Informatik 1

```
((lambda () (begin (display "Hello world!") (newline))))
```

Mitschrift von www.kuertz.name

Hinweis: Dies ist **kein offizielles Script**, sondern nur eine private Mitschrift. Die Mitschriften sind teweilse **unvollständig, falsch oder inaktuell**, da sie aus dem Zeitraum 2001–2005 stammen. Falls jemand einen Fehler entdeckt, so freue ich mich dennoch über einen kurzen Hinweis per E-Mail – vielen Dank!

Klaas Ole Kürtz (klaasole@kuertz.net)

Inhaltsverzeichnis

1 Grundbegriffe **1**
1.1 Grundbegriffe . . . 1
1.2 Eigenschaften von Algorithmen . . . 1
1.3 Mittel . . . 1
1.3.1 unwichtig . . . 2
1.3.2 wichtig . . . 2

2 Abstraktion mit Prozeduren **2**
2.1 Programmelemente . . . 2
2.1.1 Ausdrücke . . . 2
2.1.2 Abstraktion durch Benennung mit Objekten . . . 3
2.1.3 Auswertung von Termen/Kombinationen . . . 3
2.1.4 Prozeduren: Verfahren zur Wertberechnung abstrahieren 4
2.2 Programmauswertung mit dem Substitutionsmodell . . . 5
2.2.1 informelle Definition des Substitutionsmodells . . . 5
2.2.2 Definitionen für Programmteile . . . 5
2.2.3 formelle Definition des Substitutionsmodells . . . 7
2.2.4 unterschiedliche Berechnungswege . . . 8
2.3 weitere Sonderformen . . . 9
2.3.1 `cond` . . . 9
2.3.2 `if` . . . 10
2.3.3 `and`, `or`, `not` . . . 10
2.3.4 Beispiel: Quadratwurzelberechnung nach Newton . . . 10
2.4 Namensgebung . . . 11
2.5 Prozeduren und Prozesse . . . 12
2.5.1 Iteration, Rekursion etc. . . . 12
2.5.2 Größenordnungen . . . 14
2.5.3 Beispiele für effiziente Algorithmen . . . 14
2.5.4 Prozeduren höherer Ordnung . . . 17
2.5.5 λ-Abstraktionen . . . 17
2.5.6 Sonderform `let` . . . 18
2.5.7 universelle Berechnungsverfahren . . . 18
2.5.8 Prozeduren als Ergebnis . . . 19

3 Abstraktion mit Daten **20**
3.1 Einführung . . . 20
3.2 Hierachische Datenstrukturen . . . 23
3.2.1 Darstellung von Baumstrukturen . . . 24
3.2.2 Quotieren . . . 24

3.2.3 Beispiel: Symbolisches Differenzieren 25
3.2.4 Beispiel: Implementierung von Mengen 27
3.2.5 Huffmann-Bäume (siehe auch externes Blatt) 30
3.2.6 Konstruktion der Huffmann-Bäume 31
3.3 Daten mit Mehrfachrepräsentation 31
3.3.1 Beispiel: komplexe Zahlen 32
3.3.2 Implementierung oberste Ebene (Arithmetik) 32
3.3.3 Implementierung der Selektoren und Konstruktoren . . 32
3.4 Systeme mit generischen Operatoren 35
3.4.1 Anfügen einer Typetikette 36
3.4.2 Verbindung der Arithmetikoperationen mit Zahloperatoren 36
3.4.3 Definition der generischen Operatoren 36
3.4.4 Operanden mit unterschiedlichen Typen 37
3.4.5 Typanpassung 38
3.4.6 Generische Arithmetik auf Polynome 39

4 Modularität, Objekte, Zustände 41
4.1 Zuweisungen und lokale Zustände 42
4.1.1 Anwendung: Monte-Carlo-Simulation 46
4.1.2 Umgebungsmodell 47
4.1.3 Lokale Zustände, Umgebungen, Definitionen 49
4.2 Veränderbare Datenstrukturen 51
4.2.1 Warteschlangen/Queues 53
4.2.2 Tabellen 54
4.2.3 Tabellen als aktive Objekte 56
4.3 Anwendungsbeispiele 56
4.3.1 Simulation digitaler Systeme 56
4.3.2 Modelle mit Beschränkungen 60
4.4 Zustände in nebenläufigen Systemen 63
4.5 Datenströme (*streams*) als Standardschnittstellen 65
4.5.1 Implementierung der Ströme 67
4.5.2 Universelle Prozeduren für Datenströme 69
4.5.3 Beispiele 70
4.5.4 Geschachtelte Abbildungen, `collect` 71
4.5.5 Implementierung von Datenströmen 74
4.5.6 Implementierung der verzögerten Auswertung 76
4.5.7 Datenströme unendlicher Länge 77
4.5.8 Imlizite Definition von Strömen 78
4.5.9 Datenströme ↔ lokale Zustände 81
4.5.10 Diskussion: Datenströme ↔ Objekte mit Zuständen . . 83

5 Einführung in Java **85**
5.1 Allgemeines 85
5.2 Ausdrücke und Anweisungen 86
5.2.1 Grundtypen in Java 87
5.2.2 Anweisungen 87
5.2.3 Konzept zur Fehler- und Ausnahmebehandlung 88
5.3 Deklarationen 89
5.3.1 Felder („arrays“) 90
5.3.2 Funktionen und Prozeduren 91
5.4 Klassen und Objekte 91
5.5 Vererbung, Überladen 93
5.6 Schnittstellen und Pakete 95
5.7 weitere Aspekte in Java 96

1 Grundbegriffe

1.1 Grundbegriffe

- *programmieren:* Algorithmus in einer Programmiersprache formulieren
- *Algorithmus:* Beschreibung einer Vorgehensweise, wie man zu einer Klasse von gleich- bzw. ähnlichartigen Problemen eine Lösung findet (Bspl. Quadratwurzel).
- *Parameter:* legt konkrete Probleme der Klasse fest
- *partielle Korrektheit:* Falls der Algorithmus anhält, ist das Ergebnis eine Lösung des Problems.
- *totale Korrektheit:* Der Algorithmus hält immer an und ist partiell korrekt.

1.2 Eigenschaften von Algorithmen

- Endlichkeit der Beschreibung
- Effektivität der Schritte (jeder Schritt ist ausführbar und terminiert auch)
- z.T. Determiniertheit (Wirkung jedes Schrittes und nachfolgender Schritt ist eindeutig)

► Es gibt Probleme, zu denen es *keine* Algorithmen gibt (Bspl. Hält ein Pascal-Programm an?)

- Algorithmen sollten immer korrekt sein, formal ist das aber nicht nachprüfbar, Verfahren zur Prüfung nicht allgemein vorhanden, Testen kann nur die Anwesenheit von Fehlern zeigen

1.3 Mittel

- Beherrschung der *Komplexität* durch geeignete Strukturierung: Zerlegung großer Probleme in kleine Einheiten (Module) und Techniken zur Komposition
- *Programmiersprachen:* Hilfsmittel zur Formulierung von Algorithmen, ausführbar auf Rechnern, Bearbeitung mit Wekzeugen (Bibliotheken, Versionsverwaltung etc.)

- *Syntax:* Was sind zulässige Zeichenfolgen? → Bildungsgesetze
- *Semantik:* Was bedeuten diese Zeichenfolgen?
- *Pragmatik*: Wie wendet man die Sprache an (Systemumgebung, Kodierungshinweise, Debugger, ...)?

- *Programmelemente:* elementare Ausdrücke, Mittel zur Kombination und zur Abstraktion
- Anforderungen an eine Programmiersprache: Universiladität (Formulierung beliebiger Algorithmen) und automatische Analysierbarkeit; Programmiersprache muß elementare Daten/Prozeduren beschreiben können und diese kombinieren können

⇒ SCHEME , entstanden aus LISP und ALGOL

1.3.1 unwichtig

- bestimmte Sprache bis zur letzten Feinheit erlernen
- effiziente Programme schreiben
- trickreiche Programmierung

1.3.2 wichtig

- Gefühl für guten Programmierstil
- Techniken zur Beherrschung komplexer Zusammenhänge
- Denken in Strukturen und Algorithmen
- Wissen, was ist wichtig/unwichtig beim Lesen großer Programme

2 Abstraktion mit Prozeduren

2.1 Programmelemente

2.1.1 Ausdrücke

- *Zahl:* Wert ist die Zahl selbst
- *Kombinationen:* Verknüpfung mit durch elementare Prozeduren, wie z.B. $+, \cdot, ...$

- *Präfix-Notation:* `(<Operator> <Operand1> <Operand2>)`
  Vorteile: beliebig viele Operanden, beliebig verschachtelte Ausdrücke, eindeutige Hierachie

  ```
  > (+ 10 21)
  31
  > (* 5 42)
  210
  > (/ 45 (+ 10 (/ 10 5) 20 (* 10 10)))
  9
  ```

  Konvention zur Notation: Operanden vertikal untereinander ausrichten

2.1.2 Abstraktion durch Benennung mit Objekten

- identifiziere Objekte durch Namen, genauer: Ein Name (Identifikator) benennt eine Variable, deren Wert ein Objekt ist.
- Operator zur Namensgebung: $define$

```
> (define groesse 2)
> (+ 10 groesse)
12
```

- abstrahiere (komplexe) Ausdrücke zu einem Namen
- notwendig: SCHEME-System benötigt Speicher für die Namenszuordnung: → *Umgebung*
- *Umgebung:* Menge von Zuordnungen Namen zu Objekten; $define$ verändert die Umgebung

2.1.3 Auswertung von Termen/Kombinationen

1. Alle Teile auswerten (keine feste Reihenfolge!)
2. Wende Wert des linken Elements (eine Prozedur) auf Werte der restlichen Elemente an

- rekursiver Prozess, aber wohldefiniert (da Teile immer kleiner werden), also von innen nach außen

- linkes Element (Operator) kann komplexer Ausdruck sein (später ...)
- *Wert* von elementaren Prozeduren (+, ·, ...): Maschinenbefehle zum Ausrechnen
- Auswertung parallel auf gleicher Ebene (theoretisch)
- *Sonderform*
  - Sonderformen bilden die *Syntax* (nicht in der Umgebung definiert, sondern durch den Interpreter behandelt)
  - Auswertung:
    1. Werte letztes Element aus
    2. Binde zweites Element an den Wert des dritten Elements in der Umgebung
    3. Ergebnis: $undefined$
  - Beispiel: $define$
  - Scheme hat sehr wenig Sonderformen (im Gegensatz zu C, Pascal, Modula)

2.1.4 Prozeduren: Verfahren zur Wertberechnung abstrahieren

- Beispiel: Berechnung der Quadratfläche ($2 \rightarrow 4$ etc.)
- immer gleiches Schema: $F(x) = x \cdot x$
- Definition in Scheme: `(define (<name> <formale Parameter>) <Rumpf>)`
- nicht unterscheidbar von elementaren Prozeduren
- dienen als Bausteine für komplexere Prozeduren

```
> (define (quadrat x) (* x x))
> (define (qsumme x y) (+ (quadrat x) (quadrat y)))
> (define (f a) (qsumme (+ a 1) (* a 2)))
> (f 5)
136
```

2.2 Programmauswertung mit dem Substitutionsmodell

2.2.1 informelle Definition des Substitutionsmodells

- Werte des Rumpf aus, nachdem jeder formale Parameter durch das entsprechende Argument ersetzt (subsitutiert) wurde.
- Beispiel: Auswertung `(f 5)`:
  Rumpf von f: `(quadratsumme (+ a 1) (* a 2))`
  Ersetze a durch 5: `(quadratsumme (+ 5 1) (* 5 2))`
  Wende Quadratsumme an: `(+ (quadrat 6) (quadrat 10))`
  Wende Quadrat an: `(+ (* 6 6) (* 6 6))`
  Ergebnis: 136
- in der Praxis keine textuelle Ersetzung, sondern Zugriff über *lokale Umgebung* (→ Kapitel 3)
- Substitutionsmodell versagt später bei veränderbaren Daten, aber zunächst sehr nützlich

2.2.2 Definitionen für Programmteile

1. Terme/Ausdrücke enthalten Funktionssymbole, Zahlen, Variablen
2. Zusammenfassung aller Funktionssymbole in einer Signatur: Eine *Signatur* Σ ist die Menge von Paaren der Form f/n
   - f ist ein Name und n eine ganze Zahl ≥ 0.
   - Falls $f/n \in \Sigma$ und $n > 0$, dann heißt f *n-stelliges Funktionssymbol*;
   - Eindeutigkeit: $f/n, f/m \in \Sigma \Rightarrow n = m$ (nicht unbedingt notwendig, z.B. überladene Bezeichner).
   - Falls $c/0 \in \Sigma$, heißt c *Konstante*, Zahlen werden als Konstaten ersetzt.
   - Häufig: Signaturen mit Typen, hier aber nicht (weil Scheme untypisiert)
   - Beispiel für Signatur: $\Sigma = \{+/2, -/2, */2, quadrat/1, quadratsumme/2\}$
3. Sei Σ die Signatur und X die Menge von Variablen (disjunkt zu Σ). Die *Menge der Terme* $T_\Sigma(X)$ über Σ und X ist die kleinste Menge, die folgende Eigenschaften erfüllt:
   1. $\forall c/0 \in \Sigma : c \in T_\Sigma(X)$

   2. $\forall x \in X : x \in T_\Sigma(X)$
   3. $\forall f/n \in \Sigma, t_1...t_n \in T_\Sigma(X) : (ft_1...t_n) \in T_\Sigma(X)$

   - Terme sind Konstanten, Variablen und Kombinationen mit richtiger Argumentzahl.
   - ⋆ *Induktive Definition*: Rückführung von komplexen Fällen auf einfachere (zentral in der Informatik)

   Beispiel:

   - $\Sigma = \{+/2, -/2, */2, quadrat/1, quadratsumme/2, 1, 2, 3, ...\}$
   - $X = \{x, y, z\}$

4. Ein *Programm* ist ein Tripel (Σ, X, R) mit:

   1. Signatur Σ
   2. Variablenmenge X disjunkt zu Σ
   3. Menge von Regeln R in der Form $(fx_1...x_n) \rightarrow r$ mit $f/n \in \Sigma; x_1, ..., x_n \in X$ und $r \in T_\Sigma(\{x_1, ..., x_n\})$

   - häufig: höchstens eine Regel für jede Funktion (Funktionen ohne Regeln: Konstruktoren für Datenstrukturen, siehe (3))
   - rechte Regelseite (r) darf nur Parameter der Regel enthalten
   - manchmal: linke Seite ist ein Term ($\rightarrow$ patternmatching, Termersetzungssystem)
   - Zusammenhang zu Scheme: Definitionen `(define (f (x1 ... xn) r)` entspricht obiger Regel

5. Eine *Substitution* σ ist eine Funktion $\sigma : X \mapsto T_\Sigma(X)$, die jeder Variablen einen Term zuordnet. Anwendung einer Substitution Σ mit Term $t \in T_\Sigma(X)$ geschrieben $t\sigma$, ist induktiv wie folgt definiert:

   1. Falls $t \in X$, dann $t\sigma := \sigma(t)$
   2. Falls $t/0 \in \Sigma$, dann $t\sigma := t$
   3. Falls $t = (ft_1...t_n)$, dann $t\sigma = (ft_1\sigma...t_n\sigma)$

   - Beispiel: $\sigma(x) = 2, \sigma(y) = (+z1), \sigma(z) = z$, dann $(*x(+yz))\sigma = (*2(+(+z1)z))$.

6. Eine *Position* identifiziert eindeutig eine Stelle im Term. Ist t ein Term, dann ist $Pos(t)$ die Menge aller Positionen im Term t.

- üblich: Position ≈ Folge natürlicher Zahlen:
- Leere Folge <>: Position der Wurzel (Einstiegspunkt)
- Nichtleere Folge $< p_1, p_2, ..., p_n >$: Im Argument p_1 die Position $< p_2, ..., p_n >$
- Beispiel: $t =$ `(* x (+ y z))`; $Pos(t) = \{<>, <1>, <2>, <2, 1>, <2, 2>\}$.

7. Ist $p \in Pos(t)$, dann heißt $t|_p$ *Teilterm* von t an der Stelle p und ist wie folgt definiert:

   1. $t|_{<>} = t$
   2. $t = (ft_1...t_m) \rightarrow t|_{<p_1,p_2,...,p_n>} = t_{p_1}|_{<p_2,...,p_n>}$
   - Beispiel: $t =$ `(* x (+ y z))`; $t|_{<2,2>} = (+yz)|_{<2>} = z|_{<>} = z$

8. Ist $p \in Pos(t)$ und t' Term, dann heißt $t[t']_p$ *Ersetzung* von $t|_p$ durch t' an der Stelle p. Dies ist folgendermaßen definiert:

   1. $t[t']_{<>} = t'$
   2. $t = (ft_1...t_n) \rightarrow t[t']_{<p_1,p_2,...,p_n>} = (ft_1...t_{p-1}t_{p_1}[t']_{p_2,...,p_n}t_{p+1}...t_m)$
   - Beispiel: $t =$ `(* x (+ y z))`; $t' = (+zy)$; $t[t']_{<2>} =$ `(* x (+ z y))`

2.2.3 formelle Definition des Substitutionsmodells

Definition: Sei $P = (\Sigma, X, R)$ ein Programm.

1. Ein *Berechnungsschritt* $t_1 \Rightarrow t_2$ ist definiert, falls $p \in Pos(t_1), l \rightarrow r \in R$ und Substitution σ mit

   1. $\sigma(l) = t_1|_p$ (σ ersetzt formale Parameter in der durch aktuelle Terme)
   2. $t_2 = t_1[\sigma(r)]_p$ (Ergebnis ist Ersetzung durch rechte Regelseite mit Parametersubstitution)

2. Eine *Berechnung* $t_1 \Rightarrow^* t_2$ ist eine (evtl. leere) Folge von Berechnungsschritten, d.h. es existieren $n \geq 0$ und $s_1, ..., s_n \in T_\Sigma(X)$ mit $t_1 = s_1$, $t_2 = s_n$ und $s_1 \Rightarrow s_{i+1}$ für $i = 1, 2, ...n - 1$.

3. Auswertung vordefinierter Funktionen: Konzeptionell durch unendliche Menge von Regeln:
   `(+ 0 0)` → `0`

```
(+ 1 0) → 1
(+ 0 1) → 1
(+ 1 1) → 2
usw.
```

4. Ein Term t heißt *in Normalform*, falls es keinen Term t' gibt mit der Eigenschaft $t \Rightarrow t'$ (d.h. falls kein Berechnungsschritt möglich ist). Somit ist die Normalform meist das Endergebnis bzw. ein Wert, kein Zwischenergebnis.

- Randbemerkung: i.d.R. kommen in Berechnungstermen keine Variablen mehr vor (d.h. $t_1 \in T_\Sigma(\emptyset)$).

- Beispiele:

```
(quadrat x) → (* x x) ∈ R
(quadrat (* 1 2)) ⇒ (* (+ 1 2) (+ 1 2))
mit p =<>; σ(x) = (+ 1 2)

(* 3 (quadrat 4)) ⇒ (* 3 (* 4 4)
mit p =< 2 >, σ(x) = 4

(quadrat (+ 1 2))
⇒ (* (+ 1 2) (+ 1 2))
⇒ (* 3 (+ 1 2))
⇒ (* 3 3)
⇒ 9

(quadrat (+ 1 2))
⇒ (quadrat 3)
⇒ (* 3 3)
⇒ 9
```

2.2.4 unterschiedliche Berechnungswege

Es existieren z.T. unterschiedliche Berechnungswege, wenn es verschiedene Funktionssybole in einem Term gibt:

1. Auswertung in applikativer Reihenfolge: Werte die Argumente vor der Anwendung einer Prozedur aus (formal: Falls $t_1 \Rightarrow t_2$ mit $t_2 = t_1[\sigma(r)]_p$, dann ist $t_1|_p = (f s_1 ... s_n)$ und jedes s_1 ist in Normalform)

2. Auswertung in Normalordnung: ersetze alle Prozeduren so weit, bis nur Elementare Prozeduren vorhanden sind und werte dann diese aus (formal: Falls $t_1 \Rightarrow t_2$ mit $t_2 = t_1[\sigma(r)]_p$, dann ist $t_1|_p$ äußerster Teilterm, an den ein Schritt möglich ist, d.h. falls $p =< p_1, p_2, ..., p_n >$, dann ist ein Schritt an $< p_1, ..., p_i >$ mit $i < n$ nicht möglich), beginnend von außen nach innen.

Beispiel: `(quadratsumme (+ 5 1) (* 5 2))`

1. ```
   (quadratsumme 6 10)
   (+ (quadrat 6) (quadrat 10))
   (+ 36 100)
   136
   ```

2. ```
   (+ (quadrat (+ 5 1)) (quadrat (* 5 2)))
   (+ (* (+ 5 1) (+ 5 1)) (quadrat (* 5 2)))
   (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
   136
   ```
   Beobachtung: `(+ 5 1)` und `(* 5 2)` werden zweifach ausgewertet. Trotzdem: Ergebnisse in beiden Reihenfolgen gleich.

Beachte: Die Normalordnung kann ein Ergebnis liefern, wo die applikative Ordnung nicht terminiert, daher ist die Normalordnung *berechnungsstärker*. Die Normalordnung liefert die Normalform, falls diese existiert. Aber die applikative Ordnung ist im allgemeinen effizienter, SCHEME arbeitet im Prinzip mit applikativer Ordnung.

2.3 weitere Sonderformen

2.3.1 cond

Beispiel Absolutbetrag: Falls x negativ ist, gebe $-x$ aus, andernfalls x. Notation mit der Sonderform `cond` in SCHEME:

```
(cond (<p1> <a1>)
      (<p2> <a2>)
      ...
      (<pn> <an>))
```

<p> ist eine Bedingung/Prädikat, also ein Ausdruck mit dem Ergebnis `#t` oder `#f`. Bedeutung: Werte nacheinander die Prädikate aus, falls der Wert von <pi> gleich `#t` ist, dann ist das Ergebnis <ai>, falls keine Bedingung zutrifft, ist das Ergebnis undefiniert (`#unspecified`). Als letze Bedingung kann auch `else` verwendet werden. Beispiel: `(define (abs x) (cond ((> x 0) x) (else (- x))))`

2.3.2 `if`

Eine weitere (eigentlich nicht notwendige) Sonderform zur Fallunterscheidung ist `if`: `(if <p> <a1> <a2>)` $\approx$ `(cond (<p> <a1>) (else <a2>))`.

2.3.3 `and`, `or`, `not`

- `(and <p1> <p2>)` $\Leftrightarrow p_1 \wedge p_2$
- `(or <p1> <p2>)` $\Leftrightarrow p_1 \vee p_2$
- `(not <p>)` $\Leftrightarrow \neg p$

2.3.4 Beispiel: Quadratwurzelberechnung nach Newton

Mathematische Wurzelfunktion $y = \sqrt{x}$ mit $y \geq 0$ und $y^2 = x$ ist eine präzise Funktion, aber nicht effektiv und damit kein Algorithmus. Als Berechnungsverfahren dient das *Newtonsche Iterationsverfahren.* Gegeben ist eine Schätzung für y, ausgegeben wird als bessere Schäzung der Mittelwert aus y und $\frac{x}{y}$.
Formulierung:

```
(define (wurzel-iter y x) (if (gut-genug? y x) y (wurzel-iter (verbessern
y x) x)))
(define (verbessern y x) (mittelwert y (/ x y)))
(define (mittelwert x y) (/ (+ x y) 2))
(define (gut-genug? y x) (< (abs (- (quadrat y) x)) 0.001))
(define (quadrat x) (* x x))
(define (abs x) (if (< x 0) (- x) x))
(define (wurzel x) (wurzel-iter 1 x))
```

Anmerkungen:

- numerische Anwendung möglich
- keine iterativen Konstrukte (Schleifen) notwendig
- lediglich Prozeduraufrufe, also Schleifen $\approx$ Rekursion
- Effizienz: kein Problem (Implementierung als Schleife)

Vorgehensweise:

- Problem zerlegen in Teilprobleme (logische Einheiten)
- Teilprobleme als Prozeduren implementieren
- Prozeduren erfüllen bestimmte Aufgaben, das *wie* ist dabei uninteressant (prozedurale Abstraktion, *BlackBox*)

- top-down- bzw. -bottom-up-Entwurf, Praxis: Mischung aus beiden Verfahren

2.4 Namensgebung

- Prozedurnamen sind wichtig (Abstraktion), auch um Prozeduren unabhängig zu entwickeln
- Namen von lokalen Variablen sind nur intern relevant und damit relativ unwichtig
- formale Parameter sind *gebundene Variablen*, Gegeteil: *freie Variablen* (Bspl `abs`, `quadrat`, `gut-genug?`)
- *Geltungsbereich, Gültigkeitsbereich*: Ausdruck(smenge), in der Variablenbindung bekannt sind (Bspl: für `(define (f x) <e>)` ist `<e>` der Geltungsbereich von `x`)
- *Blockstruktur:* Bisher: alle Funktionen etc. auf gleicher Ebene definiert, daher gleich bekannt, aber der Name kann von anderen Programmierern nicht mehr genutzt werden (bzw. diese definieren es um, es kommt zu Problemen). Daher: Blockstruktur, d.h. Hilfsoperationen »verstecken«:

```
(define (wurzel x)
. (define (wurzel-iter y x) (...))
. (define (verbessern y x) (...)) ...
. (wurzel-iter 1 x))
```

- innere Definitionen sind damit außerhalb von `wurzel` unsichtbar, bessere Modularisierung (`wurzel` als BlackBox)
- Vorteil: Wurzel-Parameter `x` gültig in allen lokalen Definitionen, d.h. Übergabe der Variablen unnötig:

```
(define (wurzel x)
        (define (gut-genug? y)
                (< (abs (- (quadrat y) x))) 0.0001)
        (define (verbessern y)
                (mittelwert y (/ x y)))
        (define (wurzel-iter y)
                (if (gut-genug? y) y
                    (wurzeliter (verbessern y))))
(wurzel-iter 1))
```

- *Lexikalische (statische) Bindung*: Namen beziehen sich auf den nächsten umgebenden Block, in dem sie gebunden sind. Damit sind gleiche Namen für unterschiedliche Objekte möglich.

2.5 Prozeduren und Prozesse

- *Prozeduren:* Muster zur Berechnung; *Prozesse:* dynamischer Rechenablauf, gesteuert durch Prozeduren; *Frage:* Ressourcenverbrauch (Speicher, Zeit) von Prozessen?

2.5.1 Iteration, Rekursion etc.

- *linearer rekursiver Prozeß*
  - Beispiel Fakultät: $n! = n \cdot (n-1) \cdot (n-2) \cdot ...2 \cdot 1 = n \cdot (n-1)!$ (falls $n > 1$).
  - `(define (fak n) (if (= n 1) 1 (* n (fak (- n 1)))))`
  - Berechnung von `(fak 4)`:

    ```
    (fak 4)
    (* 4 (fak 3))
    (* 4 (* 3 (fak 2)))
    (* 4 (* 3 (* 2 (fak 1))))
    (* 4 (* 3 (* 2 1)))
    (* 4 (* 3 2))
    (* 4 6)
    24
    ```
  - Beobachtung: »verzögert« Multiplikation

- *linear iterativer Prozeß*
  - Beispiel Fakultät: Multipliziere 1*2, das Ergebnis mit 3, das Ergebnis mit 4, ..., das Ergebnis mit n.
  - Parameter in jedem Schritt: Zähler (1 bis n), Produkt (1 bis n!), n für den Abbruch
  - ```
    (define (fak n) (fak-iter 1 1 n))
    (define (fak-iter p z max)
    ........(if (> z max) p
    ......................(fak-iter (* p z) (+ z 1) max)))
    ```
  - Berechnung von `(fak 4)`:

    ```
    (fak 4)
    (fak-iter 1 1 4)
    (fak-iter 1 2 4)
    ```
    ```

```
(fak-iter 2 3 4)
(fak-iter 6 4 4)
(fak-iter 24 5 4)
24
```

  - Beobachtung: eine feste Anzahl von Variablen beschreibt vololständig den Zustand eines jeden Schrittes (Zustandsvariablen)

  ⇒ beide Prozesse haben von der Größenordnung her die gleiche Länge (linearer Prozess); in der ersten Version ist der Platzbedarf linear zur Eingabe, Platzbedarf ist bei der zweiten Version konstant.

  ⊗ rekursiver Prozeß $\neq$ rekursive Prozedur!

- *Baumrekursion*
  - Beispiel Fibonacci-Zahlen: Jede Zahl ist die Summe der beiden vorhergehenden (0 1 1 2 3 5 8 13 21 usw.)
  - $fib(n) = fib(n-1) + fib(n-2)$, $fib(0) = 0 \wedge fib(1) = 1)$
  -
    ```
    (define (fib n) (cond ((= n 0) 0) ((= n 1) 1)
    (else (+ (fib (- n 1)) (fib (- n 2))))))
    ```
  - Problem: Redundanz (`(fib 2)` doppelt berechnet)! Damit: schlechte Laufzeit, letztendlich werden $fib(n)$ Einsen addiert, damit eine *exponentielle Laufzeit*
  - Abschätzung $fib(n) = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}\right]$ (mit Gaußklammer [.]).

  ⇒ Gefahr von Ineffizienz, trotzdem häufig geeignet, insbesondere bei sog. hierachischen Datenstrukturen)

- *iterativer Prozeß*
  - Schrittweises Aufsummieren der Zahlen
  - Parameter: a, b (letzte und vorletzte Zahl)
  - in jedem Schritt: $a = a + b$ und gleichzeitig $b = a$
  -
    ```
    (define (fib n) (fib-iter 1 0 n))
    (define (fib-iter a b z) (if (= z 0)
    ............................0
    ............................(fib-iter (+ a b) a (- z 1))))
    ```
  - lineare Laufzeit, konstanter Speicherbedarf

⇒ *Iteration* ist effizient, benötigt genaue Überlegungen zu den Zustandsvariablen; *Rekursion* ist mit der Gefahr der Ineffizienz behaftet, aber natürlichere Formulierung

2.5.2 Größenordnungen

- wichtig: Vergleich von Resourcen-Verbrauch (Zeit, Speicherbedarf)
- gemessen in Abhängigkeit von einem Parameter n (Genauigkeit bei Wurzel, n-te Zahl bei Fibonacci, Größe der Matrix etc.)
- $R(n)$: Betrag der Resourcen des Prozesses, um Problem der Größe n zu lösen
- exakt schwer meßbar (abhängig von Implementierung)

⇒ $R(n)$ hat Größenordnung $O(f(n))$, falls es Konstanten c und n_0 gibt (unabhängig von n) mit $R(n) \leqq c \cdot f(n)$ für alle $n \geqq n_0$.

- Beispiele:
  - Linearer rekursiver Fakultätsprozeß: Zeit $O(n)$, Speicher $O(n)$
  - Iterative Version: Zeit $O(n)$, Speicher $O(1)$ (konstant!)
  - Baumrekursion bei Fibonacci: Zeit $O(\phi^n)$, Speicher $O(n)$
- etwas vereinfacht, z.B. die Multiplikation hat laut Annahme konstanten Aufwand, teilweise ist dies aber tatsächlich abhängig von der Zahlengröße, gibt aber grobes Maß für die Effizienz von Algorithmen
- Prozesse mit exponentieller Laufzeit $O(c^n)$ in der Praxis unbrauchbar, für manche Probleme gibt es aber offenbar keine effizienteren Algorithmen; Hilfsmittel/Work-Around: Heuristiken

2.5.3 Beispiele für effiziente Algorithmen

Beispiel Potenzrechnung (Eingabe b und n, Ausgabe b^n):

- Unmittelbare Lösung: rekursive Definition
  `(define (pot b n) (if (= n 0) 1 (* b (pot b )- n 1)))`
  Zeit und Speicher: $O(n)$
- iterative Version:

```
(define (pot b n)
        (define (pot-iter z p)
                (if (= z 0) p
                    (pot-iter (- z 1) (* p b))))
(pot-iter n 1))
```

Zeit: $O(n)$, Speicher $O(1)$

- schneller: Quadratbildung (z. B. $b^8 = ((b^2)^2)^2$)
  Allgemein: $b^n = (b^{\frac{n}{2}})^2$ falls n gerade, sonst $b^n = b \cdot b^{n-1}$

  ```
  (define (fixpot b n)
          (cond ((= n 0) 1)
                ((gerade? n) (quadrat (fix-pot b (/ n 2))))
                (else (* b (fixpot b (- n 1))))))
  (define (gerade? n) (= (remainder n 2) 0))
  ```

  Anzahl der Multiplikationen: $\log_2 n$, Zeit $O(\log n)$

Riesenunterschied zu $O(n)$ bei großen Zahlen: $O(n)$ ist beim letzten Algorithmus für $n = 1000$ mit nur 14 Multiplikationen durchzuführen.

Beispiel gröster gemeinsamer Teiler $ggT(a, b)$:

- Probieren aller Zahlen als Teiler aus (von 1 bis 16), Aufwand linear
- Zerlege a und b in Primfaktoren finde gleiche Primfaktoren
- Euklidscher Algorithmus: $ggT(a, b) = ggT(b, a \mod b)$; diese Reduktion läuft, bis $ggT(x, 0) = x$ resultiert.

  ```
  (define (ggt a b)
          (if (= b 0)
              a
              (ggt b (remainder a b))))
  ```

  Zeitberechnung mit *Satz von Lamé (1845)*: Wenn der euklidsche Algorithmus k Schritte benötigt, ist $min(a, b) \geqq Fib(k)$. $\Rightarrow$ Sei $n = min(a, b)$, der Algorithmus benötigt k Schritte, wobei $n \geqq Fib(k) \approx \phi^k$, d.h. Schrittzahl ist kleiner als $\log_\phi(n)$
  iterativer Prozeß, Zeit $O(\log n)$, Speicher $O(1)$

Beispiel Finden von Primzahlen (siehe Kryptographie etc., public/private-Key-Verfahren mit tatsächlich *wirklich* richtig echt RIESIG **großen** Primzahlen mit mehr als 100 Stellen)

- klassisch: Suche nach Teilern, Teile durch alle ganzen Zahlen von 2 bis $\sqrt{n}$, wenn d Teiler ist, ist auch $\frac{n}{d}$ Teiler.

```
(define (kleinster-teiler n)
        (define (finde-teiler test)
                (cond ((> (quadrat test) n) n)
                      ((teilt? test n) test)
                      (else (finde-teiler (+ test 1)))))
        (finde-teiler 2))
(define (teilt? a b) (= (remainder b a) 0))
(define (primzahl n) (= (kleinster-teiler n) n))
```

- Ergebnis aus Zahlentheorie: *kleiner Fermatscher Satz*: Wenn n Primzahl ist, dann $0 < a < n \Rightarrow a^n \mod n = a$. Falls n also keine Primzahl ist, dann gibt es gibt es mit großer Wahrscheinlichkeit viele Zahlen $0 < a < n$ mit $a^n \mod n \neq a$. Teste also $a^n \mod n = a$ für zufällige Zahlen a, je mehr positive Tests, um so wahrscheinlicher ist n eine Primzahl. Schnelle Funktion für Potenz mit Modulo:

```
(define (potmod b e n)
        (cond ((= e 0) 1)
              ((gerade? e) (remainder (quadrat (potmod b (/ e 2) n)) n))
              (else (remainder (* b (potmod b (- e 1) n)) n))))
```

  Zum testen: Wähle Zufallszahl a mit $2 \leqq a \leqq n-1$, teste $a^n \mod n = a$.

```
(define (fermat-test n)
        (define a (+ 2 (random (- n 2))))
(= (potmod a n n) a))
```

  Dieser Test soll x-mal ausgeführt werden:

```
(define (fast-prime? n x)
        (cond ((= x 0) \#t)
        ((fermat-test? n) (fast-prime? n (- x 1)))
        (else \#f)))
```

Besonderheit: *Probalistische Methode*: falls Ergebnis *true* ist die Zahl sicher keine Primzahl, andernfalls ist sie wahrscheinlich eine Primzahl - je mehr Tests, umso höher die Wahrscheinlichkeit.
Argument für probalistische Algorithmen: Auch völlig korrekte Algorithmen können eventuell falsch sein, es gibt eben ohnehin nur mit einer gewissen Wahrscheinlichkeit eine richtige Lösung.

2.5.4 Prozeduren höherer Ordnung

Bisher: Parameter und Ergebnisse sind Zahlen, boolsche Werte, ... *Rightarrow* Abstraktion ähnlicher Verfahren auf Zahlen

Neu: Parameter und Ergebnisse dürfen selbst Prozeduren sein: Prozeduren höherer Ordnung. Motivation: häufig ähnliche Prozeduren nach gleichem Schema

Beispiel: Summe der ganzen Zahlen und der Quadratzahlen zwischen a und b und Näherungslösung für $\frac{\pi}{8} = \frac{1}{3 \cdot 5} \cdot \frac{1}{7 \cdot 9} \cdot \frac{1}{11 \cdot 13} \cdot ...$:

```
(define (ganz-summe a b)
        (if (> a b)
            0
            (+         a          (ganz-summe (+ a 1) b))))
(define (quad-summe a b)
        (if (> a b)
            0
            (+ (quadrat a)        (quad-summe (+ a 1) b))))
(define (pi-summe a b)
        (if (> a b)
            0
            (+ (/ 1 (* a (+ a 2))) (pi-summe   (+ a 4) b))))
```

Verallgemeinerung:

```
(define (summe fun naechst a b)
        (if (> a b)
            0
            (+ (fun a) (summe fun naechst (naechst a) b))))
```

Analogie zur Mathematik: $\sum_{n=a}^{b} f(n)$ mit den Parametern a, b, f. Damit ist die Quadratsumme nur ein Spezialfall der Summe:

```
(define (inc1 a) (+ a 1))
(define (inc4 a) (+ a 4))
(define (ganz-summe a b) (summe a inc1 a b))
(define (quad-summe a b) (summe quadrat inc1 a b))
(define (pi-summe a b)
        (define (pi-fun x) (/ 1 (* x (+ x 2))))
        (summe pi-fun inc4 a b))
```

2.5.5 λ-Abstraktionen

Um Funktionen nicht einzeln vorher definieren zu müssen, gib es die *Lambda-Abstraktion.* `pi-fun` etc. werden nicht explizit definiert, sondern anonym verwendet, möglich durch eine neue Sonderform Lambda: `(lambda (x) (+ x 4))` (Wert: „eine Funktion, die vier dazu addiert“)

```
(define (pi-summe a b)
        (summe (lambda (x) (/ 1 (* x (+ x 2))))
               (lambda (x) (+ x 4))
               a b))
```

Allgemein: `(lambda (<formale Prarmeter>) <Rumpf>)`, Wert: Prozedur. Statt Prozedurnamen können immer auch λ-Definitionen verwendet werden: statt `(mittelwert x y)` kann man folgendes verwenden:

```
 ((lambda (x y) (/ (+ x y) 2)) 2 6)
```

Das λ-Kalkül (Church 1941) ist Grundlage der funktionalen Programmierung.

2.5.6 Sonderform `let`

Statt `define` (lokal) auch Sonderform `let`:

```
(let ((var1 ausdr1)
      (var2 ausdr2)
      ...
      (varn ausdrn))
     rumpf)
```

Leseweise: Lasse var_1 den Wert von $ausdr_1$ haben, ... lasse var_n den Wert von $ausdr_n$ haben im $rumpf$.
Konzeptionell nicht neu, sondern `let` ist äquivalent zu lokalen Prozeduren im Rumpf („sytaktischer Zucker"für praktische Schreibweise).
Alle Einzelterme in let werden im Prinzip gleichzeitig ausgewertet:

```
(define x 5)
(let ((x 2) (y x)) (+ x y))
```

Dabei benutzt `(y x)` die vorherige Definition von $x = 5$, das `(+ x y)` benutzt $x = 2$.

2.5.7 universelle Berechnungsverfahren

Mathematik: z.B. Nullstellenbestimmung, Differenzieren für (fast) beliebige Funktionen. Implementierung durch Prozeduren hörerer Ordnung, Beispiel Nullstellenbestimmung durch Invervallhalbierung.
Gegeben: beliebige Funktion f, Punkte a, b mit $f(a) < 0 \wedge f(b) > 0$. Idee Intervallhalbierung: Sei x der Mittelwert von a und b. Falls $f(x) > 0$ liegt die Nullstelle in $]a, x[$, bei $f(x) < 0$ liegt die Nullstelle in $]x, b[$. Laufzeit: wegen Halbierung in jedem Schritt: $O(\log(\frac{|b-a|}{T}))$ mit der Toleranzgrenze T (Länge des Zielintervalls). Implementierung:

```
(define (suche f a b)
        (let ((x (mittelwert a b)))
             (if (nah-genug? a b)
                 x
                 (let ((fx (f x)))
                      (cond ((positive? fx) (suche f a x))
                            ((negative? fx) (suche f x b))
                            (else           x))))))
(define (nah-genug? x y) (< (abs (- x y)) 0.0001))
(define (intervall-halb f a b)
        (let ((fa (f a))
              (fb (f b)))
             (cond ((and (negative? fa) (positive? fb))
                     (suche f a b))
                    ((and (positive? fa) (negative? fb))
                     (suche f b a))
                    (else (error "falsche Vorzeichen")))))
```

2.5.8 Prozeduren als Ergebnis

Möglich, falls Ergebnis ein λ-Ausdruck ist, z.B. die Funktion $differenzieren(f)$ liefert eine neue Funktion f' als Ergebnis (Beispiel: $\frac{d(x^2)}{dx} = 2 \cdot x$).
Numerische Näherung: $\frac{d(f(x))}{dx} = \frac{f(x+dx)-f(x)}{dx}$ für kleine dx:

```
(define (ableitung f dx)
        (lambda (x) (/ (- (f (+ x dx)) - (f x)) dx)))
```

Anwendungsbeispiele:

```
((ableitung quadrat 0.0001) 4.0)
```

Beispiel: Komposition von Funktionen: $(f \circ g)(x) = f(g(x))$

```
>(define (komp f g) (lambda (x) (f (g x))))
>((komp quadrat quadrat) 4)
256
>((ableitung (komp quadrat quadrat) 0.0001) 3.0)
108.054
```

3 Abstraktion mit Daten

3.1 Einführung

Bisher im Prinzip nur numerische Daten, nun auch komplexere Daten, also z.B. zusammengesetzte Daten; Abstraktion mit

- Prozeduren: interne Berechnung unsichtbar
- Daten: interne Darstellung unwichtig

Abstrakte Daten:

- Anwender: kennt die innere Sturktur nicht, benutzt nur festgelegte Operationen
- Implementierer: wählt eine beliebige (passende) Struktur und stellt die Operationen (Schnittstelle) bereit

Vorteile dieser Methode:

- interne Implementierung leicht austauschbar (Effizienz)
- mehr Modularität
- bessere Verständlichkeit der Programme

Zentrales Element ist die *Schnittstelle:*

- *Selektoren:* Extraktion von Teilinformationen
- *Konstruktoren:* Konstruktion komplexer Daten
- *Operatoren:* Verknüpfung dieser Daten

Beispiel: Rationale Zahlen

- definiert durch ganzzahlige Zähler und Nenner
- Konstruktor: `(konstr-rat z n)`
- Selektoren: `(zaehler r)`, `(nenner r)` etc.
- arithmetische Operationen:

```
(define (+rat x y)
        (konstr-rat (+ (* (zaehler x) (nenner y))
                       (* (zaehler y) (nenner x)))
                    (*    (nenner x)  (nenner y))))
(define (*rat x y)
        (konstr-rat (* (zaehler x) (zaehler y))
                    (* (nenner  x) (nenner  y))))
```

Darstellung der rationalen Zahlen als Paar von zwei Zahlen:

- Konstruktor: `(cons x y)` (liefert im Prinzip $(x|y)$)
- Selektoren: `(car p)` liefert erstes Element, `(cdr p)` liefert zweites Element

Genaue Implementierung unwichtig, wichtig sind nur Eigenschaften (Gesetze): [1]

```
(car (cons x y)) = x
(cdr (cons x y)) = y
```

Idee der Datenabstraktion: spezifiziere nur die Eigenschaften der Daten, Beispiel: Implementierung rationaler Zahlen

```
(define (konstr-rat x y) (cons x y))
(define (zaehler r) (car r))
(define (nenner  r) (cdr r))
(define (display-rat r) (display (zaehler r))
                        (display "/")
                        (display (nenner r))
                        (newline))
(define (konstr-rat-gekuerzt x y)
        (let ((g (ggt x y))
              (cons (/ x g) (/ y g)))))
```

Konstruktionsmethode: Schichtenentwurf von Programmen

- *Unterste Schicht:* Elementare Objekte der Programmiersprache
- *Oberste Schicht:* Anwendungsobjekte
- wichtig: kein „Durchgreifen“durch Schichten, sondern Schnittstellen benutzen (*Abstraktionsbarrieren*)

[1] „Objekte sind auch nur Prozeduren, die Nachrichten verstehen!“

Schichten beim Beispiel rationale Zahlen:

1. cons, car, cdr
2. konstr-rat, zaehler, nenner
3. +rat, *rat
4. Anwendungsprogramm

Vorteile:

- Implementierung einzelner Schichten austauschbar
- Beherrschung der Entwurfskomplexität
- Verifikation einzelner Schichten

Abstrakte Datentypen: beschrieben durch Konstruktoren, Selektoren (und Operationen) und Gesetze
Beispiel Datenpaare: Funktionen cons, car, cdr und die beiden oben genannten Gesetze
Beispiel rationale Zahlen: Funktionen und Gesetz

```
(= (/ (zaehler (konstr-rat x y))
      (nenner  (konstr-rat x y)))
   (/ x y))
```

Abstrakt heißt dabei, daß keine konkrete Implementierung festgelegt ist (z.B. kann die Kürzung bei rationalen Zahlen eingebaut oder weggelassen werden).

Andere Implementierung von Paaren:

```
(define (cons x y)
        (define (zuteilen m)
                (cond ((= m 0) x)
                      ((= m 1) y)
                      (else (error "Falsches Argument"))))
        zuteilen)
(define (car p) (p 0))
(define (cdr p) (p 1))
```

ungeöhnliche Implementierung, aber sie erfüllt alle Gesetze für Paare
⇒ Sprachen mit Prozeduren höherer Ordnung benötigen nicht unbedingt Datenstrukturen
Definition: Ein *abstrakter Datentyp* (ADT) (Σ, X, E) besteht aus

- einer Signatur Σ (Operationen des Datentyps)
- einer Variablenmenge X disjunkt zu Σ
- einer Menge von Gleichungen E der Form $t = t'$, wobei $t, t' \in T_\Sigma(X)$ Terme sind

Wichtig: Gleichungen sind nicht die Regeln, sondern: Implementierung muß Gleichungen erfüllen, Gleichungen sind im Gegensatz zu Regeln z.T. nicht ausführbar

Definition: Ein Programm (Σ, X, R) ist eine Implementierung eines abstrakten Datentyps (Σ, X, E), falls diese alle Gleichungen in E erfüllt, d.h. falls für jede Gleichung $(t = t') \in E$ und alle Substitutionen $\sigma : X \to T_\Sigma(\{\})$ gilt: $\sigma(t) = t_1 \Leftrightarrow t_2 \Leftrightarrow ... \Leftrightarrow t_n = \sigma(t')$, wobei $n \geq 0$. Das heißt intuitiv: alle Anwendungen linker und rechter Gleichungsseite müssen durch beliebige Berechnungsschritte ineinander überführt werden können.

3.2 Hierachische Datenstrukturen

Graphische Darstellung: Kasten-Zeiger-Diagramme (siehe Buch); in Scheme: `(list a b c)` entspricht `(cons a (cons b (cons c)))`; `car` und `cdr` liefern das erste Element bzw. die Restliste. Um das n-te Element einer Liste zu bekommen:

```
(define (n-tes n l)
        (if (= n 0)
            (car l)
            (n-tes (- n 1) (cdr l))))
```

Anhängen von Listen aneinander:

```
(define (append l1 l2)
        (if (null? l1)
            l2
            (cons (car l1)
                  (append (cdr l1) l2))))
```

Transformation einer Liste durch Anwendung einer Funktion auf alle Elemente:

```
(define (mapcar f l)
        (if (null l)
            ()
            (cons (f (car l))
                  (mapcar (cdr l)))))
```

Länge einer Liste:

```
(define (laenge l)
        (if (null? l)
            0
            (+ 1 (laenge (cdr l)))))
```

Erweiterungen von `car` bzw. `cdr`:

```
(define (cadr  x) (car (cdr x)))
(define (caddr x) (car (cdr (cdr x))))
```

3.2.1 Darstellung von Baumstrukturen

Elemente von Baumstrukturen: Wurzel, innere Knoten mit Sohnknoten und Blättern, mögliche Darstellung von Knoten durch Listen: Wann ist ein Baum ein Blatt?

```
(define (leaf? x) (not (pair? x)))
```

Anzahl der Blätter in einem Baum?

```
(define (blattanzahl b)
        (cond ((null? b) 0) ; Knoten hat keine Soehne
              ((leaf? b) 1) ; Knoten ist Blatt
              (else (+ (blattanzahl (car b))
                       (blattanzahl (cdr b))))))
```

3.2.2 Quotieren

Wünschenswert: neben Zahlen auch Symbole als atomare Objekte; Beispiel: Liste mit Symbolen (a, b, c) oder $(Klaus, 4)$ etc. Wird dies als `(list a b c)` definiert, gibt es einen Fehler. Lösung: Quotieren, d.h. im Prinzip in Anführungszeichen setzen. In SCHEME wird ein Apostroph vor nicht auszuwertende Ausdrücke gesetzt:

```
>(define a 1)
>(define b 2)
>(list a b)
(1 2)
>(list 'a 'b)
(a b)
```

Es ist auch möglich, komplexe Ausdrücke zu quotieren:

```
>(cdr '(a b c))
(b c)
```

Notwendig wir eine Möglichkeit zur Prüfung von Gleichheit zweier Symbole: `eq?`; Beispiel: Extrahieren der Restliste einer Liste, die mit einem bestimmten Symbol beginnt (falls nicht vorhanden: false):

```
(define (memq e l)
        (cond ((null? l) #f)
              ((eq? e (car l)) l)
              (else (memq e (cdr l)))))
>(memq 'b '(a b c d))
(b c d)
```

3.2.3 Beispiel: Symbolisches Differenzieren

- nicht numerische, sondern: $ax^2 + bx + c \mapsto 2ax + b$
- eine Motivation für List, Teil großer Systeme (z.B. Maple)
- Hier Funktionen ausgebaut auf: Zahlenkonstanten, Variablen, Summen und Produkte

Darstellung von Funktionen zur Eingabe:

1. Stringdarstellung: $ax + b =$`(a * x + b)`
2. Präfixnotation: $ax + b =$`(+ (* a x) b)` (Vorteil: Prioritäten klar)

Datentyp für die Funktionsdarstellung:

- Konstruktoren: (trivial: Zahlen und Variablen)

  ```
  (define (konstr-summe a1 a2) (list '+ a1 a2))
  (define (konstr-produkt a1 a2) (list '* a1 a2))
  ```

- Prädikate:

  ```
  (define (konstante? x) (number? x))
  (define (variable? x) (symbol? x))
  (define (summe? x) (if (pair? x) (eq? (car x) '+) #f))
  (define (produkt? x) (if (pair? x) (eq? (car x) '*) #f))
  (define (gleiche-variable? v1 v2)
          (and (variable? v1) (variable? v2) (eq? v1 v2)))
  ```

- Selektoren

```
(define (operand1 S) (cadr S))
(define (operand2 S) (caddr S))
```

mathematische Regeln:

- $\frac{dc}{dx} = 0$
- $\frac{dx}{dx} = 1$
- $\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$
- $\frac{d(u \cdot v)}{dx} = u \cdot \frac{dv}{dx} + v \cdot \frac{du}{dx}$

Mögliche Implementierung:

```
(define (ableitung a v)
  (cond ((konstante? a) 0)
        ((variable? a) (if (gleiche-variable? a v) 1 0))
        ((summe? a) (konstr-summe (ableitung (operand1 a) v)
                                  (ableitung (operand2 a) v)))
        ((produkt? a) (konstr-summe (konstr-produkt (operand1 a)
                                                    (ableitung (operand2 a) v))
                                    (konstr-produkt (operand2 a)
                                                    (ableitung (operand1 a) v))))
        (else (error "Ableitung dieser Funktion nicht implementiert"))))
```

Beispiele für Ausgaben:

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

Die Ausgabe ist also richtig, aber nicht optimal (Produkt mit 0 ist immer 0 etc.), Verbesserung: in die Konstruktoren mathematische Regeln einbauen ($x \cdot 0 = 0; x \cdot 1 = x; 0 + x = x$):

```
(define (konstr-summe a1 a2)
        (cond ((and (number? a1) (number? a2)) (+ a1 a2))
              ((number? a1) (if (= a1 0)
                                a2
                                (list '+ a1 a2)))
              ((number? a2) (if (= a2 0)
                                a1
                                (list '+ a1 a2)))
```

```
              (else              (list '+ a1 a2))))
(define (konstr-produkt a1 a2)
        (cond ((and (number? a1) (number? a2)) (* a1 a2))
              ((number? a1) (cond ((= a1 1) a2)
                                  ((= a1 0) 0)
                                  (else (list '* a1 a2))))
              ((number? a2) (cond ((= a2 1) a1)
                                  ((= a2 0) 0)
                                  (else (list '* a1 a2))))
              (else              (list '* a1 a2))))
```

3.2.4 Beispiel: Implementierung von Mengen

Menge: Ansammlung einzelner Objekte (keine Reihenfolge, keine doppelten Elemente), Datentyp mit Operationen *vereinigen*, *schnitt*, *hinzufuegen*, *element?*; Gesetze:

```
(element? x (hinzufuegen x S)) = #t
(element? x (vereinigung S T)) = (or (element? x S) (element? x T))
(element? x (schnitt S T)) = (and (element? x S) (element? x T))
(element? x leer) = #f
(hinzufuegen x (hinzufuegen x S)) = (hinzufuegen x S)
(hinzufuegen x (hinzufuegen y S)) = (hinzufuegen y (hinzufuegen x S))
```

Idee: Menge darstellen als Liste der Elemente (ohne Duplikate)

```
(define (element? x m)
        (cond ((null? m) #f)
              ((equal? x (car m)) #t)
              ( else (element? x (cdr m)))))
(define leer ())
(define (hinzufügen x m)
        (if (element? x m) m (cons x m)))
(define (schnitt m1 m2)
        (cond ((null? m1) leer)
              ((element? (car m1) m2)
               (cons (car m1) (schnitt (cdr m1) m2)))
              ( else          (schnitt (cdr m1) m2))))
(define (vereinigung m1 m2)
        (cond ((null? m1) m2)
              ((element? (car m1) m2)
                    (vereinigung (cdr m1) m2))
              (else (cons (car m1)
```

```
                    (vereinigung (cdr m1) m2)))))
```

Wichtiger Aspekt: Effizienz? n: Mäachtigkeit der Menge(n).

- `element?` $O(n)$
- `hinzufuegen` $O(n^2)$
- `schnitt` $O(n^2)$ wegen `element?` und Rekursion über `m1`

Damit auf jeden Fall zu langsam für große Mengen. *Verbesserung:* Ordne die Elemente in aufsteigender Reihenfolge, Voraussetzung: es existiert eine totale Ordnung auf Elementen; d.h. hier werden nur Zahlen betrachtet. Notwendig: Veränderung der `hinzufügen`-Funktion

```
(define (hinzufügen x m)
        (cond ((null? m) (cons x m))
              ((= x (car m)) m)
              ((< x (car m)) (cons x m))
              (else (cons (car m) (hinzufügen x (cdr m))))))
```

Komplexität: Im Durchschnitt $\frac{n}{2}$ Vergleiche, aber Größenordnung $O(n)$. Verbesserung von `schnitt`: Gleichzeitiges Durchlaufen beider Liste: Vergleiche Anfangselemente. Sind beide gleich, so kommt das Element in den Schnitt; ansonsten kommt das kleinere Element nicht in den Schnitt.

```
(define (schnitt m1 m2)
        (if (or (null? m1) (null? m2)) leer
            (let ((x1 (car m1)) (x2 (car m2)))
                 (cond ((= x1 x2) (cons x1
                                        (schnitt (cdr m1)
                                                 (cdr m2))))
                       ((< x1 x2) (schnitt (cdr m1) m2))
                       ((> x1 x2) (schnitt (cdr m2) m1))))))
```

Komplexität: in jedem Schritt Wegnahme eines Elementes, daher $|m_1| + |m_2|$ Schritte, also $O(n)$ statt $O(n^2)$. Analog dazu ist die Vereinigung. Trotzdem ist diese Implementierung noch relativ aufwendig: `elemenet?` und `hinzufügen`. *Bessere Idee:* Mengen als geordnete binäre Bäume darstellen. D.h. es gibt immer zwei Teilbäume, die so geordnet sein sollen, daß der linke Teilbaum nur kleinere als der Knoten und der rechte Teilbaum nur größere Elemente als der Knoten enthält, die Darstellung ist jedoch nicht eindeutig. Der Vorteil dieser Darstellung: die `element?`-Funktion wird schneller:

1. $x = Wurzel(m)$: fertig
2. $x < Wurzel(m)$: linken Teilbaum nehmen
3. $x > Wurzel(m)$: rechten Teilbaum nehmen

Ein Schritt bedeutet (im Durchschnitt) eine Halbierung der Anzahl der Elemente, daher ist die Komplexität $O(\log n)$. Datentyp: Knoten ist `(eintrag linker-baum rechter-baum)`.

Selektoren:

```
(define (eintrag b) (car b))
(define (linker-baum b) (cadr b))
(define (rechter-baum b) (caddr b))
```

Konstruktor:

```
(define (kontr-baum e l r) (list e l r))

(define (element? x m)
        (if (null? m) #f
            (let ((e (eintrag m)))
                 (cond ((= x e) #t)
                       ((< x e) (element? x
                                 (linker-baum m)))
                       ( else   (element? x
                                 (rechter-baum m)))))))
(define (hinzufügen x m)
        (if (null? m) (konstr-baum x () ())
            (let ((e (eintrag m)))
                 (cond ((= x e) m)
                       ((< x e) (konstr-baum e
                                 (hinzufuegen x
                                  (linker-baum m))
                                 (rechter-baum m)))
                       (else    (konstr-baum e
                                 (linker-baum)
                                 (hinzufuegen x
                                  (rechter-baum m))))))))
```

Komplexität: analog zu `element?`, d.h. $O(\log n)$! Bei Schnitt/Vereinigung: keine effizientere Möglichkeit wie bei geordneten Listen, sondern analog zu ungeordneten Listen ($O(n \cdot logn)$). Die logarithmischen Laufzeiten gelten

nur für ausgewogene Bäume! Lösung: Bäume ausgewogen halten, siehe z.B. Vorlesung Informatik II mit AVL-Bäumen oder Red-Black-Trees).

Anwendung von Mengen:

- in Datenbanksystemen, Elemente sind ein Paar `(Schlüssel Inhalt)`, so daß eine totale Ordnung der Elemente aufgrund der totalen Ordnung der Schlüssel möglich wird, damit ist ein schneller Zugriff auf die Daten möglich; in solchen Systemen ist auch die logarithmische Komplexität wichtig.

- Darstellung von Daten: 0-1-Folgen (Bitfolgen), z.B. ASCII-Code (7 Bit, 128 verschiedene Zeichen); Nachricht mit 10 Zeichen: Codierung 70 Bits; allgemein: Codierung N verschiedener Zeichen mit $\log_2 N$ Bits pro Symbol

  Beispiel: Nachricht mit einem Symbolvorrat von 8 Zeichen (A, B, C, D, E, F G, H), damit reichen 3 Bits aus: A = 000; B = 001; C = 010; D = 011; E = 100; F = 101; G = 110; H = 111. Da der Code eine feste Länge hat, kann man die Zeichenfolge 000001010011 eindeutig zu ABCD decodieren. Nachteil: unterschiedliche Häufigkeit einzelner Zeichen wird nicht berücksichtigt! Ansatz: Code variabler Länge, z.B. kommt A besonder häufig vor, so z.B. A = 0; B = 100; C = 1010; D = 1011; E = 1100; F = 1101; G = 1110; H = 1111. Damit ist die Codierung einer Zeichenfolge mit vielen As kürzer; schwierig: wo fängt bei einer Zeichfolge das nächste Zeichen an? Lösung: Präfixcode: kein Code ist Präfix eines anderen Codes.

3.2.5 Huffmann-Bäume (siehe auch externes Blatt)

Konstruktion solcher Codes durch Huffman-Codierung; Idee: Code als Binärbaum; Blätter: einzelne Zeichen mit Gewichtung (Häufigkeit); Innere Knoten: Zeichenmengen (Vereinigung der Sohnknoten) mit Gewichtung (Summe der Gewichtungen der Sohnknoten); zum Beispiel:

```
+ 0 A (8)
  1 0 0 B (3)
      1 0 C (1)
        1 D (1)
    1 0 0 E (1)
        0 F (1)
      1 1 G (1)
        1 H (1)
```

- Codierung: Beschrift Söhne immer mit 0 oder 1; Code für ein Zeichen: Folge der Beschriftungen (Wurzel bis Blatt)
- Decodierung: Folge entsprechend der Codeirung von der Wurzel bis zu einem Blatt; wenn ein Blatt erreicht ist: Zeichencode zu Ende, starte wieder mit Wurzel

Datenstrukturen: Darstellung Blatt: Liste `('blatt <symbol> <Gewichtung>)`; Operationen: `konstr-blatt` etc.; Darstellung innerer Knoten: Liste `(<linker Sohn> <rechter Sohn> <Symbolliste> <Wichtung>`; Decodierung bei gegebenem Baum: Aus Bitfolge und aktueller Teilbaum wird eine Symbolfolge generiert.

3.2.6 Konstruktion der Huffmann-Bäume

- Anfangs: von Blättern ausgehen (Zeichen/Häufigkeiten)
- Schritt: suche die zwei Teilbäume mit geringster Gewichtung und vereinige diese Teilbäume zu einem
- Ende: nur noch ein Baum vorhanden

Beispiel $\{(E, 1); (F, 1); (H, 1); (G, 1)\}$:

1. Zusammensetzen von $\{(E, 1); (F, 1)\}$ zu einem Teilbaum
2. Zusammensetzen von $\{(G, 1); (H, 1)\}$ zu einem Teilbaum
3. Zusammensetzen beider Teilbäume (1) und (2) zu einem Teilbaum

Wichtig: Effizientes Auffinden von Teilbäumen bzw. Blättern mit kleinster Wichtung; Menge von Teilbäumen entspricht einer Liste geordnet nach Wichtungen (kleinste vorne), das Auffinden passiert also in konstanter Zeit. Damit muß aber `hinzufuegen-menge` nach Wichtung geordnet einfügen

Konstruktion der initialen Menge: Wiederholtes Einfügen der Blätter (Eingabe: Liste von Symbol/Häufigkeiten-Paaren); Vereinigungsschritt siehe Übung

3.3 Daten mit Mehrfachrepräsentation

Datenabstraktion: Implementierung schützen durch Abstraktionsbarriere; Vorteil: Implementierung austauschbar (z.B. vom Prototyp zur effizienteren Implementierung); mehrere Implementierungen: Entscheidung zur Laufzeit, welche Implementierung benutzt wird:

- generische Operatoren: können mit mehreren Implementierungen umgehen
- manifeste Typen: Daten haben explizite Angabe über die Art der Implementierung
- datengesteuerte Programmierung: Daten entscheiden selbst über Verarbeitung

3.3.1 Beispiel: komplexe Zahlen

Darstellung durch Real- und Imaginärteil bzw. als Koordinaten in $\mathbb{R}^2$ (Rechteckkoordinaten, gut für Addition) oder durch Betrag und Winkel (Polarkoordinaten, gut für Multiplikation);

Daher gewünschte Programmebenen:

1. `+c`, `-c`, $\times$`c`, $\div$`c`
2. `konstr-rechtecke`, `konstr-polar`;
   `real-teil`, `im-teil`, `abs-wert` und `winkel`
3. Paare und normale Arithmetik

3.3.2 Implementierung oberste Ebene (Arithmetik)

Addition in Rechteckkoordinaten $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$; Multiplikation in Polarkoordinaten $(a_1, \varphi_1) \cdot (a_2, \varphi_2) = (a_1 \cdot a_2, \varphi_1 + \varphi_2)$:

```
(define (+c z1 z2)
        (konstr-rechteck (+ (real-teil z1) (real-teil z2))
                         (+ (im-teil   z1) (im-teil   z2))))
(define (*c z1 z2)
        (konstr-polar     (* (abs-wert  z1) (abs-wert  z2))
                         (+ (winkel    z1) (winkel    z2))))
```

3.3.3 Implementierung der Selektoren und Konstruktoren

mit $x = a \cdot \cos\varphi$; $y = a \cdot \sin\varphi$; $a = \sqrt{x^2 + y^2}$; $\varphi = \arctan(x, y)$

verschiedene Ansätze:

1. intern nur Rechteckskoordinaten

```
(define (kontr-rechteck x y) (cons x y))
(define (konstr-polar a w)   (cons (* a (cos w)) (* a (sin w))))
(define (real-teil z) (car z))
(define (im-teil z)   (cdr z))
(define (abs-wert z)  (wurzel (+ (quadrat (car z))
                                 (quadrat (cdr z)))))
(define (winkel z)    (atan (cdr z) (car z)))
```

2. intern nur Polarkoordinaten

```
(define (konstr-rechteck x y) (cons (wurzel (+ (quadrat x)
                                               (quadrat y)))
                                    (atan y x)))
(define (konstr-polar a w)    (cons a w))
(define (real-teil z) (* (car z) (cos (cdr z))))
(define (im-teil z)   (* (car z) (sin (cdr z))))
(define (abs-wert z)  (car z))
(define (winkel z)    (cdr z))
```

3. intern beide Darstellungen möglich (dann müssen Selektoren „wissen“, welche Darstellung ein Paar repräsentiert) ⇒ *Manifeste Typen* (Implemetierung durch ein Paar `(<typ> <inhalt>)`):

```
(define (typ-anbinden typ inhalt) (cons typ inhalt))
(define (typ objekt)
        (if (pair? objekt) (car objekt)
            (error "falscher Datentyp!")))
(define (inhalt objekt)
        (if (pair? objekt) (cdr objekt)
            (error "falscher Datentyp!")))
(define (konstr-rechteck x y)
        (typ-anbinden 'rechteck (cons x y)))
(define (konstr-polar    x y)
        (typ-anbinden 'polar (cons a w)))
```

Zur Definition der Selektoren verschiedene Ansätze:

(a) *naiv:* Erkennung des Darstellungstyps, Definition generischer Selektoren (d.h. sie können mit mehreren Typen umgehen):

```
(define (rechteck? z) (eq? (typ z) 'rechteck))
(define (polar? z)    (eq? (typ z) 'polar   ))
(define (real-teil z)
        (cond ((rechteck? z) (real-teil-rechteck (inhalt z)))
              ((polar?    z) (real-teil-polar    (inhalt z)))))
(define (real-teil-rechteck z) (car z))
(define (real-teil-polar    z) (* (car z) (cos (cdr z))))
```

analog folgen `im-teil`, `abs-wert` und `winkel` bzw. `im-teil-rechteck` und `im-teil-polar` usw.

Vorteil dieses Ansatzes: modularer Entwurf: jedes Modul realisierbar nur durch seine Schnittstellen

Nachteil: generische Prozeduren (`real-teil`, ...) müssen alle Darstellungen kennen (Zuordnungen zu eigentlichem Selektor nötig); Folge: jede neue Darstellung bringt Änderungen aller generischen Prozeduren mit sich

(b) *datengesteuerte Programmierung*: Daten (und nicht Prozeduren) entscheiden selbst, wie sie bearbeitet werden; Beispiel: eine komplexe Zahl in Rechteckdarstellung liefert den Realteil durch Prozedur `real-`
`teil-rechteck`, eine Zahl in Polardarstellung durch die Prozedur `real-teil-polar`. Notwendig: Zuordnung *(generische Operation, Typ)* $\mapsto$ *Prozedur*; Realisierung als Tabelle:

	rechteck	**polar**
real-teil	`real-teil-rechteck`	`real-teil-polar`
im-teil	`im-teil-rechteck`	`im-teil-polar`
abs-wert	`abs-wert-rechteck`	`abs-wert-polar`
winkel	`winkel-rechteck`	`winkel-polar`

Aufbau der Tabelle: `(put <typ> <operation> <eintrag>)`; Durchsuchen der Tabelle: `(get <typ> <operation>` liefert `<eintrag>`; (Implementierung in Kapitel (4)).

Aufbau der konkreten Tabelle mit Prozeduren (nicht Namen!) als Einträge:

```
(put 'rechteck 'real-teil real-teil-rechteck)
(put 'polar    'im-teil   im-teil-polar)
...
```

Ausführen einer Prozedur für ein Objekt:

```
(define (op-ausfuehren op obj)
        (let ((proz (get (typ obj) op)))
             (if (not (procedure? proz))
                 (proz (inhalt obj))
                 (error "Operater undefiniert!"))))
```

Nun wird eine typunabhängige Definition generischer Operationen möglich:

```
(define (real-teil obj) (op-ausfuehren 'real-teil obj))
```

Allgemeine Strategie: *dispatching on type*, d.h. Typprüfung und Aufruf einer geeigneten Prozedur; codeunabhängige Realisierung

Vorteil: Beim Hinzufügen neuer Darstellungsform muß diese nur in die Tabelle eingetragen werden und die in die Tabelle eingetragenen Funktionen programmiert werden.

(c) *nachrichtengesteuerte Programmierung*: Nachrichtenweitergabe (intelligente Datenobjekte, nicht mehr intelligente Operatoren); Datenobjekt enthält eigene Zuordnung *Operatoren* $\mapsto$ *Implementierung*; am Beispiel der Rechteckdarstellung:

```
(define (konstr-rechteck x y)
        (define (zuteilen n)
                (cond ((eq? n 'real-teil) x)
                      ((eq? n 'im-teil  ) y)
                      ((eq? n 'abs-wert ) (wurzel
                                            (+ (quadrat x)
                                               (quadrat y))))
                      ((eq? n 'winkel   ) (atan yx))
                      (else (error "Unbekannte Operation"))))
        zuteilen)
```

Beachte: Analogie zu alternativer Paarimplementierung in Kapitel (3.1)! Ausführung einer Operation auf Objekt:

```
(define (op-ausfuehren op obj) (obj op))
```

Programmiertechnisch Nachrichtenweitergabe („*message passing*“), zusammen mit Zuständen (Kap. (4)) Basis der objektorientierten Programmierung

Vorteil: Modularität: Datenobjekte enthalten Informationen über ihre Implementierung

3.4 Systeme mit generischen Operatoren

voriges Kapitel: arbeiten mit verschiedenen Darstellungen; jetzt: arbeiten auf verschiedenen Arten von Operanden

Beispiel: generisches Arithmetikmodul; Operatoren für Zahlen aller Art (im konkreten Beispiel für reelle, rationale und komplexe Zahlen):

<table>
<tr><td colspan="4">Anwendung</td></tr>
<tr><td colspan="4"><code>add</code>, <code>sub</code>, <code>mul</code>, <code>div</code></td></tr>
<tr><td colspan="4">generisches Arithmetikpaket</td></tr>
<tr><td><code>+rat</code>, <code>-rat</code>
<code>*rat</code>, <code>/rat</code></td><td colspan="2"><code>+c</code>, <code>-c</code>
<code>*c</code>, <code>/c</code></td><td><code>+zahl</code>, <code>-zahl</code>
<code>*zahl</code>, <code>/zahl</code></td></tr>
<tr><td rowspan="2">rationale Zahlen</td><td colspan="2">komplexe Zahlen</td><td rowspan="2">reelle Zahlen</td></tr>
<tr><td>polar</td><td>rechteck</td></tr>
<tr><td colspan="4">Listenstruktur, primitive Funktionen</td></tr>
</table>

Vorgehensweise in mehreren Stufen:

3.4.1 Anfügen einer Typetikette

ähnlich wie bei Mehrfachdarstellungen (`'rat`, `'komplex`, `'zahl`)

```
(define (konstr-zahl x) (typ-anbinden 'zahl x))
(define (konstr-komplex z) (typ-anbinden 'komplex z))
(define (+zahl x y) (konstr-zahl (+ x y)))
(define (-zahl x y) (konstr-zahl (- x y)))
(define (*zahl x y) (konstr-zahl (* x y)))
(define (/zahl x y) (konstr-zahl (/ x y)))
(define (+komplex z1 z2) (konstr-komplex (+c z1 z2)))
(define (-komplex z1 z2) (konstr-komplex (-c z1 z2)))
(define (*komplex z1 z2) (konstr-komplex (*c z1 z2)))
(define (/komplex z1 z2) (konstr-komplex (/c z1 z2)))
```

3.4.2 Verbindung der Arithmetikoperationen mit Zahloperatoren

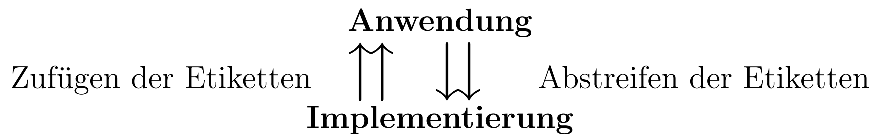
```
(put 'zahl    'add +zahl   )
(put 'zahl    'sub -zahl   )
(put 'zahl    'mul *zahl   )
(put 'zahl    'div /zahl   )
(put 'komplex 'add +komplex)
(put 'komplex 'sub -komplex)
(put 'komplex 'mul *komplex)
(put 'komplex 'div /komplex)
```

3.4.3 Definition der generischen Operatoren

```
(define (add x y) (op-ausfuehren-2 'add x y))
(define (op-ausfuehren-2 op arg1 arg2)
        (let ((t1 (typ arg1)))
```

```
(if (eq? tq (typ arg2))
    (let ((proz (get t1 op)))
         (if (not (null? proz))
             (proz (inhalt arg1)
                   (inhalt arg2))
             (error "Operator undefiniert")))
    (error "Operanden ungleichen Typs"))))
```

Interessant: für komplexe Zahlen existiert ein zweistufiges Typsystem, Beispiel komplexe Zahl in Rechteckkoordinaten: `('komplex 'rechteck 3 4)`; Ablauf der Typenverarbeitung:



3.4.4 Operanden mit unterschiedlichen Typen

- mögliche (naive) Lösung: Operatoren für gemischete Operanden: `+zahl|komplex`, `+rational|zahl` etc. und dreidimensionale Tabelle $Typ \times Typ \times Operation \mapsto Prozedur$

  Nachteil: Aufwand ist riesig (bei n verschiedenen Typen n^2 verschiedene Versionen jedes generischen Operators!)

- besser Lösung: Typanpassung; Versuche, einen Typ in einen anderen zu überführen. Beispiel: Zahlen zu komplexen Zahlen

  ```
  (define (zahl->komplex x)
          (konstr-komplex (konstr-rechteck (inhalt x) 0)))
  ...
  ```

  Datengesteuerte Programmierung, alle möglichen Typanpassungen in spezielle Tabelle eintragen

  ```
  (put-typanpassung 'zahl 'komplex zahl->kopmlex)
  ...
  ```

  Anpassung der Funktion zum Ausführen einer Operation:

  ```
  (define (op-ausfuehren-2 op a1 a2)
          (let ((t1 (typ a1)))
               ((t2 (typ a2)))
  ```

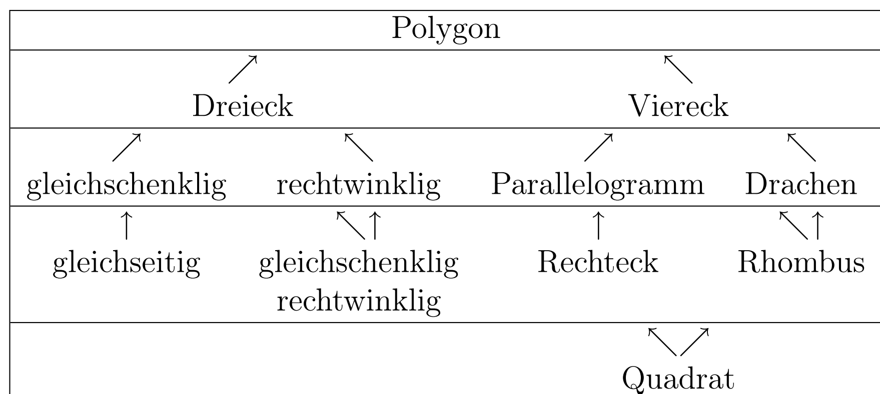
```
(if (eq? t1 t2)
    (let ((proz (get t1 op)))
         (if (not (null? proz))
             (proz (inhalt a1)
                   (inhalt a2))
             (error "Operator n. definiert")))
    (let ((t1->t2 (get-typanpassung t1 t2)))
         ((t2->t1 (get-typanpassung t2 t1)))
         (cond ((not (null? t1->t2))
                (op-ausfuehren-2 op
                                 (t1->t2 a1)
                                 a2))
               ((not (null? t2->t1))
                (op-ausfuehren-2 op
                                 a1
                                 (t2->t1 a2)))
               (else (error "keine Konvertierung
                            möglich")))))))
```

3.4.5 Typanpassung

Umwandlung von Daten unterschiedlichen Typs (`op-ausfuehren-2` mit Typanpassung)

- *Vorteil:* bei n Typen nur noch n Versionen jedes generischen Operators (statt n^2); allerdings bis zu n^2 Typanpassungen (besser: transistive Umwandlungen)
- *häufig:* Objekte von Typen mit t_1 und t_2, aber weder $t_1 \to t_2$ noch $t_2 \to t_1$ möglich, aber eventuell existiert ein gemeinsamer Typ t_3 mit $t_1 \to t_3$ und $t_2 \to t_3$.
- *Daher:* Betrachtung von Typhierachien, t_1 ist Untertyp von t_2:
  - Jedes t_1-Objekt ist auch t_2-Objekt
  - Jede auf t_1-Objekte anwendbare Operation ist auch auf t_1-Objekte anwendbar
  - t_1-Konzept ist Spezialfall von t_2-Konzept
- Beispiel Zahlen: $\mathbb{N} \subseteq \mathbb{N}_0 \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$; d.h. *Turmstruktur*, für Typanpassung sind nur noch drei Prozeduren $t_n \to t_{n+1}$; Typanpassung ganzzahlig $\to$ komplex: drei Schritte (ganzzahlig zu rational zu reell zu komplex).

- Änderungen in `op-ausfuehren-2`: bei verschiedenen Typen: erhöhe schrittweise niedrigeren Typ, bis beide gleich sind (siehe Übung)
- *Vererbung von Operationen*: Falls Operationen auf bestimmte Typen undefiniert: Statt `error` soll der Typ erhöht werden und versuchen, die Operation dann auf erhöhtem Typ auszuführen;
- *Erniedrigung von Typen:* Umwandlung von Objekten in niedrigeren Typ (falls möglich), Beispiel: $(2 + 3i) + (4 - 3i) = 6 + 0i = 6$, also ganze Zahl ausgeben; notwendig: Kriterium „Ist Objekt von niedrigerem Typ?“ (siehe Übung)
- *komplexe Typhierachien* am Beispiel der Geometrie:



  Probleme:

  - Mehrfachvererbung (siehe gleichschenkliges rechtwinkliges Dreieck); kein eindeutiger Typ für Erhöhung
  - Mehrere Untertypen (siehe Dreieck oben etc.); keine eindeutige Erniedrigung

  Typanpassung: Erhöhe Operanden mit kleinsten gemeinsamen Obertyp (existiert nicht immer); Beispiel: gleichseitiges Dreieck + Quadrat → Polygom; Notwendig: Suche im Typgraph (effiziente Implementierung ist möglich!)

3.4.6 Generische Arithmetik auf Polynome

Polynom: Summe von Termen kx^n; Darstellungsproblem: Bedeutung der Variablen (sind $5x^2 + 3x$ und $5y^2 + 3y$ gleich? Mathematisch ja, im folgenden jedoch syntaktisch nicht!).

Datenabstraktion: Polynom = `(variable termliste)`, nötig:

- Selektor `(variable (konstr-poly v tl)) = v`
- Selektor `(termliste (konstr-poly v tl)) = tl`
- Konstante `leere-termnliste`
- Abfrage `leere-termnliste?`
- Prozedur `term-anhaengen` mit höherem Grad
- Selektor `erster-term`
- Selektor `rest-term`
- Selektor `grad` Grad eines Termes
- Selektor `koeff` Koeffizienten eines Termes

Rechnen mit Polynomen: Addition (und analog Multiplikation)

```
(define (+poly p1 p2)
        (if (gleiche-variable (variable p1) (variable p2))
            (konstr-poly (variable p1) (+terme (term-liste p1)
                                               (term-liste p2)))
            (error "Polynome haben ungleiche Variablen")))
```

Erweiterung der generischen Arithmetik:

```
(put 'poly 'add +poly)
(put 'poly 'mult *poly)
```

Implementierung von +terme (siehe Zettel): Vergleiche Grad der ersten Terme, wenn diese gleich sind, so addiere Koeffizienten, andernfalls füge Term mit größerem Grad hinzu. `*terme`:

- multipliziere jeden Term mit allen Termen des anderen Polynoms
- addiere diese Polynome
- Multiplikation von Termen: Addiere Grade, Multipliziere Koeffizienten

Koeffizienten mit generischen Operatoren add/mult verknüpft $\Rightarrow$ Polynome mit beliebigen Koeffizienten aus Arithmetikmodul (reell, komplex, rat)

- mit Typanpassung: $(5x^2 + (4 - 2i)x \cdot (6x^5 + \frac{3}{2}x^2)$
- auch Polynome als Koeffizienten: $(y + 1)x^2 + (y^3 + 5y^2)x$

Implementierung der Termlisten:

- Variable in jedem Term gleich, d.h. keine explizite Speicherung in der Termliste
- zwei mögliche Darstellungen (Beispiel $5x^4 + 2x^2 + 3$):
  - `(5 0 2 0 3)`
  - `((5 4) (2 2) (3 0))`

```
(define (term-anhaengen t tl)
        (if (=null? (koeff t))
            tl
            (cons t tl)))
(define (leere-termliste) ())
(define (leere-termliste? tl) (null? tl))
(define (erster-term tl) (car tl))
(define (rest-terme tl) (cdr tl))
(define (konstr-term grad koeff) (cons grad koeff))
(define (grad t) (car t))
(define (koeff t) (cdr t))
(define (konstr-poly var tl) (typ-anbinden 'poly (cons var tl)))
(define (variable p) (car p))
(define (term-liste p) (cdr p))
```

Typhierachie mit Polynomen: mit Turmstrukktur könnten Polynome über komplexen Zahlen stehen (durch Konvertierung von $a + bi = (a + bi) \cdot x^0$); Aber: unendlicher Term, da Polynome selbst koeffizienten von Polynomen sein können; Problem zusätzlich: Polynome nicht vergleichbar (verschiedene Variablen), daher unendlich viele Obertypen

4 Modularität, Objekte, Zustände

Prinzip der Software-Erstellung:

Orientierung an der Struktur des Anwendungsproblems
und
geeignete Organisationsstruktur der Programme

Beispiele:

- mathematische Berechnungen: Prozeduren und Lokalität
- datenintensive Anwendungen (z.B. Datenbanken): hierarchische Datenstruktur und Datenabstraktion

„Reale Welt“: komplexe Objekte mit „Geschichte“ (Bankkonto, Flugreservierung, Buchungssysteme); Organisationsprinzipien hier:

- Objekte mit Zuständen
- Ströme

4.1 Zuweisungen und lokale Zustände

Objektverhalten häufig abhängig von Vorgeschichte, Beispiel: Abhebung von 1000 DM von meinem Konto möglich? Je nachdem...

Abstraktion der Geschichte auf einen veränderbaren Zustand, z.B. aktueller Kontostand; Beispiel: abheben vom Konto: Falls möglich: Ausgabe Kontostand, sonst: entsprechende Mitteilung

```
>(abheben 50)
80
>(abheben 50)
30
>(abheben 50)
Deckung nicht ausreichend!
```

Bisher: Variable entspricht eindeutigem Wert; Nun: Änderung von Variablenwerten durch die Sonderform `(set! <name> <neuer-wert>)`; Effekt bei Auswertung von `set!`:

- Werte Ausdruck `<neuer-wert>` aus.
- Verändere Wert von `<name>` zum berechneten Wert
- Ergebnis von `set!`: undefiniert

```
(define kontostand 130)
(define (abheben betrag)
        (if (>= kontostand betrag)
            (begin (set! kontostand (- kontostand betrag))
                   kontostand)
            "Deckung nicht ausreichend!"))
```

Notwendig: Sonderform `(begin <a1> <a2> ... <an>)`; Auswertung:

- Werte *nacheinander* die Ausdrücke $a_1, a_2, ..., a_n$ aus
- Wert der Sonderform = Wert von a_n

Bisherige Lösung: globale, veränderbare Variablen (von jedem veränderbar);
Lösung: Definiere Kontostand lokal zur Prozedur `abheben`

```
(define neu-abheben
        (let ((kontostand 130))
             (lambda (betrag)
                     (if (>= kontostand betrag)
                         (begin (set! kontostand
                                      (- kontostand betrag))
                                kontostand)
                         "Deckung nicht ausreichend!"))))
```

Gleiches Verhalten wie abheben, aber Kontostand von außen nicht zugreifbar; im Moment nicht einsehbar, da keine exakte Definition von `set!` in Substitutionsmodell möglich ist (siehe Kapitel 4.2); weitere Nachteile: Fixer Anfangskontostand und ein festes Konto; daher: Konstruiere „Abhebungsprozessoren“ für beliebige Konten:

```
(define (konstr-abheben kontostand)
             (lambda (betrag)
                     (if (>= kontostand betrag)
                         (begin (set! kontostand
                                      (- kontostand betrag))
                                kontostand)
                         "Deckung nicht ausreichend!")))
```

Benutzung zweier Abhebungsprozessoren:

```
(define A1 (konstr-abheben 100))
(define A2 (konstr-abheben 100))
>(A1 50)
50
>(A2 70)
30
>(A2 40)
Deckung nicht ausreichend
>(A1 40)
10
```

Auch einzahlen: Kontostand, abheben, einzahlen müssen lokal in gleicher Umgebung arbeiten ⇒ Nachreichtenweitergabe:

```
(define (konstr-konto kontostand)
```

```
        (define (abheben betrag)
                (if (>= kontostand betrag)
                    (begin (set! kontostand (- kontostand betrag))
                           kontostand)
                    "Deckung nicht ausreichend!"))
        (define (einzahlen betrag)
                (set! kontostand (+ kontostand betrag))
                kontostand)
        (define (zuteilen m)
                (cond ((eq? m 'abheben) abheben)
                      ((eq? m 'einzahlen) einzahlen)
                      (else (error "Falsche Nachricht!"))))
        zuteilen)

>(define konto (kontstr-konto 100))
>((kto 'abheben) 70)
30
```

Probleme der Zuweisung: Substitutionsmodell:

- ersetze Variablen durch ihre Werte
- ersetze Kombinationen durch ausgerechneten Wert

Beispiel:

- Nach Aufruf von `(define kontostand 130)` steht `kontostand` für den Wert `130`.

⇒ Vorkommen von `kontostand` ersetzen durch `130`.

⇒ `(+ e kontostand)` und `(+ e 130)` liefern gleiche Werte

! Jedoch: `(+ (abheben 110) kontostand)` liefert `(+ 20 20)`...

- Somit: bei Benutzung von `set!`:

Variablen sind nicht Namen für Objekte oder Werte, sondern Namen für Orte, wo Werte gespeichert werden!

⇒ komplexeres Auswertungsmodell

⇒ Auswertungsreihenfolge relevant:

```
(+ (abheben 110) kontostand)
```

kann `(+ 20 20)` oder `(+ 20 130)` liefern!

- Jedoch: Auswertungsreihenfolge bei Kombinationen *beliebig*.

! Bei Veränderungen immer `(begin ...)` benutzen!

- unzulässig: Ausdrücke in Programmen durch Werte ersetzen, z.B.

  `(let ((x (abheben 110))) (+ x 2))` ⇝ `(+ 20 2)`

- Konsequenz: Optimierung und Verifikation von Programmen: schwieriger!

Referentielle Transparenz:

- Eigenschaft von Sprachen, bei denen „Gleiches durch Gleiches“ ersetzt werden kann (z.B. Ausdrücke durch ihre Werte) ohne Änderung der Ergebnisse

- Vorteil für Optimierung, Verifikation, Wertung

- wird zerstört duch die Benutzung von `set!`

- Begriff der *Gleichheit* wird schwieriger durch `set!`: Wertgleichheit und Objektgleichheit (-identität) entscheidend:

- Beispiel: verschiedene Konten:

```
(define peter-kto (konstr-konto 100))
(define paul-kto  (konstr-konto 100))
```

  gleiche Konten, verschiedene Namen:

```
(define peter-kto (konstr-konto 100))
(define paul-kto  peter-konto)
```

! Beachte: kein Problem der Programmiersprache, sondern Resultat der Modellbildung von Objekten mit veränderbarem Zustand.

- Vorzüge der Zuweisung: mehr Modularität durch Objekte mit lokalem Zustand.

Beispiel: Zufallszahlengenerator. Dargestellt durch Zufallsfolge (statistisch gleich verteilt), nicht einfach definierbar (Beispiel: $x_{n+1} = (ax_N + b) \mod m$ mit passend gewählten a, b, m). Notwendig: Funktion, die aus einer Zufallszahl die nächte erzeugt:

```
x2 = (zufall-aktuell x1)
x3 = (zufall-aktuell x2)
...
```

Mit Verwendung von `set!`:

```
(define zufall (let ((x zufall-init))
                    (lambda () (set! x
                                     (zufall-aktuell x))
                               x)))
```

Vorteil: Programm weiß aktuellen Zufallswert, kein *Durchreichen* notwendig.

4.1.1 Anwendung: Monte-Carlo-Simulation

- große Megen von Experimenten
- zufällige Auswahl von Stichproben
- Tabellierung der Ergebnisse

⇒ Wahrscheinlichkeiten erlauben Schlußfolgerungen

- konkret: Wahrscheinlichkeit, daß zwei zufällig gewählte ganze Zahlen keinen gemeinsamen Teiler größer als 1 besitzen: $\frac{6}{\pi^2}$ (Cesero-Test), daraus kann man einen Schätzwerk für π berechnen:

```
(define (schaetzwert-pi versuche)
        (sqrt (/ 6 (monte-carlo versuche
                                cesaro-test))))
(define (cesaro-test)
        (= (ggt (zufall) (zufall)) 1))
(define (monte-carlo versuche experiment)
        (define (iter versuche-uebrig versuche-erfolgreich)
                (cond ((= versuche-uebrig 0)
                       (/ versuche-erfolgreich versuche))
                      ((experiment)
                       (iter (- versuche-uebrig 1)
                             (+ versuche-erfolgreich 1)))
                      (else
                       (iter (- versuche-uebrig 1)
                             versuche-erfolgreich))))
        (iter versuche 0))
```

Ein Cesaro-Test mit `zufall-aktuell` statt `zufall` hingegen (d.h. ohne `set!`) würde mit der Modularität brechen:

- kein allgemeines `monte-carlo`:
  * Zustand des Zufallsgenerators muß immer weitergereicht werden
  * Verquickung von Experimentzählern und Zufallsgenerierung
- selbst Schätzwert für π enthält Zufallsgenerator

$\Rightarrow$ Zustände können Modularisierung unterstützen!

Andere Alternative: Datenströme (später!)

4.1.2 Umgebungsmodell

Substitutionsmodell:

- Variable entspricht einem Namen für den Wert
- Auswertung von `(f a1 ... an)`:
  1. Werte Teilausdrücke aus
  2. Wende Wert den Operators `f` auf Werte der Operanden `a1` ... `an` an

Jetzt neu: Variable entspricht einem *Ort*, wo Werte gespeichert sind. Strukturen zum festhalten von Orten sind *Umgebungen* (*environments*), sie sind nötig, da lokale Variablen und Parametervariablen existieren.

- *Umgebung*: Folge von (Bindungs-)Rahmen
- *Rahmen (frame)*: Tabelle von Bindungen und Zeiger auf zugehörige Umgebung (abgesehen vom globalen Rahmen).
- *Bindung*: Zuordnung von Variablen zu Werten

Wert einer Variablen bezüglich einer Umgebung: Wert derjenigen Bindung dieser Variablen, die sich im ersten Rahmen der Umgebung befindet, der eine Bindung für die Variable enthält. Existiert keine Bindung, ist die Variable ungebungen. Bindungen können einander überschneiden.

Die Umgebung bestimmt den Kontext der Auswertung:

- Wert einer Variablen bezüglich einer Umgebung

- Auswertung eines Ausdrucks bezüglich einer Umgebung

Auswertung:

- neu: Prozedur ist Code (Paramter + Rumpf) und *Zeiger* auf Umgebung (wo ist die Prozedur definiert?)

- Beispiel: `(define (quadrat x) (* x x))` ist syntaktischer Zucker für `(define quadrat (lambda (x) (* x x)))`, somit wird der entsprechenden Umgebung (ggf. globale Umgebung) die Bindung von `quadrat` auf ein Paar aus Code (Parameter `x` und Rumpf `(* x x)`) und Zeiger auf die Umgebung.

- Auswertung von z.B. `(quadrat 5)`:

  - Es wird eine neue Umgebung angelegt mit Zeiger auf die Umgebung, in der `quadrat` definiert ist
  - in die Umgebung wird eingetragen: `x` ist an `5` gebunden
  - in der Umgebung wird der Rumpf der Prozedur ausgewertet (`(* x x)`)

Auswertungsregeln formal:

- Werte Teilausdrücke bezüglich *dieser Umgebung* aus (ohne Reihenfolge!)

- Wende den Wert des Operatorausdrucks (Prozedur) auf die Operandenwerte wie folgend an:

  - Konstruiere neuen Rahmen mit Umgebungsteil der Prozedur als zugehöriger Umgebung
  - Trage dort die Bindungen *formale Parameter* → *aktuelle Argumente* ein
  - Werte den Rumpf der Prozedur bezüglich der neuen Umgebung aus

- Auswertung eines λ-Ausdrucks bezüglich einer Umgebung: Erzeuge neues Prozedurobjekt (Parameter + Rumpf) des λ-Ausdrucks mit Zeiger auf aktuelle Umgebung

- Auswertung eines `define` bezüglich einer Umgebung: Erzeuge Bindung *Variable* → *Wert* in *aktueller Umgebung* (eventuell nur Änderung einer schon existierenden Bindung

- Auswertung von `(set! x a)` bezüglich einer Umgebung:

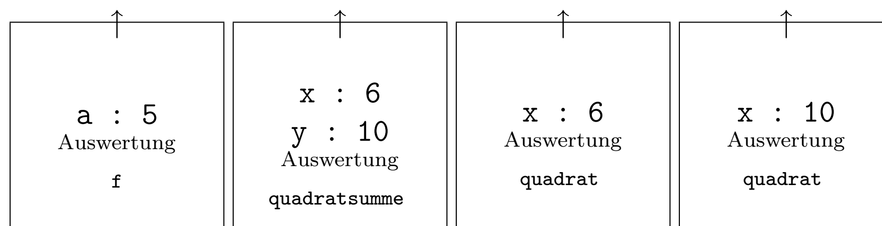
- Werte `a` aus bezüglich dieser Umgebung
- Suche ersten Rahmen in diser Umgebung mit Bindung für Wariable `x` (falls dies nicht existiert: Fehlermeldung!)
- Ändere den dortigen Wert von `a`

Beispiel:

```
(define (quadrat x) (* x x))
(define (quadratsumme x y) (+ (quadrat x) (quadrat y)))
(define (f a) (quadrat (+ a 1) (* a 2)))
(f 5)
```

globale Umgebung

- `quadratsumme` (Parameter: `x`, Rumpf: `(+ ...)`, Zeiger: globale Umgebung)
- `quadrat` (Parameter: `x`, Rumpf: `(* x x)`, Zeiger: globale Umgebung)
- `f` (Parameter: `a`, Rumpf: `(...)`, Zeiger: globale Umgebung)



4.1.3 Lokale Zustände, Umgebungen, Definitionen

Beispiel: vereinfachten „Abheben-Prozessor“ aus (4.1):

```
(define (konstr-abheben kontostand)
          (lambda (betrag)
                (if (>= kontostand betrag)
                    (begin (set! kontostand
                                 (- kontostand betrag))
                           kontostand)
                    "Deckung nicht ausreichend!")))
```

Auswertung von `(define W1 (konstr-abheben 100))` in globaler Umgebung:

- Neue Umgebung U_1 mit Zeiger auf globale Umgebung
- Auswertung Rumpf in U_1 liefert neues Prozedurobjekt
- in globaler Umgebung wird Prozedur W_1 definiert: Parameter Betrag, Rumpf mit Abhebe-Funktion und Zeiger auf U_1 (nicht auf globale Umgebung!)

Abhebevorgang `(W1 50)`:

- Neue Umgebung U_2 für Parameter mit Bindung an U_1
- Auswertung Rumpf in U_2 verändert Kontostand in U_1

⇒ lokaler Zustand von W_1 wird in U_1 gespeichert

Lokale Definitionen, zum Beispiel:

```
(define (wurzel x)
        (define (gut-genug? y)
                (< (abs (- (quadrat y) x))) 0.0001)
        (define (verbessern y)
                (mittelwert y (/ x y)))
        (define (wurzel-iter y)
                (if (gut-genug? y) y
                    (wurzeliter (verbessern y))))
(wurzel-iter 1))
```

Auswertung:

- globale Umgebung: `wurzel` ist definiert (Parameter `x`, Rumpf einfach direkt aufgelistet – siehe oben, Zeiger: globale Umgebung)
- bei Aufruf von `wurzel 2`
  - neue Umgebung U_1 (mit Zeiger auf globale Umgebung), in der zunächst `x` auf `2` gebunden wird
  - als Code wird der Rumpf der Bindung von `wurzel` eingetragen
  - die `define`-Ausdrücke werden direkt ausgewertet, also die drei Prozeduren `gut-genug?`, `verbessern` und `wurzel-iter` gebunden auf Paae Code (Parameter + Rumpf) und Zeiger (diesmal auf U_1!)
  - der Befehl `(wurzel-iter 1)` erzeugt wieder eine neue Umgebung U_2, in der dann der übergebene Parameter `1` gebunden wird auf `y` und der Code von `wurzel-iter` ausgeführt wird.

Umgebungsmodell erklärt Effekt lokaler Definitionen:

- keine Überschneideung lokaler mit globlaem Namen (nur „Überschattung“ globaler Namen durch lokale)
- innerhalb lokaler Prozeduren sind Parameter globaler Prozeduren sichtbar
- Namenskonflikte bei Parameternamen bei Prozeduren auf gleicher Ebene

4.2 Veränderbare Datenstrukturen

- Komplexe Systeme benötigen zusammengesetze Daten (Abstraktion: Konstruktoren und Selektoren), Veränderung: *Mutatoren* („Veränderer“) für Datenobjekte (Beispiel: `put` für Tabellen),
- veränderbare Datenobjekte (*mutable data objects*): Datenobjekte, für die es Datenobjekte gibt
- Mutatoren for Listenstrukturen:
  - für Paare: `(set-car! p o)` ersetzt car-Verweis von Paar `p` durch einen Verweis auf `o` bzw.
  - `(set-cdr! p o)` ersetzt cdr-Verweis von Paar `p` durch einen Verweis auf `o`
  - Modifikation ist Seiteneffekt, d.h. das Ergebnis ist undefiniert
  - Beispiel:

```
> (define l (list 1 2))
> (set-car! l 3)
> l
(3 2)
> (set-cdr l (list 4 5))
> l
(3 4 5)
```

  - Beispiel aus Kapitel (3.2): `(append l1 l2)` vereinigt `l1` und `l2`, Mutator:

```
(define (append! l1 l2)
        (define (letztes-paar l)
                (if (null? (cdr l))
                    l
```

```
                  (letztes-paar (cdr l))))
        (set-cdr! (letztes-paar l1) l2)
        l1)
```

– Problem: `append!` nur eingeschränkt verwendbar, Zirkelverweise möglich:

```
> (define l (list 1 2))
> (set-cdr! l l)
> l
#0=(1 . #0#)
```

(die Ausgabe ist DrScheme-spezifisch!)

- *Structure sharing* und Identität: Problem der Zuweisung (siehe (4.1)): Unterschied zwischen *gleich* und *identisch*, ähnglich bei veränderbaren Strukturen:

```
> (define l '(a b))
> (define l1 (cons l l))
> (define l2 (cons '(a b) '(a b)))
```

Die beiden Listen `l1` und `l2` stellen die „gleichen“ Listen dar: `((a b) a b)`. Der Unterschied in den Strukturen von `l1` und `l2` ist unproblematisch, solange nichts verändert wird, aber relevant bei Mutatoren:

```
> (define (set-caar-z! l)
          (set-car! (car l) 'z))
> (set-caar-z! l1)
> (set-caar-z! l2)
> l1
((z b) z b)
> l2
((z b) a b)
```

Daher: sorgfältige Verwendung von Mutatoren, da sonst leicht ungeahnte Veränderungen! !

- *Mutation* = *Zuweisung*: Sharing + Mutation problematisch, daher nur Anweisung, keine Mutation? Keine Lösung, da Mutation durch Zuweisung implementierbar (Analogie: Paare duch Funktionen implementieren), Implementierung:

```
(define (cons x y)
        (define (set-x! v) (set! x v))
        (define (set-y! v) (set! y v))
        (define (zuteilen m)
                (cond ((eq? m 'car) x)
                      ((eq? m 'cdr) y)
                      ((eq? m 'set-car!) set-x!)
                      ((eq? m 'set-cdr!) set-y!)
                      (else (error "undefiniert!"))))
        zuteilen)
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
(define (set-car! z v) ((z 'set-car!) v))
(define (set-cdr! z v) ((z 'set-cdr!) v))
```

mit set! alleine sind alle Probleme präsent; rein funktionale Sprachen (mit referentieller Transparenz) haben keine Zuweisung; bei disziplinierter Benutzung ist Zuweisung nützlich

4.2.1 Warteschlangen/Queues

Neue Datenstrukturen mit Mutatoren: Warteschlangen („*queue*“) mit FIFO (first in, first out); Eigenschaft: Hinzufügen von Elementen am Ende, Entfernen von Elementen am Anfang (im Gegensatz zu LIFO, d.h. Stack bzw. Keller). Implementierung:

- Intuitiv: Liste der Elemente, Vorteil: Entfernen in $O(n)$, Nachteil: Einfügen in $O(n)$
- besser: Zusätzlicher Zeiger auf das letzte Element, d.h. Warteschlange ist implementiert als Paar bestehend aus Zeigern auf Anfang und Ende der Warteschlange
- Konstruktor:

  ```
  (define (konstr-warteschlange)
          (cons () ()))
  ```

- Selektor:

  ```
  (define (leere-warteschlange? q)
          (null? (car q)))
  ```

```
(define (anfang q)
        (if (leere-warteschlange? q)
            (error "Warteschlange leer!")
            (car (car q))))
```

- Mutator:

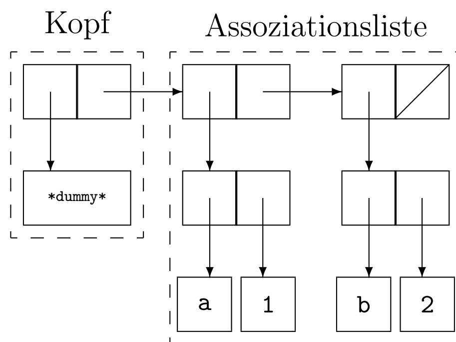
```
(define (hinzufuegen-warteschlange! q e)
        (let ((neues-paar (cons e ())))
        (if (leere-warteschlange q)
            (begin (set-car! q neues-paar)
                   (set-cdr! q neues-paar)
                   q)
            (begin (set-cdr! (cdr q) neues-paar)
                   (set-cdr! q neues-paar)
                   q))))
(define (entfernen-warteschlange! q)
        (if (leere-warteschlange? q)
            (error "Warteschlange leer!")
            (begin (set-car! q (cdr (car q)))
                   q)))
```

Anmerkung: Umsetzen Ende-Zeiger bei leerer Warteschlange unwichtig, `if` durch `cond` ersetzbar (ohne `begin!`), da Klauseln in cond auch Folgen von Ausdrücken.

4.2.2 Tabellen

n-dimensionale Tabelle: Abbildung $\langle key_1 \rangle, \langle key_2 \rangle, ..., \langle key_n \rangle \rightarrow werte$; Beispiel: Operator-Typ-Tabelle für generisches Arithmetikmodul Implementierung 1-dimensionaler Tabellen:

- Liste von Paaren $(key, wert)$
- Einfügen am Anfang
- „Dummy"-Element am Anfang, damit Tebellenobjekt immer gleich (auch bei Einfügen in leere Tabelle)
- Beispiel: Tabelle $a \mapsto 1, b \mapsto 2$:



Selektor: Suche nach Wert mit gegebenem Schlüssel

```
(define (suche-wert key t)
        (let ((satz (ass-eq key (cdr t))))
             (if (null? satz)
                 '()
                 (cdr satz))))
```

Suche in einer Assoziationsliste mit einem Schlüsselvergleich `eq?`

```
(define (ass-eq key al)
        (cond ((null? al) '())
              ((eq? key (caar al)) (car al))
              (else (ass-eq key (cdr al)))))
```

Mutator: Eintragen eines neuen Schlüssel-Wert-Paares: Falls Schlüssen vorhanden: Ändere Wert, sonst: füge am Anfang ein

```
(define (eintragen! key wert t)
        (let ((satz (ass-eq key (cdr t))))
             (if (null? satz)
                 (set-cdr! t
                           (cons (cons key wert)
                                 (cdr t)))
                 (set-cdr! satz
                           wert))))
```

Zwei- und mehrdimensionale Tabellen: $K_1, K_2 \to V \Leftrightarrow K_1 \to (K_2 \to V)$, d.h. Implementierung einer zweidimensionalen Tabelle als eindimensionale Tabelle mit eindimensionalen Tabellen als Werten (bei denen auf das Dummy-Element verzichtet werden kann), Realisierung: analog zum eindimensionalen Fall.

4.2.3 Tabellen als aktive Objekte

Die Funktionen `suche-satz` und `eintragen!` haben Tabelle als Parameter, also sind mehrere Tabellen möglich. Objektorientierte Sichtweise: Tabelle ist Objekt, das Nachrichten verarbeiten kann, ⇒ Lokalität des Zustandes, Modularität; Realisierung durch Nachrichtenweitergabe:

```
(define (konstr-tabelle)
        (let ((t (list '*dummy*)))
             (define (suche-wert key)
                     (wie oben))
             (define (eintragen! key wert)
                     (wie oben))
             (define (zuteilen m)
                     (cond ((eq? m 'suche-satz-proz)
                            suche-satz)
                           ((eq? m 'eintragen-proz!)
                            eintragen!)
                           (else (error "Falsche Op!"))))
             zuteilen))
```

Implementierung einer globalen Tabelle analog zu Kapitel (3.4):

```
(define op-tabelle (konstr-tabelle))
(define get (op-tabelle 'suche-satz-proz))
(define put (op-tabelle 'eintragen-proz))
```

4.3 Anwendungsbeispiele

4.3.1 Simulation digitaler Systeme

„Computing is Simulation“ (Alan Key); wichtiger Anwendungsbereich: Chipentwurf, Auto- und Flugzeugbau, Klimavorhersage etc.

- Hier: Digitale Schaltkreise mit Zeitverzögerung
- Schaltkreis: Menge von Schaltelementen: Verbunden mit Drähten, die den „Wert“ 0 oder 1 haben
- Schaltelemente:
  - Inverter: $\neg a$
  - UND-Gatter: $a \wedge b$
  - ODER-Gatter: $a \vee b$

- Komplexerer Schaltungen: z.B. Halbaddierer (Addiere 2 Bits mit Übertrag)

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- Ziel: Simulation mit Berücksichtigung der Verzögerung
- Objekte in Programmen:
  - Drähte: variable Schaltelemente, haben Werte 0 oder 1, veranlassen Aktionen bei Wertänderung
  - Schaltelemente: leiten Werte zwischen angeschlossenen Drähten weiter
- Beispiel: Erzeugung von Drähten:

```
(define a (konstr-draht))
```

  B

- Verbindung mit Schaltelementen:

```
(und-gatter a b c)
```

- Abstraktion: Definition komplexere Schaltwerke

```
(define (halbaddierer a b s c)
        (let ((d (konstr-draht))
              (e (konstr-draht)))
             (oder-gatter a b d)
             (und-gatter a b c)
             (inverter c e)
             (und-gatter d e s)))
```

⇒ Sprache für Schaltkreisentwurf (keine Spezialsprache notwendig!)

- Spezifikation des „Draht"-Datentyps:
  - Konstruktor: `(konstr-draht)`

- Selektor: `(get-signal draht)`
- Mutatoren: `(set-signal! draht neuer-wert)`, `(add-vorgang! draht prozedur)` (die übergebene Prozedur ohne Parameter wird automatisch ausgeführt, wenn Drahtwert sich ändert)
- Globale Prozedur: `(verzoegert zeit proz)`: Führt `proz` nach Zeitverzögerung `zeit` aus ($\approx$ Simulation)
- Globale Variablen: `(inverter-verzoegerung)` sowie `(und-gatter-verzoegerung)` usw., Verzögerungszeiten aller Schaltelemente
- Quellcode der Schaltelemente siehe externes Blatt bzw. Download

• Implementierung des „Draht"-Datentyps:

- Lokaler Zustände: `signal-wert`: aktueller Wert (0/1), und `vorgang-prozeduren`: Liste von Prozeduren, die bei Wertänderung auszuführenden Prozeduren
- Modellierungstechnik: aktives Objekt, das Nachrichten empfangen kann
- Simulation durch Nachrichtenweitergabe
- Wertänderung im Draht: $\rightarrow$ Teile Wertänderung allesn angeschlossenen Gattern mit

• Implementierung bisher: zeitlos (z.B. Wertermittlung), noch zu implementieren: `verzoegert`

- Realisierung durch „Zeitplan" für auszuführende Aktionen:

Zeit	*Aktionen*
0	$proz_1, proz_{13}$
2	$proz_3$
5	$proz_7, proz_9$
⋮	⋮

- Operationen für Zeitplan:
  * `(leerer-plan? plan)`
  * `(erster-plan-eintrag plan)` liefert ersten (geordnet nach Zeit) Eintrag in den Plan
  * `(entferne-ersten-plan-eintrag! plan)`

    * `(hinzufuegen-plan! zeit vorgang plan)` geordnetes Hinzufügen,
    * `(aktuelle-zeit plan)` liefert aktuelle Zeit $\approx$ erster Eintrag
  - Implementierung der Simulation mit Zeitplan-Operationen:
    * verzoegert: Implementierung durch Hinzufügen des Vorgangs zum definierten Zeitpunkt
    * Ausführen: Ausführung des ersten Vorgangs im Zeitplan, bis Zeitplan leer ist
    * Globale Variable `der-plan` (globaler Zeitplan)
  - Implementierung des Zeitplans:
    * Zeitplan ist Tabelle von Zeitsegmenten
    * Zeitsegment ist Paar (Zeit, Warteschlange)
    * Tabelleneinträge geordnet nach Zeitpunkten, statt `*dummy*`: aktuelle Zeit des ersten Eintrags
    * Zeitplan ist leer $\Leftrightarrow$ keine Zeitsegmente vorhanden
    * Eintragen eines Vorgangs: Füge Vorgang in passendes Zeitsegment ein bzw. füge neues Zeitsegment passend ein
    * Ersten Eintrag entfernen: aus erster Warteschlange streichen (falls Wartschlange dann leer: lösche Zeitsegment)
    * Erster Eintrag suchen: erstes Element in erster Warteschlange, zusätzlich: Aktualisierung der aktuellen Zeitplanzeit
  - Beispielsimulation: bisher keine Ausgabe; Ausgabe durch Anbringen von „Sonden“ an Drähten; Vorgang zur Ausgabe

- Initialisierung der Simulation:
  - globale Variablen:

```
(define der-plan (konstr-plan))
(define (inverter-verzoegerung    2))
(define (und-gatter-verzoegerung  3))
(define (oder-gatter-verzoegerung 5))
```

  - Definition von Drähten (s.o.)
  - Definition von Gattern
  - Anbringen von Sonden an Drähte
  - Setzen von Signalen und Aufruf von `(fortfuehren)`

4.3.2 Modelle mit Beschränkungen

Traditionelle Programmierung: Wertberechnung in einer Richtung (funktional); Voraussetzung: Berechnungsrichtung zur Compilerzeit bekannt;

ALTERNATIV: Richtung unbekannt, Werte durch Relationen verknüpft, Ausnutzung der Relation in verschiedenen Richtungen, sobald Werte bekannt sind.

Beispiel: Temperaturumrechnung Celsius ↔ Fahrenheit. Relation: $9 \cdot C = 5 \cdot (F - 32)$; Falls C bekannt kann man F berechnen, falls F bekannt, kann man C berechnen - beide Richtungen sind machbar und notwendig.

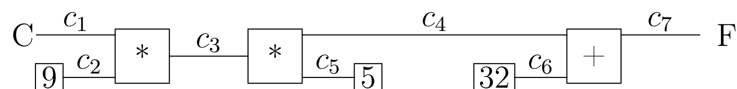
⇒ Idee des *Rechnens mit Beschränkungen* (constraint), Propagierung von Werten; Anwendung:

- Künstliche Intelligenz (Wissensverarbeitung, unvollständiges Wissen)
- Operations Research: Optimierung bezüglich gegebener Beschränkungen, etwa
  - Containerverladung
  - Produktionsabläufe
  - Terminalvergabe bei Flughafen

Wertpropagierung? Realisierung durch Beschränkungsnetze bestehend aus

- elementare Beschränkungen: Relation zwischen Größen:
  - `(addierer x y z)` entspricht der Addition $x + y = z$
  - `(multiplikator x y z)` entspricht der Multiplikation $x * y = z$
  - `(konstante c x)` entspricht Relation $x = c$
- Konnektoren: Verknüpfung von Relationen zur Programmierung von Welten

Beispielnetzwerk für $9 \cdot C = 5(F - 32)$:



Berechnung im Netzwerk:

- Falls Konnektor Wert erhält: Wecke alle angeschlossenen Beschränkungen (außer die, von der der Wert kommt) und informiere über ihren neuen Wert
- Falls Beschränkung geweckt: Prüfe, ob angeschlossene Konnektoren genug Werte haben, um Werte weiterzuleiten, und leite dann Werte weiter.

Beispiel: Falls C und F ohne Wert: keine Propagation.

- Setze Wert von C auf 25

$\Rightarrow$ $c_1 = 25$

$\Rightarrow$ $c_3 = 225$ (1. Multiplikator geweckt)

$\Rightarrow$ $c_4 = 45$ (2. Multiplikator geweckt)

$\Rightarrow$ $c_7 = 77$ (Addierer geweckt)

Implementierung des Netzwerks mit Beschränkungssystem:

- `(konstr-konnektor)` erzeugt neuen Konnektor, Argument für elementare Beschränkungen

```
(define C (konstr-konnektor))
(define F (konstr-konnektor))
(define (celsius-fahrenheit-konverter c f)
        (let ((c2 (konstr-konnektor))
              (c... (konstr-konnektor))
              (c6 (konstr-konnektor)))
             (multiplikator c c2 c3)
             (multiplikator c4 c5 c3)
             (addierer c4 c6 f)
             (konstante 9 c2)
             (konstante 5 c5)
             (konstante 32 c6)))

(celsius-fahrenheit-konverter C F)
```

Analogie zur Schaltkreissimulation: Elemente Beschränkungsobjekte und Abstraktionen der Programmiersprache, $\Rightarrow$ Sprache zur Definition komplexer Beschränkungen

Beobachtung der Netzaktivitäten durch Anbringen von „Sonden“, die Wertänderungen an Konnektoren ausgeben:

```
> (sonde "celsius" C)
> (sonde "fahrenheit" F)
> (set-wert! C 25 'benutzer)
sonde: celsius = 25
sonde: fahrenheit = 77
```

Weiter mit ungekehrter Reihenfolge:

```
> (set-wert! F 212 'benutzer)
error: Widerspruch
```

Ursache: Konnektor F hat noch Wert 77 - also zunächst Wert löschen

```
> (vergessen-wert! C 'benutzer)
sonde: celsius = ?
sonde: fahrenheit = ?
> (set-wert! F 212 'benutzer)
sonde: fahrenheit = 212
sonde: celsius = 100
```

Implementierung des Beschreibungssystems:

- Datentyp der Konnektoren: Grundoperationen
- `(hat-wert? k)` Hat der Konnektor k einen Wert?
- `(get-wert k)` liefert aktuellen Wert
- `(set-wert! k wert informant)` ein Informant möchte Wert setzen
- `(vergiss-wert! k rueckziehender)` ein Rückziehender möchte den Wert vergessen lassen
- `(verbinde k neue-beschraenkung)` Beteilige k an `neue-beschraenkung`

Implementierung von elementaren Beschränkungen:

- Objekte, die Nachrichten verarbeiten, z.B. $'ich - habe - wert$, $'ich - verlor - meinen - wert$
- Falls neue Wert verfügbar: Prüfe, ob Wertpropagation möglich ist (z.B. bei Multiplikator: falls ein Argument null ist, ist das Ergebnis null)

Implementierung Konnektoren: lokaler Zustand

- Wert: Aktueller Wert oder `()`
- Informant: Objekt, das Wert gesetzt hat (Beschränkung oder 'benutzer)
- Beschränkungen: Liste der Beschränkungen, mit denen der Konnektor verbunden ist

4.4 Zustände in nebenläufigen Systemen

Bisher in Kapitel (2) und (3): keine Zustände, referenzielle Transparenz, damit: einfaches Substitutionsmodell, Ausdrücke sind zeitlos.

Kapitel (4): Lokale Zustände und veränderbare Datenstrukturen führen zum Umgebungsmodell, Werte in Ausdrücken zeitabhängig (Wert des Ausdrucks abhängig von der Form des Ausdrucks und der Vorgeschichte, d.h. allen vorherigen Auswertungen (diese sind eventuell überschaubar bei streng sequentieller Ausführung)).

Aber: Die reale Welt ist nebenläufig (concurrent): unabhängige Objekte, die allein agieren und interagieren. Programme sollten reale Welt möglichst einfach abstrahieren $\Rightarrow$ Anwendungsprogramme sind auch nebenläufig zu organisieren (auch wenn diese dann sequentiell verarbeitet werden).

Nebenläufige Anwendungen werden immer wichtiger: Internet, verteilte Informationssysteme, Simulationen, ... (Programmiertechniken: z.B. herkömmliche Sprachen mit prozeßorientierten Bibliotheken oder neuere Programmiersprachen, die nebenläufige Konstrukte enthalten, wie z.B. Java).

Nebenläufigkeit bringt Probleme bei Zuständen, die von mehreren „Agenten“ verändert werden, da Zeitaspekte[2] besonders relevant werden.

Betrachte Abheben vom Bankkonto:

```
(define (abheben betrag)
        (if (>= kontostand betrag)
            (begin (set! kontostand (- kontostand betrag))
                   kontostand)
            "Deckung nicht ausreichend"))
```

Annahme: Peter und Paul haben ein gemeinsames Konto mit Kontostand 130 und sie heben jeweils 100 an verschiedenen Automaten ab.

Möglichkeit bei „gleichzeitigem“ Abheben:

[2] „Stellen Sie sich mal vor, eine Person hebt gleichzeitig bei zwei Geldautomaten Geld ab.“

- Beide dürfen abheben und die `begin`-Ausdrücke werden nacheinander ausgeführt: Peter hebt ab, `kontostand` ist 30; Paul hebt ab, `kontostand` = -70.
- Beide dürfen abheben und die `begin`-Ausdrücke werden nebeneinander ausgeführt:

Peter	Konto	Paul
	130	
$(>= \ 130\ 100) = \#t$	130	$(>= \ 130\ 100) = \#t$
$(-\ kontostand\ betrag) = 30$	130	
	130	$(-\ kontostand\ betrag) = 30$
	30	$(set!\ kontostand\ 30)$
$(set!\ kontostand\ 30)$	30	

Notwendig: Kontrolle kritischer Programmbereiche! *Semaphore*: Signale (z.B. Werte 0,1) mit Operationen P und V (von „passieren“/„vrijgeven“, DIJUSTRA 1968, Signale bei Eisenbahnen):

- $(P\ s)$: falls $s = 1$, setze $s = 0$. Sonst: stoppe Ausführung bis $s = 1$ (Eintrag in Warteschlange)
- $(V\ s)$: Setze s auf 1, aktiviere einen Wartenden (falls vorhanden)

Wichtig: P und V sind unteilbare Operationen (Betriebssystem garantiert, daß nur ein Prozess P oder V für einen Semaphor ausgeführt wird).

Verhinderung der Bankpleite: Erzeuge Semaphor s und verändere den Code folgendermaßen:

```
(define (kontr-abheben betrag)
        (P s)
        (abheben betrag)
        (V s))
```

Damit kann nur ein Prozess gleichzeitig abheben. Nachteil: kein gleichzeitiges Abheben von verschiedenen Konten möglich; Lösung: Erzeuge für jedes Konto eine eigene lokale Semaphor:

```
(define (konstr-konto kontostand)
        (let ((s (konstr-semaphor)))
             (define (abheben betrag)
             ...)))
```

Nebenläufigkeitskontrolle führt zu neuen Problemen: Verklemmung (deadlock), Beispiel: Überweisung zwischen Konten K_1 und K_2, da Beträge in K_1 und K_2 verändert werden, muss diese Operation kontrolliert werden. Sei s_i Semaphor des Kontos K_i,

```
(define (ueberweisen k1 k2 s1 s2 betrag)
        (P s1) (P s2)
        <überweisung>
        (V s2) (V s1))
```

mögliche Problemsituation: Parallele Überweisung von K_1 auf K_2 und umgekehrt, jede Überweisung sperrt zunächst das Quellkonto, beide warten (endlos) auf das andere.

Vermeidung: sorgfältige Planung von „Vermeidungsstrategien“, aber: oft schwierig zu überschauende Zeitabhängigkeiten. Fazit: Zuständsänderung $\Rightarrow$ Zeitaspekt relevant, schwierige Kontrolle in realen Systemen.

Konsequenzen für Programmentwurf:

- zuständsfrei (referentielle Transparenz) wo möglich
- Zustände lokal halten, alle Änderungen kontrollieren (Nachrichtenweitergabe)
- Änderungsoperationen synchronisiren, mögliche Verklemmungen ausschließen

4.5 Datenströme (*streams*) als Standardschnittstellen

- weitere Abstraktionstechnik zur Organisation größer Programme
- besserer modularer Aufbau
- Modellierung von Objekten mit Zuständen ohne explizite Zustände

$\Rightarrow$ Vermeidung der theoretischen Probleme von Zustandstransformationen

Zur Erinnerung: Prozeduren höherer Ordnung: Herausziehen von Gesetzmäßigkeiten bezüglich Prozeduraufrufstruktur; wünschenswert: auch für Datenabstraktion/-fluss.

Beispiele:

1. Summiere in Binärbaum mit ganzzahligen Blättern die Quadrate aller ungeraden Blätter:

```
(define (summe-ug b)
        (if (blatt? b)
            (if (ungerade? b) (quadrat b) 0)
            (+ (summe-uq (linker-ast b))
               (summe-uq (rechter-ast b)))))
```

2. Liste der ungerade Fibonacci-Zahlen bis n

```
(define (ungerade-fiber n)
        (define (naechstes k)
                (if (> k n) '()
                    (let ((f (fib k)))
                         (if (ungerade? f)
                             (cons f (naechstes (+ k 1)))
                             (naechstes (+ k 1))))))
        (naechstes 1))
```

Unterschiedlicher Programmaufbau, aber „logische“ Gemeinsamkeiten:

		`summe-uq`	`ungerade-fibs`
Schritte		Aufzählung der Blätter	Aufzählung aller ganzen Zahlen
		Filtern der ungeraden	Fib-Berechnung
		Quadrieren	Filtern der ungeraden
		Akkumulieren	Akkumulieren der Liste
Datenfluss	Generator*	Blätter	Zahlen
	Filter	ungerade	ungerade
	Abbilder	Quadrat	fib
	Akkumulator**	Summe	Liste

* Daten*strom* wird erzeugt; ** Ergebnisse werden gesammelt

- Jedoch: Struktur im Programm nicht explizit sichtbar, alle Elemente vermischt
- Ziel: modularer Aufbau bezüglich des Datenflusses
- Vorteile: übversichtlich, lesbar, universelle Bausteine
- Methode: statt Berechnungsreihenfolge Datenfluss betrachten!
- Datenflussorientierte Programmstruktur mit *Datenströmen*: Folge von Elementen

Operationen auf Strömen:

- `cons-strom`: Konstruiere neuen Strom
- `kopf`: erstes Element
- `rest`: restlicher Strom
- `der-leere-strom`: leerer Strom ohne Elemente
- `leerer-strom?`: ist Strom leer?

Gesetze für Ströme:

```
(kopf (cons-strom a b)) = a
(rest (cons-strom a b)) = b
(leerer-strom? (cons-strom a b)) = #f
(leerer-strom? der-leere-strom) = #t
```

Implementierung:

- z.B. als Listen (`cons`, `car`, `cdr`, `'()`, `null`)
- später: spezielle Implementierung (größe Ströme)

4.5.1 Implementierung der Ströme

1. Summieren im Binärbaum

    - **Generator** für Blattstrom aus Baum:

    ```
    (define (gen-baum b)
            (if (blatt? b)
            (cons-strom b der-leere-strom)
            (append-strom (gen-baum (linker-ast b))
                          (gen-baum (rechter-ast b)))))
    ```

    - **Filter** für ungerade Zahlen:

    ```
    (define (filter-ungerade s)
            (cond ((leerer-strom? s) der-leere-strom)
                  ((ungerade? (kopf s))
                   (cons-strom (kopf s)
                               (filter-ungerade (rest s))))
                  (else (filter-ungerade (rest s)))))
    ```

- **Abbilder** für Quadrate

```
(define (abb-quadrat s)
        (if (leerer-strom? s)
            der-leere-strom
            (cons-strom (quadrat (kopf s))
                        (abb-quadrat (rest s)))))
```

- **Akkumulator**: Aufsummieren der Stromelemente

```
(define (akkumulierer-+ s)
        (if (leerer-strom? s)
            0
            (+ (kopf s) (akkumuliere-+ (rest s)))))
```

- **Neudefinition** von `summe-uq`:

```
(define (summe-uq b)
        (akkumuliere-+
            (abb-quadrat
                (filter-ungerade
                    (gen-baum b)))))
```

2. Summieren ungerade Fibonacci-Zahlen

- **Generator**: Erzeugt Liste der Zahlen 1 bis n
- **Abbilder**: Fibonnaci
- **Filter**: ungerade
- **Akkumulator**: Liste (`cons`)
- **Neudefinition** von `ungerade-fibs`:

```
(define (ungerade-fibs n)
        (akkumuliere-cons
            (filter-ungerade
                (abb-fib
                    (gen-intervall 1 n)))))
```

Zunächst: (sehr) umständlich, aber Vorteil: einfaches Zusammensetzen zu neuen Funktionen, Beispiel: Summe der Quadrate der ersten n Fibonacci-Zahlen;

```
(define (summe-qf n)
        (akkumuliere-+
            (abb-quadrat
```

```
                                (abb-fib
                                    (gen-intervall 1 n)))))
```

Nachteil: Prozeduren wie `abb-fib` oder `abb-quadrat` sind zu speziell, Verallgemeinerung:

4.5.2 Universelle Prozeduren für Datenströme

Verschiedene Abbilder, Filter und Akkumulatoren unterscheiden sich nur durch Abbildungs-, Filter- bzw. Akkumulierfunktion ⇒ praktisch: Funktion als Parameter

```
3(define (filter f s)
        (if (leerer-strom? s)
            der-leere-strom
            (if (f (kopf s))
                (cons-strom (kopf s)
                            (filter f (rest s)))
                (filter f (rest s)))))
(define (akk f n s)
        (if (leerer-strom? s)
            n
            (f (kopf s)
               (akk f n (rest s)))))
```

Dritte Definition von `summe-uq` und `ungerade-fibs`:

```
(define (summe-uq b)
        (akk + 0
            (abb quadrat
                (filter ungerade?
                    (gen-baum b)))))
(define (ungerade-fibs n)
        (akkumuliere cons '()
            (filter ungerade?
                (abb fib
                    (gen-intervall 1 n)))))
```

Universeller Akkumulator vielfach anwendbar: Umwandlung von Strom von Strömen in einen Strom:

```
(define (glaetten s)
        (define (append-stroeme s1 s2)
```

```
                (if (leerer-strom? s1)
                    s2
                    (cons-strom (kopf s1)
                                (append-stroeme (rest s1)
                                                s2))))
        (akk append-stroeme
             der-leere-strom s))
```

Anwendung einer Prozedur (mit evtl. Zustandsänderung) auf jedes Stromelement:

```
(define (fuer-jedes proz s)
        (if (leerer-strom? s)
            'fertig
            (begin (proz (kopf s))
                   (fuer-jedes proz (rest s)))))
```

4.5.3 Beispiele

Neue Beispiele einfach realisierbar:

- Produkt der Quadrate der ungeraden Zahlen zwischen 1 und n:

```
(define (prod-qu n)
        (akk * 1
            (abb quadrat
                (filter ungerade?
                    (gen-intervall 1 n)))))
```

- Polynomauswertung mit Hornerschema (Motivation: möglichst wenige Additionen und Multiplikationen):

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x_1 + a_0 = ((\cdots((a_n x + a_{n-1}) \cdot x + a_{n-2}) \cdot x \cdots a_1) \cdot x + a_0)$$

```
(define (horner-schema x koeff-strom)
        (define (add-term koeff hoehere-terme)
                (+ koeff (* x hoehere-terme)))
        (akk add-term 0 koeff-strom))
```

- nicht nur für mathematische Probleme auch für konventionelle Datenverarbeitungs-Probleme; Gegeben: Strom von Studentendaten; Gesucht: Liste der Studierenden mit Klausurqualifikation

```
(define (klausur-qualif studenten-strom)
        (akk cons '()
             (filter >-haelfte-uebungspunkte?
                  (filter hoert-informatik-I
                      studenten-strom))))
```

4.5.4 Geschachtelte Abbildungen, `collect`

Abbilder bisher: $Strom \stackrel{abb\ f}{\longrightarrow} Strom$; Manchmal notwendig: Abbildungen auf mehreren verschachtelten Strömen,

Beispiel: Gegeben: n, Gesucht: ganze Zahlen i, j mit $1 \leq j < i \leq n$ mit $i + j = x$ und x Primzahl; gewünschtes Ergebnis für $n = 5$:

i	j	$i+j$
2	1	3
3	2	5
4	1	5
4	3	7
5	2	7

Lösungsansatz:

1. Erzeuge den Strom von Paaren (i, j) mit $1 \leq j < i \leq n$ durch:
   (a) Erzeuge einen Strom der natürlichen Zahlen $1, 2, ..., n$
   (b) Erzeuge für jedes Element i aus (a) einen Strom $(i, 1) \cdots (i, i-1)$
2. Filtere Paare mit $i + j = x_{prim}$

Einfache Realisierung mit universellen Stromprozeduren:

```
(glaetten
     (abb (lambda (i)
          (abb (lambda (j) (list i j))
               (gen-intervall 1 (- i 1))))
          (gen-intervall 1 n)))
```

Kombination:

$$Strom \stackrel{Abbildung f. Element \rightarrow Strom}{\longrightarrow} Strom\ von\ Strömen \stackrel{glaetten}{\longrightarrow} Strom$$

Damit neue Verallgemeinerung:

```
(define (glaettabb f s) (glaetten (abb f s)))
```

Gesamtlösung:

```
(define (primzahl-summe-paare n)
        (abb (lambda (paar)
                     (list (car paar)
                           (cadr paar)
                           (+ (car paar)
                              (cadr paar))))
             (filter (lambda (paar)
                             (primzahl? (+ (car paar)
                                           (cadr paar))))
                     (glattabb (lambda (i)
                                       (abb (lambda (j)
                                                    (list i j))
                                            (gen-intervall 1 (- i 1))))
                               (gen-intervall 1 n)))))
```

Allgemeines Prinzip geschachtelter Abbildungen:

- Generiere Strom für erste Komponente (oben: i)
- Generiere Strom für zweite Komponente (oben: j)
- ...
- Generiere Strom für n-te Komponente
- Filtere n-Tupel (oben: Primzahlsumme)
- Produziere Ergebnis $(oben\ (i,j)) \mapsto (i,j,i+j)$

Folgende Sonderform (syntaktische Zucker) hierfür praktisch ($\notin$ SCHEME-Standard):

```
(collect <ergebnis>
         ((<v1> <menge1>)
          (<v2> <menge2>)
          ...
          (<vn> <mengen>))
         <einschränkung>)
```

Bedeutung von `collect`:

$\{\langle ergebnis\rangle \mid \langle v_1\rangle \in \langle menge_1\rangle, ..., \langle v_n\rangle \in \langle menge_n\rangle \text{ mit } \langle einschraenkung\rangle\}$

Beispiel:

```
(define (primzahl-summe-paare n)
        (collect (list i j (+ i j))
                 ((i (gen-intervall 1 n)
                  (j (gen-intervall 1 (- i 1)))))
                 (primzahl? (+ i j))))
```

Implementierung von „collect“ durch Ausdruck

```
(abb (lambda (tupel) (let ((<v1> (car tupel))
                           (<vn> (cad..dr tupel)))
                          <ergebnis>))
     (filter (lambda (tupel) (let ((...))
                                  <einschraenkung>))
             (glattabb (lambda (<v1>)
                       (glattabb (lambda (<v2>)
                       ...
                           (abb (lambda (<v_n>) (list <v1>...<vn>))
                                <menge-n>)
                              ...
                           <menge-2))
                       <menge-1>))))
```

Beachte: `<menge-i>` nicht nur Intervall, sondern beliebige Menge ($\hat{=}$ Strom) sein, weitere interessante Anwendungen:

Beispiel: Erzeuge Permutationen einer Elementmenge S, z.B. $S = \{a, b, c\}$
Lösung durch Rekursion: Für jedes Element $x \in S$:

- Erzeuge Permutationen von $S \setminus \{x\}$
- Füge dann x an den Anfang all dieser an

Implementierung:

```
(define (permutationen S)
        (if (leerer-strom? S)
            leere-permutationen
            (glattabb (lambda (x)
                              (abb (lambda (p)
                                           (cons-strom x p))
                                   (permutationen (entfernen x S)
            )))       S)))
(define leere-permutationen (cons-strom der-leere-strom
                                        der-leere-strom))
(define (entfernen x S)
        (filter (lambda (e) (not (equal? e x)))
                S))
```

Implementierung mit „collect“:

```
(define (permutationen S)
       (if (leerer-strom? S)
           leere-permutationen
           (collect (cons-strom x p)
                    ((x S)
                     (p (permutationen (entfernen x S))))
                    #t)))
```

4.5.5 Implementierung von Datenströmen

- Einfache Implementierung: Strom als Liste (`const-strom` ist `cons`, `kopf` ist `car` und `rest` ist `cdr`);
- Nachteil: unnötiger Aufbau großer Strukturen (wg. Scheme-Auswertung)
- Beispiel: Berechne zweite Primzahl im Intervall 10.000 bis 1.000.000:

  ```
  (kopf (rest (filter primzahl? (gen-intervall 10000 1000000))))
  ```

  Argumentauswertung: Aufbau von Liste mit fast einer Million Elementen!

⇒ *Verbesserung:* nicht Listen als Datenströme, kein vollständiger Aufbau von Stromstrukturen, sondern nur soweit, wie benötigt:

  ```
  cons-strom -> (cons e <prozedur zur berechnung des reststroms>)
  ```

- Realisierung durch *verzögerte Auswertung*
- SCHEME-Sonderform: `(delay <ausdruck>)`, Ergebnis: keine Auswertung von `<ausdruck>`, sondern (verzögertes) Objekt `O` (auch *Versprechen (promise)* genannt), das später mittels `(force O)` ausgewertet werden kann.
- Gesetz für verzögerte Objekte: `(force (delay O)) = O`
- Alternativimplementierung:

  ```
  (cons-strom a b) = (cons a (delay b))
  (define (kopf s) (car s))
  (define (rest s) (force (cdr s)))
  ```

  Hierfür gilt:

```
(kopf (cons-strom a b)) = (car (cons a (delay b))) = a
(rest (cons-strom a b)) = (force (cdr (cons-strom a b)))
                        = (force (cdr (cons a (delay b))))
                        = (force (delay b)) = b
```

- Effekt der verzögerten Auswertung:

```
(define (gen-intervall u 0)
        (if (> u 0)
            der-leere-strom
            (cons-strom u (gen-intervall (+ u 1) 0))))
(kopf (rest (filter primzahl? (gen-intervall 10000 1000000))))
```

Auswertung[3]:

1. Auswertung von `(gen-intervall 10000 1000000)` ergibt:
   `(cons 10000 »Verz. (gen-intervall 10001 1000000)«)`
2. `Filter`-Auswertung: 10000 ist keine Prizahl, filtere den Rest von
   `(cons 10000 »Verz. (gen-intervall 10001 1000000)«)`
3. Auswerung der Verzögerung mittels force ergibt:
   `(cons 10001 »Verz. (gen-intervall 10002 1000000)«)`
4. ...
5. `Filter`-Auswertung: 10007 ist Prizahl und gibt zurück
   `(cons 10007 »Verz. (filter prim? (cons 10008 »...«))«)`
6. `Rest`-Auswertung: Werte die Verzögerung aus
   `(filter primzah? (cons 10008 »Verz. (gen-intervall...)«)`
7. `Filter`-Auswertung: 10008 ist keine Prizahl, filtere den Rest von
   `(cons 10009 »Verz. (gen-intervall 10001 1000000)«)`
8. `Filter`-Auswertung: 10009 ist Prizahl und gibt zurück
   `(cons 10009 »Verz. (filter prim? (cons 10010 »...«))«)`
9. `Rest`-Auswertung: Gibt direkt zurückt:
   `(cons 10009 »Verz. (filter prim? (cons 10010 »...«))«)`
10. `Kopf`-Auswertung: gibt `10009` zurück

! fertig

⇒ kein Aufbau von Strom `10000..1000000`, sondern nur so weit ausgewertet, wie Elemente benötigt werden, sogenannte *anforderungsgesteuerte Auswertung*.

[3]„Das SCHEME-System macht das natürlich schneller, das braucht ja nicht so viel an die Tafel zu schreiben.“

- Hierdurch: Kombination von
  - datenorientierten Programmstrukturen (erzeuge Daten, manipuliere Daten, ..., akkumuliere Daten etc.)
  - inkrementelle Auswertung (erzeuge Element, manipuliere dieses, erzeuge nächstes Element, ...)

⇒ Entkopplung der Programmstruktur von Auswertungsstruktur

4.5.6 Implementierung der verzögerten Auswertung

- Ausdrücke, die nicht sofoert ausgewertet werden: Prozedurrümpfe! Mögliche Implementierung:

```
(delay <ausdruck>) = (lambda () <ausdruck>)
(define (force objekt) (objekt))
```

- Nachteil bei Anwendungen, in denen ein Strom eventuell mehrfach verwendet wird: Mehrfache Auswertung desselben verzögerten Objekts.
- Verbesserung: Speichere Ergebnis bei erster Anwendung und liefere dieses Ergebnis direkt bei weiterer Auswertung; Implementierung mit *tabellierter* Prozedur
- Generelle Prozedur zur verbesserung einer parameterlosen Prozedur:

```
(define (tab-proz proz)
        (let ((ausgewertet? #f)
              (ergebnis #f))
             (lambda () (if ausgewertet?
                            ergebnis
                            (begin (set! ergebnis (proz))
                                   (set! ausgewertet? #t)
                                   ergebnis)))))
```

- Neue Version von `delay`:

```
(delay <ausdruck>) = (tab-proz (lambda () <ausdruck>))
```

4.5.7 Datenströme unendlicher Länge

- Konsequenz der verzögerten Auswertung: nur benötigte Anteile ausrechen, daher unendliche Ströme möglich, falls endliche Anteile benötigt.
- Beispiel: Strom ganzer Zahlen:

```
(define (zahlen-von n)
        (cons-strom n (zahlen-von (+ n 1))))
(define alle-zahlen
        (zahlen-von 0))
```

- Zugriff auf das n-te Stromelement ($n = 0$ bedeutet Kopf):

```
(define (n-tes-strom n s)
        (if (= n 0)
            (kopf s)
            (n-tes-strom (- n 1) (rest s))))
```

- Auswertung von `(n-tes-strom 5 alle-zahlen)`: Bei Implementierung als Liste würde unendliche Berechnung erfolgen, mit Verzögerung wird das korrekte Ergebnis ausgegeben
- Unendliche Ströme wie üblich bearbeiten, Beispiel: Strom aller Zahlen, die nicht durch 7 teilbar sind;

```
(define (teilbar? x y) (= (remainder x y) 0))
(define ohne-sieben
        (filter (lambda (x) (not (teilbar? x 7)))
                alle-zahlen))
> (n-tes-strom 50 ohne-sieben)
59
```

- weiteres Beispiel für unendliche Ströme: Strom aller Fibonacci-Zahlen

```
(define (fibgen a b)
        (cons-strom a (fibgen b (+ a b))))
(define fibs (fibgen 0 1))
```

Beachte: rekursive Definition (ohne Interator); lineare Laufzeit!

- noch ein weiteres Beispiel: Primzahlen; Berechnungsmethode: Sieb des Erathostenes

  1. konstruiere ganze Zahlen ab 2
  2. streiche im Reststrom alle durch 2 teilbaren Zahlen
  3. Ergebnis: Strom, beginnend mit 3
  4. Streiche im Rest von diesen alle durch 3 teilbaren Zahlen
  5. Generell: Streiche im Rest eines Stromes (beginnend mit x) alle durch x teilbaren Zahlen
  6. Ergebnis: Die Köpfe aller Ergebnisströme sind Primzahlen (da nicht gestrichen)

```
(define (siebe strom)
  (cons-strom kopf strom)
    (siebe (filter (lambda (x)
                     (not? (teilbar x (kopf strom))))
                   (rest strom))))
(define primzahlen (siebe (zahlen-von 2)))
> (n-tes-strom 50 primzahlen)
233
```

4.5.8 Imlizite Definition von Strömen

bisher:

- durch Generator (`gen-intervall`...)
- durch rekursive Prozedur zum Aufzählen der Elemente

durch verzögerte Auswertung möglich: **Ströme durch Ströme definieren**
Beispiel:

- Unendliche Strom der Einsen:
  `(define einsen (cons-strom 1 einsen))`
- Verwendung durch Kombination mit „Stromaddierer“

```
(define (add-strom s1 s2)
        (cond ((leerer-strom? s1) s2)
              ((leerer-strom? s2) s1)
```

```
               (else (cons-strom (+ (kopf s1)
                                    (kopf s2))
                                 (add-strom (rest s1)
                                            (rest s2))))))
(define nats (cons-strom 1 (add-stroeme einsen nats)))
(define fibs (cons-strom 0
               (cons-strom 1
                 (add-stroeme (rest fibs) fibs))))
```

- Berechnung:

```
0 1 1 2 3 5  8 13
  0 1 1 2 3  5  8
  - - - - - -- --
  1 2 3 5 8 13 ...
```

- Damit: Ströme mit „Rückkopplung“ (Analogie zu Schaltkreisen)
- Vorteil unendlicher Ströme: Trennung zwischen *Logik* (was soll berechnet werden?) und *Kontrolle* (wieviel soll berechnet werden?)
- Beispiel `fibs`: Logik ist Strom aller Fibonacci-Zahlen; Kontrolle ist: Berechne n-te oder erste n Fibonacci-Zahlen; ohne Ströme: verschiedene Programme
- Geschachtelte Abbildungen mit unendlichen Strömen, Beispiel: Strom von Zahlen mit „collect“:

```
(define zahl-paare
        (collect (list i j)
                 ((i alle-zahlen)
                  (j alle-zahlen))
                 #t))
```

Direkt mit „glattabb“

```
(define zahl-paare (glattabb (lambda (i) (abb (lambda (j)
                                                  (list i j))
                                               alle-zahlen))
                             alle-zahlen)
```

Filtern der Paare mit erster Komponente gleich 2:

```
(filter (lambda (paar) (= (car paar) 2)) zahl-paar)
```

- terminiert nicht! Ursache: `glattabb` benutzt `glaetten` (siehe (4.5.2)), Eingabe dafür: unendlicher Strom von unendlichen Strömen, die Ausgabe ist

  ```
  ((0 0) (0 1) (0 2) (0 ...) ...)
  ```

- Verbesserung: Konkateniere unendliche Ströme nicht sequentiell, sondern „verzahne" sie (Reißverschlußverfahren):

  ```
  (define (verzahnen s1 s2)
          (if (leerer-strom? s1)
              s2
              (cons-strom (kopf s1)
                          (verzahnen s2 (rest s1)))))
  (define (glaetten s)
          (akk verzahnen der-leere-strom s))
  ```

- terminiert immernoch nicht! Ursache: Auswertung von `akk`:

  ```
  (define (akk f n s)
          (if (leerer-strom? s)
              n
              (f (kopf s)
                 (akk f n (rest s)))))
  ```

  Auswertung der `op`-Kombination: Auswertung von `(akk ...)` vor Anwendung von `op`. Damit: unendlich viele Instanzen von `akk`.

  Lösung: Auswertung dieses Arguments mit `delay` verzögern, Konsequenz: `verzahnen` hat verzögerte Objekte als zweites Argument, damit: Änderung im Rumpf von `verzahnen`:

  ```
  (define (verzahnen s1 s2)
          (if (leerer-strom? s1)
              (force s2)
              (cons-strom (kopf s1)
                          (verzahnen (force s2)
                                     (delay (rest s1))))))
  ```

- Beachte:

  - Letzte Änderung nur notwendig wegen fester (applikativer) Auswertungsreihenfolge
  - Unnötig bei Auswertung in Normalordnung, d.h. Übergabe der Parameter ohne Auswertung:
  - Konstruktoren verzögern Auswertung (mit `delay`), Selektoren aktivieren Auswertung (mit `force`)
  - realisiert in modernen funktionalen Sprachen (Miranda, Haskell)
  - Schwerwiegendes Problem der Normalordnung:
    * Berechnungsreihenfolge schwierig vorhersagbar (nicht durch Programmstruktur, sondern durch Datenfluß)
    * Kombination mit Veränderungen (Zuweisungen) unklar
    * aktiver Forschungsgegenstand

4.5.9 Datenströme ↔ lokale Zustände

- Was sind lokale Zustände? Größen (Objektteile), die sich zeitlich ändern; alternative Modellierung von Zuständen: Datenstrom der Werte; Vorteil:
  - Vermeidung von `set!`
  - Vermeidung der Probleme bei Zustandsänderungen
  - durch verzögerte Auswertung: nur benötigte Teile generieren
- Beispiel: `konto` mit `abheben` (vereinfacht!):

```
(define (konstr-abheben kontostand)
        (lambda (betrag) (set! kontostand
                               (- kontostand betrag))
                         kontostand))
> (define K (konstr-abheben 200))
> (K 30)
170
> (K 40)
130

(define (strom-abheben kontostand bestragsstrom)
  (cons-strom kontostand
    (strom-abheben (- kontostand
                      (kopf betragsstrom))
                   (rest betragsstrom))))
```

- `strom-abheben`: wohldefinierte mathematische Funktion (keine Seiteneffekte!)
- Aus Benutzersicht: gleiches Verhalten wie mit Zuständen, Implementierung ohne die theoretischen Probleme der Programmierung mit Zuständen, da Zustände nur durch die Zeitsicht des Benutzers entstehen
- neue generische Operation (wird gleich benötigt): Abbilder für binäre Abbildung auf aufeinanderfolgende Zahlen:

```
(define (abb-paar f s)
        (cons-strom (f (kopf s)
                       (kopf (rest s)))
                    (abb f (rest (rest s)))))
```

- Beispiel: Monte-Carlo-Simulation aus (4.1.1):

```
(define zufall
        (let ((x zufalls-init))
             (lambda () (set! x (zufall-aktuell x))
                        x)))
```

Alternativ: Strom von Zufallszahlen (implizite Definition), dann Cesaro-Test und π-Näherung:

```
(define zufalls-zahlen
        (cons-strom zufalls-init
                    (abb zufall-aktuell
                         zufalls-zahlen)))
(define cesaro-strom
        (abb-paare (lambda (z1 z2)
                           (= (ggt z1 z2) 1))
                   zufalls-zahlen))
(define (monte-carlo experimente aw af)
        (define (naechstes aw af)
                (cons-strom (/ aw (+ aw af))
                            (monte-carlo (rest experimente)
                                         aw af)))
        (if (kopf experimente) (naechstes (+ aw 1) af)
                               (naechstes aw (+ 1 af))))
(define pi (abb (lambda (p) (wurzel (/ 6 p)))
                (monte-carlo cesaro-strom 0 0)))
```

- kein fester Wert für die Anzahl der Experimente (mehr Modularität), stetige Verbesserung des Wertes, ähnliche Modularität wie mit Zuständen: `monte-carlo` unabhängig von Zufallsberechnung

4.5.10 Diskussion: Datenströme ↔ Objekte mit Zuständen

- Modellierung zeitlicher Änderungen durch Ströme ⇒ Vermeidung von Zustandsproblemene
  - logische Schlußfolgerung über Programme leichter
  - Vermeidung von Zeitgrenzen („wann" ist nicht wichtig)
  - Wert einer Varaiblen unterscheidet sich vor und nach einer Zuweisung

  ⇒ Berücksichtigung der Zeit für Korrektheitsbeweise
- Funktionale Programmiersprachen
  - Prozeduren $\hat{=}$ Funktionen ihrer Argumente (wohldefiniert!)
  - keine Zuweisung („Scheme ohne `set!`")
- Vorteile:
  - einfache Korrektheitsbeweise
  - hochgradig für Parallelverarbeitung geeignet: parallele Berechnung von Ausdrücken, nur Berechnung von Werten, keine Synchronisationsprobleme bei Nebenläufigkeit
  - wichtiger Berechnungsmodell für nebenläufige Systeme (Beispiel: Erlang)
- Offenes Problem: Zustände immer vermeidbar?

- Modellierung interaktiver Systeme:
  - (natürlicher) Ansatz: Objekte mit lokalen Zuständen, die Nachrichten austauschen
  - mit Strömen: Sequenzialisierung der Nachrichten

  Beispiel: Peter und Paul teilen sich ein Bankkonto:
  - objektorientiert: Bankkonto = Objekt (Zustand: Betrag), reagiert auf Nachrichten von Peter und Paul

- stromorientiert: Konto verarbeitet Strom von Nachrichten, Problem: wie liefern sowohl Peter als auch Paul Nachrichten an den Strom?
- Mögliche Lösung: Mischen von Nachrichten durch Mischer, der Peters Transaktionen und Pauls Transaktionen zusammenmischt; nicht einfach `verzahnen` (da Peter und Paul dann immer abwechselnd zugreifen müssen)
- Fairer/nicht deterministischer Mischer:
  - nimmt aus einer der beiden Ströme Eingabe, falls vorhanden
  - wechselt „fair" zwischen den Strömen (keiner muß beliebig lange warten)
    - Zeitbegriff relevant
    - nichtdeterministische Funktion
    - Erweiterung funktionaler Sprachen notwendig

- weiterer Nachteil stromorientierter Modelle:
  - jede Komponente („Objekt") hat Eingabe und Ausgabe
  - problematisch, falls diese Aufteilung nicht bekannt, z.B. Beschränkungsnetze: $A+B=C$: sowohl A, B als auch A, C können Eingabe sein
  - Relationale statt funktionale Sichtweise

- Logische Programmiersprachen (z.B. Prolog)
  - Elementare Einheiten: Relationen statt Funktionen (Beispiel: Relation `MutterVon`)
  - Rechnen durch Schlußfolgerungen: Implikationen statt Prozeduren `A MutterVon B, B MutterVon C => A GrossMutterVon C`
  - keine festgelegten Ein- und Ausgaben, Beispiel: `X GrossMutterVon C` liefert `A`
  - keine Zuweisung: leichte Verifizierbarkeit (Prädikatenlogik)

Zusammenfassung:

- es gibt kein universelles Programmierparadigma, sondern: imperativ/-zustandsorientiert, funktional, datenorientiert, relational etc.
- es gibt keine universelle Programmiersprache (d.h. gleich gut geeignet für jedes Problem)

- wähle je nach Problem passende Modellierung (und Sprache)
- Tendenz:
  - für größe, interaktive Systeme: zustandsorientiert, objektorientiert
  - für algorithmische Apsekte (Korrektheit!): datenorientiert, funktional
  - Integration ermöglichen

5 Einführung in Java

5.1 Allgemeines

- Scheme:
  - einfach erlernbar
  - flexibel: Realisierung unterschiedlicher Programmierstile
  - kompakte Programme
  - Anwendung: rapid prototyping, KI, (Web-)Script-Programmierung, emacs
- Entwicklung großer Systeme:
  - gleiche Programmiertechniken (Abstraktionen)
  - mehr Programmiersicherheit → Deklaration von Programmobjekten (Prüfung durch Compiler) ⇒ größere Programme mit Redundanzen (nur Nachteil bei kleinen Programmen, bessere Lesbarkeit)
- Java:
  - imperative objekt-orientierte Sprache
  - Typdeklarationen
  - Module (packages)
  - Prozesse
  - Netzwerkprogrammierung
  - plattformunabhängig („write once run everywhere“)
  - Sicherheitsprüfungen möglich
  - automatische Speicherverwaltung

- Grundsätzliches Arbeiten mit Java:
  1. Erstellung eines Programmtextes (bel. Editor)

     ```
     class HelloWorld
     { public static void main(String[]args)
          { system.out.println("Hello world"); }}
     ```

     in Datei `hello.java`
  2. Übersetzen in plattformunabhängigen Code:
     `> javac hello.java` in Datei `HelloWorld.class`
  3. Ablauf (Interpretation) des übersetzten Programms:
     `> java HelloWorld` → `Hello world`
- Literatur (klitzekleine Auswahl) siehe Website

5.2 Ausdrücke und Anweisungen

- Kommentare:
  - normal: `// ... bis zum Zeilenende ...`
  - über mehrere Zeilen: `/*    ... */`
  - für automatische Dokumentation mit `javadoc`: `/** ... */`
- Namen:
  - beginnen mit Buchstaben
  - gefolgt von Buchstaben, Ziffern, `_` und $
  - Groß- und Kleinschreibung relevant
- Konstanten: je nach Datentyp
- streng getypte Sprache: jede Variable hat einen festen Typ, nur diese Were dürfen zugewiesen werden (Compilerprüfung), in Scheme möglich:

  ```
  (define x 5)
  (set! x #t)
  ```

  mit Java:

  ```
  int x = 5;
  x = true;
  ```

die zweite Zeile liefert einen error: *incompatible types*. Auch bei Integer/Float etc.

- Konsequenz:
  1. Bei Deklaration einer Variable *muß* Typ angegeben werden
  2. Konstanten (Werte) gehören zu bestimmten Typen

5.2.1 Grundtypen in Java

Typname	Werte	Initialwert	Beispiel
boolean	true, false	false	
char	16-Bit-Unicode-Zeichen	u0000	'a', 'u1111', 'n'
byte	8-Bit ganze Zahlen mit Vorz.	0	
short	16-Bit ganze Zahlen mit Vorz.	0	
int	32-Bit ganze Zahlen mit Vorz.	0	
long	64-Bit ganze Zahlen mit Vorz.	0	
float	32-Bit Gleitkomma-Zahlen mit Vorz.	0.0	314.159E-2f
double	64-Bit Gleitkomma-Zahlen mit Vorz.	0.0	314.159E-2
string	Zeichenkette		"Kieln"

5.2.2 Anweisungen

- in Scheme:
  - alles sind Ausdrücke
  - relvant: Werte von Ausdrücken
  - Zuweisung: Ausdruck mit Seiteneffekt auf Umgebung
  - Variablen: Namen für Werte (veränderbar)
- in Java:
  - Variablen: Namen für veränderbare (!) Speicherzelle
  - Relevant: Veränderung der Variablen durch Anweisungen
  - Alles sind Anweisungen, Strukturierung dieser Anweisungen
- Elementare Anweisung:
  - Zuweisung `x = a;` (muß typkompatibel sein!)
  - Kurzformen: `x++;` sowie z.B. `x += 5;`
- Folge von Anweisungen: `{a1; a2; ...; an }`

- Prozeduraufruf: `p(a1, ..., an);` (Prozeduren sind Funktionen mit Ergebnistyp `void`)
- Fallunterscheidung: `if (ausdruck) anweisung else anweisung` oder

  ```
  switch (ausdruck) {
  case k1: a1; break;
  case k2: a2; break;
  ...
  default: an+1
  }
  ```

- Schleifen: Wiederholungen in SCHEME: rekursive Aufrufe, in JAVA: diverse Schleifenanweisungen:
  - `while (ausdruck) anweisung`
  - `do anweisung while (ausdruck)`
  - `for (init; bexp; iexp) anweisung`
  - `for (int i=1; i <= 1; i++) { x = x + 2 * i }`
- keine generellen Sprünge (`goto`), sondern nur strukturierte Sprünge:
  - `break;` verlässt eine Schleifenstruktur oder `switch`-Anweisung
  - `continue;` verlässt den Schleifenrumpf und fahre mit nächster Iteration fort
  - bei `break` und `continue` sind auch Marken zulässig, um mehrere umgebende Schleifen zu verlassen
  - `return;` verlässt eine Prozedur (Methode)
  - `return ausdruck;` verlässt eine Prozedur (Methode) mit Ergebniswert Ausdruck, falls der Prozedurergebnistyp ungleich `void` ist.

5.2.3 Konzept zur Fehler- und Ausnahmebehandlung

- wichtig bei robuster Software: Behandle Fehler im Programm selbst (ohne Absturz!); beispielsweise das Lesen einer Datei, falls Datei nicht vorhanden: Statt Programmabsturz Ausgabe einer Fehlermeldung und evtl. Alternativaktionen
- Anweisungen können Fehler auslösen („exceptions“)

- Spezielle Anweisungen zur Behandlung von Fehlerfallen
- Generelle Form:

  ```
  try { normaleBerechnung }
  catch (exceptiontype1 error1) { behandlung1 }
  ...
  catch (exceptiontypen errorn) { behandlungn }
  finally { immerausführen }
  ```

  sowohl `catch` als auch `finally` kann fehlen

- Bedeutung:
  - zunächst Abarbeitung der Anweisung bei `try`
  - falls Abarbeitung fehlerfrei: führe noch finally aus (falls vorhanden)
  - sonst: suche für Fehler passende `catch`-Anweisung, führe diese aus (falls vorhanden), falls keine passende vorhanden: reiche Fehler nach „oben" weiter
- Auslösen von Fehlern:
  - durch vordefinierte Prozeduren (Dateizugriffe etc.)
  - explizit durch `throw`-Anweisung:
    `throw new InvalidException("negative Zahle")`

5.3 Deklarationen

- Nachteil Scheme: Hauptfehler: zur Laufzeit `undefined varable`
- Daher in Java: Einschränkung der zulässigen Programme zugunsten von Fehlerprüfungen (hinzufügen von Redundanzen)
- Prinzip: **alle im Programm verwendeten Namen müssen deklariert werden** (falls nicht vordefiniert)
- Weiterer Fehler im Scheme: unpassende Typen:

  ```
  > (define p (cons 1 2))
  > (+ p 1)
  error: expects <number> as 1st argument, given: (1 . 2)
  ```

- Jedes Objekt hat einen *festen Typ*, der bei der Deklaration angegeben werden muß; Typ $\approx$ Menge der zulässigen Werte
- Variablendeklaration (kann als Anweisung auftreten): `<typ> <name>;` oder `<typ> <name> = <ausdruck>;` Auch mehrere Variablendeklarationen, z.B. `int i, j, k;`
- Konstantendeklaration: wie Varaiablen mit Präfix `final`, nur eine Zuweisung erlaubt: `final double pi = 3.1415926;`
- Anforderungen an Zuweisungen `x = y`: Typ von `y` muß in Typ von `x` konvertierbar sein, d.h. Wert von `y` ist auch zulässiger Wert für `x`!

```
double x = 1.5;
int i = 12;
x = i;       //   zulässig
i = x;       // unzulässig
i = (int) x; //   zulässig
i = true;    // unzulässig
             // (anders als in C!)
```

  wobei `(int)` („typecast") eine Typkonversion erzwingt (meist mit Informationsverlust)

- Beachte: hierdurch sind bestimmt Programmiertechniken von SCHEME sind in JAVA nicht anwendbar, insbesondere Nachrichtenweitergabe; Ersatz in JAVA: direkte Definition von Klassen !

5.3.1 Felder („arrays")

- geordnete Menge indizierter Variablen, d.h. Abbildung $[0, ..., n] \rightarrow Variablen$
- vergleichbar zu Tabellen in SCHEME, aber mit festem Indexbereich und nicht erweiterbar
- Deklaration: `int[] a;` gibt ein Array von Integer-Werten zurückt, Größe wird erst bei Erzeugung festgelegt
- Erzeugung:
  - dynamische Generierung: `a = new int[100]` (d.h. $[0..99] \rightarrow \mathbb{Z}$)
  - Initialisierung: `a = {1, 2, 4, 8, 16}` (d.h. Inzizies 0..4)

- Zugriff auf einzelne Elemente durch Indizierung: `a[i]`
- Beispiel: Aufsummieren aller Feldelemente:

```
int[] a = {1,2,4,8,16};
int s = 0;
for (int i=0, i < 5, i++) s += a[i];
```

- mehrdimensionale Felder: `int[][] a = new int[5][10];`

5.3.2 Funktionen und Prozeduren

- Java objektorientiert ⇒ keine seperaten Prozeduren, sondern immer in Kombination mit Objekten (Klassen) auf, genannt *Methoden*
- Deklaration besteht aus Funktionsnamen und Ergebnistyp, Parameternamen und -typen, Prozedurrumpf
- `<typ> <name> (<typ-1> <param-1>, ...., <typ-n> <param-n>) { Rumpf }`
- Beispiel: `int quadrat (int x) { return x * x; }`
- Aufruf: wie in SCHEME, aber aktuelle Parameter typkonvertierbar in Format der deklarierten Parameter
- Ergebnistyp `void`:
  - Prozedur liefert kein Ergebnis
  - Prozeduraufruf als Anweisung (Seiteneffekte!)
  - `return` ohne Ausdruck
  - Beispiel:

```
void display(int x)
    { System.out.println(x) }
display(99);
```

5.4 Klassen und Objekte

- JAVA-Programm ist Definition von Klassen, die Verhalten von Objekten beschreiben
- Ausführung eines Programms: sende Nachricht „mein“ an bestimmte Klasse

- Objekte
  - haben lokalen Zustand (Attribute)
  - verstehen Nachrichten (vgl. Bankkonto (4.1))
  - Struktur + Verhalten definiert in Klassen (≈ Typ des Objektes
- Merkmal eines Objektes: Zustandvariablen und Nachrichten
- Klassen
  - beschreiben Eigenschaften von Objekten eines Typs
  - enthalten Variablendeklarationen (Attribute), Methodendeklarationen (Implementierung der Nachrichten) und Konstruktordeklaration (Erzeugen von Objekten)
  - Generelle Struktur:

    ```
    class C {
        <variablendeklarationen>
        C(...) {<initialis.>}    // Konstruktor
        <methodendeklarationen>  // Prozeduren/Funktionen
            }
    ```

  - Bedeutung: wie bei der Nachrichtenweitergabe in SCHEME; die Definition von `zuteilen` ist nicht nötig
- Anmerkung zu Bankkonto (siehe Folie):
  - Konstruktoren: Prozeduren (ohne `void`!) mit gleichem Namen wie Klasse. Diese werden bei Objekterzeugung mittels `new` aufgerufen
  - Objekterzeugung: wie in Scheme (Konstuktoraufruf), aber statt Klassennamen muß man `new C(...)` mit Parametern aufrufen
  - Ergebnis: Referenz auf neues Objekt (≈ Prozedur `zuteilen`)
  - Nachricht an Objekt `o` senden:

    ```
    ((o 'm) parameter) ; Scheme
    o.m(parameter)     // Java
    ```

- Spezifikation der Sichtbarkeiten von Merkmalen durch optionales Schlüsselwort
  - `public`: überall sichtbar (≈ in `zuteilen` eingetragen)
  - `private`: nur in der Klasse sichtbar (≈ nicht in `zuteilen` eingetragen)

  - `protected`: nur in der Klasse, Unterklassen (siehe unten) und in gleichen Paketen sichtbar
  - ohne Angabe: nur innerhalb der Klasse und dem Paket, dem die Klasse angehört, sichtbar
  - `main` muß immer public sein!

- Statische Merkmale (`static`): Merkmal gehört zur Klasse und nicht zu Objekten der Klasse
  - Merkmal wird nur einmal repräsentiert (auch ohne Existenz von Objekten!)
  - Zugriff außerhalb der Klasse durch `<Klassenname>.m`
  - Beispiel: Durchnummerieren aller Bankkonten

```
class Konto {
    int kontostand, kontonr;
    private static int lfdnr = 0;
    Konto (int i) {
        kontostand = i;
        kontonr = lfdnr;
        lfdnr++;
        }
    ...
    }
```

  - noch besser: `final int kontonr`: keine nachträgliche Änderung der Kontonummer

- jetzt zu verstehen:

$$\underbrace{\texttt{public}}_{\text{zugreifbar}}\ \overbrace{\texttt{static}}^{\text{zur Klasse}}\ \underbrace{\texttt{void}}_{\text{Ergebnistyp}}\ \texttt{main}\,(\texttt{String[]}\ \underbrace{\texttt{args}}_{\text{Param.}})\ \{\ \ldots\ \}$$

5.5 Vererbung, Überladen

- Wünschenswert bei Software-Entwicklung: Wiederverwendung existierender Programmteile

- bisher:
  - universelle Prozeduren (höherer Ordnung)

- cut/copy/paste: kopiere Programmcode und anpassen (fehleranfällig, Lesbarkeit, Größe...)
- Strukturierte Lösung: Vererbung

- Vererbung:
  - Übernahme von Merkmalen einer Obeklasse A in eine Unterklasse B
  - zusätzlich neue Merkmale bzw. Abänderung existierender Merkmale möglich
  - wichtig: nur Änderungen müssen in B berücksichtigt werden
- Syntax: verb.class B extends A code .
- Sprechweise: B „erbt" automatisch alle Merkmale von A
- logisch: jedes B-Objekt ist ein A-Objekt
- Beispiel: Aufspalten der Konten in Priat- und Geschäftskonten
- Anmerkungen:
  - Konstruktoren: gehören zur Klasse und werden nicht vererbt, Benutzung des Konstruktors der Oberklasse durch `super(...)` !
  - allgemein: Merkmale von Oberklassen sind benutzbar durch Präfix `super.`
  - analog: Benutzung der eigenen Klasse durch Präfix `this.` (z.B. `this.einzahlen(10)`, sinnvoll z.B. in Oberklassen, falls Code generisch in Unterklassen benutzt werden soll)
  - beachte: Redefinitionen von Methoden nur bei gleichen Namen und Parameter/Ergebnistypen, ansonsten wird deis als neue Methode angesehen (siehe unten: Überladen von Methoden)
- Beispiel zum Überladen:

```
class C {
    int id(int x) { return x; }
    double id(double x) { return -x; }
}
...
new C.id(1) -> 1
new C.id(1.0) -> -1.0
```

5.6 Schnittstellen und Pakete

- Abstrakte Klassen:
  - Klassen, von denen man keine Instanzen (Objekte) bilden kann
  - dienen zur Strukturierung fasse Gemeinsamkeiten mehrerer Klassen in einer Oberklasse zusammen
  - Beispiel: Abstrakte Klasse für Benchmark, konkreter Benchmark unbekannt, aber Wiederholung mit Zeitmessung möglich

```
abstract class Benchmark {
    abstract void benchmark();
    publich long repeat (int count) {
        long start = System.currentTimeMillis();
        for (int i = 0; i < count; i++) benchmark();
        return (System.currentTimeMillis() - start);
    }
}
```

  - konkreter Benchmark:

```
class MyBenchmark extends Benchmark {
    void benchmark () { ... }
    ... main {
        repear(10000)
        ...
    }
}
```

- Sonferfall: Schnittstellen (*interfaces*)
  - Abstracte Klasse, wobei alle Methoden (implizit!) `abstract` und alle Attribute (implizit) `final static` (d.h. Konstanten) sind
  - dienen zur Steuerung und Dokumentation: was implementiert eine Klasse?
  - eine Klasse kann viele Schnittstellen implementieren ($\rightsquigarrow$ Mehrfachvererbung)
  - Beispiel: Schnitstelle für Tabellen:

```
interface Lookup { Object lookup (String name); }
interface Insert { void insert (String name, Object value); }
class MyTable implements Lookup, Insert {
    ... // Implementierung der Methoden lookup, insert }
```

wobei `Object` die implizite Oberklasse aller Klassen ist.

  – Schnittstellen können wie (abstrakte) Klassen benutzt werden:

```
... void processValues(String[] names, Lookup table)
    ... table.lookup(names[i]); ...
```

- Pakete
  – Strukturierung von Klassensammlungen zu größeren Einheiten
  – jede Klasse gehört zu einem Paket: Angabe des Paketnamens zu Programmbeginn

```
package mh.lehre.info1
```

  – jede Klasse in einem Paket kann über den Paketnamen angesprochen werden ⇒ Vermeidung von Namenskonflikten

```
java.util.Date now = new java.util.Date();
```

  – Import kompletter Pakete: mache Paketklassen sichtbar

```
import java.util.Date // oder
import java.util.*    // alle Klassen
Date now = new Date();
```

  – nur möglich, falls keine andere Klasse `Date` aus anderen Paketen importiert wurde
  – viele Pakete sind im Java-API (*application(s) programming interface*) vorhanden:
    * `java.applet`
    * `java.awt` (abstract window toolkit): Graphik, Fenster
    * `java.net` Netzwerkprogrammierung

5.7 weitere Aspekte in Java

Ausgelassen wurde...

- Threads: zur nebenläufigen Programmierung mit Synchronisationsmöglichkeiten
- Viele Pakete, z.B. AWT: GUI-Programmierung, Applets: Web-Client-Software, java.net: Netzwerkprogrammierung (Client/Server)
- Remote Method Invocation (RMI): Methodenaufruf von Objekten, die auf anderen Rechnern sind