  $\{1, 3, 5, 7, 9, 11\}$



## Darstellung binärer Bäume durch Listen

```
(define (entry tree) (car tree))  
  
(define (left-branch tree) (cadr tree))  
  
(define (right-branch tree) (caddr tree))  
  
(define (make-tree entry left right)  
  (list entry left right))
```

## Darstellung von Mengen durch binäre Bäume

Vorteil der Baumdarstellung:

Wir wollen überprüfen, ob eine Zahl  $x$  in einer Menge enthalten ist. Vergleich von  $x$  mit dem Wurzelknoten des Baums: Ist  $x$  kleiner, wird im linken Teilbaum weitergesucht; ist  $x$  größer, im rechten.

Ist der Baum "ausbalanciert", so ist jeder Teilbaum etwa halb so gross wie der ganze  $\Rightarrow$  Reduktion der Suche in Baum der Größe  $n$  auf Suche in Baum der Größe  $n/2$ .

Halbierung in jedem Schritt: Anzahl der Schritte wächst mit  $O(\log n)$ .

## adjoin-set bei Baumdarstellung ( $O(\log n)$ Schritte)

```
(define (adjoin-set x set)  
  (cond ((null? set)  
        (make-tree x '() '()))  
        ((= x (entry set)) set)  
        (< x (entry set))  
        (make-tree (entry set)  
                    (adjoin-set x  
                                (left-branch set))  
                    (right-branch set)))  
        (> x (entry set))  
        (make-tree (entry set)  
                    (left-branch set)  
                    (adjoin-set x  
                                (right-branch set))))))
```

## Vor- und Nachteile der Baumdarstellung

Bei großen Mengen: Baumdarstellung am effizientesten für Suche und Hinzufügen neuer Elemente.

Durchschnittsbildung: keine bessere Strategie als im Fall ungeordneter Listen. Für jedes Element von  $set1$  ist zu testen, ob es in  $set2$  enthalten ist. Suche erfordert Zeitaufwand  $O(\log card(set2))$  für jedes Element von  $set1$ , also insgesamt  $card(set1) \cdot \log card(set2)$ . Mit  $card(set1) \approx card(set2): O(n \log n)$

Aber: Dennoch vorzuziehen, da i.a. mehr Vergleiche als Schnittbildungen durchzuführen sind.

Abschätzung des Suchaufwands mit logarithmischem Zeitaufwand beruht auf der Annahme, dass der Baum balanciert ist; es kann sich jedoch im Lauf der Verarbeitung ein Ungleichgewicht ergeben.

Abhilfe: Transformation nach jedem `adjoin-set`. *Kosten?!*

family-union ist einfach:

```
(define (family-union s)
  (if (empty-set? s)
      the-empty-set
      (let ((elem (pick s)))
        (union
         elem
         (family-union ((residue elem) s)))))))
```

`the-empty-set` spielt die Rolle eines "Einselements" für `union` — analog zur 0 für `plus` und 1 für `times`.

## Mengenfamilien (1)

Hat eine Menge als Elemente wieder Mengen, spricht man auch von einer *Mengenfamilie*.

Die Vereinigung der Mengen, die Elemente einer Menge  $S$  sind, wird mit  $\bigcup S$  bezeichnet: `family-union`

z.B.  $\bigcup\{\{a, b\}, \{b, c, d\}, \{a, e\}\} = \{a, b, c, d, e\}$

Analog wird der Durchschnitt  $\bigcap S$  definiert: `family-intersection`

z.B.  $\bigcap\{\{a, b, c\}, \{a, c, e\}, \{a, b, c, f\}\} = \{a, c\}$

## Mengenfamilien (2)

Für Durchschnittsbildung gibt es kein solches Einselement — `the-empty-set` ist ein "Annihilator" für `intersection` (analog zur 0 für `times`).

Daher muss Rekursion bei `family-intersection` terminiert werden, wenn eine Menge erreicht ist, die nur noch eine (einzelne) Menge als Element enthält:

```

(define (family-intersection s)
  (define (fam-int s)
    (let ((elem (pick s)))
      (let ((rest ((residue elem) s)))
        (if (empty-set? rest)
            elem
            (intersection
             elem
             (fam-int rest)) ))))
  (if (empty-set? s)
      the-empty-set
      (fam-int s)))

```

## MAP für Mengen

Anwendung einer Funktion auf alle Elemente einer Menge;  
 Resultat ist Menge der Funktionswerte

```

(define (map-set proc s)
  (if (empty-set? s)
      the-empty-set
      (let ((elem (pick s)))
        (adjoin-set
         (proc elem)
         (map-set
          proc
          ((residue elem) s)) ))))

```

## Geordnete Paare, Relationen und Funktionen

Geordnete Paare der Mengenlehre als ADT:

Konstruktor	make-op
Selektoren	op-1st op-2nd
Typprädiikat	op?

Mengentheoretische Definition *geordneter Paare* (Wiener, Kolakowski):

$(x, y) \neq \{x, y\}$ , denn:  $x \neq y \rightarrow (x, y) \neq (y, x)$ , aber:  $\{x, y\} = \{y, x\}$

Daher:  $(x, y) \Leftrightarrow \{\{x\}, \{x, y\}\}$

Damit gilt:  $(y, x) = \{\{y\}, \{x, y\}\} \neq (x, y)$

Das erste Element des geordneten Paares ist das einzige Element im  
 Durchschnitt der beiden Mengen  $\{x\}$ ,  $\{x, y\}$ :

$$\bigcap \{\{x\}, \{x, y\}\} = \{x\}$$

Analog: Für  $x \neq y$  ist das zweite Element des geordneten Paares in der  
 Differenz von  $\bigcup \{\{x\}, \{x, y\}\}$  und  $\bigcap \{\{x\}, \{x, y\}\}$

Falls  $x = y$ , wähle das erste Element des geordneten Paares.

*Beachte:* Dasselbe geordnete Paar  $(x, y)$  wird dargestellt durch:

$\{\{x\}, \{x, y\}\}$  bzw.  
 $\{\{x\}, \{y, x\}\}$  bzw.  
 $\{\{x, y\}, \{x\}\}$  etc.

## Implementation geordneter Paare (1)

```
(define (make-op x y)
  (make-set (make-set x) (make-set x y)))
```

sofern make-set beliebig-stellig definiert (sonst mehrfach adjoin-set)

```
(define (op? s)
  (and (set? s)
        ((for-all set?) s)
        (= (cardinal (family-intersection s)) 1)
        (or (= (cardinal s) 1)
              ((both
                 (lambda (x) (= (cardinal x) 2)))
                 s
                 (family-union s))))))
```

## Implementation geordneter Paare (2)

```
(define (op-1st p)
  (pick (family-intersection p)))

(define (op-2nd p)
  (let ((fam-int (family-intersection p)))
    (let ((diff (difference (family-union p) fam-int)))
      (pick
        (if (empty-set? diff) fam-int
            diff))))))
```

Erklärung des or-Terms:

Die Kardinalität eines geordneten Paares kann 1 sein:

$$(a, a) = \{\{a\}, \{a, a\}\} = \{\{a\}, \{a\}\} = \{\{a\}\}$$

Selbstverständlich kann man geordnete Paare auch durch den elementaren Datentyp **PAIR** implementieren:

```
(define make-op cons)
(define op? pair?)
(define op-1st car)
(define op-2nd cdr)
```

... oder durch zweielementige Listen:

```
(define (make-op x y) (list x y))

(define (op? arg)
  (and (pair? arg)
        (pair? (cdr arg))
        (null? (cddr arg))))

(define op-1st car)
(define op-2nd cadr)
```

## Cartesisches Produkt

$S_1 \times S_2$  der Mengen  $S_1$  und  $S_2$  ist die Menge aller geordneten Paare  
 $\{(x, y) \mid x \in S_1 \wedge y \in S_2\}$

Beispiel:  $S_1 = \{a, b, c\}, S_2 = \{d, e\}$   
 $S_1 \times S_2 = \{(a, d), (a, e), (b, d), (b, e), (c, d), (c, e)\}$

```
(define (cartesian-product s1 s2)
  (if (empty-set? s1) the-empty-set
      (let ((elem (pick s1)))
        (union (map-set (lambda (x) (make-op elem x))
                       s2)
                (cartesian-product ((residue elem) s1)
                                   s2)))))
```

## Relationen (1)

Eine *Relation* von einer Menge  $X$  in eine Menge  $Y$  ist eine Teilmenge des cartesischen Produkts von  $X$  und  $Y$ :  $R \subset X \times Y$ ,  
d.h. Menge von geordneten Paaren  $(x, y)$  mit  $x \in X$  und  $y \in Y$ .

Leere Relation:  $\emptyset$

Definitionsbereich ("domain") von  $R$ :  
 $\text{domain}(R) \Leftrightarrow \{x \mid (x, y) \in R\} \subset X$

Wertebereich ("range") von  $R$ :  
 $\text{range}(R) \Leftrightarrow \{y \mid (x, y) \in R\} \subset Y$

```
(define (domain rel)
  (map-set op-1st rel))
```

```
(define (range rel)
  (map-set op-2nd rel))
```

Eine zweistellige ("binäre") Relation auf einer Menge  $S$  ist eine Teilmenge des cartesischen Produkts  $S \times S$

z.B. mit  $\text{bob}, \text{tom}, \text{jim} \in \text{boys}$

```
(define is-older-than-relation
  (make-set (make-op 'tom 'bob)
            (make-op 'tom 'jim)
            (make-op 'bob 'jim)))
```

## Relationen (2)

Wir definieren nun ein Prädikat `is-older-than?` mit zwei Parametern  $b_1, b_2 \in \text{boys}$ , das `#t` ergibt, wenn das geordnete Paar  $(\text{make-op } b_1 \text{ } b_2)$  Element der binären Relation `is-older-than-relation` ist:

```
(define (is-older-than? b1 b2)
  ((contains is-older-than-relation)
   (make-op b1 b2)))
```

## Bildung von Teilrelationen

Beispielsweise: `subrelation/1st` bzgl.  $c \in X$ , fest  
 $\{(x, y) \mid x = c \in X, (x, y) \in R\}$

```
((subrelation/1st is-older-than-relation) 'tom)
=> {(tom,bob), (tom,jim)}
```

```
(define (subrelation/1st rel)
  (lambda (arg)
    ((set-builder
      (lambda (x) ((set-equal (op-1st x)) arg))
      the-empty-set)
     rel)))
```

Das Prädikat `function?` prüft, ob eine Relation eine Funktion ist:

```
(define (function? rel)
  (or (empty-set? rel)
      (let ((subrel ((subrelation/1st rel)
                     (op-1st (pick rel)))))
        (and (= (cardinal (map-set op-2nd subrel)) 1)
              (function? (difference rel subrel)))))
```

Bestimmung des Werts einer Funktion:

```
(define (value fun)
  (lambda (arg)
    (op-2nd (pick ((subrelation/1st fun) arg)))))
```

## Funktionen als rechtseindeutige Relationen

Eine *Abbildung (Funktion)* von einer Menge  $X$  in eine Menge  $Y$  ist eine Relation  $F$  von  $X$  in  $Y$ , für die gilt:

für alle Paare  $(x, y) \in F$  gibt es zu jedem  $x \in X$  *genau ein*  $y \in Y$ .

$\Rightarrow$  Funktion  $F$  ist die Relation, die aus den geordneten Paaren  $(x, y)$  besteht, die  $y = F(x)$  erfüllen;

z.B.  $FAK = \{(1, 1), (2, 2), (3, 6), (4, 24), \dots\}$

## EXKURS: Huffmansche Codierungs-Bäume

Anwendung der ADT Menge und Baum auf die Darstellung von Daten als Folgen von Nullen und Einsen (Bits).

*Beispiel:* ASCII-Code,  
codiert jedes Zeichen durch 7 Bits, d.h.  $2^7 = 128$  verschiedene Zeichen darstellbar.

Allgemein: Um  $N$  Zeichen zu unterscheiden, brauchen wir  $\log_2 N$  Bits pro Zeichen.

Beispiel:

Alphabet sei {A, B, C, D, E, F, G, H}

Durch 3-Bit-Code (*Code fester Länge*) codiert

A 000	C 010	E 100	G 110
B 001	D 011	F 101	H 111

Die Nachricht BACADAEAFABBAAAGAH wird durch folgende Kette aus 54 bits codiert:

001000010000011000100000101000001001000000000110000111

## Huffmansche Codierungs-Bäume (2)

Manchmal sind *Codes variabler Länge* vorteilhaft, z.B. der Morse-Code.

Häufig vorkommende Zeichen werden durch kurze, weniger häufige durch längere Bitfolgen codiert  $\Rightarrow$  Effizienzgewinn!

Beispiel:

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

Damit wird obige Nachricht durch folgende Bitsequenz codiert (42 Bits, 20% Ersparnis):

100010100101101100011010100100000111001111

## Huffmansche Codierungs-Bäume (3)

**Problem:**

Erkennung des Endes der Codierung für jedes Zeichen

*Lösungen:*

1. Trenncode: Pausenzeichen im Morsecode
2. Präfixcode: Kein vollständiger Code für ein Zeichen ist Anfang des Codes für ein anderes Zeichen (z.B. oben)

Ersparnis durch Verwendung von Präfixcodes variabler Länge, die die relative Häufigkeit der Zeichen in den zu codierenden Nachrichten berücksichtigen, z.B. nach der Methode von David Huffman.

## Huffmansche Codierungs-Bäume (4)

Ein Huffman-Code kann durch einen *binären Baum* dargestellt werden, dessen Blätter die zu codierenden Zeichen bilden.

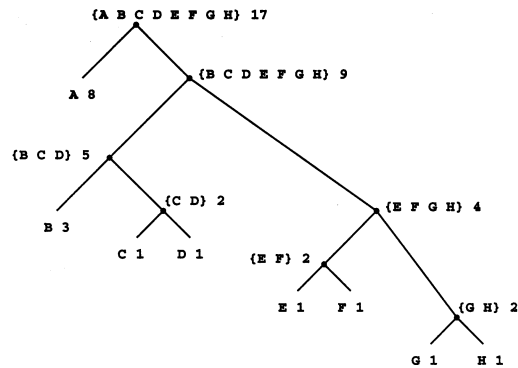
Jedem anderen Knoten ist die Menge aller Zeichen zugeordnet, die unterhalb dieses Knotens liegen.

Jedem Symbol an einem Blatt wird eine *relative Häufigkeit* zugeordnet und jedem anderen Knoten ein

*Gewicht* = Summe der Häufigkeiten aller darunter liegenden Blätter.

(Die Gewichte werden nicht bei Codierung und Decodierung benutzt, aber zur Konstruktion des Baums)

## Huffman-Baum für den Beispiel-Code



## Grundidee des Huffman-Algorithmus

Symbole mit der geringsten relativen Häufigkeit sollen am weitesten von der Wurzel des Baums entfernt sein.

Beginn: Blätter mit relativen Häufigkeiten

- Suche zwei Blätter mit den geringsten rel. Häufigkeiten
- Verschmelze sie zu einem Knoten, der diese als linken und rechten Zweig hat und als Gewicht die Summe der beiden Häufigkeiten
- Ersetze die beiden Blätter in der ursprünglichen Menge durch den neuen Knoten
- Wiederhole diesen Vorgang, bis die Menge nur noch einen Knoten enthält

## Huffmansche Codierungs-Bäume: Codierung und Decodierung

Konstruktion der **Codierung** eines Zeichens:

Abstieg von der Wurzel durch den Baum bis zum entsprechenden Blatt, wobei für jeden linken Zweig eine 0, für jeden rechten Zweig eine 1 zum Code hinzugefügt wird. *Beispiel: D* ⇒ 1011

**Decodierung:**

Beginnend an der Wurzel folge an jedem Knoten der durch den Code angegebenen Richtung, bis ein Blatt erreicht wird.

*Beispiel: 10001010* ⇒ BAC

**Konstruktion von Huffman-Bäumen**

Gegeben: Alphabet mit relativen Häufigkeiten.

Gesucht: Baum, der Nachrichten mit der kleinsten Anzahl von Bits codiert (HUFFMAN: Algorithmus [+ Optimalitätsbeweis]).

## Grundidee des Huffman-Algorithmus (Beispiel)

Beginn

{(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)}

Verschmelzung

{(A 8) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)}

{(A 8) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)}

{(A 8) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)}

(Wahlmöglichkeit)

{(A 8) (B 3) ({C D} 2) ({E F G H} 4)}

{(A 8) ({B C D} 5) ({E F G H} 4)}

{(A 8) ({B C D E F G H} 9)}

{{(A B C D E F G H} 17)}

Der Baum ist nicht in jedem Fall eindeutig bestimmt wegen Wahlmöglichkeiten bei Verschmelzung und Wahl von linkem bzw. rechtem Knoten.

## Darstellung von Huffman-Bäumen

```
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))

(define (leaf? object)
  (eq? (car object) 'leaf))

(define (leaf-symbol x) (cadr x))

(define (leaf-weight x) (caddr x))
```

Die Menge der Symbole an einem Knoten kann einfach durch eine Liste der Symbole dargestellt werden.

## Selektoren für Huffman-Bäume

```
(define (left-branch tree) (car tree))

(define (right-branch tree) (cadr tree))

(define (symbols tree)
  (if (leaf? tree)
      (list (leaf-symbol tree))
      (caddr tree)))

(define (weight tree)
  (if (leaf? tree)
      (leaf-weight tree)
      (caddr tree)))
```

Da jedes Symbol genau einmal vorkommt, kann die Mengenvereinigung durch `append` implementiert werden.

```
(define (make-code-tree left right)
  (list left
        right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right))))
```

## Die Decodierungsprozedur

```
(define (decode bits tree)
  ; Liste Bit-Sequenz Baum
  (decode-1 bits tree tree))

(define (decode-1 bits tree current-branch)
  ; Position im Baum
  (if (null? bits) '()
      (let ((next-branch
              (choose-branch (car bits) current-branch)))
        (if (leaf? next-branch)
            (cons (leaf-symbol next-branch)
                  (decode-1 (cdr bits) tree tree))
            (decode-1 (cdr bits)
                      tree next-branch))))))
```

## Mengen gewichteter Elemente (2)

```
(define (choose-branch bit branch)
  (cond
    ((= bit 0) (left-branch branch))
    ((= bit 1) (right-branch branch))
    (else
     (error "CHOOSE-BRANCH: bad bit" bit))))
```

Die Prozedur `make-leaf-set` konstruiert aus einer Liste von Paaren Symbol/Häufigkeit (z.B. ((A 4) (B 2) (C 1) (D 1))) eine geordnete Menge von Blättern zur Verschmelzung:

```
(define (make-leaf-set pairs)
  (if (null? pairs)
      '()
      (let ((pair (car pairs)))
        (adjoin-set
         (make-leaf (car pair) ; Symbol
                    (cadr pair)) ; Häufigkeit
         (make-leaf-set (cdr pairs))))))
```

## Mengen gewichteter Elemente

Da bei der Konstruktion Mengen von Blättern und Teilbäumen verarbeitet werden, so dass jeweils die "kleinsten" Elemente verschmolzen werden und also wiederholt die kleinsten Elemente in einer Menge gesucht werden müssen, empfiehlt sich eine geordnete Darstellung:

```
(define (adjoin-set x set)
  (cond
    ((null? set) (list x))
    ((< (weight x) (weight (car set)))
     (cons x set))
    (else
     (cons (car set)
           (adjoin-set x (cdr set))))))
```