Modellieren & Programmieren Lernen mit *DrScheme*

Gernot Lorenz

März 2009

Modellieren & Programmieren Lernen
mit *DrScheme*

Begleitmaterial
zum Informatikunterricht

Version 4.1

glorenz@rz-online.de

Vorwort

Liebe Schüler,

das vorliegende Schülerbuch soll Euch beim Thema „Programmieren Lernen" im Informatikunterricht begleiten, zum einen als Nachschlagewerk, zum anderen als Quelle von Übungs- und Aufgabenmaterial.
Aufgrund der ausführlichen Darstellung ist es auch zum Selbststudium geeignet.

Das Konzept basiert auf

- dem didaktischen Konzept HtDP („How to Design Programs", [1]) und [3]
- der Entwicklungsumgebung *DrScheme*
- den EPAs („Einheitliche Prüfungsanforderungen in der Abiturprüfung Informatik")

Höhr-Grenzhausen, im Februar 2005
Gernot Lorenz

Zur Version 4.1:

Neben lokalen Fehlerbeseitigungen und allgemeinen Korrekturen unterscheidet sich die vorliegende Version von der letzten folgendermaßen:

- Kapitel 4, *Modellierung mit Grafik*, ist neu gestaltet. Der bisherige Imperative Ansatz ist durch einen konsequent funktionalen Ansatz ersetzt worden
- Kapitel 5, *Datenmodellierung*, ist umstruktiert und ergänzt worden durch das Unterkapitel *Graphen*
- Kapitel 6, *Funktionale Modellierung, Teil 2* ist ergänzt worden durch das Unterkapiel *Funktionen aus der Kryptographie*
- die Gesamtstruktur inkl. Inhaltsverzeichnis ist an mehreren Stellen gestrafft worden

Herzlichen gilt Frau *Nina Pfeil* für die zahlreichen Verbesserungs- und Erweiterungsvorschläge.

Bendorf, im März 2009
Gernot Lorenz

Inhaltsverzeichnis

1 Einführung

Warum Programmieren lernen?

Die meisten Menschen wissen heutzutage, dass ein Computer – wir werden ihn von jetzt ab auch „Rechner" (lat. computare = berechnen) nennen – erst dann zum Leben erweckt wird, wenn Programme (Software) auf ihm installiert sind, ähnlich wie bei einem CD-Spieler, der ohne CDs nutzlos ist. So ist es kein Wunder, dass sich ein Großteil der IT(= Informationstechnik)-Berufe damit beschäftigt, Programme neu zu entwickeln oder vorhandene zu verändern, zu warten, sei es im technischen oder kaufmännischen Bereich. (–> *http://www.it-berufe.de/berufsbilder/berufsbilder.htm*)
Aber auch diejenigen im IT-Bereich, die nicht direkt mit Softwareentwicklung befasst sind, müssen wenigstens solide Grundkenntnisse im Programmieren haben.
Aber warum sollen sich diejenigen, die nichts oder kaum etwas mit dem IT-Bereich zu tun haben, mit dem Programmieren beschäftigen?
Schauen wir genauer hin, was Programmieren bedeutet: Wir haben ein Problem oder eine Aufgabenstellung, die mit Hilfe eines Rechners gelöst werden soll, dabei geht man – stark vereinfacht – folgendermaßen vor:

1. Problemspezifikation - Was genau soll das Problem leisten, was nicht ?
2. Modellierung - Modell einer Lösung (verbal, in Diagrammen, usw.)
3. Codierung - Übersetzung in eine Programmiersprache
4. Testen - gehe ggfs. zu (2) oder (3) zurück, bis das Programm das Gewünschte leistet.

Dabei fällt uns auf, dass insbesondere (1) und (2) Fähigkeiten sind, die wir alle auch außerhalb des IT-Bereiches sehr gut gebrauchen können, vor allem geht es dabei um

- scharfes Differenzieren,
- exaktes Analysieren,
- logisches Schließen,
- Konkretisieren und Abstrahieren,
- usw.,

oder, wie es jemand prägnanter formuliert hat:

Programmieren trainiert ...

- Problemlösungen planen
- Gedanken diszipliniert organisieren
- auf Details achten
- Selbstkritik üben

... allgemeine Fähigkeiten für die aktive Teilnahme an der Gesellschaft

Der Anfänger

Ein Musiklehrer, der uns nur das Notenschreiben und -lesen beibringt oder uns nur mit der Popmusik beschäftigt oder uns nur Flötespielen lernen lässt, wird uns ein einseitiges und unsystematisches Bild von der Musik vermitteln. Wer in der Schule oder im Verein nur Schwimmen oder nur Volleyball gelernt hat, wird beim Sportstudium oder als Sportlehrer Schwierigkeiten haben, weil er

einseitig ausgebildet ist.
Wenn wir eine Fremdsprache lernen, werden wir nicht sofort mit Cäsars „De bello Gallico" oder Shakespeares „The Merchant of Venice" konfrontiert, sondern behutsam von bescheidenen Anfängen aus Schritt für Schritt und altersgemäß im Sprechen, Hören, Schreiben und Lesen systematisch geübt.
In meiner Jugend auf dem Land haben wir versucht, mit dem schweren Fahrrad aus Stahl unserer Mutter (warum nicht Vaters Rad ?) Radfahren zu lernen und haben uns dabei oft blutige Ellbogen und Knie geholt, bis es dann endlich irgendwie ging: heutzutage hat man dafür Kinderräder.

Was bedeutet das fürs Programmieren lernen ?

Wir brauchen

- anfängergerechte Werkzeuge
  – geeignete Programmiersprache
  – geeignete Entwicklungsumgebung
- ein systematisches Lernkonzept (= didaktisches Konzept)

Was heißt das im Einzelnen ?

Die Programmiersprache

Wir brauchen also eine Programmiersprache, die auf die Bedürfnisse des Anfängers abgestimmt ist,

- die mit wenigen, klaren Sprachelementen auskommt
- mit der man schnell und einfach kodieren kann
- mit der ohne große Umschweife die allgemeinen wichtigen Programmierparadigmen, die man auch in anderen Sprachen findet, erarbeiten kann,

dazu ein Vergleich zwischen der professionellen Industriesprache *Java* und *DrScheme*:

```
// Mittelwert d. Quadratzahlen 0..n-1
public class mittelwert
{
   public static void main(String[] parameter)
   {
      int i;
      int anzahl = Integer.parseInt(parameter[0]);
      double Summe = 0;
      for (i=0;i<anzahl;i++)
         {
         Summe = Summe + i*i;
         }
      System.out.print(Summe/anzahl);
   }
}
```

```
;Mittelwert d. Quadratzahlen 0..n-1
(define (mittelwert n)
  (/ (apply + (build-list n sqr)) n))
```

Entwicklungsumgebung

Professionelle Entwicklungsumgebungen sind sehr komplexe Softwarepakete mit überladenden Menüs und zahllosen Menü-Unterpunkten, die der Anfänger meistens weder versteht noch benötigt. *DrScheme* hingegen

- ist einfach und überschaubar
- erlaubt, auf verschiedenen, skalierbaren Scheme-Dialekten zu arbeiten
- erlaubt die direkte Rückmeldung und Kontrolle aller Prozeduraufrufe

Letzteres ist möglich durch die beiden Besonderheiten

- REPL („Read-Evaluate-Print-Loop"), s. Kapitel I. und Anhang A
- Stepper, durch den man schrittweise das Prinzip der Substitution (Einsetzung) beim Abarbeiten funktionaler Prozeduraufrufe verfolgen kann

Didaktisches Konzept

Das sollt Ihr, liebe Schüler, im Unterricht selbst erfahren und erkennen; das ist natürlicherweise eher ein Thema für Lehrer, weshalb es auch im Lehrerbuch ausgebreitet wird. Wer sich dafür interessiert, sollte sich in dem im Anhang D angegebenen Quellen umsehen. Jedenfalls ist eines der wichtigsten Prinzipien dieses Konzepts

Programmmieren lernen ≠ Eine Programmiersprache lernen

Das heißt, die zum Lernen zur Verfügung stehende Zeit sollte in erster Linie damit verbracht werden,

1. das gestellte Problem mit informatorischen Mitteln zu lösen und dabei allgemein gültige Lösungs- und Modellierungstechniken zu lernen,
2. die von der Wahl der Sprache möglichst unabhängige Sprach- und Programmierparadigmen kennen zu lernen

und nicht, den Großteil der Zeit mit Kodieren, d.h. Erstellen des Quelltextes zu verbringen; denn diejenigen von Euch,

- die später programmieren, werden ohnehin mit mehreren, und i.A. anderen Sprachen zu tun haben
- die später nicht programmieren, werden von allgemeinen grundlegenden und übertragbaren (Er-)Kenntnissen mehr haben als von sehr speziellen

Eine (vermutlich übertriebene) Statistik behauptet bezüglich der Nutzung der zur Verfügung stehenden Zeit:

	Problemlösen Modelllieren, Testen	**Programmierwerkzeug** Sprache u. Umgebung lernen
Ist(*C, Java, Pascal, Delphi)*	10%	90%
Soll	90%	10%

Modellieren vs. Programmieren

Das Ziel von erstellter Software ist es bekanntlich, damit bestimmte Aufgaben zu erledigen oder anders formuliert, ein bestimmtes Problem zu lösen. Dazu führt man zunächst eine Problemanalyse durch, um sich klar zu werden über

- die Ausgangslage
- das angestrebte Ziel

Z. B. wird ein Architekt

- Lage, Art und Beschaffenheit des Baugrundstücks sowie den finanziellen Spielraum des Bauherren in Erfahrung bringen
- die Vorstellungen des Bauherren über sein neues Haus genauestens verinnerlichen und dann ein Modell erstellen (vielleicht im Maßstab 1:50), bevor es ans eigentliche Bauen geht.

Der Bauplan zum Haus ist ebenfalls ein Modell; ebenso wie ein Schaltplan der Elektrizitätsversorgung oder der Wasserversorgung. Sogar die Zeitschiene für Abfolge der einzelnen Phasen des Hausbaus ist ein Modell!
Die Art der Modellierung und die Darstellung des Modells hängen also von der Natur des Problems und der damit verbundenen Fragestellung im Hinblick auf die gewünschte Lösung ab; in der Informatik sind die gebräuchlichsten Modellierungsmethoden hier zusammengestellt:

Modellierungstechnik	typische Fragen	Beispiele	Visualisierung
imperativ	Was ist in welcher Reihenfolge zu tun?	- Kochrezept - Hausbau - ...	Flussdiagramm, PAP, Struktogramm
zustandsorientiert	Welche Zustände kann das System haben? Unter welchen Bedingungen geht das System in einen anderen Zustand?	- Maschinen-/Gerätesteuerung - Theater-Beleuchtungssystem - ...	Automatendiagramme
objektorientiert (OO)	Welche Objekte sind nötig? Welche Eigenschaften haben sie? Was sollen sie können?	- Rangierbahnhof (Züge, Gleise, ..) - Schulverwaltung (Schüler, Lehrer, Räume) - ...	UML-Diagramme
funktional	Welche Funktion(en) sind nötig? Wie sehen die Ein- und Ausgaben aus?	- e-mail-Verschlüsselung (Kryptologie) - Theater-Beleuchtungssystem - Flugbahnberechung von Raumschiffen - ...	selbstmodellierend (!)
datenbankorientiert	Welche Entitäten sind nötig? Welche Struktur haben sie? Welche Beziehungen bestehen?	- Schulverwaltung (Schüler, Lehrer, Räume) - ...	ER-Diagramme

Bei komplexen Situationen sind erwartungsgemäß verschiedene Modellierungstechniken – je nach Art des Teilproblems – nötig; bei kleinen Problemstellungen kann man durchaus mit einem Ansatz auskommen.
Wie man an den typischen Fragestellung in obiger Tabelle sieht, kann eine einseitige Ausrichtung auf eine einzige Modellierungstechnik den Blick auf eine einfachere Lösung des Problems verstellen. Um so wichtiger ist es, dass der Anfänger in der Ausbildung mit verschiedene Techniken konfrontiert wird. Im Konzept des vorliegenden Buches wird mit der *funktionalen Modellierung* begonnen, und zwar weil sie u.a.

- dem Grundprinzip eines „Rechners" am nächsten kommt, indem jeder Eingabe durch Verarbeitung eine eindeutige Ausgabe (EVA) zugeordnet wird (vgl. mit dem Funktionsbegriff aus der Mathematik !)
- mit den geringsten sprachlichen/syntaktischen Mitteln auskommt

- frühzeitig wie *Substitution* (ohne Verwirrung durch Seiteneffekte) und *Rekursion* verinnerlichen lässt
- besonders gut zum allgemeinen Prinzip der *Abstraktion* geeignet ist
- bei Befolgung der Konstruktionsanleitung für Funktionen (s. a.a.O) in Scheme in gewissem Maß selbstmodellierend ist

Danach sind die weiteren Modellierungstechniken vorgesehen (vgl. Inhaltsverzeichnis) wie Zustandsmodellierung/imperatives Programmieren.

2 Grundlagen

2.1 Arbeiten im Interaktionsfenster: „Der Universalrechner"

2.1.1 Formen, Terme, Rechenbäume, Präfix

(Fast) alles, was in die beiden Fenster eingeben wird, ist eine sog. **Form**:

(<Operator > <Operand1> <Operand2> ....)

Zunächst können wir eine Form als einen Term (wie in der Mathematik) auffassen, der durch runde Klammern eingeschlossen ist und in dem zuerst ein Rechenzeichen und anschließend die Zahlen stehen, man spricht von der **Präfix**-Schreibweise (im Gegensatz zur üblichen algebraischen Schreibweise oder Infix-Notaion):

Präfix-Notation	Infix-Notation
(+ 5 16 8)	$5 + 16 + 8$
(* 3 (- a b))	$3(a - b)$

Dabei gilt:

- Formen können geschachtelt werden
- Klammern treten immer paarweise auf

Da Terme im folgenden im Mittelunkt stehen, hier eine kleine Wiederholung:

Verknüpfung	*Term*	*Scheme*	*Bezeichnung*	a *heißt*	b *heißt*
Addition	a+b	(+ a b)	Summe	1. Summand	2. Summand
Subtraktion	a-b	(- a b)	Differenz	Minuend	Subtrahend
Multiplikation	a·b	(* a b)	Produkt	1. Faktor	2. Faktor
Division	a:b	(/ a b)	Quotient	Dividend	Divisor
	a/b		Bruch	Zähler	Nenner

Mathematische Terme können direkt im Interaktionsfenster (=REPL) ausgewertet werden, da die meisten Operatoren in Scheme eingebaut sind:

```
> (+ 11 30)
41
> (- 30 23)
7
> (- 23 30)
-7
> (/ 15 3)
5
> (/ 4 8)
0.5
> (* 12 7)
84
```

Aufgrund der Präfix-Schreibweise können wir im Gegensatz zur üblichen algebraischen Schreibweise (Infix-Notation) mit manchen Operatoren auch mehr als zwei Operanden verknüpfen:

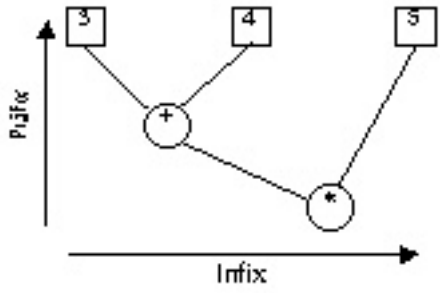
```
> (+ 3 4 5 6 7)  ==> 25
> (* 4 4 4) ==> 64
```

Aber was ist mit `(- 20 5 2)`? oder `(/ 20 5 2)`? Ausprobieren!

Durch Verschachteln können wir auch komplizierte Terme berechnen:

```
> (* (- 5 3) 6) ==> 12
> (- (* 3 5) (* 2 6)) ==> 3
> (/ 1 (/ 1 2)) ==> 2
```

Der Zusammenhang sowie die Umwandlung von Infix in Präfix und umgekehrt wird anhand eines Rechenbaumes besonders deutlich:



Liest man einen Rechenbaum

- von unten nach oben, ergibt sich der Term in Präfix-Notation: `(* (+ 3 4) 5)`
- von links nach rechts, ergibt sich der Term in Infix-Notation: $(3 + 4) \cdot 5$

Übungen

1. Fülle die leeren Felder aus:

Nr	Infix	Termtyp	Präfix	Ergebnis
1	$19,25 \cdot 0,4 - 7,7$			
2	$3 \cdot 4 + 5$			
3	$3 \cdot 4 + 5 + 2 \cdot 7$			
4	$15 \cdot (67 + 13)$			
5	$5 \cdot 6 \cdot 7$			
6	$(1 + 4) \cdot (2 + 5) \cdot 8 \cdot 9$			
7	$1 \cdot 1 + 2 \cdot 2 + 3 \cdot 3$			
8	$((5 + 9) \cdot 3) : 6$			
9	$(14 : 7) \cdot (37 - 33)$			
10	$25 - 3 \cdot (12 : 4)$			
11	$63 : (8 + 13)$			
12	$(23 - 7) : (11 + 5)$			
13			`(* (+ 12 13) 6)`	
14			`(+ 33 34 35)`	
15			`(+ 1 2 3 (- 7 5))`	
16			`(* 1/2 1/3)`	
17			`(- 26 5 6)`	
18			`(- (+ 15 11) (/ 8 4))`	
19			`(/ (+ 13 23) (- 17 8))`	
20			`(/ 12 3 2)`	
21			`(+ 1.5 (+ 1.5 (+ 1.5 0.5)))`	
22			`(* (+ 2 2) (/ (* (+ 3 5) (/ 30 10 )) 2))`	
23			`(+ (- 5/6 3/7) 4/7)`	
24			`(/ (* 1 2 3 4) (+ 1 2 3 4))`	

2. Fertige für 1 jeweils einen Rechenbaum an.

3. Schreibe die Terme
   $234 + 332 + 55 + 7$
   $333957 : 25689$
   $234 \cdot 33 \cdot 123$
   $236864 - 33458$

$(117 - 26) : 13$
$117 - 26 : 13$

a) handschriftlich in Präfix-Notation in eine Zeile
b) berechne sie dann in der REPL

2.1.2 Terme im Editor

In der Präfix-Schreibweise gibt es keine Punkt-vor-Strich-Regel, da alles durch Klammern geregelt wird. Wenn Du den Term $3+5\cdot 6$ in die Präfix-Schreibweise übersetzten willst, musst Du dir vorher über die Reihenfolge im klaren sein, damit dieser korrekt ausgewertet wird. Du musst den Term im Kopf gliedern. Erste Überlegung: Was für ein Term ist es insgesamt? Eine Summe. Dann überlegst du dir die beiden Summanden usw. In dieser Reihenfolge, von außen nach innen, schreibst Du ihn auch auf. Obiger Term ist (wegen Punkt vor Strich) eine Summe, wobei der 1. Summand die 3 ist und der zweite Summand das Produkt mit den Faktoren 5 und 6. In der Präfix-Schreibweise sieht das so aus:

```
(+ 3 (* 5 6))
```

Wenn man verschachtelte Terme statt in der REPL im Editor eingibt, werden sie auch noch übersichtlich gegliedert:

```
(+ 3
    (* 5 6))
```

(Erinnerung: Im Editor muß nach der Eingabe `Ausführen` angeklickt werden !) Diese Möglichkeit ist ideal für Termbestimmungen mit den Begriffen Summe, Differenz, Produkt und Quotient, wie das Beispiel $(38 + 112 \cdot 5) : [(711 - 698) \cdot 3 + 7]$ zeigt:

```
(/              ; Der Term ist ein Quotient.
 (+             ;   Der Dividend ist eine Summe mit
  38            ;     dem 1. Summand 38 und
  (* 112 5))    ;     dem Produkt aus 112 und 5 als 2. Summand.
 (+             ;   Der Divisor ist eine Summe aus
  (*            ;     dem Produkt mit der
   (- 711 698)  ;        Differenz der Zahlen 711 und 698 und
   3)           ;        der Zahl 3
  7))           ;     und der Zahl 7
```

Übungen

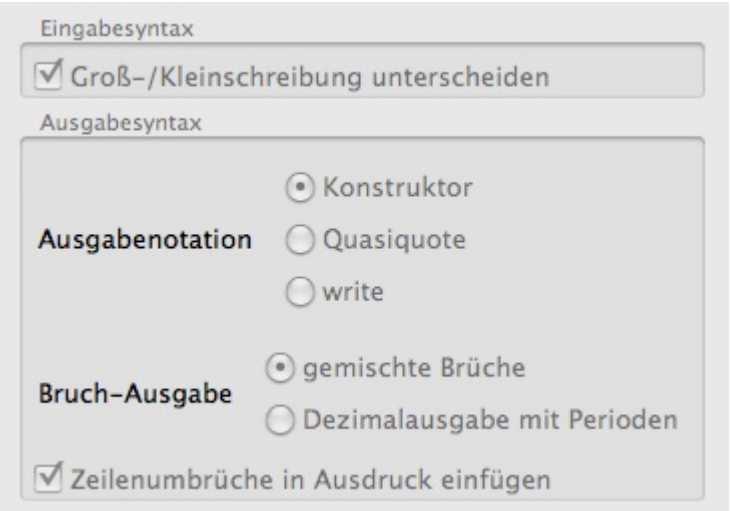
1. Gliedere folgende Terme im Editor wie im obigen Beispiel. Verfolge die Berechnung jedes Terms mit dem *Stepper*:
   a) $(42 + (25 - 3 \cdot 4)) : 11$
   b) $(24 + 32) : 7 + 3 \cdot (17 - 15)$
   c) $34428 : 38 - (1103 - 197)$
2. Formuliere folgende Texte als Terme
   a) in üblicher Schreibweise
   b) in Präfix-Notation (handschriftlich!)
      i. Addiere zum Produkt der Zahlen 7 und 8 den Quotienten der Zahlen 84 und 21.
      ii. Subtrahiere vom Produkt der Zahlen 8 und 9 den Quotienten von 76 und 4
      iii. Multipliere den Quotienten der Zahlen 51 und 17 mit der Differenz dieser Zahlen
      iv. Addiere zur doppelten Differenz der Zahlen 49 und 27 deren halbe Summe.
   c) Berechne die Terme in der REPL
3. Verfahre umgekehrt wie in Übung 3: Formuliere den Term als Text
   a) `(+ (- 88 55) (/ 32 4))`
   b) `(- (/ 56 4) (+ 2 3 4))`

2.1.3 Brüche

Folgende Terme führen zu Dezimalbrüchen, wobei Perioden - sofern sie nicht zu lange - in der REPL entsprechend dargestellt werden:

```
> (/ 17 4)
4.25
> (/ 1 7)
0.142857
> (/ 1 97)
0.010309278350515463917525773...
```

Es können aber auch echte bzw. gemischte Brüche dargestellt werden ! Dazu muß man in DrScheme im Menue `Sprache` die Option `Sprache auswählen...` anklicken dann unter Ausgabe Syntax `gemischte Brüche` anklicken.



Wir erhalten dann für die gleichen Terme folgendes:

```
> (/ 17 4)
4 1/4
> (/ 1 7)
1/7
> (/ 1 97)
1/97
```

Hinweis: Die Eingabe von Brüchen ist in jedem Modus möglich!
Neben den Grundrechenarten sind auch andere Operatoren verfügbar, die wir aus der Mathematik als Funktionen kennen, z.B.

```
(abs <Zahl>) |<Zahl>|  (Betrag)
(sqr <Zahl>) <Zahl>2   (Quadrieren)
(sqrt <Zahl>) <Zahl>  (Quadratwurzel)
(random <Zahl>) 0 < Zufallszahl < <Zahl> -1
```

Die in Scheme eingebauten mathematischen Funktion werden auch PRIM-OPS, d.h. „primitve operations" genannt - auch wenn sie in der Tat oft nicht „primitiv" sind. (–> *Help Desk, Manuals*)
Obige Beispiel-Funktionen sind alle vom Typ *Zahl –> Zahl*, d.h. einer Zahl als Argument wird wieder eine Zahl als Funktionswert zugeordnet.
Tatsächlich aber unterscheidet Scheme zwischen

- ganzen Zahlen (`integer`)
- rationalen Zahlen (`rational`)
- rellen Zahlen (`real`)

- komplexen Zahlen (`complex`)

so z.B verträgt

- `random` nur ganze Zahlen, genau genommen positive Zahlen als Argument und liefert auch nur ganze Zahlen
- `sqrt` nur Zahlen ≥ 0 als Argument

Zusätzlich wird bei den reellen und komplexen Zahlen zwischen exakten und „unexakten" (i.a. irrationale) Zahlen unterschieden:

```
> (sqrt 2)
#i1.4142135623730951
> (log 3/4)
#i-0.2876820724517809
```

Näheres Kap. II, Funktionale Modellierung: „Zahlen, Funktionen & Zufall"

Übungen

1. Gliedere folgende Terme im Editor wie im obigen Beispiel. Verfolge die Berechnung jedes Terms mit dem *Stepper*:
   a) $\frac{4}{9} \cdot (1 - \frac{7}{2} : 8)$
   b) $\frac{1}{3} - \frac{3}{5}$
   c) $\frac{7-\frac{3}{4}}{2+6,3}$
2. Formuliere folgende Texte als Terme
   a) in üblicher Schreibweise
   b) in Präfix-Notation (handschriftlich!)
      i. Dividiere die Summe von $\frac{1}{3}$ und $1\frac{1}{3}$ durch das Produkt dieser Zahlen
      ii. Addiere die Hälfte von $1\frac{2}{3}$ und drei Viertel von $-\frac{2}{9}$
   c) Berechne die Terme in der REPL
3. Schreibe in mathematischer Notation:

   ```
   (sqrt (+ (sqr 3) (sqr 4)))
   (abs (+ 5 (abs -6)))
   ```

4. Schreibe in Präfix: $\sqrt{(72 - 2,22)}$
5. Schreibe in mathematischer Notation und berechne

   ```
   > (/ 1 (+ 1 1))
   > (/ 1 (+ 1 (/ 1 (+ 1 1))))
   > (/ 1 (+ 1 (/ 1 (+ 1 (/ 1 (+ 1 1))))))
   ```

   usw. Was fällt bei den Ergebnissen auf ?

2.2 Die Dinge bekommen einen Namen

Wenn wir die Werten, mit denen wir rechnen, nicht jedesmal neu eintippen wollen, können wir ihnen auch einen Namen geben, z. B. geben wir der Zahl 8 den Namen `a`.
Diesen Vorgang nennen wir eine Definition einer „Variablen" (oder „Platzhalter") und geschieht durch die Sonderform `(define ...)`. Dies geschieht – wie der Name schon sagt – im Definitionsfenster:

Nach dem Eintippen klicken wir auf Ausführen, damit DrScheme diesen Vorgang auf Korrektheit prüft und „lernt".
Jetzt können wir im Interaktionsfenster durch Eingabe von `a` sofort prüfen, ob DrScheme den Namen `a` kennt.
Bei dem Wert, der einem Namen zugewiesen wird, muß es sich keineswegs um eine Zahl handeln, vielmehr kann es sich um auch einen Term – oder allgemeiner einen Ausdruck (engl. *expression*) handeln:

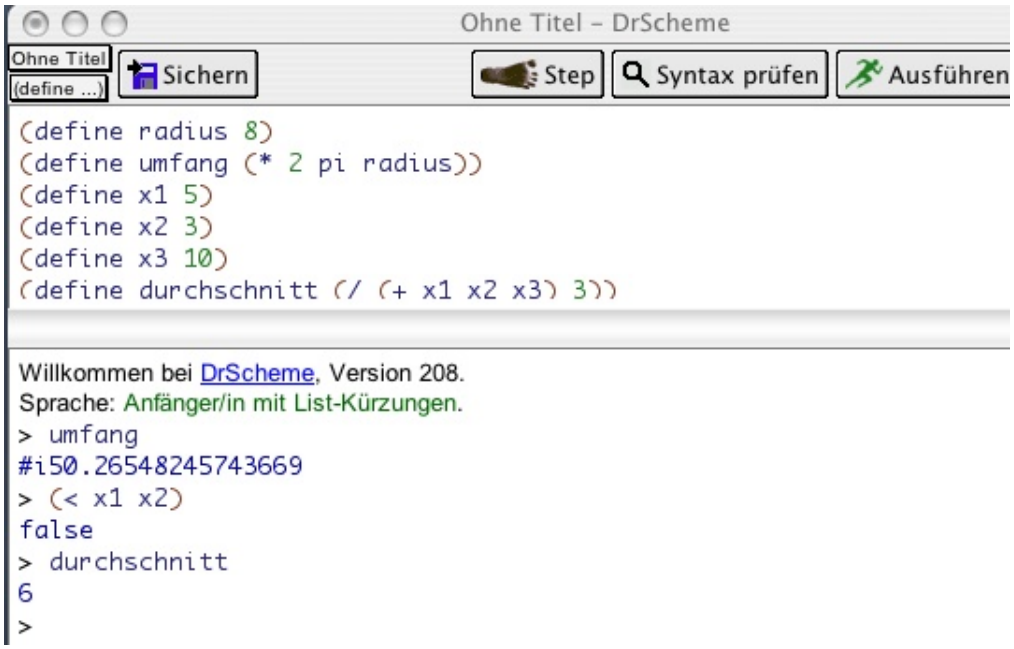
Allgemein gilt: Die Sonderform

```
(define <Name> <Ausdruck> )
```

weist einem Namen einen Ausdruck bzw. einen Wert zu. Genau genommen, wie man bei funktionalen Sprachen sagt, der Bezeichner wird an einen Ausdruck bzw. Wert gebunden.
Bei einem Namen/Bezeichner muß es sich – anders als in der Mathematik – nicht um einen einzelnen Buchstaben handeln, vielmehr sind ganze „Worte" erlaubt, die nicht nur aus Buchstaben, sondern auch aus Ziffern oder anderen Tastatur-Zeichen bestehen können.

Im Einzelnen gilt:

- „Name" und „Bezeichner" sind im folgenden gleichwertige Begriffe
- ein Name aus beliebig vielen Tastatur-Zeichen bestehen, ausgenommen sind nur `" ' , ; ` ( ) [ ] { } \ #|`, sowie Leertaste/Leerzeichen
- der Begriff „Variable" ist im funktionalen Sprachmodell eigentlich nicht korrekt, da ja sie ja an einen Wert gebunden wird; dennoch wollen wir ihn zulassen



Zur Sprachregelung

Zur Form `(define umfang (* 2 pi radius))`
können wir sagen: – „dem Term `(* 2 pi radius)` wird der Name `umfang` gegeben" oder – „die Variable `umfang` wird an den Wert des Ausdrucks `(* 2 pi radius)` gebunden"

Übungen

1. Ein Rechteck hat die Länge 34 und die Breite 13. Definiere Länge und Breite sowie Diagonale und berechne die Diagonale.

2.3 Die Logik-Maschine

2.3.1 Wahrheitswerte und BOOLEsche Terme

Neben den Grundrechenarten kann man mit *DrScheme* auch Vergleiche durchführen, die als Ergebnis **Wahrheitswerte**, nämlich **wahr** oder **falsch** (engl.: *true* und *false*) liefern:

```
> (= 4 4)
true
> (= -3 3)
false
> (= 3 3 3)
true
> (= 3 4 3)
false
> (< 100 101)
true
> (<= 3 5 7)
true
```

Gleichungen und Ungleichungen mit konkreten Zahlen gehören zu den sog. *Aussagen*, die entweder wahr oder falsch sind; man nennt die zugehörigen Terme (wie die Beispiele) auch logische Terme oder **BOOLEsche Terme**. Von den Aussagen zu unterscheiden sind *Aussageformen*, wie z.B. (=

x 5), die nur der Form nach Aussagen sind, aber erst durch Einsetzen von konkreten Werten zur Aussage werden, wie z.B. `(<= a x b)`, was bei Eingabe zur Fehlermeldung *a: name is not defined, not an argument, and not a primitive name* führt; deshalb muss vorher

```
(define a -1)
(define b 2)
(define x 2)
```

erfolgen, wonach wir Ergebnisse erhalten:
`(<= a x b)` `;--> true` liegt zahl im abgeschlossenen Intervall $]a;b[$?
`(< a x b)` `;--> false` oder im offenen Intervall $[a;b]$?

Beispiel: Teilbarkeitstests
In der Grundschule haben wir die **Ganzzahldivision** (oder Division mit Rest) kennengelernt: 13 geteilt durch 5 = 2 Rest 3 (und **nicht** 2,5) Dies geht auch in Scheme:

```
(quotient 13 5)       ;--> 2  (Ganzahlquotient)
(remainder 13 5)    ;--> 3   (Rest)
```

d.h. wir können jede ganze Zahl x bei der Division durch y darstellen als x = q·y + r, wobei der q der (Ganzzahl-)Quotient und r der Rest ist. In Präfix-Notation mit Scheme müßte dann der Term

```
(= 13 (+ (* (quotient 13 5) 5) (remainder 13 5)))
```

den Wert `true` liefern. (Ausprobieren !)

Mit der `remainder`-Funktion können wir also testen, ob die Zahl x durch y teilbar ist, in dem wir feststellen, ob bei der Ganzzahldivision ein Rest $\neq 0$ bleibt:

```
(remainder 21 15)    ;--> 6
(remainder 21 7)    ;--> 0, also ist 21 durch 7 teilbar
(remainder 123342348833315 6)    ;--> 5
```

Komfortabler ist es, wenn wir gleich das Ergebnis wahr oder falsch erhalten, also:
`(= 0 (remainder 123342348833315 6))` `;--> false`

Dies geht noch einfacher mit der BOOLEschen (oder logischen) Funktion `zero?`[1]:
`(zero? (remainder 1233423488333156))` `;--> false`

Übungen

1. Teste, ob 490458039881102987 durch 6, 7, 11, 13 oder 17 teilbar ist. Wie sieht das Ganze mit dem Taschenrechner aus ?

2.3.2 Logische Verknüpfungen

Wie wir wissen, ist „x ist teilbar durch 6" gleichbedeutend „x ist teilbar durch 2" und „x ist teilbar durch 3".
Wie drückt man dieses „und" zwischen den beiden Aussagen aus ?
Wir probieren es mal mit „and" und erhalten:

```
(and (zero? (remainder 18 2)) (zero? (remainder 18 3)))    ;--> true
```

was wir erwartet haben. Zu besseren Übersicht geben wir den Dingen Namen:

```
(define x 18)
(define A (zero? (remainder x 2)))  ;Teilbarkeit von x durch 2
(define B (zero? (remainder x 3)))  ;Teilbarkeit von x durch 3
```

[1] die Funktion **zero?** liefert kein Zahl, sondern einen Wahrheitswert als Ergebnis

dann geben wir

```
A
B
(and A B)
```

ein und erhalten – wie nicht anders erwartet – für alle drei Aufrufe, egal ob im Editor oder in der REPL jeweils den Wert `true`.
Machen wir das gleiche noch mit 19, 20 und 21, so erhalten wir insgesamt:

Zahl	A (durch 2 teilbar)	B (durch 3 teilbar)	`(and A B)`
18	`true`	`true`	`true`
19	`false`	`false`	`false`
20	`true`	`false`	`false`
21	`false`	`true`	`false`

Tabelle 2.1: Wahrheitswerttabelle

Dies ist die uns bereits bekannte Konjunktion bzw. das „logische UND", dessen Wahrheitswertetabelle folgendermaßen angeordnet ist, wobei f = falsch (`false`) und w = wahr (`true`) bedeutet:

A	B	A und B
f	f	f
f	w	f
w	f	f
w	w	w

Die **Konjunktion**, d.h. das logische **UND** ist eine Verknüpfung zweier Aussagen, die nur dann den Wert „wahr" liefert, wenn beide Aussagen wahr sind (sonst „falsch").
In DrScheme wird sie durch die Form
`(and` <Aussage1> <Aussage2>`)` realisiert.

Entsprechend gibt es in Scheme auch die Disjunktion bzw. das „logische ODER":

A	B	A oder B
f	f	f
f	w	w
w	f	w
w	w	w

Die **Disjunktion**, d.h. das logische **ODER** ist eine Verknüpfung zweier Aussagen, die nur dann den Wert „wahr" liefert, wenn mindestens eine beiden Aussagen wahr sind (sonst „falsch").
In DrScheme wird sie durch die Form
`(or` <Aussage1> <Aussage2>`)` realisiert.

Hinweise

- `(and ...)` und `(or ...)` können mehr als 2 Argumente haben !
- Hinweis: Für das ausschließende ODER (XOR) gibt es – wie in praktisch allen Programmiersprachen – keinen speziellen Befehl (s. u. Übung 7)

Und schließlich gibt es auch die **Negation** bzw. das „logische **NICHT**" bzw. die **Verneinung**:

A	nicht A
f	w
w	f

, die in DrScheme durch die Form `(not`<Ausssage>`)` formuliert wird.
Da die logischen Verknüpfungen nur auf die BOOLEschen Werte true und false anwendbar sind, ergeben `(not 3)` oder `(and 3 4)` Fehlermeldungen. Deshalb ist die Testfunktion (Prädikator) `(boolean? <objekt>)` oft hilfreich.

Übungen

1. Definiere eine Zahl x und teste damit die log. Terme (=Aussageformen)

   ```
   (> x 3)
   (or (< x 3) (< -1 x))
   (= (* x x) x)
   (and (< 4 x) (> x 3))
   ```

   für x = 2, x = 4, x = $\frac{7}{2}$.

2. Prüfe nach (durch Nachdenken und/oder durch Aufstellen einer Wahrheitswerttabellen für (2), ob die Terme/Aussageformen

   ```
   (1)   (>= x y)
   (2)   (or (> x y)   (= x y))
   ```

   gleichwertig sind. Finde einen Term, in dem die Operatoren `or` und $<$ vorkommen, mit dem man (1) auch ersetzen kann.

3. Angenommen, man definiert eine Zahl y und teste damit den Term
   `(or (and (<= 1 y) (<= y 5)) (and (<= 9 y) (<= y 12))).`
   Wo auf einem Zahlenstrahl kann diese Zahl liegen, wenn der Term den Wert `true` liefert?

4. Um eine Zahl z auf Teilbarkeit durch die Zahl p zu prüfen, kann man statt der Untersuchung des Restes bei Ganzzahldivision auf 0 (also mit `(remainder ...))` auch den Term
   `(= (/ z p) (quotient z p)))` benutzen.
   a) Prüfe das an Beispielen
   b) Begründe die Gleichwertigkeit der beiden Ausdrücke

5. Definiere eine Zahl x und teste, ob sie
   a) innerhalb des Intervalls $[3; 4]$ liegt
   b) außerhalb des Intervalls $[3; 4]$ liegt
      1. mit Hilfe von `(not ...)`
      2. mit Hilfe von `(or ...)`

6. Der Manager der Firma Ywindow&Co benötigt ein Programm, das feststellt, ob ein Mausklick innerhalb eines Rechteckes erfolgt oder nicht. Der Mausklick wird durch zwei Zahlen x und y repräsentiert. Vorgegeben sind Zahlen `nord, sued, west` und `ost`, die die Ränder des Rechtecks festlegen:
   `(define nord 100), (define sued 350), (define west 30), (define ost 220)`
   Gib einen logischen Term an, der die (vorher definierten) Koordinaten x y des Mausklicks benutzt und den Wert `false` liefert, falls der Klick außerhalb des Rechtecks erfolgt.

7. Neben der Disjunktion (auch „einschließendes" oder „inklusives" ODER genannt) gibt es noch das ausschließende ODER oder exklusive ODER (XOR), was dem „entweder - oder " entspricht.
   a) Stelle eine Wahrheitswerttabelle für das exklusive ODER auf.
   b) Angenommen, man hat zwei Zahlen x und y sowie das Intervall [5; 7]. Entwickle einen Term, der den Wert wahr nur dann liefert, wenn entweder x oder y im Intervall liegt.

3 Funktionale Modellierung, Teil 1

3.1 Mit Funktionen geht's besser

Problem

Nehmen wir an, wir haben die Funktion

$$f(x) = -(x-3)2 + 0,5x + 2 \tag{3.1}$$

und möchten ihren Wert für verschiedene x berechnen, etwa um das Maximum zu finden; wir vermuten es bei dem Argument x = 3 und definieren im Editor

```
(define x 3) ; Argument-Definition
(define f (+  (- (sqr (- x 3))) (* 0.5 x) 2))      ; Term-Definition
```

Wir erhalten als Ergebnis 3,5. Um weitere Werte zu erhalten, müssen wir jetzt jedesmal einen anderen Wert für x definieren und die REPL neu starten, d.h. wir arbeiten andauernd im Editor, was sehr umständlich ist. Bequemer wäre es, wir könnten die Funktionswerte in der REPL genauso berechnen wie bei den bisherigen bekannten Funktionen, wie also z.B. `(sqr 3)` *oder* `(abs -8)`.

3.1.1 Die Funktionsdefinition

Dazu benötigen wir die Möglichkeit – ähnlich wie in der Mathematik – Funktionsname, Argument und Funktionsterm in einem Zug aneinander zu binden. Die geschieht in Scheme folgendermaßen:

```
(define (f x) (+ (- (sqr (- x 3))) (* 0.5 x) 2))      ; Funktions-Definition
```

oder allgemein:

Scheme	*Mathematik*
`(define` (<Name><Parameter1><Parameter2>...) (<Rumpf>)	<Name>(<Argument(e)>)=<Term>

Abb. 3.1: Struktur einer Funktionsdefinition

Trotz der prinzipiellen Ähnlichkeit beider Definition gibt es formale Unterschiede; im Gegensatz zur Mathematik

- wird die unabhängige Variable, die in der Mathematik auch Argument genannt wird, in der Informatik **Parameter** genannt
- haben Funktionen in der Informatik häufig mehrere Parameter (= Argumente)
- wird eine Funktion in der Informatik häufig auch als **Prozedur** (engl. procedure) genannt
- wird der Funktionsterm in der Informatik Funktions-**Rumpf** genannt
- werden Funktionen der besseren Übersicht wegen nicht in einer Zeile definiert
- wird als Funktionsname ein sinntragender Bezeichner gewählt (vgl. Variablen)

Beispiele:

```
(define (kreis-umfang radius)
        (* 2 pi radius))

(define (volumen-zylinder radius hoehe)
        (* pi radius radius hoehe))
```

3.1.2 Der Funktionsaufruf

Die Berechnung des Funktionswertes aus den eingegebenen Parameter gestaltet sich genauso wie bei den bereits in Scheme eingebauten Funktionen; man nennt diesen Vorgang Funktions**aufruf** oder Prozedur**aufruf**: entweder im Editor oder in der REPL: *Beispiele:*

```
(kreis-umfang 3) --> 18.84955592153876
(volumen-zylinder 3 4) --> 113.09733552923255
```

Eine typische Fehlerquelle ist dabei, daß ein Parameter beim Aufruf vergessen wird, so würde etwa

```
(volumen-zylinder 3)
```

zu einer Fehlermeldung führen.
Da Funktionen Werte liefern, die an anderer Stelle wieder eingesetzt werden, ist auch - ähnlich bei den Termen – auch die Schachtelung oder Einsetzung (mathematisch: Verkettung) von Funktionen möglich:

```
(define (grundflaeche radius)
                (* pi radius radius))
(define (volumen-zylinder radius hoehe)
                (* (grundflaeche radius) hoehe))
```

allerdings muß dabei Vorsicht walten; z.B. würde

```
(define (kleiner-als-3? x)
                (< x 3))
(define (abstand y z)
                (- (kleiner-als-3? y) z))
```

beim Aufruf `(abstand 2 5)` zu einer Fehlermeldung führen, da der Funktionswert von `(kleiner-als-3? 2)` vom Typ BOOLEsch ist, bei `(abstand y 5)` aber eine Zahl für `y` gebraucht wird!

Allgemein gilt:

Beim Aufruf einer Funktion müssen die Parameter („aktuelle" Parameter) in

- Anzahl
- Art (Datentyp)
- Reihenfolge

mit denen der Definition („formale" Parameter) übereinstimmen !

Daraus folgt, daß man sich bei einer Funktionsdefinition sorgfältig überlegen muß, von welchem Typ die Funktionswerte sind; ein Vergleich mit einer verarbeitenden Maschine macht das klar:

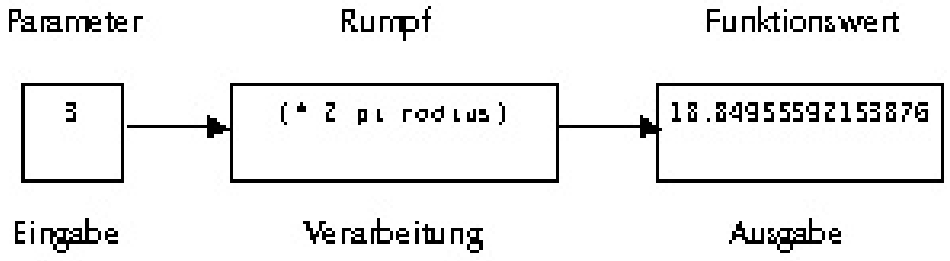


Abb. 3.2: EVA-Prinzip

Deshalb ist bei der Planung einer Funktionsdefinition systematisch vorzugehen, wie im nächsten Abschnitt gezeigt wird.

3.1.3 Systematische Konstruktion einer Funktion

Problem:

Du hast 200€ am Geburtstag geschenkt bekommen, und möchtest das Geld bei einer der Banken an Deinem Wohnort als Sparguthaben für ein Jahr anlegen. Du möchtest wissen, wieviel Zinsen Du bei welchem Zinssatz erhältst.

Wir gehen schrittweise vor:

1. **Problem- und Datenanalyse**
   Wir wissen, dass wir außer dem Anfangskapital - was schon bekannt ist - noch den Zinssatz, eine Dezimalzahl, brauchen; als Ausgabe erhalten wir die Jahreszinsen, je nach Zinssatz ebenfalls eine Dezimalzahl (die Einheit € lassen wir beim Rechnen einfach weg)

2. **Beschreibung der Aufgabe der Funktion**
   Aus Kapital und Zinssatz soll nach der bekannten Formel aus Kl. 7 der Jahreszins berechnet werden. (Wir nehmen an, dass das Geld vom 1. Januar bis zum 31. Dezember des Jahres auf dem Konto bleibt, dazu s. später)

3. **„Vertrag" für die Funktion** Nach Eingabe des (noch) unbekannten Zinssatzes sollen die Jahreszinsen berechnet werden:

```
;jahres-zins: zahl1 --> zahl2   „Vertrag"
;Eingaben: zahl1 = Zinssatz, als Deziamalzahl
;Ausgabe: zahl2 = Jahreszinsen
```

4. **Funktionsgerüst**

```
(define (jahres-zins zinssatz)
                <Rumpf> )
```

5. **Beispiele:**

```
(jahres-zins 0.02)              ==> 4
(jahres-zins 0.015)             ==> 3
(jahres-zins 0.0125)            ==> 2.5
```

6. **Funktionschablone:**

```
(define (jahres-zins zinssatz)
                ...zinssatz... )
```

7. Vervollständigung des **Funktionsrumpfes:**

```
(define (jahres-zins zinssatz)
                (* 200 zinssatz))
```

8. **Tests:** Die Tests bestätigen die Ergebnisse von 5:

```
(jahres-zins 0.02)              ==> 4
(jahres-zins 0.015)             ==> 3
(jahres-zins 0.0125)            ==> 2.5
```

Insgesamt steht jetzt Folgendes im Editor:

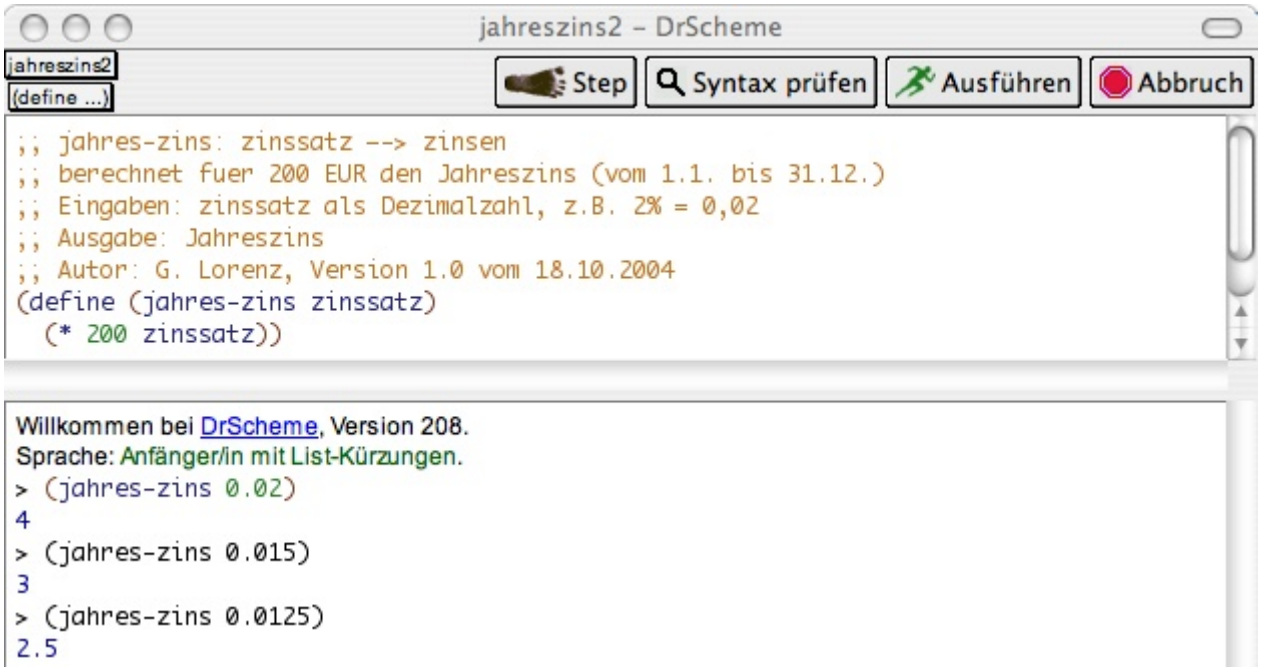


Abb. 3.3: Programm-Test

Konstruktion einer Funktion/Prozedur in Kürze:

1. Schreibe Vertrag, Zweck und Funktionskopf
2. Mache Beispiele
3. Schreibe den Rumpf der Funktion
4. Teste die Funktion!

Hinweise

- Die soeben vorgestellten 8 Schritte zur Problemlösung nennen wir **Konstruktionsanleitung** für Programme; sie ist von jetzt an bei **jeder** Problemlösung schriftlich anzufertigen.
- Für die Bezeichner von Funktionsnamen und Parametern sind **sinntragende** Namen zu geben. (Das gilt natürlich auch für den Dateinamen, unter dem es gesichert wird)
- Der **Quelltext** einer Funktion/Prozedur sollte als vorangestellter Kommentar enthalten:
  - den Vertrag
  - die Ein- und Ausgaben (Parameter) mit ihren Bedeutungen
  - Sinn und Zweck

  Außerdem sollte er laufend kommentiert sein (alles, was in einer Zeile hinter einem Semikolon steht, ist Kommentar)
- Der gesamte Quelltext einer Scheme-Datei (.ss oder .scm), der i.a. mehrere Funktionsdefinitionen enthält), sollte am Anfang einen Kommentar enthalten, der angibt
  - worum es geht
  - Autor
  - Version
  - Erstellungsdatum

Übungen

1. Die obige Zinsfunktion sollte so erweitert werden, dass Zinssatz und Startkapital eingegeben werden können
2. Bei einer Pizza soll aus dem vorgegebenen Durchmesser die Fläche des notwendigen Belags bestimmt werden; dabei ist zu beachten, dass es einen Rand von 1 cm gibt, der aus Kruste besteht und nicht belegt wird:

   ```
   Vertrag:
   ; pizza-belag: durchmesser --> fläche
   ```

3. In den USA wird zur Temperaturmessung die Fahrenheit-Skala benutzt, benannt nach Gabriel Daniel Fahrenheit aus Danzig, der 1714 das erste brauchbare Quecksilberthermometer entwickelte.
   Dabei benutzte er die nach ihm benannte Skala von 212 Grad, wobei 32 Grad als der Schmelzpunkt des Eises und 212 Grad als der Siedepunkt des Wassers definiert wurde.
   Anders Celsius aus Upsala schlug 1742 die nach ihm benannte Skala von 0 bis 100 Grad vor.
   Entwickle komplett Funktionen, die
   a) Fahrenheit in Grad Celsius
   b) Grad Celsius in Fahrenheit umrechnen
4. Eine Funktion quadrat soll 2 Zahlen quadrieren.
   a) Welche Definitionen sind falsch und warum?

   ```
   (define (quadrat) (* n n))
   (define (quadrat n))
   (define (quadrat n) (* n n)
   (define (quadrat n) (+ n n))
   ```

   b) Welche Funktionsaufrufe sind falsch und warum?

   ```
   (quadrat n)
   (quadrat 3 4)
   (quadrat)
   ```

5. Einsetzung von Funktionen in andere (Substitution):

   ```
   (define (T1 x)
      (+ x (+ x 1) (+ x 2)))
   (define (T2 x)
      (* x x))
   (define (T x)
      (* (T1 x) (T2 x)))
   ```

   a) Was bewirkt die Funktion T?
   b) Welchen Wert liefern

   ```
   (T1 4)
   (T1 (T1 4))
   (T2 (T1 4))
   ```

6. Gegeben sind:

   ```
   (define (einnahme zuschauer preis)
      (* zuschauer preis))
   (define (kosten zuschauer)
      (+ (* zuschauer 1.60) (* 0.04 (- zuschauer 120))))
   (define (zusch preis)
      (+ (* (/ 15 0.10) (- 5.00 preis)) 120))
   (define (profit preis)
      (- (einnahme (zusch preis) preis) (kosten (zusch preis))))
   ```

   a) In welchem Verhältnis stehen die vier Funktionen zueinander?
   b) Gib bei allen Funktionen an: Funktionsnamen, Parameter, Kopf, Rumpf
   c) Welchen Zweck könnte die Funktion profit haben?
   d) Man könnte die vier Funktionen durch eine einzige ersetzen:

   ```
   (define (profit2 preis)
   (- (* (+ (* (/ 15 0.10) (- 5.00 preis)) 120) preis)
   (+ (* (+ (* (/ 15 0.10) (- 5.00 preis)) 120) 1.6)
   (* 0.04 (- (+ (* (/ 15 0.10) (- 5.00 preis)) 120) 120)))))
   ```

   Warum ist das nicht so sinnvoll?

3.2 Zahlen, Funktionen & Zufall

3.2.1 Zahlen

Im folgenden bedeuten die Platzhalter:	
<any>	irgendein Objekt
<num>	Zahl
<real>	relle Zahl
<rat>	rationale Zahl
<int>	ganze Zahl

Eine Einteilung der Zahlen in aufsteigende Teilmengen gibt es auch in Scheme:

Menge:	ganze Zahlen $\mathbb{N}$	$\subset$	rationale Zahlen $\mathbb{Q}$	$\subset$	reelle Zahlen $\mathbb{R}$	$\subset$	komlexe Zahlen $\mathbb{C}$
Beispiele:	-4; 0; 123		-4; 2,35; 4,333...		-4; 2,35; $\sqrt{2}$; π		$\sqrt{-2}$
Prädikator	`(integer? <any>)`		`(rational? <any>)`		`(real? <any>)`		`(complex? <any>)`

Tabelle 3.1: Zahlenübersicht

Zusätzlich gibt es noch weitere, von einander unabhängige Unterteilungen durch Prädikatoren

positiv/negativ: `(positive? <real>)`, `(negative? <real>)`
gerade/ungerade: `(even? <int>)`, `(odd? <int>)`
exakt/unexakt: `(exact? <num>)`, `(inexact? <num>)` bei Zahlen, die DrScheme dezimal nicht exakt darstellen kann, wie z.B. $\sqrt{2}$ = `(sqrt 2) --> #i1.4142135623730951`

und schließlich

```
(number? <any>)
(zero? <num>)
```

3.2.2 Eingebaute Funktionen, in Auswahl

Diese sind hier nur mit ihrem Bezeichner, also ohne Parameterliste und ohne Typ des Funktionswertes angegeben (genaueres siehe `HelpDesk`):
Für <num> sind definiert

`+, -, *, /`	auch mit mehr als 2 Operanden
`=`	oder `(equal? <num> <num>)`

Nur für komplexe Zahlen sinnvoll sind
`angle, magnitude, imag-part, real-part, conjugate`

Für <real> sind definiert

`<, <=, >, >=`	auch mit mehr als 2 Operanden
`abs`	Betrag: |...|
`round`	rundet auf ganze Zahlen
`max, min`	auch mit mehr als 2 Operanden
`floor, ceiling`	untere/ober Gaussklammer
`make-polar`	Umkehrung zu `angle` und `magnitude`
`sign`	Vorzeichenfunktion (1, 0 -1)
`exp`	e-Funktion
`log`	ln (nat. Log.)
`expt`	allg. Exponentialfunktion
`sin, cos, tan, asin, acos, atan`	trigon. Funktionen inkl. Umkehrung
`sinh, cosh`	lhyperbolische Funktionen

Für <rat> sind definiert

`numerator, denominator`	Zähler, Nenner eines gekürzten Bruches

Für <int> sind definiert

`quotient`	Ganzzahldivision
`remainder, modulo`	Rest bei Ganzzahldivision
`gcd, lcm`	ggT, kgV
`random`	Zufallszahlen 0 ... n-1

Umwandlungsfunktionen

```
exact->inexact
inexact->exact
integer->char
```

Sonstige Funktionen

`gcd`	ggT
`lcm`	kgV

Konstanten

`e`	EULERsche Zahl e `(#i2.718281828459045)`
`pi`	π `(#i3.141592653589793)`

3.2.3 Zufallszahlen

Der eingebaute Zufallsgenerator erzeugt (Pseudo-)Zufallszahlen mit der Funktion

```
(random <int>) --> 0..<int>-1,
```

wobei <int> im Bereich [1, 2147483647] liegen muß.
Also ergibt `(+ 1 (random 49))` eine Zufallszahl x mit $1 \leq x \leq 49$.

Übungen

1. Simuliere einen Würfel
2. Entwickle eine Funktion *;zufall: a b -> zahl* mit a $\leq$ zahl $\leq$ b
3. In einer Bäckerei kostet ein Berliner 60Cent.
   Zusätzlich gibt es das Angebot „3 Berliner für 1,50€".
   a) Was kosten 2, 3, ..., 8 Berliner ?
   b) Entwickle eine Funktion *;berliner-preis: anzahl –> preis*!
      Tip: Benutze die Funktionen `quotient` und `remainder`
   c) Was kosten n Berliner durchschnittlich ?
   d) Entwickle zu b) die Umkehrfunktion.

3.3 Weitere einfache Datentypen

3.3.1 Zeichenkette - `string`

Die Funktion

```
;teilbar?: zahl teiler--> BOOLEsch
(define (teilbar? zahl teiler)
        (zero? (remainder zahl teiler)))
```

liefert die BOOLEschen Werte `true` oder `false`.
Vielleicht wäre jetzt jemand als Antwort lieber „wahr" und „falsch" oder „ja" und „nein". Dazu brauchen einen neuen Datentyp, nämlich die sog. Zeichenkette oder string: Das sind alle Zeichenfolgen, die mit " eingeschlossen sind:
`"Hans", "x1", "32", "\#007", "1. Ableitung", "Adam und Eva"`, usw.
Im Gegensatz zu den Namen bzw. Bezeichner für Variable sind alle Zeichen erlaubt. Ob ein Objekt eine Zeichenkette ist, kann mit dem Prädikator
`(string? <Objekt>)`
getestet werden:

```
(string? "Hallo ?") ==> true
(string? "333") ==> true
(string? 333) ==> false (!)
(number? "333") ==> false (!)
(number? 333) ==> true
(string? "!§$%&/()=?") ==> true
```

Jetzt könnnen wir `teilbar?` auch wie gewünscht definieren, allerdings ist der Funktionswert vom Typ Zeichenkette:

```
;; teilbar?: zahl teiler–> zeichenkette

(define (teilbar? zahl teiler)
   (cond
     ((zero? (remainder zahl teiler)) "ja")
     (else
      "nein")))
```

Nützliche Funktionen über Zeichenketten sind die Umwandlungsfunktionen

```
(string->number <zeichenkette>) --> <zahl>
(number->string <zahl>) --> <zeichenkette>
```

sowie

```
(string-length <<zeichenkette>) --> <zahl>
(string-append <<zeichenkette1> <zeichenkette2>) --> <zeichenkette>
(string-ref <<zeichenkette1> <ganzzahl>) --> <zeichen>
```

Angenommen, beim Würfelspiel „Mäxchen" werden die beiden Würfel durch

```
(define z1 (+ 1 (random 5)))
(define z2 (+ 1 (random 5)))
```

simuliert. Haben wir einen „Pasch" geworfen ? Dann hilft

```
;; pasch: zahl zahl–> zeichenkette

(define (pasch zzahl1 zzahl2)
   (if (= zzahl1 zzahl2)
       (string-append (number->string zzahl1) "er-Pasch")
       "kein Pasch"))
```

z.B. ergibt sich

```
(pasch 3 3) ==> "3er-Pasch"
(pasch 2 3) ==> "kein Pasch"
```

3.3.2 Zeichen - `char`

Um einzelne Zeichen verarbeiten zu können, gibt es auch in Scheme einen eigen Datentyp, nämlich `char` (von „character"[1]), womit die mit einer Tastatur editierbaren Zeichen wie die Buchstaben (alphabetische Zeichen), Ziffern (numerische Zeichen) und Sonderzeichen gemeint sind. Sie werden in der Form

`#\a` für a
`#\B` für B
`#\4` für 4
`#\?` für ?

dargestellt. Die Kennzeichnung von Zeichen durch die Voranstellung von `#\` (#+alt-shift /) ist notwendig, damit u.a.

- alphabetische Zeichen von Bezeichnern (Namen) unterschieden werden können, `#\a` ≠ `a`
- numerische Zeichen (Zifffern) von Zahlen unterschieden werden können, `#\4` ≠ `4`

Auch dafür gibt es einen Prädikator `(char? ...)`, z.B.

```
(char? #\4) ==> true
(char? 4) ==>  false
(char? #\a) ==>  true
```

Bekanntlich sind die alphabetischen, die numerischen und die Sonderzeichen nach dem ASCII[2] codiert, und zwar einheitlich für alle Betriebssysteme und Tastaturen von 0 bis 127, z.B. entspricht

```
A       -->     65
B       -->     66
0       -->     48
```

Für diese Kodierung bzw. Dekodierung sind die Funktionen

```
(char->integer <Zeichen>) --> <Zahl>
(integer->char <Zahl>) --> <Zeichen>
```

vorgesehen:

```
(char->integer #\a) ==> 97
(integer->char 97) ==> #\a
```

Zu den Nummern 0 bis 31 gehören keine sichtbaren Zeichen, vielmehr verbergen sich dahinter Steuerbefehle aus der Steinzeit der Computer, die hier keine Rolle spielen.
Einen Sonderfall stellt die Nr. 32 dar, womit das Leerzeichen (space) gemeint ist:
`(integer->char 32) ==> #\space`
Aufgrund der ASCII-Numerierung gibt es auch eine Ordnungsrelation innerhalb der Zeichen durch z.B.

```
(char>? #\a #\b)   ==> false
(char<? #\a #\b)   ==> true
(char<? #\a #\b #\c)  ==> true
(char<=? #\Z #\W)   ==> false
(char=? #\+ #\+)   ==> true
```

Wichtig ist der Unterschied zwischen Zeichenkette und Zeichen: Eine Zeichenkette der Länge 1 ist kein Zeichen, z.B. `"A"` ≠ `#\A` !

Hilfreich für Umwandlung zwischen Zeichenkette und Zeichen sind die Funktionen

```
(string->list "abc")   ==> (list #\a #\b #\c)
(list->string (list #\a #\b #\c))  ==> "abc"
```

Es gibt noch weitere Funktionen für Zeichen, siehe dazu Anhang D, „Sprachumfang"

[1] *altgr.* „eingekratztes (Zeichen)"
[2] = American Standard Code for Information Interchange, siehe Internet

3.3.3 Symbol - `symbol`

Symbole sind atomare Daten, und zwar Zeichenfolgen, auf deren einzelne Zeichen man im Gegensatz zu den Zeichenketten (strings) man nicht zugreifen kann. Sie werden durch ' (ASCII-Nr. 39) eingeleitet und dürfen kein Leerzeichen enthalten: `'x1 '1234 'a '007` usw.
Auch hier gibt es einen Prädikator `(symbol? ...)`,

```
> (symbol? 'hallo)
true
> (symbol? hallo)
reference to an identifier before its definition: a
> (symbol? 1)
false
> (symbol? '1)
true
> (symbol? "1")
false
```

3.4 Von Fall zu Fall - `cond` und `if`

Problem

In der gymnasialen Oberstufe haben wir zusätzlich zu den Verbalnoten eine Verfeinerung der Notenstufen durch Punkte, also

Punkte	*Note*
13, 14, 15	*sehr gut*
10, 11, 12	*gut*
...	*...*
1, 2, 3	*mangelhaft*
0	*ungenügend*

Wenn z.B. ein Zeugnisstellungsprogramm nach Eingabe der Punktzahl zusätzlich die Verbalnote auf das Zeugnisformular drucken soll, wird eine Funktion benötigt, die hier 6 verschiedene Ausgaben liefern soll, je nachdem, welche Punktzahl vorliegt; d.h. es müssen Fälle unterschieden werden, in Abhängigkeit von Bedingungen, also etwa

```
Bedingung
13, 14, 15 Punkte: Ausgabe „sehr gut"
10, 11, 12 Punkte: Ausgabe „gut"
...
1, 2, 3 Punkte: Ausgabe „mangelhaft"
0 Punkte: Ausgabe „ungenügend"
```

Dies wird in Scheme ermöglicht durch **Fallunterscheidung** (auch *Verzweigung* oder bedingte Anweisung oder Mehrfachauswahl genannt):

```
(cond
     (<Bedingung1> <Ausdruck1>
     ( <Bedingung2> <Ausdruck2>)
     ....
(else Ausdruck>))
```

Hinweise

- die letzte Klausel - der `else`-Fall - ist optional, d.h. kann auch weggelassen werden
- die Bedingungen werden der Reihe nach (von oben nach unten) überprüft

  - bis eine der Bedingungen zutrifft und der zugehörige Ausdruck zurückgegeben wird. Daher ist die Reihenfolge wichtig !
  - wenn keine zutrifft, wird eine Fehlermeldung ausgegeben, oder - falls vorhanden - der Ausdruck der else-Klausel zurückgegeben.

  Daher empfiehlt es sich, i.a. die else-Klausel einzubauen, sofern keine unsinnigen Rückgaben möglich sind.
- jede Klausel muß in ein Klammerpaar (Bedingung und Ausdruck) eingeschlossen werden

Gibt man `(define punktzahl 6)` vor, erhält man für unser Eingangsbeispiel

```
(cond
   ((>= punktzahl 13) "sehr gut")
   ((>= punktzahl 10) "gut")
   ((>= punktzahl 7) "befriedigend")
   ((>= punktzahl 4) "ausreichend")
   ((>= punktzahl 1) "mangelhaft")
   (else "ungenügend"))
```

und die Ausgabe "ausreichend", wobei man für die letzte Zeile auch `((= punktzahl 0) "ungenügend"))` nehmen könnte.
Diese Version hat den Nachteil, dass bei allen Zahlen außer $0, 1, ..., 15$ eine Fehlermeldung erscheint, da keine wahre Bedingung eintritt; bei der else-Version andererseits enstünden in diesem Fall zwar keine Fehlermeldungen, dafür aber falsche Zuordnungen wie z.B:

```
0.4 --> "ungenügend"
-2 --> "ungenügend"
3.7 --> "mangelhaft"
17--> "sehr gut"
```

Will man ganz sicher gehen, dass

- überhaupt nur Zahlen
- nur die ganzen Zahlen $0, 1, ..., 15$ verwertet werten und sonst ein Hinweis gemeldet wird,

wählt man die elegante Variante (hier als Funktion):

```
;msspunkte->verbalnote: zahl --> string
(define (msspunkte->verbalnote punktzahl)
        (cond
                ((not (number? punktzahl)) "Zahl eingeben!")
                ((not (integer? punktzahl)) "Ganze Zahl 0 <= x <= 15 eingeben!")
                ((<= 13 punktzahl 15) "sehr gut")
                ((<= 10 punktzahl 12) "gut")
                ((<= 7 punktzahl 9) "befriedigend")
                ((<= 4 punktzahl 6) "ausreichend")
                ((<= 1 punktzahl 3) "mangelhaft")
                ((= 0 punktzahl) "ungenügend")
                (else "Ganze Zahl 0 <= x <= 15 eingeben!")))
```

Zusätzlich zur `(cond ...)`-Form gibt es in Scheme auch die *zweiseitige Entscheidung* oder Verzweigung:

(if <Bedingung>	
<*Ausdruck1*>	wenn wahr
<*Ausdruck2*>**)**	wenn falsch

Sie stellt einen Spezialfall von `(cond ...)` dar, und zwar

```
(cond
     (<Bedingung> <Ausdruck1>)
     (else <Ausdruck2>))
```

Ein Beispiel:

```
(if (zero? (remainder punktzahl 2))
     "durch 2 teilbar"
     "nicht durch 2 teilbar")
```

Umgekehrt kann man `(cond ...)` mit `(if ...)` nur durch Schachtelung darstellen, wie man an (*) sieht:

```
(if (> temperatur 35)
    "heiß"
    (if (> temperatur 25)
        "warm"
        (if (> temperatur 15)
            "mittel"
            "kalt"))))
```

Übungen

1. Darf man bei folgenden Beispielen die `(cond ...)`-Klauseln vertauschen?

   a)
   ```
   (cond
        ((> temperatur 35) "heiss")
        ((> temperatur 25) "warm")
        ((> temperatur 15) "mittel")
   (else "kalt"))
   ```

   ```
   (cond
           ((zero? (remainder zahl 2)) "durch 2 teilbar")
           ((zero? (remainder zahl 3)) "durch 3 teilbar")
   (else "werde durch 2 noch durch 3 teilbar"))
   ```

2) Analysiere folgende Funktion:

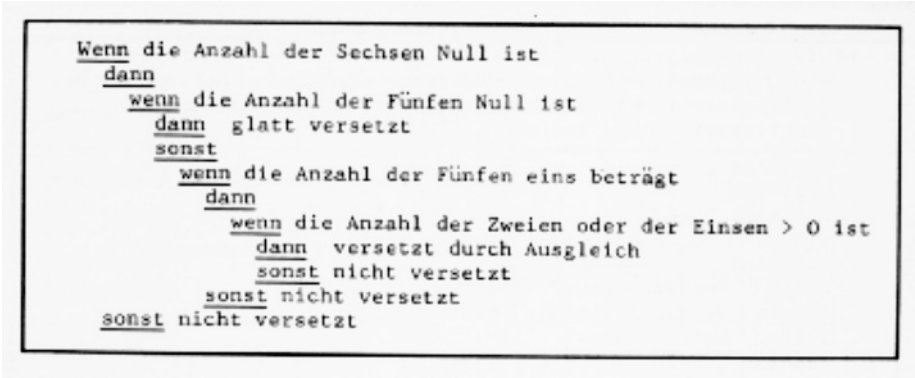
   ```
   (define (maut gesamtgewicht)
   (cond
       ((not (number? gesamtgewicht)) "Eingabe muss Zahl sein !")
        ((<= gesamtgewicht 0) "Zahl muss größer 0 sein !")
        ((<= gesamtgewicht 1000) 20)
        ((<= gesamtgewicht 2000) 30)
        ((<= gesamtgewicht 5000) 50)
        ((<= gesamtgewicht 10000) 100)
        (else 250)))
   ```

   a) Welchen Sinn haben die beiden ersten Klauseln ?
   b) Begründe, weshalb man hier die Reihenfolge der Klauseln nicht ändern darf
   c) Nenne ein konkretes Beispiel, bei dem man die Klauseln ändern darf

   Zur Wiederholung: Der BOOLEsche Ausdruck `(number? gesamtgewicht)` testet, ob der Wert von `gesamtgewicht` eine Zahl ist.
   Allgemein gilt: Formen der Art (<typname>? <Objekt>) nennt man *Prädikatstester* oder *Prädikator*

3. Entwickle eine Funktion „Rechner" mit den 3 Parametern *Rechenzeichen, Operand1* und *Operand2*, die je nach Rechenzeichen die Summe, die Differenz, das Produkt oder den Quotient ausgibt. Tip: Benutze hier `equal?` statt =.
4. Entwickle eine Funktion *zensur*, die MSS-Punkte in einer Verbalnote („sehr gut", „gut", usw.) ausgibt.

5. Angenommen, an einer Schule gilt folgende einfache Versetzungsordnung:



   Entwickle dafür eine Funktion „Versetzung" mit

   - `(cond...)`
   - `(if...)`

6. Bei Skelettfunden schließt man aus der Länge Knochen auf die Körpergröße; und zwar gilt (als statistischer Mittelwert) in cm:
   Körpergröße = 69,089 + 2,238 · Oberschenkellänge bei Männern
   Körpergröße = 61,412 + 2,317 · Oberschenkellänge bei Frauen
   Ab dem 30. Lebensjahr nimmt die Körpergröße um 0,06 cm pro Jahr ab. Entwickle ein Programm *k-laenge*, das aus Oberschenkellänge, Alter und Geschlecht die vermutete Körpergröße berechnet.

7. Gegeben ist

```
(* (cond ((> a b) a)
 ((< a b) b)
 (else -1))
    (+ a 1))
```

   a) Handelt es sich um einen korrekten Scheme-Ausdruck ? Begründung !

   b) Definiere a und b jeweils so, dass alle Klauseln einmal zutreffen. Ergebnisse ?

8. a) Ersetze

```
(cond
(test1 (aktion1))
       (test2 (aktion2))
       (test3 (aktion3))
       (else (aktion4)))
```

   durch geschachtelte `(if...)`-Formen

   b) Könnte man `else` durch die BOOLEsche Konstante true ersetzen ?

9. Bekanntlich wird im Straßenverkehr die Geschwindigkeitsübertretung durch Kraftfahrzeuge je nach Ausmaß als Ordungswidrigkeit, Vergehen oder Straftat geahndet. Neben anderen Maßnahmen wird in der Regel auch eine Geldbuße verhängt.

   a) Informiere Dich, evtl. im Internet, über den Katalog der Maßnahmen und stelle eine Funktion auf, die - je nach Höhe der Übertretung - den Betrag der Geldbuße ausgibt.

   b) Kannst Du zusätzlich auch berücksichtigen, ob der Tatbestand innerorts oder außerorts stattgefunden hat ?

10. Aus der Parkzeit (in Min.) soll die Gebühr berechnet werden:

Parkdauer	Parkgebühr
bis zu 2 Stunden	1,50 €
jede weitere angefangene Std.	1 €
Höchstgebühr pro Tag	10 €

    Das Programm

```
(define (pg1 min)
   (cond
     ((<= min 120) 1.5)
     (else
       (+ 1.50 (+ (quotient (- min 121) 60) 1)))))
```

leistet das Gewünschte nur teilweise.

a) Erläutere den Quellcode

b) Entwickle eine zweite Funktionen *pg2*, die auch die Höchstgebühr berücksichtigt. Dabei soll *pg1* benutzt werden.

11. Finde für die Gleichungen

$$\begin{aligned} x^2 + px + q &= 0 \\ ax^2 + bx + c &= 0 \end{aligned}$$

jeweils Funktionen, die die *Anzahl* der Lösungen liefern.

3.5 Zusammengesetzte Typen: Verbund - `struct`

Bisher haben wir lediglich einfache Datentypen kennengelernt: Zahlen, BOOLEsche Werte, Zeichenketten. Es gibt aber auch Situationen, wo mehrere Werte vom Sinn her zusammengehören, wie etwa bei einer Datenbank für Vereinsmitglieder: z.B. Nachname, Vorname, Geburtsdatum, Mitgliedsnummer und Beitrag:

mitglied

Nachname
Vorname
Mitgliedsnummer
Beitrag

oder einem Punkt im Raum:

punkt

x
y
z

In Scheme spricht man von einem **strukturierten Typ** genannt **`struct`**, oder auch von einem *Verbund* oder *record*.
Da Anzahl und Art der einzelnen Komponenten (hier auch Felder genannt) je nach Bedarf schwanken, müssen solche Typen selbst definiert werden:
`(`**`define-struct`** `mitglied (Nachname Vorname Mitgliedsnummer Beitrag))`
bzw.
`(`**`define-struct`** `punkt (x y z))`
Damit hat man aber lediglich eine Typendefinition, aber noch kein Exemplar dieses Typs. Letzteres geschieht durch den Konstruktor

```
(make-mitglied "Lorenz" "Gernot" 4711 12)
```

bzw.

```
(make-punkt 20 30 40)
```

wobei die Werte vom gleichen Typ wie die zugehörigen Felder sien müssen; allerdings hat das so erzeugte Objekt keinen Namen.
Will man ein Objekt dieses Typs an einen Namen binden, ist eine Definition nötig:

`(define` <Name> `(make-`<Typname> <Wert1> <Wert2> ... <Wertn>`))`

Beispiel

Bei der Geometrie der Ebene hat ein Punkt zwei Koordinaten:

```
(define-struct punkt (x y))  ; Typ-Definition
(make-punkt 7 3)               ; Konstruktor-Aufruf
(define p1 (make-punkt 3 4))    ; Variablen-Definition
```

Mit der Definition eines zusammengesetzten Datentyps werden neben dem Konstruktor-Aufruf automatisch zwei weitere Funktionen deklariert:

```
(punkt? p1)                          ; Prädikatstest-Aufruf: (<Typ>? <Objekt>)
(punkt-x p1)                         ; Selektor-Aufruf: (<Typ>-<Feld> <Ausdruck>)
```

Strukturierte Datentypen sind im Gegensatz zu bisherigen einfachen Datentypen nicht vordefiniert[3]),
sie müssen mit
(define-struct <Typname> (<Feld1> <Feld2> ... <Feldn>)
selbst definiert werden.
Dabei werden automatisch die drei Funktionen
(make-<Typname> <Wert1> <Wert2> ... <Wertn>) ;*Konstruktor*
(<Typname>-<Feld> <Objekt>) ;*Selektor*
(<Typname>? <Objekt>) ;*Prädikator*
erzeugt

Beispiel, Forts.:

```
(define (abstand0 p)                  ; Abstandsberechnung vom Ursprung (0|0)
   (sqrt (+ (* (punkt-x p) (punkt-x p)) (*(punkt-y p) (punkt-y p)))))
```

Übungen

1. Definiere eine zusammengesetzten Datentyp `zeitdauer` mit den Felder Stunden, Minuten und Sekunden und darauf aufbauend eine Funktion *gesamtdauer* mit einem Parameter vom Typ `zeitdauer`, der daraus eine Gesamtzeit in Sekunden liefert.
2. Definiere eine Funktion, die - ähnlich wie Beispiel 2 - den Abstand zweier Punkte P1 und P2 voneinander berechnet.
3. Im Internet findet man unter www.immobilien.de unter der Rubrik „Haus zu verkaufen" z.B. folgende 3 Anzeigen:

Einfamilienhaus in ruhiger Lage	**Wohnen wie im Urlaub**	**Für Naturliebhaber**
56459 Berzhahn	57647 Enspel	56459 Mähren
4 Zimmer	4 Zimmer	5 Zimmer
Wohnfläche 125 qm	Wohnfläche 80 qm	Wohnfläche 110 qm
Bj: 1968	Bj: 1994	Bj: 1994
179000 EUR	160000 EUR	160000 EUR

Dies Angaben (zumindest Teile davon) sollen mit EDV ausgewertet werden. Dazu möchte man mit Hilfe von Scheme-Funktionen
   - den Preis pro qm Wohnfläche eines Hauses berechnen können
   - die durchschnittliche Wohnfläche von 3 Häusern berechnen können
   - den Titel der Annonce des Hauses (von 3 Häusern) mit dem niedrigsten qm-Preis

ermitteln können
   a) Entwirf ein passendes Datenmodell
   b) Gib die Scheme-Def. für „haus" an (Beachte, dass nur die für (1) bis (3) nötigen Angaben zu berücksichtigen sind!).
   c) Definiere die o.g. Funktionen (1) bis (3)
   d) Wende (2) und (3) auf die o.a. Anzeigen an.

Schachtelung von strukturierten Typen

Wenn z.B. eine Personendatei mit dem Ansatz

Schüler

Nachname
Vorname
Adresse
Klasse

mit

Adresse

Straße u. Nr.
PLZ
Ort

modellieren wollen, fällt uns auf, dass Adresse selbst wieder als Verbund dargestellt werden kann:

```
(define-struct adresse (str PLZ Ort))
(define meine-adresse (make-adresse "Neubergsweg 49" 56170 "Bendorf"))
(define-struct person (VN NN adresse))
(define ego (make-person "Gernot" "Lorenz" meine-adresse))
```

z.B. ergibt dann folgender Aufruf:
`(adresse-PLZ (person-adresse ego)) --> 56170`
Wir sehen, dass wir hier das Prinzip der Substituion konsequent durchhalten können.

3.5.1 Der vordefinierte Typ `posn`

Es müssen nicht alle strukturierten Datentypen selbst definiert werden, es gibt eine Ausnahme, nämlich den Typ **`posn`** (=Position), der die Koordinaten eines Punktes im 2-dimensionalen angibt. Müßte man diesen Typ selbst definieren, könnte man es leicht mit
`(define-struct posn (x y))`
erledigen. Entsprechend sind

```
(make-posn ...)
(posn-x ...), (posn-y ...)
(posn? ...)
```

automatisch vordefiniert.

Beispiel

Wir modellieren einen Typ dreieck und definieren einige Funktionen:

```
((define-struct dreieck (A B C))
```

| die Eckpunkte A, B und C sind vom Typ `posn` und erstellen ein konkretes Objekt von diesem Typ

```
(define mein-dreieck
(make-dreieck (make-posn 60 240) (make-posn 270 210) (make-posn 180 60)))
```

Angenommen, wir interessieren uns für den Schwerpunkt S des Dreiecks; dazu benötigen wir die Seitenhalbierenden, die einen Eckpunkt mit dem Mittelpunkt der gegenüberliegenden Seie verbinden:

```
;mp-a: dreieck -> posn
;liefert den Mittelpunkt der Seite a:

(define (mp-a ein-dreieck)
   (make-posn
(/ (+ (posn-x (dreieck-B ein-dreieck)) (posn-x (dreieck-C ein-dreieck))) 2)
(/ (+ (posn-y (dreieck-B ein-dreieck)) (posn-y (dreieck-C ein-dreieck))) 2)))
```

Entsprechend kann man `(mp-b ein-dreieck)` und `(mp-c ein-dreieck)` für die Seiten b und c definieren. Man erhält z.B.
`(mp-a mein-dreieck) --> (make-posn 225 135)`
Da wir wissen, dass sich die Seitenhalbierenden (=Schwerelinien) im Verhältnis 2:1 jeweils teilen, gilt für die Schwerpunktkoordinaten $s_x = \frac{2}{3} \cdot (A - x + mp - a - x)$ bzw. $s_y = \frac{2}{3} \cdot (A - y + mp - a - y)$:

```
;schwerpunkt: dreieck -> posn
;liefert den Schwerpunkt des Dreiecks ABC:

(define (schwerpunkt ein-dreieck)
  (make-posn
   (+ (posn-x (dreieck-A ein-dreieck))
      (* 2 (/ (- (posn-x (mp-a ein-dreieck))
                 (posn-x (dreieck-A ein-dreieck))) 3)))
   (+ (posn-y (dreieck-A ein-dreieck))
      (* 2 (/ (- (posn-y (mp-a ein-dreieck))
                 (posn-y (dreieck-A ein-dreieck))) 3)))))
```

Aufruf:
`(schwerpunkt mein-dreieck) --> (make-posn 260 240)` Interessant wäre jetzt, wenn wir das Dreieck samt Seitenhalbierenden und Schwerpunkt auch zeichenen könnnten. Das geschieht im **Exkurs 1**: Modellieren mit Grafik

3.6 Listen - ein induktiver Datentyp

Während der strukturierte Datentyp aus mehreren Einzeldaten fester Anzahl besteht, kann es auch nützlich sein, wenn mehrere Einzeldaten veränderlicher Anzahl zu einer Einheit zusammengefaßt werdent können, etwa
<meine Freunde>: „Andreas", „Paula", „Sabine", „Robert"
Dafür gibt es in Scheme den Datentyp **Liste** (`list`).

3.6.1 Konstruktion einer Liste - `cons`

Wie konstruiert man eine solche Liste in Scheme ?
Zuerst hatten wir gar keinen Freund:
Die Namensliste war leer: `empty`
Dann wurde Robert der erste Freund:
Die Namensliste besteht aus: `(cons "Robert" empty)`
Dann kam Sabine hinzu:
Die Namensliste besteht aus: `(cons "Sabine" (cons "Robert" empty))`
Dann kam Paula hinzu:
Die Namensliste besteht aus: `(cons "Paula" (cons "Sabine" (cons "Robert" empty)))`
Schließlich kam Andreas hinzu:
Die Namensliste besteht aus: `(cons "Andreas" (cons "Paula" (cons "Sabine" (cons "Robert" empty))))`
Wir könnten aber auch aufgrund ihrer Entstehungsgeschichte die Namensliste so auffassen:
Meine Freunde sind

- Andreas, mein neuester Freund,
- und meine bisherigen Freunde

Genau diese Idee hat man sich in Scheme beim Datentyp Liste zu Nutzen gemacht: Listen kann man als *Paare* auffassen, die aus einem ersten Element und dem Rest der Liste besteht, der selbst wiederum eine Liste ist,
also: <Liste> = (<erstes Element> <Rest (=Liste)>),
was in Scheme durch die Paar-Bilder-Form `cons` („Konstruktor") geschieht:
`(cons` <Element> <Rest>) ;Konstruktor
Berücksichtigen wir noch, dass eine Liste auch leer sein kann, kommen wir zu einer allgemeinen Definition:

Eine **Liste**

- ist leer
  oder
- besteht aus
  - einem ersten Element
  - einem Rest, der selbst wieder eine Liste ist

oder in Scheme

Eine **Liste** ist

- **`empty`**
  oder
- **`(cons`** <Element> <Rest der Liste>**`)`**

Diese Idee der schrittweisen Konstruktion, in dem man zu einer Liste durch `(cons ...)` ein weiteres Element, das dann das erste ist, hinzunimmt, nennt man auch „Induktion" (*inducere*, lat.: (schrittweise) heranführen); entsprechend ist die Liste ein **induktiver Datentyp**.

Neben der Konstruktion einer List mit `(cons...`
- `(cons` <Element-1> `(cons` <Element-2> `(cons`<Element-3> ... `empty)..)`

gibt es auch die kürzere Form
- **`(list`** <Element-1> <Element-2> ... <Element-n>)

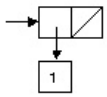
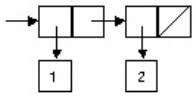
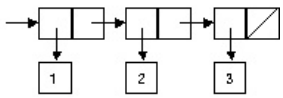
Beispiele

```
(cons "Robert" empty) ==> ("Robert")
(cons "Sabine" (cons "Robert" empty)) ==> ("Sabine" "Robert")
(list "Sabine" "Robert") ==> ("Sabine" "Robert")
(cons 1 (cons 2 (cons 3 (cons 4 empty)))) ==> (1 2 3 4)
(define eineListe (cons "Robert" empty))
eineListe ==> ("Robert")
(cons "Sabine" eineListe) ==> ("Sabine" "Robert")
(define andereListe (cons "Sabine" eineListe))
andereListe ==> ("Sabine" "Robert")
(define weitereListe (cons "Sabine" (cons "Robert" empty)))
```

wobei `andereListe` und `weitereListe` identisch sind

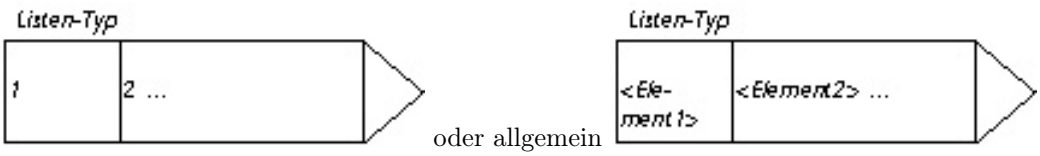
```
(list "Adam") ==> ("Adam")
(list "Adam" "Eva") ==> ("Adam" "Eva")
(list 1 2 3 4) ==> (1 2 3 4)
```

Am Beispiel der Liste `(list 1 2 3)` können wir den Aufbau mit der Zeiger-Notation verdeutlichen:

Liste	Zeigernotation
`empty`	
`(cons 1 empty)`	
`(cons 1 (cons 2 empty)`	
`(cons 1 (cons 2 (cons 3 empty)`	

Während die Zeiger-Notation der Vorstellung einer Liste und ihrer Teillisten als Paare entgegenkommt, entspricht folgende Pfeildarstellung eher der Form

`(list 1 2 3)`



3.6.2 Eingebaute Funktionen auf Listen

Um mit Listen arbeiten zu können, muß man auch auf Teile der Liste zugreifen können:

1. das erste Element einer Liste mit
   ; first: liste –> element
   `(first <Liste>)`
   Beispiele:

   ```
   (first eineListe) ==> "Robert"
   (first andereListe) ==> "Sabine"
   (first (list 1 2 3 4)) ==> 1
   ```

2. den Rest einer Liste mit
   ;rest: liste –> liste
   `(rest <Liste>)`
   Beispiele:

   ```
   (rest eineListe) ==> empty
   (rest andereListe) ==> ("Robert")
   (rest (list 1 2 3 4)) ==> (list 2 3 4))
   (first (rest andereListe) ==> "Robert"
   (rest (rest (list 1 2 3 4)) ==> 2
   ```

Zum Arbeiten mit Listen gibt es praktischer Weise zahlreiche vordefinierte Funktionen (in Auswahl):

;list?: liste –> BOOLEsch
(list? <Objekt>) — *Prädikator*: Testet, ob ein Objekt eine Liste ist

;empty?: liste –> BOOLEsch
(empty? <Liste>) — Testet, ob eine Liste leer ist

;cons?: liste –> BOOLEsch
(cons? <Liste>) — Testet, ob eine Liste nichtleer ist

;reverse: liste –> liste
(reverse <Liste>) — Kehrt die Reihenfolge einer Liste um
`(reverse (list "Adam" "Eva"))` `==> ("Eva" "Adam")`

;length: liste –> zahl
(length <Liste>) — Gibt die Länge einer Liste an
`(length (list "Adam" "Eva"))` `==> 2`

;append: liste liste–> liste
(append <Liste1> <Liste2> ...) — Macht aus mehreren Listen eine Liste
`(append eineListe andereListe)` `==> ("Eva" "Adam" "Eva")`

Beipiele

```
(define bunteListe (list (list 3 4) 5)) ==> ((3 4) 5)
(define paare (list (cons "Adam" (cons "Eva" empty)) (cons "Paul" (cons "Paula" emtpy))))
==> (("Adam" "Eva") ("Paul" Paula"))
(rest (first paare)) ==> ("Eva")
(first (rest paare)) ==> ("Paul" Paula")
(rest (rest paare)) ==> empty
(first (first (rest paare))) ==> "Paul"
(rest (first (rest paare))) ==> ("Paula")
```

Betrachten wir folgende Listen

1. `(define adresse (list "Neubergsweg 49" 56170 "Bendorf"))`
2. `(define selbst (list "Gernot Lorenz" adresse)) ==> (list "Gernot Lorenz" (list "Neubergsweg 49" 56170 "Bendorf"))`

stellen wir fest, dass

- bei 1. alle Elemente von einfachem Datentyp sind, diese sowohl vom Typ Zeichenkette als Zahl sind,

- bei 2. ein Element selbst eine Liste ist.

Daher legen wir folgende Einteilung in Listentypen fest:

Elemente der Liste	Bezeichung	Sonderfälle alle Elemente sind...
Alle Element sind von einfachem Datentyp (Zahl, BOOLEsch, Zeichen, Zeichenkette, Symbol)	sequentielle Liste	... nur von einem einfachem Typ ... von verschiedenen einfachen Typen
Es gibt Elemente zusammengesetzten Typs (Verbund und/oder Liste)	strukturierte Listen	... Listen und evtl. einfache Typen (s. Baumstruktur) ... sequentielle Listen gleicher Länge (z.B. Assozationslisten) (s. Datenmodellierung) ...Verbunde vom gleichen Typ

Tabelle 3.2: Listenübersicht

Einen häufig gebrauchten Sonderfall von strukturierten Listen liefert folgender Situation: Wenn wir etwa ein Telefonbuch modellieren wollen,

Name	*Tel-Nr.*
...	...
Maier	12358
Müller	25011
Lehmann	78892
Schmidt	1587
...	...

bietet sich eine Liste von Paaren an, wobei ein Paar durch einen Verbund oder selbst als Liste realisiert werden könnte:

```
(define telefon-buch
   (list
      (list Maier 12358)
      (list Müller 25011)
      (list Lehmann 78892)
      (list Schmidt 1587))
```

oder

```
(define-struct person (name nummer)

(define telefon-buch
   (list
         (make-person Maier 12358)
         (make-person Müller 25011)
         (make-person Lehmann 78892)
         (make-person Schmidt 1587))
```

Man spricht von einer **Assoziationsliste**, da jedem Namen ein Wert assoziiert (*it.* associare, „beigesellen") wird, wobei anhand des Namens (Schlüssel) nach der Nummer (Wert) gesucht wird. Mit einer Verallgemeinerung der Merkmale ergibt sich:

Eine **Assoziationsliste** *al*

- `empy`
  oder
- `(cons <el> al)`
  wobei
  - `<el>`
    entweder
    `(liste <p1> <p2> ...)`
    oder
    `make-<verbund> <p1> <p2> ...)`
    und `<p1> <p2> ...` einfache Datentypen sind
  - *al* eine Assoziationsliste

Übungen

1. Angenommen, folgende Definitionen liegen vor:

```
(define Liste1 (list "A" "B" "C"))
(define Liste2 (cons "A" (list "B" "C")))
(define Liste3 (cons "A" (cons "B" (cons "C" empty))))
(define xlist (list "2" 2))
(define paare (list (cons "Adam" (cons "Eva" empty)) (cons "Paul" (cons "Paula" empty))))
(define spezialListe (list 1 (list 2 3 (list 5 7) 9)))
```

2. Welche Ergebnisse bringen folgende Funktionsaufrufe ?

```
(rest (first paare)) ==>
(first (rest paare)) ==>
(rest (rest paare)) ==>
(first (first (rest paare))) ==>
(rest (first (rest paare))) ==>
(cons Liste1 xlist) ==>
(cons (first xlist) (rest xlist)) ==>
(cons? (rest (rest paare))) ==>
(empty? (rest (rest paare))) ==>
(list Liste1 xlist) ==>
(list (rest xlist) (cons (first xlist) (rest xlist)))==>
(length (rest Liste2)) ==>
(cons (length Liste2) Liste2) ==>
(list? Liste1) ==>
(first (reverse Liste3)
(length (reverse Liste3))
```

3. Gib eine Kombination von `first` und `rest` an, mit der man die 7 aus spezialListe (s.o.) herausfiltern kann.
4. Modelliere als Assoziationslisten
   a) die Lottogewinnzahlen vom Wochenende
   b) das Ersatzteillager einer Autowerkstatt
   c) eine Klassenliste in verschiedenen Varianten
   d) die Notenverteilung der letzten Kursarbeit
   e) Temperaturmessungen an verschiedenen Orten (zu verschiedenen Zeitpunkten)
5. Modelliere eine Messreihe zu Bestimmung des el. Widerstandes. Beginne mit `(define-struct messung (U I))`

3.7 Induktive Datentypen & Rekursion

3.7.1 Rekursion über Listen

Die Besonderheit der Listenstruktur legt es nahe, dass die von uns definierten Funktionen, die auf Listen operieren sollen, vermutlich auch eine besondere Struktur haben. Machen wir uns nochmal klar:

> Eine **Liste**
>
> - ist leer
>   oder
> - besteht aus
>   - einem ersten Element
>   - einem Rest, der selbst wieder eine Liste ist

Bei den meisten Aufgaben mit Listen wird es darum gehen, dass mit jedem Element der Liste gleiche gemacht wird. Dies legt folgendes Funktionsschema nahe:

> **mache-etwas-mit** Liste
>
> - ist leer: Sonderfall
>   oder
> -
>   - behandle erstes Element
>   - **mache-etwas-mit** dem Rest der Liste

Wir unterscheiden zwei Fälle von Funktionen, die auf einer oder mehreren Listen operieren:

1. Ausgabe: keine Liste, d.h. ein anderer Datentyp
2. Ausgabe: Liste

Beispiel 1:
Zum ersten Fall überlegen wir uns eine Funktion *;summe*, die jeder Zahlen-Liste die Summe ihrer Element zuordnet, d.h. der Funktionswert ist vom Typ Zahl.
Vertrag:
;summe: Liste –> Zahl
und erhalten als Funktionsschablone gemäß

```
(define (summe liste)
   (...liste...(rest liste)...))
```

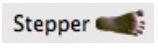
Wir müssen zuerst feststellen, ob die Liste leer ist, dann liegt es nahe, den Wert der Funktion als 0 zu betrachten; andernfalls müssen wir etwas mit dem ersten Element und dem Rest der Liste unternehmen:

```
(define (summe liste)
   (cond ((empty? liste) 0)
         (else
          (...liste...(rest liste)...))))
```

Genau genommen müssen wir das erste Element zur Summe der restlichen Elemente addieren:

```
(define (summe liste)
   (cond ((empty? liste) 0)
         (else
          ( + (first liste) (summe (rest liste))))))
```

Was passiert dabei ?

Es wird uns klar, wenn wir den Aufruf `(summe (list 1 2 3))` im Stepper 

verfolgen, hier eine Abkürzung:

```
(summe (list 1 2 3))              =>  (+ (first (list 1 2 3)) (summe (rest (list 1 2 3))))
(+ 1 (summe (list 2 3)))          =>  (+ 1 (+ (first (list 2 3)) (summe (rest (list 2 3)))))
(+ 1 (+ 2 (summe (list 3))))      =>  (+ 1 (+ 2 (+ (first (list 3)) (summe (rest (list 3))))))
(+ 1 (+ 2 (+ 3 (summe empty))))   =>  (+ 1 (+ 2 (+ 3 0)))
=> 6
```

Das heißt, es wurde im Rumpf anstelle von `(summe (rest list)) = (summe (2 3))` der komplette Rumpf eingesetzt, in dem wiederum anstelle von `(summe (rest list)) =(summe (3))` der komplette Rumpf eingesetzt, in dem wiederum... usw. bis `(summe empty)` erreicht wird, was nach Definition 0 ergibt, wobei jedesmal das erste Zahl der restlichen Liste addiert wurde, so dass sich schließlich 6 ergab.
Ergebnis: Die Funktion (oder Prozedur) summe ruft sich im Rumpf **selbst** auf!

Beispiel 2:

Wir betrachten wieder `(list 1 2 3)` als Eingabe; es soll daraus eine neue Liste entstehen, in dem die Elemente verdoppelt werden, d. h. es wird daraus `(list 2 4 6)`.
Unser Vertrag lautet also *;doppler: Liste –> Liste*
Auch hier müssen wir zuerst feststellen, ob die Liste leer ist, dann hätten wir als Ausgabe ebenfalls eine leere Liste:

```
(define (doppler liste)
   (if (empty? liste)
       empty
       (... ? ...)))
```

Falls die Liste nicht leer ist, soll das erste Element der neuen Liste doppelt so groß sein wie das erste Element von `list`, also `(cons (* 2 (first liste)) .....)`

Und wie wird der Rest konstruiert ?
Indem wir einfach unsere neue Funktion auf den Rest der Liste anwenden!

```
(define (doppler liste)
   (if (empty? liste)
       empty
       (cons (* 2 (first liste)) (doppler (rest liste)))))
```

Auch hier schafft der *Stepper* Einsicht.

Ergebnis: Die Funktion (oder Prozedur)`doppler` ruft sich im Rumpf selbst auf!

Zussammenfassung
Wir können Funktionen definieren, die sich selbst (im Rumpf) aufrufen.
Solche Funktionen nennen wir **rekursive Funktionen** oder **Rekursionen**.
Sie

- bestehen aus einer Verzweigung
  - in einen **terminierenden** Zweig (sonst läuft das Programm endlos !)
  - in einen sich **selbst aufrufenden** Zweig
- ermöglichen eine beliebig häufige **Wiederholung** eines Aufrufs einer Funktion

Hinweise

- In den Sprachwissenschaften ist es völlig unüblich, das, was man definieren will, mit Hilfe von sich selbst zu definieren (Vorsicht mit Latein- oder Deutschlehrern)
- Rekursive Funktionen sind ideal zu Behandlung von Listenstrukturen, da beide eine ähnliche Struktur, nämlich die „induktive", aufweisen.

Übungen

1. Verallgemeinere die Funktion `doppler` (aus Beispiel 2) so, dass jedes Element einer Zahlen-Liste mit einem beliebigen Faktor multipliziert wird.
2. Entwickle eine rekursive Funktion, die
   a) die Summe der Zahlen einer Zahlenliste ermittelt
   b) das Produkt der Zahlen einer Zahlenliste ermittelt
   c) den Mittelwert der Zahlen einer Zahlenliste ermittelt
   d) eine beliebige Liste kopiert
   e) die Quadrate der Zahlen einer Zahlenliste liefert
3. Nach dem wir die `length`-Funktion rekursiv nachbilden konnten, dürftes auch z.B. mit der `append`-Funktion kein großes Problem sein.
   Wir nennen sie hier `anhaenge`, und es müßte z.B. folgendes zu erwarten sein:
   `(anhaenge (list 1 2 3) (list 4 5) ==> (list 1 2 3 4 5)`
   mit dem Vertrag: *;anhaenge: liste liste –> liste*
   Lösung:

```
(define (anhaenge list1 list2)
  (cond
    ((empty? list1)
     list2)
    (else
     (cons (first list1) (anhaenge (rest list1) list2)))))
```

4. Entwickle eine Funktion, die das gleiche wie `reverse` bewirkt
   (Tip: Benutze dabei `append` bzw. `anhaenge`)
5. Entwickle eine Funktion
   ;entferne: element liste –> liste, die ein bestimmtes Element aus einer Liste entfernt
6. Versuche herauszufinden, was die Funktionen bewirken, indem Du
   - den Quellcode studierst (schwierig)
   - die Funktion auf verschiedene Listen anwendest (Lösung durch Probieren)

   a)
```
(define (f liste)
      (if (empty? liste)
         empty
          (if (empty? (rest liste))
                     (first liste)
                     (f (rest liste)))))
```

   b)
```
(define (g liste)
      (cond
        ((empty? liste) empty)
        ((empty? (rest liste)) (first liste))
        (else
         (if (> (first liste) (g (rest liste)))
             (first liste)
             (g (rest liste))))))
```

7. Entwickle eine Funktion namens `mitglied?`, die feststellt, ob ein Objekt Element einer Liste ist.
8. Konstruiere alle Übungen und Aufgaben statt mit `(if ...)` mit `(cond ...)`.
9. Mit der eingebauten Funktion string-length kann man die Anzahl der Zeichen einer Zeichenkette ermitteln.
   Entwickle ein Programm, das aus einer Liste von Wörtern
   a) das längste Wort ermittelt
   b) die Summe aller Zeichen ermittelt
10. Eine Funktion soll aus einer Zahlenliste die geraden Zahlen als neue Liste herausfiltern.
11. Untersuche und vergleiche die beiden Funktionen u1 und u2:

```
(define (u1 liste)
  (if (empty? liste)
      empty
```

```
          (if (zero? (remainder (first liste) 2))
              (cons (first liste) (u1 (rest liste)))
              (u1 (rest liste)))))

(define (u2 liste)
  (cond
    ((not (list? liste)) "Keine Liste")
    ((empty? liste) empty)
    ((not (number? (first liste))) "nicht alles Zahlen")
    ((zero? (remainder (first liste) 2)) (cons (first liste) (u2 (rest liste))))
       (else (u2 (rest liste)))))
```

12. Untersuche

```
(define (auf? liste)
   (cond
     ((< (length liste) 2) true)
     ((> (first liste) (first (rest liste))) false)
     (else
      (auf? (rest liste)))))
```

Betrachten wir die drei Listen

```
(define spezial-liste1 (list 3 "Hans" (make-posn 4 5) 6))
(define spezial-liste2 (list 3 "Hans" 5 6))
(define spezial-liste3 (list 3 "Hans" (list "Dampf" 5) 6))
```

Mit

```
;einfach-test liste -> zeichenkette (Typ)

(define (einfach-test liste)
   (cond
     ((empty? liste) "leer")
     ((empty? (rest liste))
              (cond
                ((or
                  (number? (first liste))
                  (boolean? (first liste))
                  (char? (first liste))
                  (string? (first liste))
                  (symbol? (first liste)))
"sequentielle Liste")
                (else
                 "struktrierte Liste")))
     ((not (or
            (number? (first liste))
            (boolean? (first liste))
            (char? (first liste))
            (string? (first liste))
            (symbol? (first liste))))
"struktrierte Liste")
     (else
      (einfach-test (rest liste)))))
```

können wir den Listentyp ermitteln.

Übungen

1. Angenommen, es gäbe als zusammengesetzte Datentypen nur Listen und Verbunde. Vereinfache die Funktion `einfach-test` (Hinweis: Benutze `list?` und `struct?`)
2. Ein Elektriker nimmt an einem Draht eine Messreihe vor, um festzustellen, ob der Widerstand in etwa gleich bleibt, wenn sich die Spannung ändert. Dazu definiert er
   `(define-struct messung (Volt Ampere))`
   und macht folgende Messungen:

```
(define messreihe (cons (make-messung 5 0.22)
   (cons (make-messung 8.4 0.35)
   (cons (make-messung 11 0.43) empty))))
```

a) Gib eine Funktion an, die aus der Messreihe eine Liste mit den einzelnen Widerständen liefert.

b) Gib eine zweite Funktion an, die aus dem Ergebnis von a) den Mittelwert errechnet.

3. Ergaenzung zu Aufgabe 14, S. II,31 (Der Quotient U/I gibt in den elektr. Widerstand an) Gegeben sind:

```
(define-struct messung (U I))

(define (messreihe n)
   (cond
      ((zero? n) empty)
      (else
      (cons (make-messung (+ 5 (/ (random 10) 10)) (/ (random 100) 100)) (messreihe (- n 1))))))

(define z (messreihe 5))

(define (f1 aliste)
  (cond
    ((empty? aliste) 0)
    (else
     (+ (messung-U (first aliste)) (f1 (rest aliste))))))

(define (f2 aliste)
  (/ (f1 aliste) (length aliste)))

(define (f3 aliste)
  (cond
    ((empty? aliste) empty)
    (else
     (cons (/ (messung-U (first aliste)) (messung-I (first aliste))) (f3 (rest aliste))))))
```

a) Was bewirkt messreihe?

b) Wofür die Variable z?

c) Was bewirkt f1?

d) Was bewirkt f2?

e) Was bewirkt f3?

4. Angenommen, man die Tabelle von Punkten in einer x-y-Ebene als Assoziationsliste:
`(define tab1 (list (list 2 4) (list 3 6) (list 4 9)))`
Die Funktion `quotienten-liste` liefert von jedem Punkt den x/y-Quotient:

```
(define (quotienten-liste a-liste))
  (cond
    ((empty? a-liste) a-liste)
    (else
     (cons
      (/ (first (first a-liste)) (first (rest (first a-liste))))
      (quotienten-liste (rest a-liste))))))
```

a) Wende `quotienten-liste` auf `tab1` an.

b) Ändere `quotienten-liste` und texttttab1 ab für den Fall, dass die Punkte vom Typ `posn` sind.

c) Entwickle eine Funktion `alle-quotienten-gleich?`, die als Eingabe eine Liste von Quotienten hat.

d) Entwickle eine Funktion geraden-steigung, die (unter Benutzung von `alle-quotienten-gleich?`) die Steigung der Ursprungsgeraden liefert, sofern es sich um eine Ursprungsgerade handelt.

5. Bei der Kfz-Zulassungsstelle sind die Häufigkeiten einiger Automarken als Listen vorhanden:

```
(define Ort0 (list (list 'Peugoet 200)))

(define Ort1 (list (list 'VW 124) (list 'BMW 47) (list 'Mercedes 88) (list 'Opel 69)
   (list 'Ford 57) (list 'alfa-romeo 12) (list 'Porsche 9) (list 'Toyota 50)))

(define Ort2 (list (list 'VW 194) (list 'BMW 43) (list 'Mercedes 98) (list 'Opel 62)
   (list 'Ford 47) (list 'alfa-romeo 8) (list 'Porsche 11) (list 'Toyota 70) (list 'Renault 33)))

(define Ort3 (list (list 'VW 155) (list 'BMW 38) (list 'Mercedes 92) (list 'Opel 63)
   (list 'Ford 51) (list 'alfa-romeo 5) (list 'Toyota 44) (list 'Renault 43) (list 'Fiat 28)))
```

Es gibt auch „Ortslisten", z.B.

```
(define Orts-Liste (list Ort2 Ort3))
```

a) Wozu dient folgende Funktion ?

```
(define (erhoehe a-list marke zahl)
  (cond
    ((empty? a-list)
     (cons (list marke zahl) empty))
    ((equal? marke (first (first a-list)))
     (cons (list marke (+ zahl (first (rest (first a-list))))) (rest a-list)))
    (else
     (cons (first a-list) (erhoehe (rest a-list) marke zahl)))))
```

b) Entwickle eine Funktion *auswertung-2-Listen : OrtA OrtB –> Liste*, die von zwei Orten die gemeinsame Häufigkeit der Automarken unter Benutzung von erhoehe ermittelt.

c) Entwickle eine Funktion *auswertung: Orts-Liste Ort –> Liste*, die von einer Orts-Liste und einem Ort die gemeinsame Häufigkeit der Automarken unter Benutzung von *auswertung-2-Liste* ermittelt.

3.7.2 Rekursion über n $\in \mathbb{N}$

Bisher haben wir nur rekursive Funktion konstruiert, die eine (oder mehrere) Liste(n) als Eingabeparameter hatten; wie aber können wir Funktionen konstruieren, die als Eingabeparameter keine Liste haben ?
Wir hätten einen Vertrag folgender Form:

;funktion: <Objekt> –> Liste

In diesem Fall haben wir nicht die Möglichkeit, dass der Selbst-Aufruf im Rumpf am Ende einer Liste (als Eingabeparameter) in den terminierenden Zweig, also die `(if (empty? liste) ...` gelangt.
Wir müssen uns also ein anderes Abbruchskriterium überlegen: Genau genommen hat sich in den bisherigen Beispielen eine rekursive Funktion so oft selbst aufgerufen, wie die Anzahl der Elemente betrug, d.h. wenn die Liste n Elemente hat, beträgt der Rest beim ersten Aufruf n-1, beim zweiten Aufruf n-2 usw. bis 0 erreicht ist: wir können also auch sagen, eine Zahl n wurde beim jedem Selbstaufruf um 1 erniedrigt, bis sie den Wert 0 hat.
Daher können wir die Abfrage `(if (empty? liste) ...` durch `(if (= 0 (length liste) ...` ersetzen, wofür wir auch `(if (= 0 n) ...` setzen können.

Dies wird vermutlich deshalb funktionieren, die hier verwendeten natürlichen Zahlen genau wie die Listen die Eigenschaften eines induktiver Datentyps haben:

> Eine *natürliche Zahl n*
>
> - ist 0
>   oder
> - ist Nachfolger einer natürlichen Zahl, d.h. mit n ist auch $n+1$ eine natürliche Zahl

Greifen wir diese Idee bei folgendem Problem auf:

Beispiel 1

Wir wollen eine Liste von Zufallszahlen, z.B. aus dem Bereich 0...5, also mit `(random 6)` erzeugen. Da die Liste zwar beliebig lang, aber doch endlich sein muß, nennen wir die Anzahl der Elemente n; das wäre gleichzeitig auch unser einziger Eingabeparameter:

```
(define (zufallsliste n)
    (.......(zufallsliste (- n 1)) ......)
```

Dadurch, dass wir den Selbstaufruf mit n-1 durchführen, ist sichergestellt, dass irgendwann der Wert 0 erreicht wird, was wir dann im terminierenden Zweig berücksichtigen:

```
(define (zufallsliste n)
  (cond ((zero? n)empty)
        (else
         (cons (random 6) (zufallsliste (- n 1))))))

(zufallsliste 6) ==>  (list 4 0 3 4 5 2)
```

Insgesamt haben eine Funktion mit dem Vertrag *;zufallsliste: zahl –> liste.*
Hier laufen also die Selbstaufrufe der Funktion nicht über das jeweils nächste Element einer einzugebenden Liste, sondern über den Parameter n. Wenn wir nun als Ausgabe keine Liste wollen, sondern etwa nur eine Zahl, braucht in einer Rekursion überhaupt keine Liste vorzukommen, wie folgendes Beispiel zeigt:

Beispiel 2

Wir wollen die Summe aller Zahlen von 1 bis n bestimmen; wir haben also den Vertrag

```
;; summe: zahl --> zahl
(define (summe n)
    (if (= 0 n)
        0
        (+ n (summe (- n 1)))))
```

mit z. B. `(summe 5) ==> 15`

Wenn wir Beispiel 2 abstrahieren, gelangen wir zu

Scheme

```
(define (f n)
  (cond
    ((zero? n) g0)
    (else
     (.....n....(f (- n 1))))))
```

Mathematische Schreibweise:

$$\begin{aligned} f(0) &= g_0 \\ f(n) &= h(n, f(n-1)) \text{ für n} > 0 \end{aligned}$$

(h steht für den `else`-Zweig)

In dieser Schablone fungiert der Parameter n als Zähler, der mit jedem Selbstaufruf schrittweise auf 0 heruntergezählt wird.
Beispiel 1 können wir weiter verallgemeinern, indem einen zusätzlichen Parameter maxi einführen, etwa um die Obergrenze der Zufallszahlen festzulegen, und erhalten folgenden Vertrag der Form

;zufallsliste2: maxi n –> zahl

Beispiel 3

```
(define (zufallsliste2 maxi n)
  (cond
    ((zero? n) empty)
    (else
      (cons (random maxi) (zufallsliste2 maxi (- n 1))))))
```

mit `(zufallsliste2 49 6) ==> (list 9 22 18 5 29 39)`

Beispiel 4

Bei dem zusätzlichen Parameter kann es sich aber auch ein beliebiges Objekt handeln, etwa um eine Liste von Zeichenketten, aus der wir mittels der Funktion *n-tes-element* die n-te Zeichenkette herausgreifen wollen mit dem Vertrag

;n-tes-element: liste zahl –> zeichenkette

```
(define (n-tes-element liste n)
  (cond
    ((equal? n 1) (first liste))
    (else
      (n-tes-element (rest liste) (- n 1)))))
```

mit `(n-tes-element (list "Adam" "Eva" "Corinna" "Seppl") 3) ==> "Corinna"`

Abstrahieren wir über die Beispiele 3 und 4, erhalten wir eine allgemeinere Form[4]:

Scheme

```
(define (f p1 p2 ... n)
  (cond
    ((zero? 0) (... p1 p2...))
    (else
     (...n...(f p1 p2...(- n 1))))))
```

Mathematische Schreibweise:

$$f(p1, p2, ..., 0) = g(p1, p2, ...)$$
$$f(p1, p2, ..., n) = h(n, f(p1, p2, ..., n-1)) \text{ für } n > 0$$

In der letzten verallgemeinerten Form lassen sich eine enorme Anzahl von rekursiven Funktionen definieren, z.B. viele der uns aus der Mathematik bekannten Zahlenfolgen, wie wir im folgenden sehen werden, aber wir können sie noch weiter verallgemeinern.
Insbesondere können wir auch rekursive Funktionen definieren, in denen Listen keine Rolle spielen, weder als Eingabe noch als Ausgabe, dazu folgendes

Beispiel 5

Betrachten z.B. die Folge 1, 3, 9, 27, 81, ..., d.h. jedes Folgenglied ist das 3-fache des vorangegangen; nach Schablone erhalten wir sofort:

```
(define (geo-einfach n)
  (cond
    ((equal? n 1) 1)
    (else
     (* 3 (geo-einfach (- n 1))))))
```

mit `(geo-einfach 3) ==> 9`
Dies ist aber lediglich ein Spezialfall der sog. geometrischen Folgen der Form $a_n = a \cdot q_n$ mit $0 < q$ und $q \neq 1$.

1. Verallgemeinerung: ein beliebiger Startwert a:

   ```
   (define (geo-nicht-so-einfach a n)
     (cond
       ((equal? n 1) a)
       (else
        (* 3 (geo-nicht-so-einfach a (- n 1))))))
   ```

   mit `(geo-nicht-so-einfach 4 3) ==> 36`

2. Verallgemeinerung: zusätzlich ein beliebiger Faktor q:

   ```
   (define (geo a q n)
     (cond
       ((equal? n 1) a)
       (else
        (* q (geo a q (- n 1))))))
   ```

   mit `(geo 4 5 3) ==> 100`

Am letzten Beispiel wollen wir zwei Aspekten nachgehen:

- Wir erhalten so nur das n-te Glied der Folge. Häufig würde man aber gerne die ersten n Glieder sehen; dazu müssen nur eine neue Funktion `geo-list1` konstuieren, die `(geo a q n)` aufruft und damit eine Liste der ersten n Glieder als Ausgabe liefert:

  ```
  (define (geo-list1 a q n)
         (if (= n 1)
             (list a)
  ```

[4] Ausblick auf das Thema *Berechenbarkeit*: *primitiv-rekursive* Funktionen

```
            (cons (geo a q n) (geo-list1 a q (- n 1)))))

(geo-list 4 5 3)                ==> (list 100 20 4)
```

Leider erhalten wir so die Folgenglieder nicht in der gewünschten Reihenfolge, was sich durch `(reverse (geo-list 4 5 3))` ändern lässt. Oder wir bauen die Umkehrung der Reihenfolge direkt ein:

```
(define (geo-list2 a q n)
  (if (< n 1)
      empty
      (append (geo-list2 a q (- n 1)) (list (geo a q n)))))
```

Sowohl `geo-list1` als auch `geo-list2` sind äußerst unökonomisch, da in beiden Funktionen jeweils für jedes neue Listenelement geo aufgerufen, was seinerseits als rekursive Funktion alle Glieder berechnet - also haben wir hier eine Form von „doppelt gemoppelt". Alternativen werden wir später finden.

- Genau besehen, entspricht der Rumpf von `(geo a q n)` nicht unserer allgemeinen Schablone, und zwar aus zwei Gründen:
  1. Im terminierenden Zweig hat die Abfrage nicht die Form `((zero? 0)` , sondern `(equal? n 1)` Ändern wir in `geo` die Abfrage `(zero? n)` ab, erhalten wir
     `(geo 4 5 3) ==> 500`
     da wir ja bereits bei 0 zu zählen anfangen, d.h. wir berechnen das (n+1)-te Glied. Das kann man aber mit
     `(geo 4 5 2) ==> 100`
     wieder ausgleichen. Ohne Beweis gilt: durch Umformung jede Abfrage im terminierenden Zweig auf die Form `(zero? 0)` gebracht werden kann.
  2. Im Gegensatz zur letzten allgemeinen Schablone kommt im Rumpf von `geo-list1` der Parameter q außerhalb des Selbstaufrufs vor. Also kann noch weiter verallgemeinert werden.

Die letzte Überlegung führt uns damit zur (hier) allgemeinsten Form einer rekursiven Funktion:

Scheme	*Mathematische Schreibweise:*
`(define (f p1 p2 ... n)` `(if (= n 0)` `  (..p1 p2..)` `  (..p1 p2..n..(f p1 p2..(- n 1))))`	*f(p1,p2,...,0) = g(p1,p2,...)* *f(p1,p2,...,n) = h(p1,p2,...,n,f(p1,p2,...,n-1))* für *n > 0*

Übungen

1. Erstelle eine Funktion *;quersumme: zahl –> zahl*
   Tip: Benutze dabei die Ganzzahl-Operationen `quotient` und `remainder`

2. Wozu dient folgende Funktion ?

   ```
   ;unbekannt1?: liste --> boolean
   (define (unbekannt? eineListe)
     (cond
       ((empty? eineListe) true)
       ((not (number? (first eineListe))) false)
       (else
        (unbekannt? (rest eineListe)))))
   ```

3. Erstelle mit Hilfe von 2. eine Funktion *;sichere-summe: zahl –> zahl* die Werte einer Zahlenliste nur dann addiert, wenn außer Zahlen keine anderen Werte vorkommen

4. Wozu dient folgende Funktion ?

```
; unbekannt2?: liste X X X--> liste
(define (unbekannt2 eineListe c1 c2 c3)
  (cond ((empty? eineListe)
             (list c1 c2 c3))
        ((number? (first eineListe))
             (unbekannt2 (rest eineListe) (+ c1 1) c2 c3))
        ((symbol? (first eineListe))
             (unbekannt2 (rest eineListe) c1 (+ c2 1) c3))
        ((list? (first eineListe))
             (unbekannt2 (rest eineListe) c1 c2 (+ c3 1)))
        (else
             (unbekannt2 (rest eineListe) c1 c2 c3))))
```

5. Mit *;Lottozahlen: Zahl –> Liste* können wir leicht „6 aus 49" erzeugen:

```
(define (Lottozahlen n)
                (if (= n 1)
                        (cons (+ (random 48) 1) empty)
                (cons (+ (random 48) 1) (Lottozahlen (- n 1) ))))
```

   Teste Lottozahlen mehrfach, und finde heraus, weshalb diese Prozedur in der Praxis nicht brauchbar ist. (Lösung im Kapitel: Lokale Variablen)

6. Gegeben ist

```
(define (f1 a c n)
  (cond
    ((not (number? n)) "keine Zahl")
    ((zero? n) a)
     (else (+ c (f1 a c (- n 1))))))
```

   a) Berechne für a = 2, c = 3 die ersten 3 Glieder der Folge
   b) Gib eine explizite (= nicht rekursive) Dartellung von f1 an.

7. Es sind folgende Funktionen definiert:

```
(define (n2FolgeA n)
  (cond ((= n 0) empty)
        (else
         (cons (* n 2) (n2FolgeA (- n 1))))))

(define (n2FolgeB n)
  (cond ((> n 10) empty)
        (else
         (cons (/ 1 n) (n2FolgeB (+ 1 n))))))

(define xFolge (n2FolgeA 4))

(define (ausgabe folge)
  (if (empty? folge)
      empty
      (cons (first folge) (rest folge))))

-----------Aufrufe
(n2FolgeA 14)
(n2FolgeB 14)
(ausgabe xFolge)
```

   a) Stelle fest - zunächst ohne DrScheme - was die drei Aufrufe bewirken
   b) Kannst Du die beiden ersten auch (= nicht rekursiv) formulieren ?

8. Umwandlung von ganzen Dezimalzahlen (10er-System) in Dualzahlen (als Liste)

```
;Umwandlung von ganzen Dezimalzahlen (10er-System) in Dualzahlen (als Liste)
(define (dez->dual-hilfe z)
  (cond ((zero? z) empty)
        (else
         (cons (remainder z 2) (dez->dual-hilfe (quotient z 2))))))
(define (dez->dual dezZahl)
  (cond ((zero? dezZahl) (list 0))
        (else
         (reverse (dez->dual-hilfe dezZahl)))))
; Umwandlung von Dualzahlen (als Liste) in ganze Dezimalzahlen (10er-System)
(define (dual->dez liste)
  (cond ((empty? liste) "keine Zahl")
        (else
         (dual->dez-hilfe liste))))
(define (dual->dez-hilfe liste)
  (cond ((empty? liste) 0)
        (else (+ (* (first liste) (expt 2 (- (length liste) 1))) (dual->dez-hilfe (rest lis
```

a) Teste
   - `(dez->dual 2000) -> ?`
   - `(dual->dez kByte-dual)` mit `(define kByte-dual (list 1 0 0 0 0 0 0 0 0 0 0)) -> ?`

b) Seien

```
(define kByte-dual (list 1 0 0 0 0 0 0 0 0 0 0))
(define MByte-dez (* (dual->dez kByte-dual) (dual->dez kByte-dual)))
```

Definiere daraus und mit Hilfe obiger Funktionen
   - - kByte-dez - MByte-dual - GByte-dez - GByte-dual

c) Warum funktioniert `(define MByte (* kByte-dual kByte-dual))` nicht ?

9. Mit Hilfe von Listen können wir leicht den radioaktiven Zerfall simulieren:
Zunächst fassen eine vorhandene Menge von Atomen (Isotopen gleicher Art) als Liste von Nullen auf:

```
;Atome-als-Nullen: Zahl --> Liste
(define (Atome-als-Nullen n)
  (cond
    ((zero? n) empty)
    (else
     (cons 0 (Atome-als-Nullen (- n 1))))))
```

und erhalten z.B.

```
> (Atome-als-Nullen 20)
 --> (list 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

Der eigentliche Zerfall soll nun so erreicht werden, daß die Liste der Atome von einer Prozedur zerfallsgeneration durchlaufen wird, und jede Null (also jedes Atome) mit einer Wahrscheinlichkeit $0 < p < 1$ aus der Liste entfernt wird und somit eine neue Liste mit den verbleibenden Nullen (Atomen) ausgibt, die die neue Zerfallsgeneration darstellt:

```
;zerfallsgeneration: Liste Zahl --> Liste
(define (zerfallsgeneration atom-Liste p)
  (cond
    ((empty? atom-Liste) empty)      ; keine Atome mehr da ?
    ((>= <Wahrscheinlichkeit> p)     ; wenn Zufallszahl >= p:
     (...<Atom bleibt in Liste>...)) ; kein Zerfall: 0 wird durch 0 erzetzt
    (else
     (...<Atom fällt aus Liste>...)))) ; Zerfall: 0 wird nicht erzetzt
```

Aber wie realisieren wir <Wahrscheinlichkeit>? Gehen wir davon aus, daß p max. 3 Hinterkommastellen hat, genügt der Ausdruck `(/ (random 1000) 1000)`. Überlege, weshalb das funktioniert!

a) Vervollständige *zerfallsgeneration* und teste z.B. `(zerfallsgeneration (reihe0 20) 0.5)`

b) Erstelle eine Prozedur zerfall

3.7.3 Sonderfall: Endrekursion (*tail recursion*)

Betrachten wir nochmal das Standardbeispiel (neben den Fibonacci-Zahlen) für Rekursion, die Fakultät n!:

$$n! = \begin{cases} 1 \text{ für } n = 0 \\ n \cdot (n-1) \text{ für } n \geq 1 \end{cases}$$

```
(define (fak n)
  (cond ((= n 0) 1)
        (else (* n (fak (- n 1))))))
```

Im Stepper sehen wir, dass das Verfahren im wahrsten Sinn des Wortes „recurrere" zurückläuft bis auf 0!, und dann nacheinander mit 1, 2, 3 usw. zu multiplizieren und durch Rückwärtseinsetzung auszurechnen, d.h. der Speicheraufwand wächst mit n:

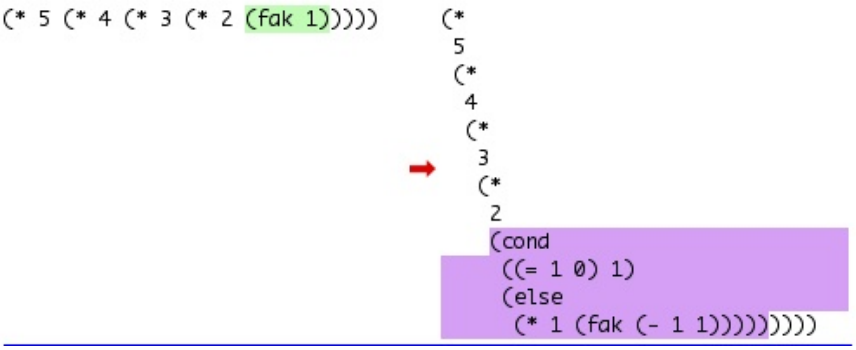


Abb. 3.4: *n !* im Stepper

Vergleichen wir jetzt die Alternative

```
(define (neu-fak n)
  (fak-iter n 0 1))
```

wobei

```
(define (fak-iter n anzahl resultat)
  (cond
    ((= anzahl n) resultat)
    (else
     (fak-iter n (+ anzahl 1) (* (+ anzahl 1) resultat)))))
```

Der Aufruf `(neu-fak 5)` wird durch `(fak-iter 5 0 1)` ersetzt und liefert auch den Wert 120, allerdings benötigt die Hilfsfunktion `fak-iter` bei jedem Selbstaufruf keinen zusätzlichen Speicherplatz, da die Zwischenergebnisse in resultat gespeichert werden:

```
(neu-fak 5)     = (fak-iter 5 0 1)
                        = (fak-iter 5 1 1)
                        = (fak-iter 5 2 2)
                        = (fak-iter 5 3 6)
                        = (fak-iter 5 4 24)
                        = (fak-iter 5 5 120)
                        = 120
```

Einen solchen Paramater nennt man auch **Akkumulator**. D.h. der Speicherbedarf bleibt während des Abarbeitens konstant; solche Rekursionen werden auch **endrekursiv** genannt - während fak „echt" rekursiv ist.
Ein weiteres Beispiel ist die Such nach dem kleinsten Element einer (Zahlen-)Liste, also

```
;min-el: liste --> zahl
```

Wenn die Liste leer ist, existiert kein kleinstes Element, dann machen wir eine entsprechende Meldung. Wenn sie nicht leer ist, rufen eine Hilfsfunktion `;min-hilfe: liste min` mit einem zusätzlichen Parameter `min` als Akkumulator:

```
(define (min-el liste)
  (cond
    ((empty? liste) "gibt es nicht")
    (else
     (min-hilfe liste (first liste)))))
```

Der Akkumulator wird mit dem ersten Element aufgerufen, das vorläufig als kleinstes Element angesehen wird, und `min-hilfe` durchsucht dann den Rest der Liste nach einem kleineren Element als im Akkumulator steht:

```
(define (minel-hilfe liste min)
  (cond
    ((empty? (rest liste)) min)
    ((< (second liste) min) (minel-hilfe (rest liste) (second liste)))
    (else
     (minel-hilfe (rest liste) min))))
```

Übungen

1. Einen Sonderfall des beschränkten Wachstum stellt das logistische Wachstum dar:
   Eine Population P_n wächst

   exponentiell mit der Wachstumsrate r

   $$P_{n+1} = P_n + r_1 \cdot P_n \tag{3.2}$$

   proportional zur Differenz aus aktueller Population und maximalem Wert:

   $$P_{n+1} = P_n + r_2 \cdot (Max - P_n) \tag{3.3}$$

   Zusammen ergeben (3.2) und (3.3) mit $r = r_1 \cdot r_2$:

   $$P_{n+1} = P_n + r \cdot P_n \cdot (Max - P_n)$$

   Betrachtet man die relative Population $p = \frac{P}{Max}$, erhalten wir

   $$p_{n+1} = p_n + r \cdot p_n \cdot (1 - p_n)$$

   Eine Codierung in Scheme ergibt zunächst:

   ```
   (define (p n r)
      (cond
      ((zero? n) 0.1)   ;Anfangswert
      (else
         (+ (p (- n 1) r) (* r (p (- n 1) r) (- 1 (p (- n 1) r)))))))
   ```

   Das bedeutet, dass p sich im Rumpf 3 mal selbst aufruft !
   Ein Test mit z.B. `(p 15)` zeigt, dass wir mit diesem Ansatz in der Sackgasse sind. Auch hier hilft Endrekursion, wenn wir wie im Eingangsbeispiel vorgehen:

   ```
   (define (neu-p n r)
      (p-iter n 0 0.1 r))
   ```

   wobei

   ```
   (define (p-iter n z res)
      (cond
         ((equal? z n) res)
         (else
             (p-iter n (+ z 1) (+ res (* r res (- 1 res)))))))
   ```

   a) Vergleiche jeweils den Zeitaufwand von p und neu-p für n = 2, 5, 10, ...

b) Untersuche beide für verschiedene Werte von $0{,}2 < r \leq 3$!

2. Die Fibonacci-Zahlen sind folgendermaßen definiert:

$$\begin{aligned} f_1 &= 1 \\ f_2 &= 1 \\ f_{n+1} &= f_n + f_{n-1} \end{aligned}$$

Führe Vergleichtests mit und ohne Endrekursion durch !

3.8 Lokale Variablen & Funktionen

Variablen, die im ganzen Programm gelten, heißen **globale** Variablen, und Variablen, die nur in einer bestimmten Umgebung gelten, heißen **lokal**. *Bisher* hatten wir folgende Fälle:

global : alle Variablen, die auf oberster Ebene mit `(define <var> <Ausdruck>)` definiert sind

lokal : alle Funktionsparameter

deshalb ist es auch möglich, innerhalb eines Programms bei verschiedenen Funktionen die gleichen Bezeichner (Namen) als Parameter zu nehmen, da sie nur innerhalb der Funktions-Definition gültig sind.
Aber es gibt Situationen, wo es sinnvoll ist, irgendwo im Rumpf einer Funktions-Definition eine Art „Hilfsvariable" zu definieren, die nur an einer bestimmten Stelle gebraucht wird.

Beispiel

Wir wollen 6 verschiedene Lottozahlen erzeugen mit

```
(define (lottozahlen1 n)
    (cond
      ((= n 1) (cons (+ (random 48) 1) empty))
      (else
        (cons (+ (random 48) 1) (lottozahlen1 (- n 1) )))))
```

Wir wir schon wissen, liefert dieses Programm leider nicht immer 6 verschiedenen Zahlen, weil jede neu erzeugte Zufallszahl nicht mit den schon vorhandenen verglichen wird. Stattdessen müßten wir

```
(define (lottozahlen2 n)
  (cond
    ((= n 1) (cons (+ (random 48) 1) empty))
    (else
      (neueZahl (lottozahlen2 (- n 1) )))))
```

haben, wo die Funktion *neueZahl* folgendes leisten muß:

1. eine neue Zufallszahl erzeugen und an einen Namen, z.B. x binden
2. prüfen, ob die x in der Liste `(Lottozahlen (- n 1))` schon vorhanden ist

Also etwa so:

```
(define (neueZahl liste)
  ((define x (+ (random 48) 1)))     ; funktioniert nicht !
  (cond
    ((mitglied? x liste) (neueZahl liste))
    (else
     (cons x liste))))
```

Leider kann man `define` so nicht innerhalb eines Funktionsrumpfes aufrufen; stattdessen gibt es die Form

Die Form
(**local** (<Definition-1> ....<Definition-n>)
<Ausdruck>)
wobei <Ausdruck> (wie ein Funktionsrumpf) der Gültigkeitsbereich der lokalen Definition(en) ist.

Wir erhalten damit

```
(define (neueZahl liste)
  (local ((define x (+ (random 48) 1)))
    (if (mitglied? x liste)
         (neueZahl liste)
         (cons x liste))))
```

und brauchen nur noch `mitglied?` zu definieren:

```
(define (mitglied? x liste)
  (cond ((empty? liste) false)
        (else
         (cond ((equal? x (first liste)) true)
               (else (mitglied? x (rest liste)))))))
```

(Die vordefinierte Funktion `member` kann hier nicht gebraucht werden, s. Hilfen)
Damit ist unser Problem gelöst.
Ebenso wie Variablen können auch Funktionen lokal definiert werden:

```
(define (zylinder-oberflaeche r h)
  (local ((define (grundflaeche r)
            (* r r Pi))
          (define (Mantel r h)
            (* 2 r Pi h)))
    (+ (grundflaeche r)(Mantel r h))))
```

Auch hier ist zu beachten, dass z.B. die Funktion Grundflaeche nur innerhalb des Rumpfes `(local ...)` von bekannt ist. Würde also eine weitere Funktion gleichen Namens außerbalb von `(local ...)` , also global definiert, hätte dies zur Folge, dass mit einem Aufruf von Grundflaeche außerhalb des Rumpfes von `(local ...)` die global definierte Variante gemeint ist.

Hinweise

- In der Form `local` können Variablen- und Funktionsdefinitionen gemischt werden
- Im professionellen Scheme wird kann bei Variablendefinitionen `(local ((define .....` durch `let` abgekürzt werden,

z.B. wird `neueZahl` zu

```
(define (neueZahl liste)
  (let ((x (+ (random 48) 1)))
    (if (mitglied? x liste)
         (neueZahl liste)
         (cons x liste))))
```

Übungen

1. Bestimme für y=5 den Wert folgenden Termes:

   ```
   (local ((define x (* y 3)) (define z 7))        ; y ist vorher definiert
              (+ x z))
   ```

2. Definiere `fak-iter` aus dem Beispiel für Endrekursionen (S. II, 42) als lokale Hilfsfunktion von `neu-fak`

4 Modellierung mit Grafik

4.1 Überblick

Für die Modellierung mit Pixel-Grafik stehen zwei Varianten zur Verfügung, die jeweils als Erweiterungen der HtDP-Lernsprachen in Form von sog. *Teachpacks* dazugeladen werden (s. Menü „Hilfe"–> „Hilfezentrum"–>„Teachpacks"):

- `image.ss` und `world.ss`, *funktionaler* Ansatz
- `draw.ss`,*imperativer Ansatz*

Im folgenden wird als konsequente Fortführung des vorangegangenen Kapitels der *funktionale* Ansatz vorgestellt.
Das Teachpack `image.ss` stellt lediglich Funktionen zur Erzeugung geometrischer Figuren bereit, während `world.ss` Funktionen zur Animation und Interaktion liefert.
Dabei ist `image.ss` eine Teilmenge von `world.ss`.

4.2 Geometrische Figuren & Bilder: `image.ss`

Im Mittelpunkt steht der Datentyp *image*, der für die Funktionswerte der Grafik-„Befehle", d.h. Zeichenfunktionen, steht.
Die Ausgabe der gezeichneten Grafiken und Bilder erscheint direkt im Interaktionsfenster (REPL):

```
> (rectangle 40 30 'solid 'green)
```

Zudem ist es auch möglich, eine Variable vom Typ *image* zu definieren und sie an eine Grafik aus einer Datei (vorzugsweise *.png*, aber auch *.jpg* und *.gif* sind möglich) zu binden:

```
(define rakete )
(overlay rakete (circle 30 'outline 'blue))
```

```
Willkommen bei DrScheme, Version 4.1 [3m].
Sprache: Fortgeschritten angepasst; memory limit: 128
Teachpack: image.ss.
```

Abb. 4.1: Ergebnisse der Grafikfunktionen werden in der REPL ausgegeben

Eine **Auswahl** der Funktionen[1]:

`rectangle` : *Number Number Mode Color -> Image*
zeichnet ein Rechteck mit Breite, Höhe, Zeichenmodus und Farbe

`circle` : *Number Mode Color -> Image*
zeichnet einen Kreis mit Radius, Zeichenmodus und Farbe

`ellipse` : Number Number Mode Color -> Image zeichnet eine Ellipse mit Breite, Höhe, Zeichenmodus und Farbe

`triangle` : *Number Mode Color -> Image*
zeichnet eine gleichseitiges Dreieck mit Seitenlänge, Zeichenmodus und Farbe

`line` : *Number Number Color -> Image*
zeichnet eine Strecke von (0|0) zum Punkt den gegebenen Koordinaten

`text` : *String Size Color -> Image*
liefert einen Text als Grafik mit Text, Schriftgröße und Farbe

`pinhole-x` : *Image -> Int*
liefert die x-Koordinate des Grafik-Mittelpunktes

`pinhole-y` : *Image -> Int*
liefert die y-Koordinate des Grafik-Mittelpunktes

`put-pinhole` : *Image Int Int -> Image*
verschiebt den Mittelpunkt der Grafik an die Stelle der beiden Koordinaten

`move-pinhole` : *Image Int Int -> Image*
verschiebt den Mittelpunkt der Grafik relativ um die angegebenen Werte (positive Werte: nach rechts bzw. unten, negative Werte: nach links bzw. oben)

`overlay` : *Image Image Image ... -> Image*
überlagert mehrere Grafiken, so dass sich ihre Mittelpunkte decken

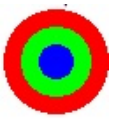
`image-width` : *Image -> Int*
liefert die Breite einer Grafik

`image-height` : *Image -> Int*
liefert die Höhe einer Grafik

Hinweise zu `image.ss`

- ▷ Vordefinierte Farben vom Typ `color` sind: `'red, 'green, 'blue, 'yellow, 'black, 'brown, 'violet`
  Weitere können als RGB-Typen mit `(make-color r g b)` erstellt werden, wobei $0 \leq$ r, g, b ≤ 255 gilt
- ▷ Für den Zeichenmodus `mode` gibt es `'outline` und `'solid`

Übungen

1. Erzeuge eine „Zielscheibe”: 
2. Untersuche

[1] Näheres s. *Hilfe –> Hilfezentrum –> teachpacks*

```
(define (unbekannt1 level)
  (if (equal? level 1) (circle 10 'outline 'blue)
      (overlay (circle (* level 10) 'outline 'blue) (unbekannt1 (- level 1)))))
```

3. Entwickle eine Funktion *;gesicht: zahl –> image*:  , wobei *zahl* den Radius angibt. Verallgemeinere die Funktion so - und zwar *rekursiv* -, dass eine Folge von größer werdenden Gesichtern erzeugt wird, wobei die Proportionen (Augen, Mund) erhalten bleiben. Tipp: Es gibt auch eine Funktion `ellipse`. Erweiterung: *smiley*?

4. Analysiere folgende Funktion nur anhand des Quellcodes:

```
(define (unbekannt2 b1 b2)
  (local ((define dx
            (+ (pinhole-x b2)
               (- (image-width b1) (pinhole-x b1)))))
    (move-pinhole
     (overlay/xy b2 dx 0 b1)
     (round (/ dx 2)) 0)))
```

4.3 Grafik im Koordinatensystem: `world.ss`

4.3.1 Statische Grafik: Visualisierung von Listen

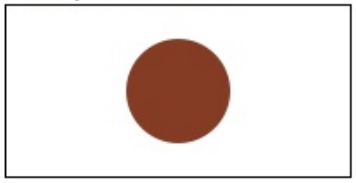
Um Figuren an einer bestimmten Stelle in einer Zeichenfläche platzieren zu können, benötigt man

1. eine *Szene* (`scene`), eine eingerahmte rechteckige Fläche, die in der REPL erscheint und in der linken oberen Ecke den Ursprung eines unsichtbaren 2-dimensionalen Koordinatensystems hat

2. die Funktion *place-image: grafik x y szene –> szene*, die eine Grafik mit ihrem Mittelpunkt an der Stelle x y in eine vorhandene Szene zeichnet und die so entstandene neue Szene in der REPL zeigt:

```
(place-image (circle 30 'solid 'brown) 100 50 (empty-scene 200 100))
```

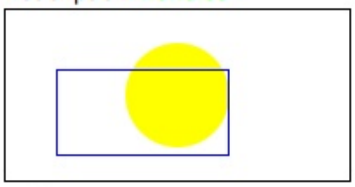
```
Willkommen bei DrScheme, Version 4.1.3 [3m].
Sprache: Fortgeschritten angepasst; memory limit: 256 megabytes.
Teachpack: world.ss.
```



oder

```
(define sz (empty-scene 200 100))
(place-image (rectangle 100 50 'outline 'blue) 80 60
             (place-image (circle 30 'solid 'yellow) 100 50 sz))
```

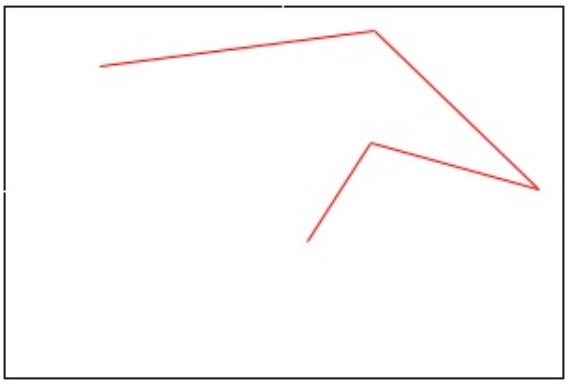
```
Willkommen bei DrScheme, Version 4.1.3 [3m].
Sprache: Fortgeschritten angepasst; memory limit: 256 megabytes.
Teachpack: world.ss.
```



Da `place-image` eine Szene verarbeitet und eine neue Szene liefert, werden mehrere Figuren durch Schachtelung von `place-image` gezeichnet.

Beispiel 1: Streckenzüge und Polygone

Die Verbindung mehrerer Strecken, bei denen der Endpunkt einer Strecke der Anfangspunkt einer anderen Strecke ist, nennt man **Streckenzug**, d.h. aus n Punkten ergeben sich n-1 Strecken.



Wie aber sieht die interne Datenrepräsentation aus ?
Da wir nur die Punkte benötigten, liegt eine Liste von Punkten auf der Hand, im 2-dimensionalen Fall also `posn`:

> Eine Liste von Punkten, kurz ***plist***, ist entweder
>
> - `empty`
>   oder
> - `(cons p plist)`, wobei p ein `posn` ist

Sind keine bestimmten Punkte vorgegeben, können wir uns zufällig eine Liste erzeugen mit

```
;erzeuge-punktliste: n xmax ymax --> liste (von posns)
```

Die Parameter `xmax` und `ymax` entsprechen den Abmessungen der Szene.
Da eine stukturelle Rekursion über $\mathbb{N}$ vorliegt, die eine Liste erzeugt, ergibt sich als Gerüst:

```
(define (erzeuge-punktliste n xmax ymax)
  (cond
    ((equal? n 0) empty)
    (else
     ...))))
```

Die Zufallskoordinaten der Punkte ergeben sich aus `(+ (random xmax) 1)` bzw. `(+ (random ymax) 1)` , also ergibt sich:

```
(define (erzeuge-punktliste n xmax ymax)
  (cond
    ((equal? n 0) empty)
    (else
     (cons (make-posn (+ (random xmax) 1) (+ (random ymax) 1))
           (erzeuge-punktliste (- n 1) xmax ymax)))))
```

z.B. ergibt der Aufruf `(define plist (erzeuge-punktliste 5 300 200))`

```
plist -->
(list
   (make-posn 51 32) (make-posn 198 13) (make-posn 286 98) (make-posn 196 73)
   (make-posn 162 126))
```

Für die graphische Darstellung müssen wir folgendes beachten: zu einem Streckenzug gehören mindestens zwei Punkte, d.h. bei n Punkten werden $n-1$ Strecken, also

> Ein *Streckenzug* `szug`, ist entweder
>
> - `(cons p empty`, wobei p ein `posn` ist
>   oder
> - `(cons p szug)`, wobei p ein `posn` ist und `szug` ein Streckenzug

Der Funktionsvertrag lautet

```
;zeichne-streckenzug: plist --> szene
```

Da jede Strecke zwei Punkte der Liste benötigt, genügt es zur Terminierung den Fall zu betrachten, dass nur noch ein Punkt übrig ist; als Gerüst ergibt sich:

```
 ergibt sich ein Gerüst der Form
(define (zeichne-streckenzug zugliste)
  (cond
    ((empty? (rest zugliste))...)
    (else
     (...
      (zeichne-streckenzug ...)))))
```

Im Rumpf der Funktion muessen bei jedem Selbstaufruf der aktuell erste und der aktuell zweite Punkt der Liste miteinander verbunden werden und anschließend mit `(place-image ...)` der Ausgangsszene, anfänglich `(empy-scene b h)` hinzugefügt werden. Wenn nur noch ein Punkt übrig ist, wird er nicht mehr zum Zeichnen benötigt und passend zur Terminierung wird `(empy-scene b h)` zurückgegeben. Es ergibt sich das Gerüst

```
(define (zeichne-streckenzug zugliste)
  (cond
    ((empty? (rest zugliste)) (empty-scene b h))
    (else
     (place-image
      (...(first punktliste)...(first (rest punktliste))...
       (zeichne-streckenzug (rest zugliste)))))))
```

Zum eigentlichen Zeichnen der einzelnen Strecken bietet sich die vordefinierte Funktion
;add-line: image x y z u symbol –> image
an, wobei `image` eine schon vorhandene Grafik, x und y die Koordinaten des ersten, z und u die Koordinaten des zweiten Punktes und eine Farbe ist. Da diese Funktion nicht in eine Szene, sondern über eine schon vorhandene Graphik zeichnet, müssen wir eine solche vorher definieren, am besten so, dass sie `(empy-scene b h)` entspricht:
`(define hintergrund (rectangle b h 'outline 'white))`
Insgesamt ergibt sich:

```
;zeichne-streckenzug: punktliste --> szene
(define (zeichne-streckenzug punktliste)
  (cond
    ((empty? (rest punktliste)) (empty-scene b h))
    (else
     (place-image
      (add-line hintergrund
                (posn-x (first punktliste)) (posn-y (first punktliste))
                (posn-x (first (rest punktliste))) (posn-y (first (rest punktliste))) 'blue)
      0 0
      (zeichne-streckenzug (rest punktliste))))))
```

und wir erhalten eine ähnliche Graphik wie im obigen Bild.

Ein **Polygon** ist ein geschlossener Streckenzug, d.h. Anfangs- und Endpunkt sind miteinander verbunden: um aus einem Streckenzug ein Polygon zu machen, müssen wir die zusätzlich die Strecke vom ersten zum letzten Punkt zeichnen. Die benötige Funktion
; zeichne-polygon: punktliste –> szene
unterscheidet sich von nur dadurch, dass die Liste `(cons (letztes-element punktliste) punktliste)` durchlaufen wird:

```
;zeichne-polygon: liste --> szene
(define (zeichne-polygon punktliste)
  (zeichne-streckenzug (cons (letztes-element punktliste) punktliste)))
```

Es bleibt noch die Definition von
`;letztes-element liste -> element`
Wir erinnern uns:

```
(define (letztes-element punktliste)
  (cond
    ((empty? (rest liste)) (first punktliste))
    (else
     (letztes-element (rest punktliste)))))
```

Aufrufe von `zeichne-streckenzug` und `zeichne-polygon` mit

```
(define beispielliste (list (make-posn 179 137) (make-posn 21 45) (make-posn 281 16)
        (make-posn 123 147) (make-posn 132 58))
```

liefern jeweils

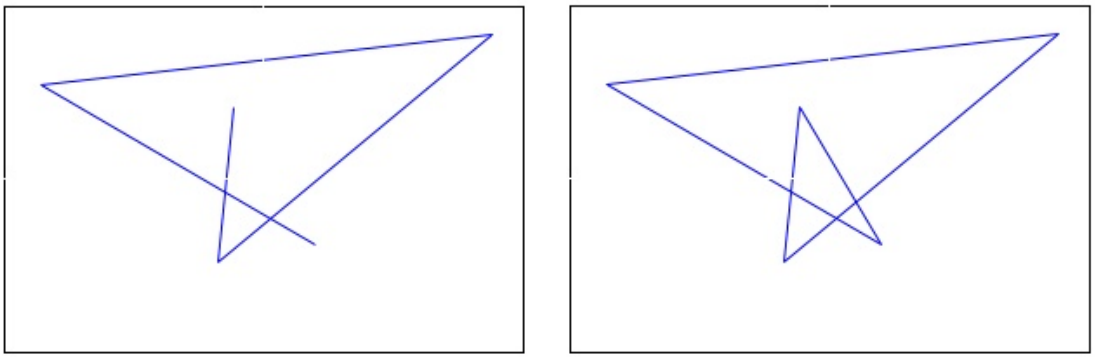


Abb. 4.2: Streckenzug und Polygon

Übungen

1. Ergänze die Funktionen `zeichne-streckenzug` und `zeichne-polygon` mit einem Parameter `farbe`

2. Erstelle folgende Funktionen
   a) *; länge-streckenzug: plist –> zahl*, die die Länge eines Streckenzugs (=Summe aller Teilstrecken) ermittelt
   b) *; abstände-streckenzug: plist –> liste*, die eine Liste der Längen der Strecken des Streckenzugs liefert
3. Erstelle eine Funktion *;polygon: anzahl –> szene*, die ein Zufalls-Polygon mit n Ecken zeichnet

Beispiel 2: Balkendiagramm

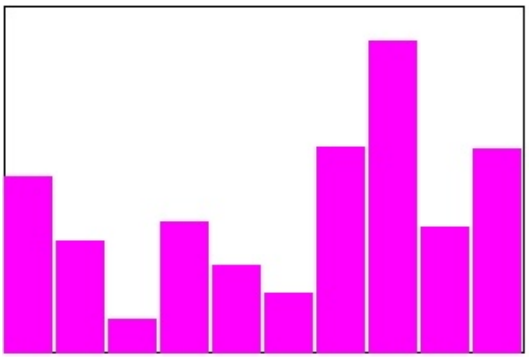
Häufig besteht das Bedürfnis, Zahlenwerte (technische Messwerte, Umsatzzahlen, Verteilungen usw.) in Form von Balkendiagrammen zu visualisieren.
Angenommen, wir haben die Zahlenliste

```
(define zliste (list 108 69 22 80 54 37 126 190 77 125))
```

und wollen sie mittels einer Funktion

```
;diagramm: liste farbe --> szene
```

in einem Balkendiagramm darstellen:



Bevor das Zeichnen der Balken umgesetzt wird, muessen einige Fragen geklärt werden, vor allem die grundsätzliche: werden die Abmessungen der Balken der vorgegebenen Breite und Höhe der Anfangszene `(empty-scene b h)` angepassst oder umgekehrt? Wir entscheiden uns für das erstere, dann bleiben die Fragen

1. Wieviel Balken sind zu zeichnen?
2. Wie groß ist der Balkenabstand?
3. Wie groß ist die Balkenbreite?
4. Wie wird die grafische Balkenhoehe bestimmt?
5. ...

Die zugehörigen Werte müssen vor dem eigentlichen Zeichnen bereitstehen, deshalb entwickeln wir für diesen Vorgang eine eigene Funktion

```
;balken-zeichnen: liste balkenbreite balkenabstand x0 maxhoehe szene farbe --> szene
```

die in nach den Vorbereitungen aufgerufen wird:

```
(define (balken-diagramm liste farbe)
  (local
      (... balkenzahl ...
       ... balkenabstand...
       ... balkenbreite...
       ... sz...
       ... maxhoehe...
       ... x0...
       )
    (balken-zeichnen liste balkenbreite balkenabstand x0 maxhoehe sz farbe)))
```

Die Anzahl der Balken ergibt sich leicht aus der Anzahl der Zahlenwerte, also `(length liste)` , für den Abstand kann man z.B. 2 Pixel nehmen, bei der Balkenbreite muss von `(/ b balkenzahl)` noch der Balkenabstand abgezogen wird, als Anfangsszene wird `(empty-scene b h)` genommen; auch muss festgelegt werden, in welchem Abstand vom linken Rand der erste Balken gezeichnet wird, z.B. x0 = 0. Eine Funktion *;maxwert: liste –> zahl*, die den größten Wert einer Zahlenliste liefert, ist bereits früher konstruiert worden.

```
(define (balken-diagramm liste farbe)
  (local
      ((define balkenzahl (length liste))
       (define balkenabstand 2)
       (define balkenbreite (- (/ b balkenzahl) balkenabstand))
       (define sz (empty-scene b h))
       (define maxhoehe (maxwert liste))
       (define x0 0)
       )
    (balken-zeichnen liste balkenbreite balkenabstand x0 maxhoehe sz farbe)))
```

Bei *balken-zeichnen* handelt es sich um strukturelle Rekursion: es wird die Zahlenliste durchlaufen, bei jedem Selbtaufruf wird zunächst der nächste Balken als `image` erzeugt, dann mit `place-image` an die berechnete Stelle in der bisherigen Szene positioniert, die durch vorherigen Selbstaufruf erzeugt wurde:

```
(define (balken-zeichnen liste bb ba lux mh szene farbe)
  (local
      <Funktion zur Erzeugung des neuen Balken als image aus bb, mh, h, hoehe und Farbe>
  (cond
    ((empty? liste) szene) ;leere Szene
    (else
     <Platziere neuen Balken>
      (balken-zeichnen (rest liste) .......)))))
```

Für die Erzeugung des nächsten Balkens als Rechteck konstruieren wir die Funktion

```
;figur: hoehe --> image
```

mit `hoehe` als einzigem Parameter, da sich bei jedem Selbstaufruf nur diese als nächster Zahlenwert der Liste ändert. Da diese an die Abmessungen der Szene angepasst werden muss, wird sie aus `(* hoehe (/ h mh))` (Proportionalität) errechnet.
Beim Zeichnen des aktuellen Balkens `(figur (first liste))` brauchen wir für `place-image` die notwendigen Koordinaten, die sich auf den Mittelpunkt der Figur beziehen: deshalb muss zum aktuellen linken unteren Rand `lux` mindestens die halbe Balkenbreite von `bb` addiert werden, damit der erste Balken komplett in der Szene liegt. Ebenso muss vom unteren Rand mindestens die halbe Bildhöhe `(/ (image-height (figur (first liste))) 2)` subtrahiert werden.
Beim Selbstauf von `balken-zeichnen` muss zusätzlich die x-Koordinate `lux` um die Summe von `bb` und `ba` nach rechts geschoben werden.
Insgesamt ergibt sich:

```
(define (balken-zeichnen liste bb ba lux mh szene farbe)
  (local
      ((define (figur hoehe) (rectangle bb (* hoehe (/ h mh)) 'solid farbe)))
  (cond
    ((empty? liste) szene)
    (else
     (place-image
      (figur (first liste))
      (+ lux (/ bb 2))
      (- h (/ (image-height (figur (first liste))) 2))
      (balken-zeichnen (rest liste) bb ba (+ lux bb ba) mh szene farbe))))))
```

Übungen

1. *Fraktale Bäume*
   „Fraktale besitzen strukturelle Selbstähnlichkeit, dergestalt, dass ein Teil des Fraktals häufig wie das

ganze aussieht", sie können - ausgehend von einem Grundmuster - schrittweise hergestellt werden. Zu dieser Beschreibung paßt folgendes Beispiel „Fraktal-Baum":

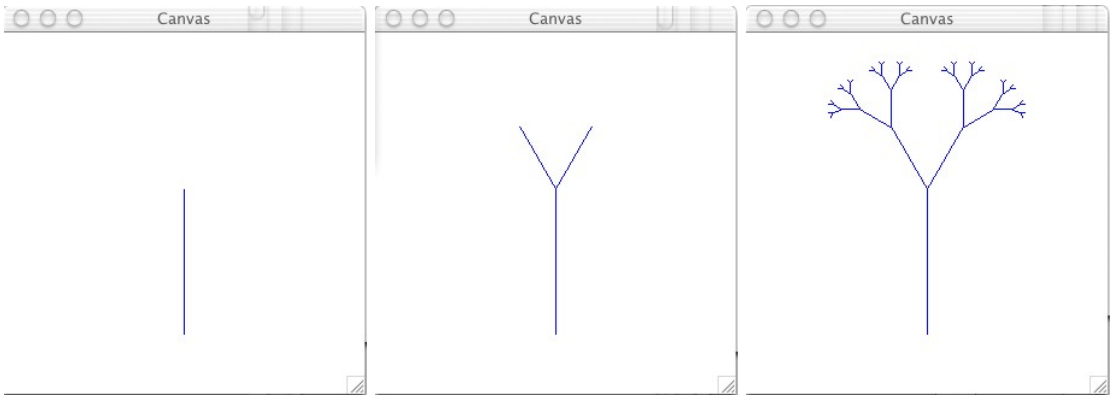


Abb. 4.3: Grundmuster, Zwischenphase und möglicher Endzustand

Eine Funktion, die diese verzweigte Figur zeichnet, muss den Zeichenprozess zwangsläufig spiegeln, also mit anderen Größen wiederholen:

- die Eingaben müssen das Grundmuster repräsentieren, mit dem gestartet wird
- der Prozess kann z.B. dadurch beendet werden, dass die wiederholten Muster zu klein werden

Die rekursive Struktur liegt auf der Hand:

Wenn er nicht zu klein ist,
Zeichne einen Stamm,
der einen linken und einen rechten Zweig hat, die kleinere Stämme darstellen,
die wir zeichnen (wenn sie nicht zu klein sind)
die jeweils wieder einen linken und einen rechten Zweig haben, die kleinere .....,
die wir zeichnen (wenn sie nicht zu klein sind),
die jeweils wieder einen linken und....., usw.

Hinzuzufügen ist, dass

- die Länge der Zweige hier halb so groß wie die des Stammes ist,
- die Zweige unter einem bestimmten Winkel zum Stamm einmal nach links und einmal nach rechts gezeichnet werden

Daher empfehlen sich *Polarkoordinaten* zur Darstellung der Anfangs- und Endpunkte der Zweige: Statt x und y werden als Koordinaten r und ϕ genommen, wobei die Transformationsgleichungen $x = r \cdot cos\phi$, $y = r \cdot sin\phi$ bzw. $\tan\phi = \frac{x}{y}$, $r = \sqrt{x^2 + y^2}$ lauten.

Entwickle eine passende Funktion mit dem Vertrag

```
;fraktal-baum: posn zahl zahl --> szene
```

Vorschlag für ein Funktionsgerüst:

```
; zeichne einen Stamm (als Strecke von A bis (r · cos winkel | y = r · sin winkel),
; vorausgesetzt,  r ist groß genug)
(define (fraktal-baum P r winkel)
   (cond
      ((zuklein? r) true)
      (else
         .......... (zeichne-ast P r winkel)........... )))
```

2. Verändere `fraktal-baum` so, dass die folgenden Figuren entstehen:

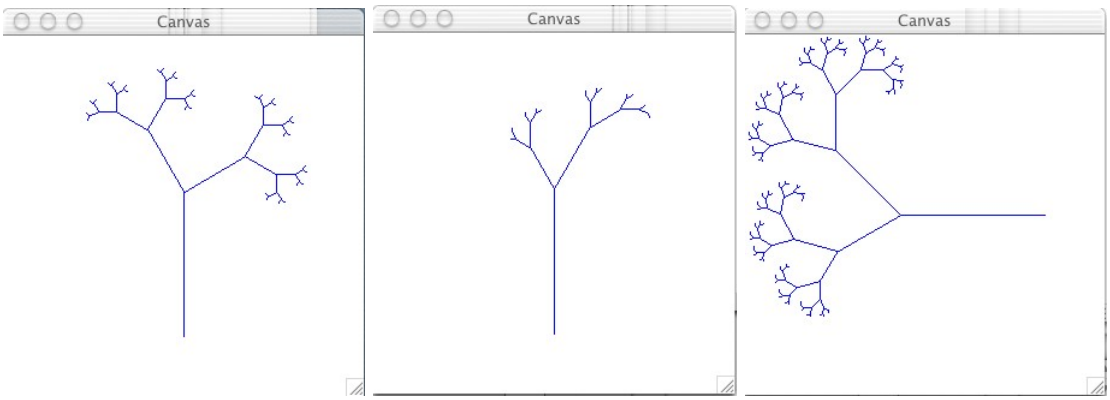


Abb. 4.4: Verschiedene Fraktalbäume

4.3.2 Bewegte Grafik: einfache Animation

Eine bequeme Möglichkeit, eine Grafik zu bewegen (ohne Interaktion durch Tastatur oder Maus), bietet die in `world.ss` eingebaute Funktion

```
;run-simulation: b h r f --> true
```

die folgendermaßen funktioniert:

- Es wird ein Fenster der Breite b und Höhe h geöffnet
- Es wird eine interne Uhr *clock* gestartet, die $\frac{1}{r}$-mal pro Sekunde tickt
- Die 1-parametrige Funktion f erzeugt bei jedem Tick eine neue Szene, wobei der Parameter von f die Werte 0, 1, 2, ... annimmt (eine Art „Endlosprogramm") und mittels f eine oder zwei Koordinaten von einer oder mehreren Grafiken steuert und die neue Szene im Grafikfenster darstellt.

Zur Erläuterung betrachten wir

Beispiel: Raketenlandung[2]

Wir legen zunächst

```
(define b 300)
(define h 200)
```

fest, was wir sowohl für die Abmessungen des Grafikfensters als auch der der Anfangsszene
`(define hintergrund (empty-scene 300 200)`
nehmen.
Für den Wert r legen 25 fest, da das menschliche Auge Bildwechsel von ca. mehr als 25 Bilder pro Sekunde nicht mehr diskret sondern als fließend wahrnimmt. Die Bewegungsfunktion nennen wir *bewege-rakete*, also

```
(run-simulation b h (/ 1 25) bewege-rakete)
```

Für eine Raketenlandung muss die y-Koordinate des Bildes in der Szene schrittweise verkleinert werden, also bietet sich der Paramter `hoehe` an, und wir erhalten als Vertrag:

```
;bewege-rakete: hoehe --> szene
```

Damit haben wir auch das Gerüst der Funktion:

```
(define (bewege-rakete hoehe)
  (place-image rakete ... hoehe ...))
```

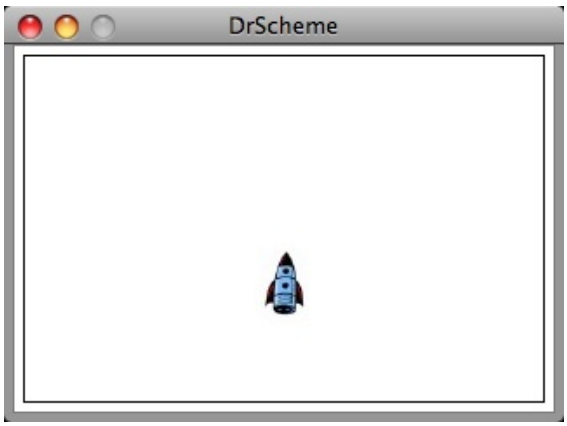
[2]Dieses und andere Beispiele:
http://www.ccs.neu.edu/home/matthias/HtDP/Worldv300/,
http://web.cs.wpi.edu/ kfisler/Courses/TeachScheme/World/flight-lander.html

Jetzt brauchen wir noch ein Raketebild, was wir z.B. im Internet als *jpg*-Datei finden:

`(define rakete`  `)`
Wenn die Rakete ab der Mitte des oberen Szenen- bzw. Fensterrandes sinken soll, wählen wir als x-Koordinate der Rakete `(/ b 2)` . Insgesamt erhalten wir:

```
(define (bewege-rakete hoehe)
  (place-image rakete (/ b 2) hoehe hintergrund))
```

Der Aufruf `(run-simulation b h (/ 1 25) bewege-rakete)` liefert:



4.3.3 Grafik mit Interaktion (Ereignissteuerung ohne GUI)

Grundgedanke ist hier eine sog. „Welt" (***world***), eine abstrakte Größe, in der hier alles, was während des Programmablaufs geändert werden soll, zusammengefasst ist. Dabei kann es sich um einen Wert irgendeines Datentyps handeln, im einfachsten Fall um eine Zahl, aber auch um eine *posn* oder *struct*. Dazu wird zunächst eine Umgebung mit einer Anfangs-Welt erzeugt durch die Funktion

- `(big-bang b h r anfangswelt)`

Dabei wird von *big-bang* folgendes veranlasst:

  * Es wird ein Fenster der Breite b und Höhe h geöffnet.
  * Es wird eine interne Uhr *clock* gestartet, die $\frac{1}{r}$-mal pro Sekunde tickt
  * Es wird eine Anfangswelt, die durch das Ticken der Uhr oder später durch Benutzer-Interaktionen geändert werden soll, bereitgestellt.

Interaktionen werden durch eine sog. **Ereignissteuerung** (*event handlling*) geregelt.
In `world.ss` gibt es drei Arten von Ereignissen, auf die reagiert werden kann:

- `(on-tick-event f1)`
  Zeit-Ereignis: Tick der inneren Uhr (*clock*)
- `(on-key-event f2)`
  Tastatur-Ereignis, z.B. welche Tast wurde gedrückt ?
- `(on-mouse-event f3)`
  Maus-Ereignis, z.B. Maus bewegt ? Maustaste gedrückt ? usw.

Dabei sind die selbst zu definierenden Funktion f_1, f_2 *und* f_3 sog. *Event-Handler*, die angeben, was als Reaktion auf das jeweilige Ereignis zu tun ist. Die möglichen Reaktionen sind i.a. Änderungen der Welt.
Damit nun die Änderungen der Welt als neue Szene sichtbar werden, wird bei jedem Ereignis die Funktion

- `(on-redraw f4)`

aufgerufen, wobei die selbst zu definierende Funktion f_4 als *Event-Handler* eine neue Szene in das Fenster zeichnet. Zur Beendigung des Programms dient

- `(stop-when f5)`,

wobei f_5 eine BOOLEsche Funktion ist.
Wir fassen zusammen:

Allgemeines Gerüst für Programme mit `world.ss`

```
;globale Daten
;---z.B. Anfangswelt u.a.-------
;Ereignis-Handler
;---------
;Hauptprogramm:
(big-bang b h 1/r anfangswelt)
(on-tick-event <Tick-Handler>)  ;ggfs.
(on-mouse-event <Maus-Handler>)  ;ggfs.
(on-key-event <Tastatur-Handler>) ;ggfs.
(on-redraw <NeueSzene) ;Visualisierung der Szenen im Fenster
(stop-when <Abbruch>) ;Vermeidung von Endlos-Programmen
```

Die Funktionsweise soll an folgenden Beispielen demonstriert werden:

Beispiel 1: Zeitsteuerung

Ersetzt man bei unserer obigen Raketenbewegung die Zeile den Aufruf `(run-simulation 100 100 (/ 1 28) erzeuge-raketen-szene)` durch

```
(define (zeit t)
  (+ t 1))
(big-bang b h (/ 1 25) 0)
(on-tick-event zeit)
(on-redraw bewege-rakete),
```

kann man die gleiche Beobachtung wie oben machen.
Anzumerken ist, dass dieses Beispiel so einfach ist, dass man noch nicht einmal ein *Welt* braucht, weshalb der vierte Parameter von `big-bang` einfach Null gesetzt ist.
Zusammenfassend erkennt man an diesem Beispiel, dass die Funktion *;run-simulation: b h r f –> true* lediglich eine Vereinfachung eines Sonderfalls des o.g. Mustergerüsts für *;big-bang: b h r w –> true* ist.

Beispiel 2: Zeit- und Tastatursteuerung

Eine Kugel erscheint zufällig an der Oberkante des Grafikfenster und fällt nach unten. Mittels der Pfeiltasten der Tastatur soll sie in einen Zielbereich gelenkt werden, bei Erfolg erscheint ein entsprechende Meldung:

Ein grundsätzliche Analyse der Aufgabenstellung ergibt, dass

1. eine (automatische) Zeitsteuerung, und zwar für das Fallen der Kugel, und
2. eine Tastatursteuerung

nötig sind, auf die entsprechend reagiert werden muss, etwa mit

```
(on-tick-event zeit-reaktion)
(on-key-event tasten-reaktion)
```

Abb. 4.5: Spiellablauf und „Endwelt"

Da nach jedem Ereignis – ein Tick der inneren Uhr oder ein Tastendruck – sich die Welt verändert und deshalb die Szene neu gezeichnet werden muss, benötigen wir einen „Event Handler", etwa *;neu-zeichnen: w -> szene* für `(redraw ...)`.
Beendet ist das Programm `(stop-when ...)` , wenn die Kugel den Zielbereich erreicht hat oder außerhalb die Unterkante des Grafikfensters bzw. der Szene erreicht hat, was durch eine BOOLEsche Funktion `(ende? w)` überprüft wird .
Somit lautet das Programmgerüst:

```
.....globale Daten.....
........für Welt...........
........für Visualisierung
.....Event-Handler.....
(big-bang b h (/ 1 25) anfangswelt)
(on-tick-event zeit-reaktion)
(on-key-event tasten-reaktion)
(on-redraw neuzeichnen)
(stop-when ende?)
```

Die „Welt" besteht aus den Größen, die sich beim Ablauf ändern, das ist hier der Zustand der Kugel, und zwar ihre Koordinaten x und y sowie eine BOOLEsche Variable, z.B. `geschafft?` , in der festgehalten wird, ob sie im Zielbereich gelandet ist oder nicht, also das Tripel

```
(define-struct welt (x y geschafft?))
```

Lässt man die Kugel in der linken Hälfte des oberen Randes der Szene fallen, erhält man als Anfangswelt:

```
(define anfangswelt (make-welt (+ 20 (random (/ b 2))) 0 false))
```

Die Anfangszene – hier `spielfeld` genannt –, also ohne Kugel, besteht aus zwei rechteckigen Blöcken und der Schrift „ZIEL", die in der unteren rechten Ecke platziert werden müssen (was hier nicht näher erläutert wird):

```
(define zielbreite 60)
(define block1 (rectangle 20 50 'solid 'magenta))
(define block2 (rectangle 20 50 'solid 'magenta))
(define ziel (text "ZIEL" 20 'green))
(define spielfeld
  (place-image block1
               (- (- b (image-width block2)) (image-width ziel) (/ (image-width block2) 2))
               (- h (/ (image-height block1) 2))
```

```
(place-image ziel
             (- (- b (image-width block2)) (image-width ziel))
             (- h (image-height ziel))
             (place-image block2
                          (- b (/ (image-width block2) 2))
                          (- h (/ (image-height block2) 2)) (empty-scene b h)))))
```

Zusätzlich benötigen wir eine Kugel und eine Erfolgsmeldung:

```
(define kugel (circle 10 'solid 'green))
(define gewinntext (text „Geschafft!" 20 'red))
```

Das Kernstück des Programmes bilden die Reaktionen auf die Ereignisse, die sog. *event handler*:

Zunächst muss auf das Ticken der inneren Uhr reagiert werden, da sich mit jedem Tick die y-Koordinate der Kugel ändert, zusätzlich muss bei jedem Tick geprüft werden, ob sich die Kugel schon im Zielbereich befindet, wenn ja, muss Weltkomponente `geschafft?` den Wert `true` erhalten.

Die Funktion *zeit-reaktion* macht aus der bisherigen Welt eine neue, um einen Tick jüngere Welt, in der die x-Koordinate Kugel und evtl. auch die Variable `geschafft?` geändert sind:

```
;zeit-reaktion: welt --> welt
```

So ergibt als Gerüst:

```
(define (zeit-reaktion w)
  (cond
    (< im Zielbereich? > --> <neue Welt mit geschafft? wird true>)
    (else
     (<neue Welt mit Weiterfalleln>))))
```

Insgesamt ergibt sich (ohne Erläuterung):

```
(define (zeit-reaktion w)
  (cond
    ((and
      (and
       (< (welt-x w) (- b (image-width block2)))
       (> (welt-x w) (- b (image-width block2) (image-width ziel))))
      (> (welt-y w) (- h (image-height block1))))
     (make-welt (welt-x w) (+ (welt-y w) 1) true))
    (else
     (make-welt (welt-x w) (+ (welt-y w) 1) (welt-geschafft? w)))))
```

Als nächstes braucht man eine Funktion

```
;tasten-reaktion: welt taste --> welt
```

deren Vertrag und damit die beiden Parameter vom Typ her vorgegeben sind, nämlich Welt und Taste. Die Funktion braucht nur auf die vier Pfeiltasten zu reagieren: es muss zwischen `'left 'right 'up 'down` unterschieden werden, bei allen anderen Tasten bleibt die Welt unverändert:

```
(define (tasten-reaktion w taste)
  (local
      ((define weite 5))
    (cond
      ((equal? taste 'right)
       (make-welt (+ (welt-x w) weite) (welt-y w) (welt-geschafft? w)))
      ((equal? taste 'left)
       (make-welt (- (welt-x w) weite) (welt-y w) (welt-geschafft? w)))
      ((equal? taste 'up)
       (make-welt (welt-x w) (- (welt-y w) weite) (welt-geschafft? w)))
      ((equal? taste 'down)
       (make-welt (welt-x w) (+ (welt-y w) weite) (welt-geschafft? w)))
      (else
       w))))
```

Der Wert der lokalen Variablen `weite` kann auch anders gewählt werden, man überlege, welchen Einfluss die Wahl auf das Spielgeschehen hat.
Die Veränderung der „Welt" durch die Ereignisse muss auch in den Szenen visualisiert werden. Eine zu `(redraw ...)` passende Funktion *neuzeichnen* muss den Vertrag

```
neuzeichnen: welt --> szene
```

haben.
Wenn die Kugel im Ziel ist, muss zusätzlich zur Kugel auch der Gewinntext angezeigt werden, andernfalls nur die neue Kugelposition:

```
(define (neuzeichnen w)
  (cond
    (<Kugel im Ziel ?> <dann Gewinntext und Kugel zeichnen>)
    (else
      <Nur Kugel zeichnen)))
```

Es bleibt noch, eine BOOLEsche Funktion, etwa namens *ende?* für `(stop-when...)` zu finden, die feststellt, ob das Spiel zu Ende ist, also mit dem Vertrag

```
;ende?: welt --> BOOLEsch
```

Dies ist offensichtlich dann der Fall, wenn die Kugel im Ziel ist oder außerhalb die Grundlinie erreicht, also:

```
(define (ende? w)
  (or
   (> (welt-y w) h) (welt-geschafft? w)))
```

Beispiel 3: Maussteuerung

„Schiff finden" als Spezialfall von „Schiffe versenken" ist ein einfaches Beispiel für Maussteuerung: Ein unsichtbares, zufälliges Schiff soll durch Mausklicken entdeckt und sichtbar gemacht werden:

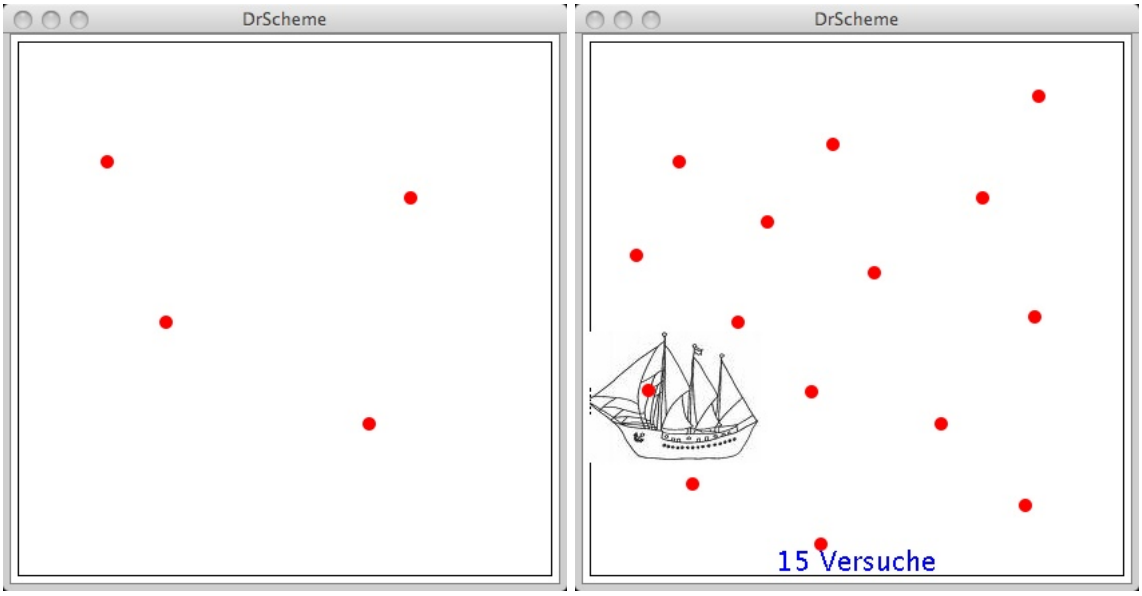


Abb. 4.6: *Schiff Finden:* Spielablauf und „Endwelt"

Die Programmentwicklung verläuft wie bei Beispiel 2. Wir haben lediglich Maussteuerung, also ergibt sich für das Programmgerüst:

```
(big-bang b h 0.5 anfangswelt)
(on-mouse-event mausaktion)
```

```
(on-redraw neuzeichnen)
(stop-when ende?)
```

Da wir keine Zeitsteuerung haben, spielt es keine Rolle, wie schnell die Uhr tickt, und wir können einen beliebigen Wert für die Zeitkonstante wählen, z.B. `0.5` .
Wie sieht hier die „Welt" aus?
Da der Benutzer sehen soll, wo er schon hingeklickt hat, müssen wir seine "Schüsse" sichtbar machen, etwa als farbige Kreise, und uns die zugehörigen getätigten Mausklicks merken, am einfachsten in einer Liste `schussliste` .
Da das Spiel beendet ist, wenn das Schiff getroffen, also gefunden wurde, benötigen wir eine BOOLEsche Variable `getroffen?` :

```
(define-struct welt (schussliste getroffen?))
(define anfangswelt (make-welt empty false))
```

Nach der Festlegung der Abmessungen für Fenster und Szenen

```
(define b 400)
(define h 400)
```

benötigen wir für die Visualisierung

- eine (leere) Anfangsszene `(define hintergrund (empty-scene b h))`
- ein Bild des Schiffes: `(define schiffbild`  `)`
- eine Markierung für die „Schüsse", z.B. `(define schuss (circle 5 'solid 'red))`

Während des Programmablaufs ist das unsichtbar, d.h. wir brauchen eine interne Repräsentation des Schiffes, also Rechteck-Struktur, die der Größe des Schiffes entspricht, um feststellen zu können, ob ein Mausklick innerhalb dieser Struktur stattgefunden hat oder nicht:

```
(define-struct schiff (kx ky))
```

Dabei sind `kx` und `ky` die Koordinaten des Mittelpunktes, der so zu wählen ist, dass das Schiff und seine interne Repräsentation `schiff-ort` ganz im Fenster liegen:

```
(define schiff-ort
  (local
      ((define x (random (- b (/ (image-width schiffbild) 2))))
       (define y (random (- h (/ (image-width schiffbild) 2)))))
    (make-schiff x y)))
```

Auf ein Maus-Ereignis wird mit einer in Anzahl und Typ der Parameter vorgegebenen Funktion reagiert, etwa mit

```
;mausaktion: w xm ym mause-ereignis --> true
```

wobei w die Welt, xm und ym die Koordinaten des Mausklicks und `maus-ereignis` die Art des Mausereignisses angeben (es gibt `'button-down 'button-up 'drag 'move 'enter 'leave`).
Im vorliegenden Fall muss geprüft werden,

1. ob ein Mausklick vorliegt oder nicht
2. wenn nein, bleibt die Welt wie sie ist
3. wenn ja, muss geprüft werden, ob der Mausklick innerhalb von `schiffort` liegt
   a) wenn ja, wird in der neuen Welt `getroffen?` wahr

   und in jedem Fall muss die Schussliste um einen Punkt erweitert werden

Daraus folgt das Programmgerüst

```
(define (mausaktion w xm ym mause-ereignis)
  (local ((define ein-punkt (make-posn xm ym))) ;Merke, wo Maus geklickt
    (cond
      ((equal? mause-ereignis 'button-up)
```

```
      (cond
        (...Treffer?...dann
         .... Merke Schuss in Liste .... getroffen: true .... )
        (else
         .... Merke Schuss in Liste .... getroffen: bleibt false ....))
      (else
       ...Welt bleibt unverändert...))))
```

Da ein Mausklick erst als vollendet gilt, wenn die Maustaste nicht nur gedrückt, sondern auch wieder losgelassen wurde, wird hier statt `'button-down` als Mausereignis `'button-up` gewählt; den zugehörigen Ort mit den Koordinaten xm und ym definieren wir als `posn` mit dem Namen `ein-punkt` im Hinblick auf die noch zu definierende Funktion *treffer?*, die feststellen soll, ob der Mausklick der Schiff getroffen hat:

```
;treffer?: schiff-record posn --> BOOLEsch
```

Es ergibt sich (ohne Erklärung —> Übungsaufgabe):

```
;treffer?: schiff-record posn --> BOOLEsch
(define (treffer? einSchiff punkt)
  (and
   (< (- (schiff-kx einSchiff) (/ (image-width schiffbild) 2))
      (posn-x punkt)
      (+ (schiff-kx einSchiff) (/ (image-width schiffbild) 2)))
   (< (- (schiff-ky einSchiff) (/ (image-height schiffbild) 2))
      (posn-y punkt)
      (+ (schiff-ky einSchiff) (/ (image-height schiffbild) 2)))))
```

Wenn das Schiff getroffen wird, dann erfolgen zwei Aktionen:

1. der Schuss wird zur Visualisierung und zum Zählen in die Schussliste aufgenommen
2. die Welt-Variable `getroffen?` wird `true`

Wenn geklickt, aber nicht getroffen wurde, wird nur der Punkt in die Schussliste aufgenommen: `(cons ein-punkt (welt-schussliste w))`
Wenn nicht geklickt wurde (andere Mausaktionen werden hier nicht abgefragt), bleibt die Welt unverändert, also `(make-welt (welt-schussliste w) (welt-getroffen? w))` , d.h. die neue Welt besteht aus den Werten der letzten Welt.
Insgesamt:

```
(define (mausaktion w xm ym mause-ereignis)
  (local ((define ein-punkt (make-posn xm ym)))
    (cond
      ((equal? mause-ereignis 'button-up)
       (cond
         ((treffer? schiff-ort ein-punkt)
          (make-welt (cons ein-punkt (welt-schussliste w)) true))
         (else
          (make-welt (cons ein-punkt (welt-schussliste w)) false))))
      (else
       (make-welt (welt-schussliste w) (welt-getroffen? w))))))
```

Da sonst keine Benutzer-Interaktionen abgefragt werden, kommen zu `(on-redraw...)` , d.h. wird müssen eine Funktion

```
;neuzeichnen: welt --> szene
```

konstruieren, die die veränderte Welt darstellt.
Dabei müssen zwei Fälle unterschieden werden, und zwar die Welt nach einem Treffer und die Welt nach einem Fehlschuss, also lautet das Gerüst:

```
(define (neuzeichnen w)
  (local
      (... einige Definitionen für übersichtlicheren Quellcode ...)
    (cond
      (... getroffen? ...
```

```
        ... Bisherige Schüsse zeigen ...
        ... Ergebnis (=Anzahl der Versuche) zeigen ...
        ... Schiff zeigen ... )
      (else
       ... Bisherige Schüsse zeigen ... )))
```

Die Fallunterscheidung erfolgt durch `(welt-getroffen? w)` ; wenn ja, muss die neue Szene, die zu zeichnen ist, aus drei Graphiken bestehen:

1. die Schussliste muss als Punkte, z.B. kleine rote Kreise, sichtbar gemacht werden
2. das Ergebnis, also die Zahl der Schüsse, wird als grafischer Text mit `(text ...)` angezeigt
3. das Bild des Schiffes

Für die Visualisierung der Schussliste konstruieren eine Funktion

```
;schussliste-zeichnen: liste szene --> szene
```

die die Punkte der Liste in eine Szene einzeichnet. Da wir eine Liste durchlaufen, liegt ein rekursives Gerüst nache:

```
(define (schussliste-zeichnen liste szene)
  (cond
    ((empty? liste) szene)
    (else
     ... Platziere ersten Schuss ...
     ... schussliste-zeichnen (rest liste) szene)))))
```

Mit `(define schuss (circle 5 'solid 'red))` ergibt sich insgesamt:

```
(define (schussliste-zeichnen liste szene)
  (cond
    ((empty? liste) szene)
    (else
     (place-image schuss (posn-x (first liste)) (posn-y (first liste))
                  (schussliste-zeichnen (rest liste) szene)))))
```

Wenn durch Schachtelung von *place-image* mehrere Grafiken in Szenen gezeichnet werden, ist die Reihenfolge dann wichtig, wenn sich die Grafiken überlagern: Damit die roten Schüsse als kleinste Objekte alle zu sehen sein sollen, muss die Schussliste als letztes hinzugefügt werden und das Schiff als größtes zuerst, also

```
(define (neuzeichnen w)
  (local
      ((define ergebnis
         (text (string-append (number->string (length (welt-schussliste w)))
                              " Versuche") 20 'blue))
       (define erg-l (image-width ergebnis))
       (define erg-h (image-height ergebnis)))
    (cond
      ((equal? (welt-getroffen? w) true)
       (schussliste-zeichnen (welt-schussliste w)
                             (place-image ergebnis
                                          (- (/ fenster-b 2)(/ erg-l 2))
                                          (- fenster-h erg-h)
                                          (place-image schiffbild
                                                       (schiff-kx schiff-ort)
                                                       (schiff-ky schiff-ort) sz))))
      (else
       (schussliste-zeichnen (welt-schussliste w) sz)))))
```

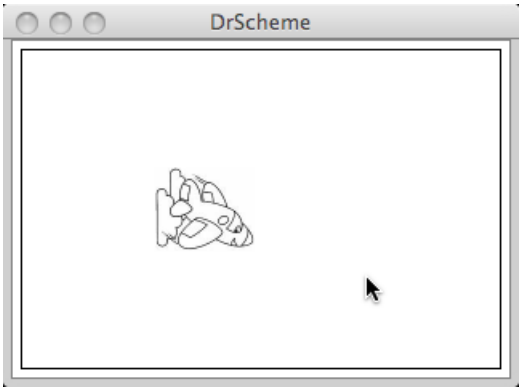
Für `(stop-when ...)` bleibt noch die Konstruktion einer BOOLEschen Funktion

```
;ende?: welt --> BOOLEsch)
```

übrig, was durch die Abfrage von `(welt-getroffen? w)` leicht möglich ist.

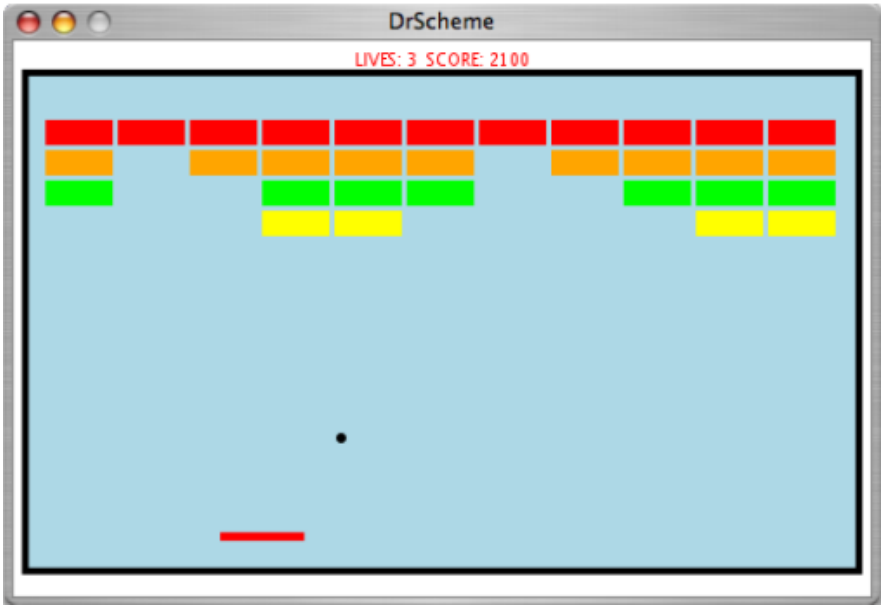
Übungen

1. Erstelle ein „Flieger"-Programm, bei dem ein Flieger (z.B. als .png oder .jpg-Grafik aus Google) automatisch waagerecht von links nach rechts bewegt wird



   a) die Bewegung wird durch ein Mausklick ins Fenster gestoppt
   b) Es genügt zum Stoppen nicht irgendein Mausklick in das Fenster, sondern der Flieger muss getroffen werden
   c) Erzeuge einen schrägen Flug von links oben nach rechts unten

2. Erstelle ein Breakout-Spiel:



5 Datenmodellierung

5.1 Komplexe Strukturen

5.1.1 Datentypen und Strukturen

Bis jetzt konnten wir alle Problemstellungen mit Hilfe der zusammengesetzten Datentypen Liste und Verbund und den atomaren Typen Zahl, BOOLEsch, Symbol sowie Zeichen und Zeichenkette beschreiben. Allerdings wird es in komplexeren Situationen schwierig, den Überblick zu behalten:

> *Angenommen, die an einer Schule beteiligten Personen sollen mit personenspezifischen Merkmalen verwaltet werden, etwa wie bei einem Datenbanksystem: Schüler, Lehrer, Sekretärinnen, Hausmeister etc. Bei einem Schüler interessiert z.B. seine Klassenzugehörigkeit und Religion, bei einem Lehrer seine Unterrichtsfächer und Klassenleitung, usw.*

Sollen wir nun die Schule - was die Schüler betrifft - als eine Liste von Klassen auffassen, wobei jede Klasse wiederum eine Liste von Schülern ist, in der wiederum jeder Schüler ein Verbund ist ? Oder als ein Verbund von Klassen, wobei jede Klasse wiederum eine Liste von Schülern ist, in der wiederum jeder Schüler ein Verbund ist ?
Wie wir schon festgestellt haben, hat die Art der Datenstruktur weitreichende Konsequenzen auf die Programmgestaltung, genauer die Funktionsmodellierung, um so mehr ist bei der Planung auf Sorgfalt zu achten.
Zu diesem verschaffen wir uns nochmal einen Überblick über die zu Verfügung stehenden Datentypen:

5.1.2 Modellierungsdiagramm

Um den Überblick nicht zu verlieren, wollen wir eine symbolische oder visuelle Beschreibungsmöglichkeit der Datenmodellierung einführen:
Die bisher bekannten Datentypen „atomar" (*Zahl, BOOLEsch, Symbol* und *Zeichen* (char)) und „zusammengesetzt" (Liste, Verbund, Zeichenkette) wollen wir dazu anders aufteilen, und zwar in „einfache" und „zusammengesetzte", indem wir die Zeichenketten (strings), obwohl sie technisch zusammengesetzte Daten sind, mit den atomaren Typen zusammen die „einfachen Typen" bilden lassen:

einfach

Zahl (Nummer/number **N**
BOOLEsch (**B**)
Zeichen (**C**)
Symbol (**S**)
Zeichenkette (string) (**Z**)

Für den Typ *Verbund* nehmen wir das *Verbunddiagramm* mit den o.a. Abkürzungen für die Komponenten einfachen Typs:

Verbund-Typ

Komp1 *Z*
Komp2
...

Für *Listen* benutzten wir das *Pfeildiagramm* und beschränken uns auf Listen mit Komponenten gleichen Typs:

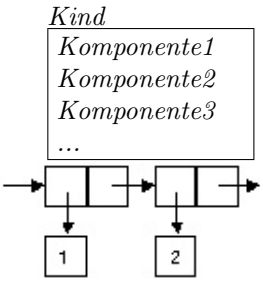
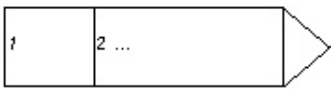
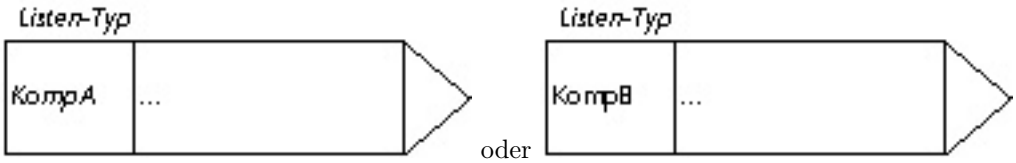
Typklasse allgemein	Typklasse hier	Typ	Visualisierung
atomar	einfach	Zahl `number?`	
atomar	einfach	BOOLEsch `boolean?`	
atomar	einfach	„Symbole" `symbol?`	
atomar	einfach	Zeichen `char?`	
zusammen gesetzt	einfach	Zeichenkette `string?`	
zusammen gesetzt	zusammen gesetzt	Verbund `struct?`	
zusammen gesetzt	zusammen gesetzt	Liste `list?`	Zeigernotation  Pfeilnotation
zusammen gesetzt	zusammen gesetzt	komplexer Typ = Liste(n) und Verbund(e) mehrfach ineinander verschachtelt	
zusammen gesetzt	zusammen gesetzt	Vektor (Reihung, array, Feld) `vector?`	

Tabelle 5.1: Überblick Datentypen



oder
wobei die Komponenten (Elemente) entweder

- alle zusammengesetzt (z.B. *KompA*s)
oder
- alle einfach (z.B. KompB) sind

Schreibweise:

kursiv: Typ-Name von *zusammengesesetzten* Daten

normal: Bezeichner/Name und Typ von *einfachen* Daten

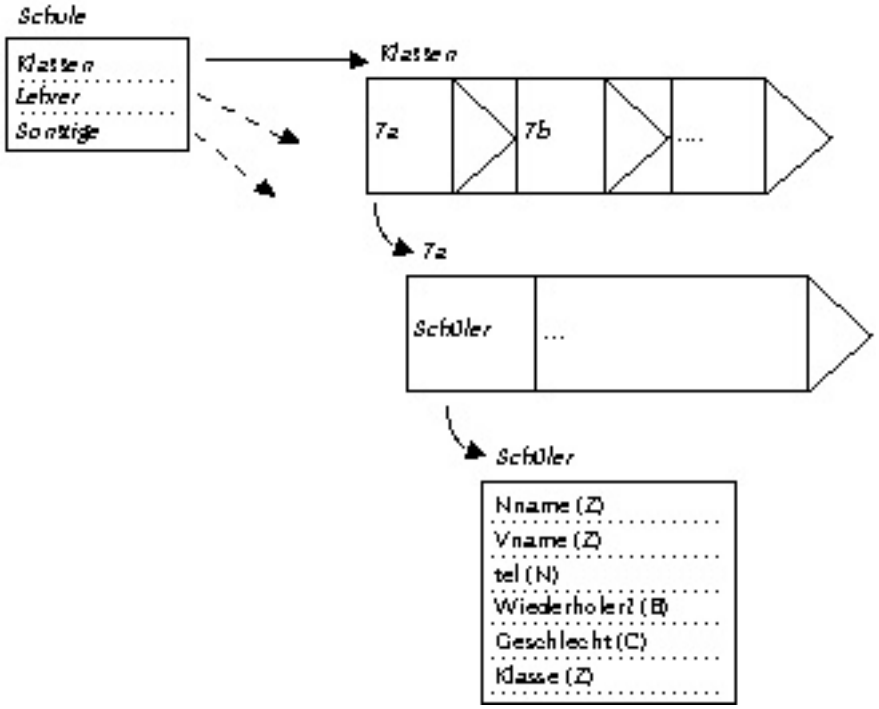
Darstellung:

ein zusammengesetzter Typ als Komponente wird durch ein Pfeil → in sein eigenes Symbol überführt

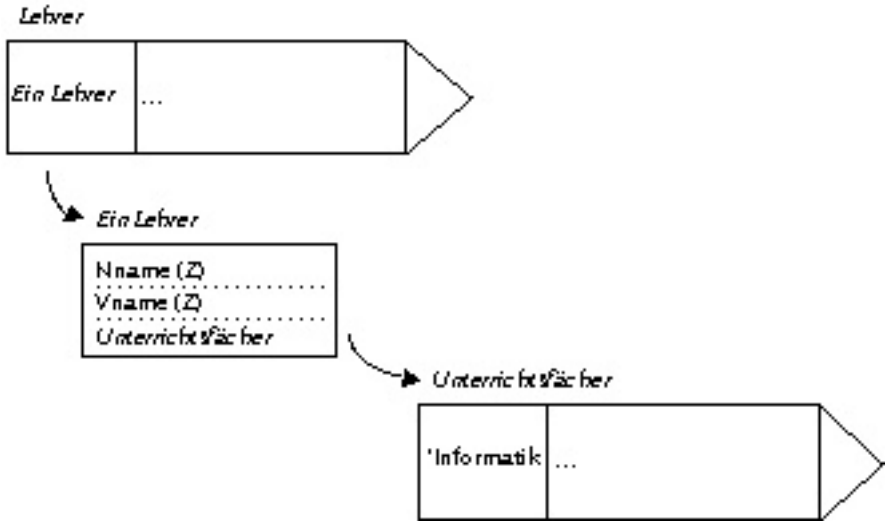
Strukturen, für die wir die Schachtelung beider Diagramme, also Verbunddiagramm und Pfeildiagramm benötigen, nennen wir **komplexe** Strukturen und das zugehörige (Gesamt-)Diagramm **Modellierungsdiagramm**

Beispiel 1: Schülerdatei

Jetzt können wir durch Kombination dieser Symbole komplexe Situationen übersichtlich strukturieren:



Wie können die Schule in diesem Modell als „Verbund von Listen von Listen von Verbunden" auffassen. Für die Lehrer könnte man sich etwa folgendes Modell vorstellen:



5.1.3 Modellierungsprinzip

Ein komplexes Datengeflecht können wir also schrittweise zerlegen, indem wir mit der obersten Ebene als Grobmodell in Form eines zusammengesetzten Datentyps (Verbund oder Liste) beginnen, dann die Komponenten entsprechend ihrem Typ mit Hilfe eines Pfeils als eigenes Symbol darstellen, usw. bis in einem Symbol nur noch einfache Datentypen auftauchen:

Ein **Modellierungsdiagramm** ist entweder

- leer, wenn nur einfache Datentypen vorliegen
  oder
- besteht aus Schachtelungen von Verbund- und/oder Listensymbolen, in deren letzten Symbolen nur noch einfache Datentypen vorkommen

oder, als Aktivität gesehen:

Ein **Datenmodellierung** ist beendet, wenn entweder

- wenn nur noch einfache Datentypen vorliegen
  oder
- die Komponenten von zusammengesetzten Datentypen sind solange in andere Datentypen zerlegt worden, bis nur noch einfache Datentypen vorkommen

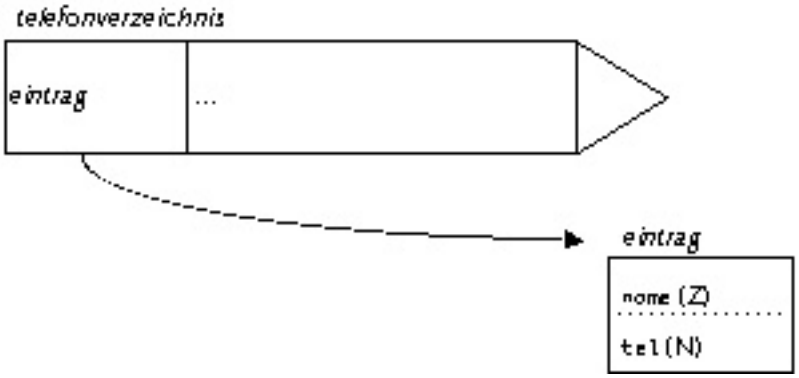
Beispiel 2: Telefonverzeichnis

Wenn wir ein Verzeichnis mit den Telefonnummern unserer Freunde anlegen wollen - vor allem, wenn unser Moblitelefon mal abhanden kommt -, ist uns schnell, dass es sich dabei um eine Ansammlung zusammengesetzter Daten handelt.
Wie können wir sie strukturieren, wie organisieren ?
Das Verzeichnis muß in der Länge flexibel sein, schließlich kommen gelegentlich Freunde dazu, manchmal werden auch Einträge gelöscht - vielleicht sogar irgendwann mal alle.
Also wählen wir eine Liste:

Ein **Telefonverzeichnis** ist entweder

- leer
  oder
- `(cons e tv)`, wobei `e` eintrag und `tv` ein Telefonverzeichnis ist

oder in symbolischer Form:



Pro Eintrag beschränken wir uns zunächst auf Nachnamen und Telefonnummer, also name und tel , was noch nicht ganz der Realität entspricht:

- Mehrdeutigkeit von Namen, da der Vorname fehlt
- jeder hat genau eine Telefonnummer
- da `tel` vom Typ Zahl ist, sind keine Klammern, Binde- oder Schrägstriche möglich

Insgesamt haben wir Datenmodell: eine Liste von Verbunden ! Also:

```
(define-struct eintrag (name tel))
```

Ein Verzeichnis können wir jetzt manuell erstellen:

```
(define ein-tverzeichnis (list
                          (make-eintrag "Lehmann" 4711)
                          (make-eintrag "Schneider" 1234)
                          (make-eintrag "Müller" 55772)))
```

Typische Anwendungen

```
; name->tel: Name --> Nummer     liefert Nummer zum Namen
   (define (name->tel tverzeichnis name)
      (cond ((empty? tverzeichnis) "Nicht vorhanden")
        ((equal? name (eintrag-name (first tverzeichnis))) (eintrag-tel (first tverzeichnis)))
        (else
         (name->tel (rest tverzeichnis) name))))

;; Neuer Eintrag
(append ein-tverzeichnis (list (make-eintrag "Schmidt" 66066)))
```

Verschiedene Anwendungen

```
;; alleNamen: Liste --> Liste     liefert Namensliste
(define (alleNamen tverzeichnis)
  (cond ((empty? tverzeichnis) empty)
        (else
         (cons (eintrag-name (first tverzeichnis)) (alleNamen (rest tverzeichnis))))))

;; name-oft: Liste  Name --> Zahl     liefert die Häufigkeit eines Namens
(define (name-oft tverzeichnis name)
  (cond ((empty? tverzeichnis) 0)
        ((equal? name (eintrag-name (first tverzeichnis)))
           (+ 1 (name-oft (rest tverzeichnis) name)))
        (else
         (name-oft (rest tverzeichnis) name))))

;; nummer-raus: Liste --> Liste     entfernt alle weiteren Personen mit gleicher Nummer
(define (nummer-raus tverzeichnis)
  (local ((define (ohne-mehrfache el liste)
      (cond
         ((empty? liste) liste)
```

```
        ((equal? el (eintrag-tel (first liste)))
           (ohne-mehrfache el (rest liste)))
        (else
            (cons (first liste) (ohne-mehrfache el (rest liste)))))))
(cond
   ((empty? tverzeichnis) empty)
   (else
       (cons (first tverzeichnis)
       (nummer-raus (ohne-mehrfache (eintrag-tel (first tverzeichnis))
          (rest tverzeichnis))))))))
```

Übungen

1. Entwickle folgende Funktionen:

   a) *alleNummern: Liste –> Liste* liefert eine Liste mit allen Telefonnummern
   b) *tel->name: Nummer –> Name* liefert Namen zur Nummer
   c) *tel-oft: Liste Zahl –> Zahl* liefert die Häufigkeit einer Nummer

2. Das Telefonverzeichnis könnte man aus als „Liste von Listen" modellieren, etwa:
   `(list (list "Lehmann" 4711) (list "Schneider" 1234) (list "Müller" 55772) ...)`

   Entwickle eine Funktion `als-Liste-von-Listen: Liste --> Liste`,
   die ein Verzeichnis der Form „Liste von Verbunden" in „Liste von Listen" umwandelt

3. Die Erstellung eines längeren Telefonverzeichnisses ist durch die Eingabe `(make-eintrag <name> <tel>)` für jeden einzelnen Eintrag mühevoll. Stattdessen kann man zum Testen der einzelnen Funktionen ein Verzeichnis mit „Zufallsnamen", also von Zeichenketten, die aus zufälligen Großbuchstaben A, .., Z ($65 \geq$ ASCII-Nr. ≥ 90) bestehen, und Zufallszahlen für die Telefonnummer erzeugen:

   ```
   (define anderes-tverzeichnis (tverzeichnis-machen 100)
   ;mit
   (define (tverzeichnis-machen anzahl)
           (cond ((zero? anzahl) empty)
             (else
           (cons (make-eintrag (zufalls-name 4 10) (zufalls-tel 1000 9999))
           (tverzeichnis-machen (- anzahl 1))))))
   ```

   Erstelle die Funktionen

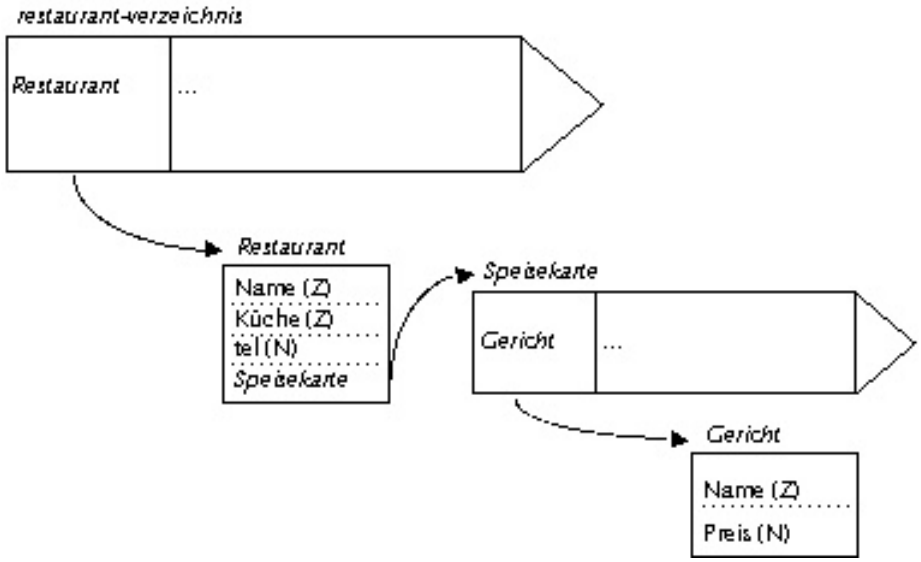
   a) *zufalls-name: Zahl1 Zahl2 –> Zeichenkette*, wobei Zahl1 die Mindestlänge (Buchstabenzahl) und Zahl2 die Maximallänge des Namens ist

   b) *zufalls-nummer: Zahl1 Zahl2 –> Zahl* , wobei gilt: Zahl1 $\leq$ Zahl $\leq$ Zahl2

   Hinweis: Verwende `integer->char` und `list->string`

4. Neben dem Nachnamen soll auch der Vorname in den Eintrag aufgenommen werden. Ändere alle Definitionen und Übungen entsprechend um.

Beispiel 3: Restaurants
Die Restaurants einer Stadt werden in einem Reiseführer durch ihren Namen, ihre Küche (z.B. „deutsch", „italienisch", „chinesisch", usw.), Tel-Nr. und Speisekarte beschrieben. Die jeweilige Speisekarte zählt die Gerichte des Restaurants auf, die wiederum jeweils durch Bezeichnung und Preis gekennzeichnet sind.
Eine Datenmodellierung könnte so aussehen:



Übungen zu Beispiel 3

1. Erläutere das Modellierungs-Diagramm
2. Begründe die Wahl der einzelnen zusammengeszten und einfachen Datentypen
3. Definiere
   a) einige Gerichte wie z.B. `(define G1 (make-Gericht "Rührei mit Spinat" 6.90)` usw.
   b) einige Speisekarten wie z.B. `(define L1 (list G1 G2 G3))` usw.
   c) ein Restaurant
   d) ein Restaurant-Verzeichnis
4. Entwickle folgende Funktionen:
   a) erstes-Gericht, die von einem Restaurant den Namen des ersten Gerichtes auf der Speisekarte ausgibt.
   b) `billig-Gerichte`, die zu einem Gericht und einem bestimmten Betrag den Wert `true` liefert, wenn das Gericht billiger sind als der o.g. Betrag.
   c) `alle-Preise`, die alle Preise einer Speisekarte in einer Liste ausgibt
   d) `Durchschnitts-Preis`, die den durchschnittlichen Preis aller Gerichte einer Gaststätte berechnet. Benutzt man dafür alle-Preise und die vordefinierten Funktionen `apply` und `length`, braucht man keine Hilfsfunktionen zu definieren.
5. Schlage eine andere Modellierung vor

5.2 Baumstrukturen

Eine Liste hat neben ihrer induktiven Natur auch eine lineare Struktur, zumindest dann, wenn ihre Elemente atomaren Typs sind. Daneben finden wir oft auch verzweigte Strukturen, sog. Baumstrukturen, wie z.B. ein Familienstammbaum; wie unterscheiden hier zwei Fälle:

- den aufsteigenden Stammbaum: Wer hat welche Eltern?
- den absteigenden Stammbaum: Wer hat welche Kinder?

Es liegt auf der Hand, dass die erste Form die einfachere ist, da jeder Mensch zwei Elternteile hat, wohingegen die Anzahl der Kinder bekanntlich stark schwankt.

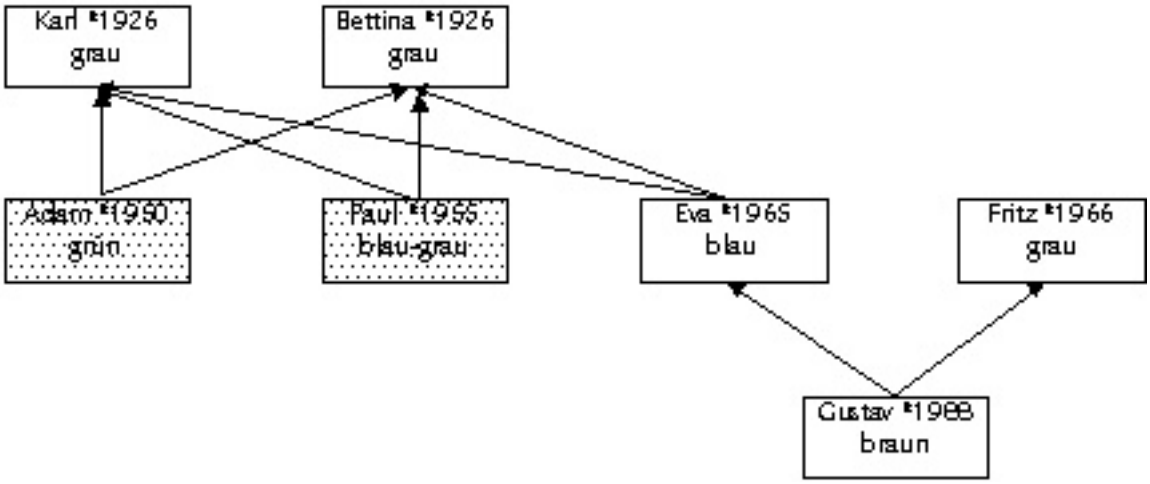


Abb. 5.1: Beispiel eines aufsteigenden Familienstammbaums

Ein solche Baumstruktur besteht allgemein aus sog. **Knoten** (pro Person) und **Kanten** (Verbindungspfeile, die auf die Eltern zeigen). In obigem Beispiel repräsentiert jeder Knoten eine Person, genauer gesagt ein Kind, wobei Name, Geburtsjahr und Augenfarbe dazugehören, aber auch die Angabe von Vater und Mutter; daher spricht man von einem *aufsteigenden Familienstammbaum.*

Dies legt folgende Datenstruktur nahe:

Kind

V
M
Name
GebJahr
AF

also in Scheme:

```
(define-struct kind (Vater Mutter Name Geburtsjahr Augenfarbe))
```

oder kürzer

```
(define-struct kind (V M Name GebJahr AF))
```

Also

```
(make-kind V M Name GebJahr AF),
```

wobei V und M wiederum von der Struktur kind sind, Name ein Symbol, GebJahr eine Zahl und AF ein Symbol.
Wir könne z.B. definieren

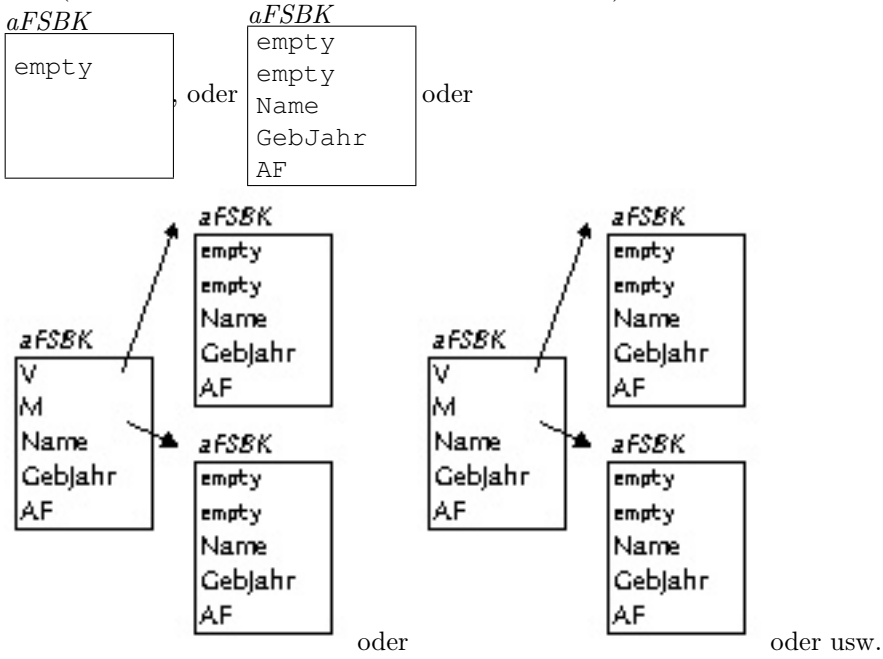
```
(define Adam (make-kind Karl Bettina 'Adam 1950 'grün)),
```

d.h. ein Knoten hat einen Namen (wie ein Variablen-Bezeichner) und enthält den gleichen Namen als Symbol.
Genau genommen stellt ein Knoten den *kompletten* (soweit die Daten bekannt sind) *Familienstammbaum* einer Person bzw. eines Kindes dar, da ja in ihm die Verweise auf die Knoten von Vater und Mutter enthalten sind, in denen wieder um die Verweise auf deren Eltern enthalten sind, usw. Ein Knoten beinhaltet somit nur die Abstammung einer Person, nicht aber sonstige Verwandte. In obigem Beispiel gehören daher Adam und Paul nicht zum persönlichen Familienstammbaum von

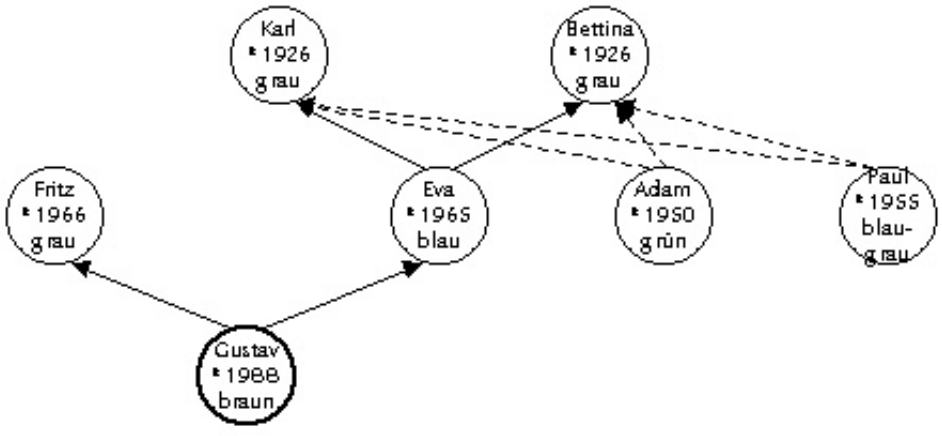
Gustav. Insgesamt modelllieren damit einen *aufsteigenden* Stammbaum, in dem abgefragt werden kann, wer wessen Vater bw. Mutter ist:

> Ein *Familien-Stammbaum-Knoten* oder kurz *aFSBK* ist entweder
>
> - `empty`
>   oder
> - `(make-kind V M GebJahr AF)`

Eine schrittweise Entwicklung eines Stammbaums sieht dann im Verbund-Schema folgendermaßen aus: (statt `aFSBK` könnten wir auch `kind` schreiben)



Drehen wir dieses Schema um 90°, erhalten wir den ursprünglichen Stammbaum (s.o.):



Beachte: *Adam* und *Paul* gehören streng genommen nicht zu diesem Stammbaum, da wir auf sie ausgehend von *Gustav* („Wurzel") nicht zugreifen können ! Siehe dazu Abschnitt *Baumstrukturen*.
Um mit obigem Beispiel arbeiten zu können, erstellen wir zunächst die Knoten:

```
;Älteste Generation
(define Karl (make-kind empty empty 'Karl 1926 'grau))
(define Bettina (make-kind empty empty 'Bettina 1926 'grau))
```

```
;Mittlere Generation
(define Adam (make-kind Karl Bettina 'Adam 1950 'grün))
(define Paul (make-kind Karl Bettina 'Paul 1955 'blau-grau))
(define Eva (make-kind Karl Bettina 'Eva 1965 'blau))
(define Fritz (make-kind empty empty 'Fritz 1966 'grau))

;Jüngste Generation
(define Gustav (make-kind Fritz Eva 'Gustav 1988 'braun))
```

Wir können jetzt den Stammbaum untersuchen:
Z.B. mit Hilfe der Selektoren nach Vorfahren fragen,
`(kind-V Eva) ==> (make-kind empty empty 'Karl 1926 'grau)`
oder nach dem Namen

```
(kind-Name (kind-V Eva))        ==>     'Karl   ;Vater von Eva
(kind-Name (kind-M (kind-M Gustav))) ==> 'Bettina  ;eine Oma von Gustav
```

oder mittels Funktionen:

Beispiel1

```
; mutter-name : aFSBK --> Name (symbol)
(define (mutter-name einkind)
  (cond
    ((empty? einkind) empty)
    ((empty? (kind-M einkind)) "unbekannt")
    (else
     (kind-Name (kind-M einkind)))))
```

mit

```
(mutter-name Paul) ==> 'Bettina
```

Beispiel2

```
; Ist ein blau-äugiger Vorfahre vorhanden ?
; BV? : aFSBK --> boolean
```

und erhalten

```
(define (BV? einFSBK)
  (cond
    ((empty? einFSBK) false)
    (else
     (cond
       ((symbol=? (kind-AF einFSBK) 'blau) true)
       ((BV? (kind-V einFSBK)) true)
       ((BV? (kind-M einFSBK)) true)
       (else false)))))
```

mit

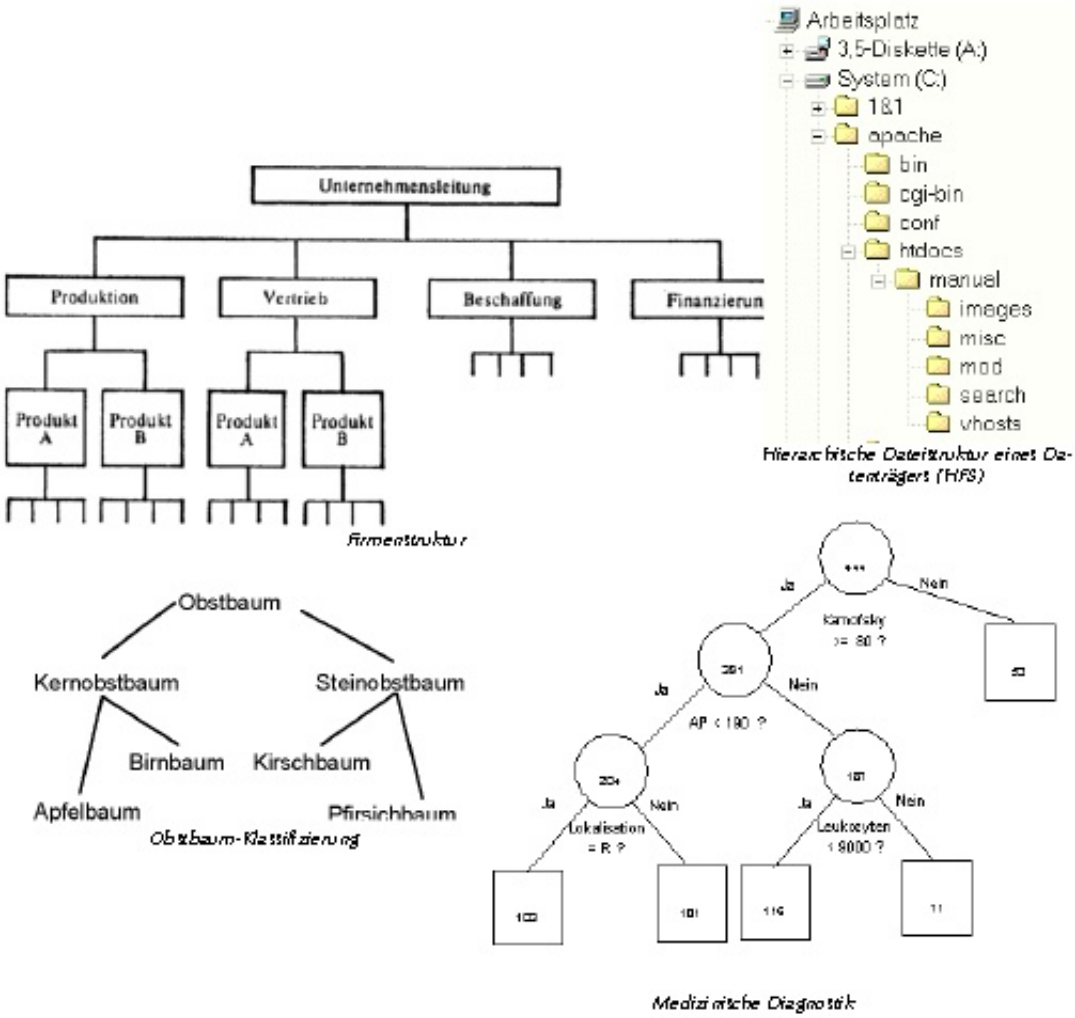
```
(BV? Eva)       ==>     true
(BV? Adam)      ==>     false
(BV? Fritz)     ==>     false
(BV? Gustav)    ==>     true
```

Übungen

1. Entwickle mit Hilfe von `mutter-name` eine Funktion
   `; omas: einkind (bzw. FSBK) --> liste`,
   die die Namen der Omas (falls vorhanden) auflistet.
2. Verallgemeinere `BV?` zu `VmAF?` (=„Vorfahre mit Augenfarbe") und führe den Paramater Augenfarbe ein: `(define (VmAF? Augenfarbe) ...)` und untersuche verschiedene Personen, also *FSBKs* daraufhin)

3. Erstelle eine Funktion anzahlPersonen: *FSBK –> Zah*l, die Anzahl der Personen eines *FSBKs* ermittelt
4. Erstelle eine Funktion *durschschnittsAlter: FSBK –> Zahl*, die das Durchschnittsalter der Personen eins *FSBKs* ermittelt
5. Erstelle eine Funktion `farbListe: FSBK -> liste`, die alle vorkommenden Augenfarben eines *FSBKs* als Liste ermittelt
6. Erstelle für Deine eigene Person einen Familienstammbaum und probiere die gefundenen Funktionen aus
7. Modelliere einen sog. absteigenden Stammbaum, in dem gefragt wird, wer ist wessen Kind.
   Mache Dir dabei den grundsätzlichen Unterschied zum aufsteigenden Stammbaum klar.

Die Struktur eines Familienstammbaums, nämlich eine gerichte, sich von einem Anfang aus wiederholt verzweigende Strukur, finden wir in vielen Bereichen:



Hierarchische Dateistruktur eines Datenträgers (HFS)

Firmenstruktur

Obstbaum-Klassifizierung

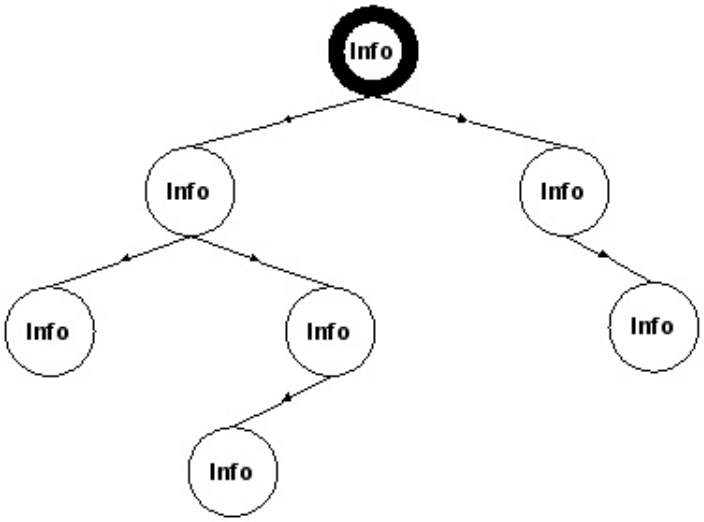
Medizinische Diagnostik

In der Informatik spricht man daher von einem **Baum**, wenn Daten eine Baumstruktur haben. Im Gegensatz zum realen Baum der Natur wird die Struktur Baum i.a. auf dem Kopf stehend graphisch dargestellt: die Wurzel ist oben, und die Verzweigungen finden nach unten statt. Wie wir an dem Beispiel der hierarchischen Dateistruktur sehen, geht es auch von links oben nach rechts unten.
Spätestens bei diesem Beispiel wird klar, daß jemand, der mit Software zu tun hat, die auf Datenträger zugreift, sich mit dem Modell der Baumstruktur beschäftigen sollte.
Die Verzweigungsstellen werden **Knoten** genannten, der Anfangsknoten **Wurzel** und die Endknoten **Blätter**.
Die Knoten enthalten neben den Verzweigungen eine Information, die durch den Baum hierarchisch

dargestellt werden soll (es gibt auch Bäumen, bei denen lediglich die Blätter Information enthalten). Eine Sonderform stellt der **binäre Baum** dar; eine solcher liegt vor, wenn die Knoten höchstens zwei Verzweigungen besitzen. Bei höchstens drei Verzweigungen hat man einen ternären Baum, usw.

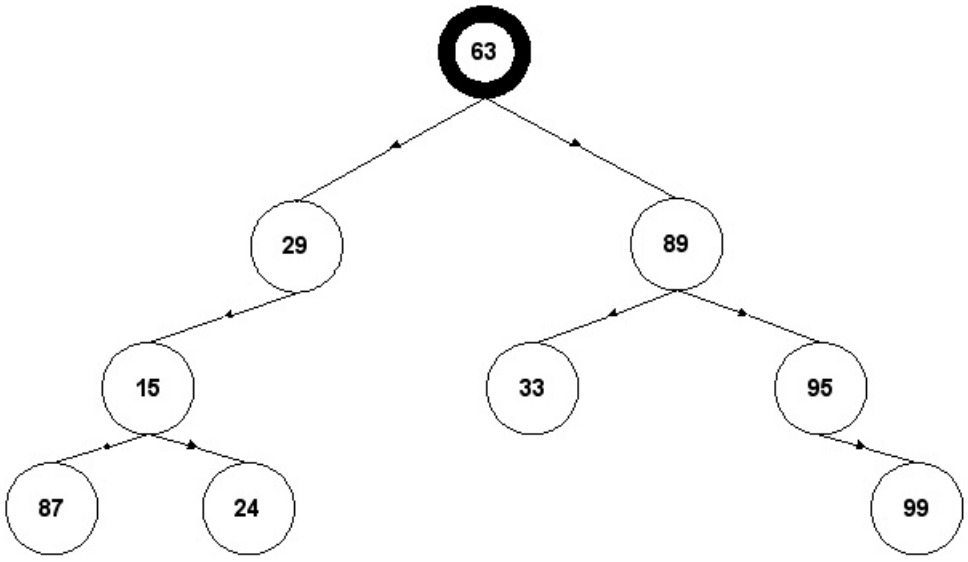
5.2.1 Binäre Bäume

Zur Visualisierung binärer Bäume wollen wir nebenstehendes Schema benutzen:



Datenrepräsentation

Wenn die Datenanalyse eines Problems zu einer Baumstruktur führt, stellt sich die Frage, mit welchem Datentyp wir den Baum repräsentieren können. Betrachten wir dazu einen einfachen Zahlenbaum:



Machen wir uns klar, daß

- jeder Knoten selbst wieder eine Wurzel eines (Teil-)Baums darstellt
- die fehlenden Verzweigungen eines Knotens als „leer" betrachtet werden können,

so liegt eine induktive Beschreibung nahe:

Ein **binärer Baum** ist entweder

- leer
  oder
- `(<info> lb rb)`, wobei `lb` und `rb` selbst binäre Bäume sind

Scheme hat keinen eigenen Datentyp für Bäume vorgesehen; es stehen uns lediglich die Liste und der Verbund als zusammengesetzte Datentypen zur Verfügung, dabei haben wir einen Familienstammbaum, der (in der absteigenden Form) ein binärer Baum ist, bereits mit verschachtelten Verbunden modelliert. Da Listen von Natur aus induktive Daten sind, liegt die Darstellung

Ein **binärer Zahlen-Baum**(`bzb`) ist entweder

- `empty`
  oder
- `(list <zahl> lb rb)`, wobei `lb`und `rb`selbst binäre Zahlen-Bäume sind

Abb. 5.2: binärer Zahlenbaum

nahe, was bei obigem Zahlenbaum zu

```
(define ein-bzb
  (list 63
            (list 29
                      (list 15
                                (list 87)
                                (list 24)))
            (list 89
                      (list 33)
                      (list 95
                                (list 99)))))
```

führt. Allerdings hat die Darstellung von Bäumen mit Listen auch Nachteile, u.a. weil

- bei komplexeren Bäumen die Bearbeitung umständlich wird, s.u. Beipiel „Kontoliste"
- das Prinzip der Kapselung nicht vorhanden ist

Daher ist einer Repräsentation durch Verbunde der Vorzug zu geben:
Mit `(define-struct bzb (zahl lb rb))`ergibt sich aus Abbildung 5.2 für obigen Zahlenbaum:

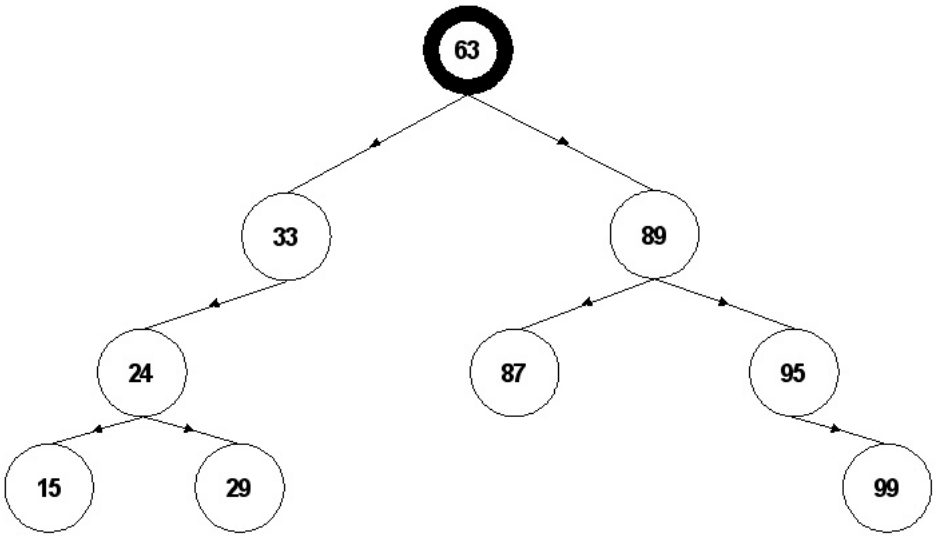
```
(define ein-bzb
  (make-bzb 63
            (make-bzb 29
                      (make-bzb 15
                                (make-bzb 87
                                          empty
                                          empty)
                                (make-bzb 24
                                          empty
                                          empty))
                      empty)
            (make-bzb 89
                      (make-bzb 33
                                empty
                                empty)
                      (make-bzb 95
                                empty
                                (make-bzb 99
```

```
                                    empty
                                    empty)))))
```

Es ist übrigens interessant zu sehen, wie schön im Editor die nebenstehende Baumstruktur deutlich wird, wenn nach jedem eine neue Zeile beginnt und anschließend „Neu einrücken" wählt.

Suchbäume

Wenn wir uns fragen, ob eine bestimmte Zahl in obigem Zahlenbaum enthalten ist, müssen wir ggfs. alle Knoten des Baumes durchforsten, im ungünstigsten Fall alle 9 Knoten.
Lägen allerdings die gleichen Zahlen in folgendem Baum vor, wären es maximal 4 Knoten !



Wieso ?
Wir sehen, daß die linke Verzweigung eine kleinere und die rechte Verzweigung eine größere Zahl als der Knoten selbst enthält, d.h. bei einer Suche nach einer Zahl brauchen wir bei einem Vergleich mit einem Knoteninhalt nur zu testen, ob – wenn sie nicht schon gleich ist – sie kleiner oder größer ist.
Einen solchen Baum nennt man einen **binären Suchbaum**:

Ein **binärer Such-Baum**(`bsb`) ist entweder

- `empty`
  oder
- `(make-bsb` <*info*> `lb rb)` , wobei
  - `lb` und `rb` selbst binäre Such-Bäume sind
  - alle <*info*> in `lb` kleiner sind als im Knoten selbst
  - alle <*info*> in `rb` größer sind als im Knoten selbst

wobei natürlich vorausgesetzt wird, daß die <*infos*> einer Ordnungsrelation unterliegen, wie z.B. Zahlen (andernfalls könnte keine Hierarchie unter den Daten (= <*infos*>) erzeugt werden.
Leider liegen in der Praxis die Informationen i.a. nicht in einem Suchbaum vor, sondern seriell (etwa in Listenform); daher muß daraus erst mal ein Baum erzeugt werden. Für obiges Beispiel definieren wir zunächst:

```
(define-struct bsb (zahl lb rb))
```

und erstellen einen leeren Baum:

```
(define mein-bsb empty)
```

Beispiel

Bei einer Großbank sind viele Tausend Kunden mit ihrer Kontonummer und dem zugehörigen Kontostand

gespeichert. Man möchte nun bei Eingabe der Kontonummer den Kontostand angezeigt bekommen. Da nun die Konten leider nur alphabetisch nach Namen geordnet sind, muß man im ungünstigsten Fall die eingegebene Kontonummer mit allen vorhandenen vergleichen, bis man die gewünschte findet.
Die Konten liegen also in einer Kontoliste der Form *(Konto-Nr1, Konto-Stand1), (Konto-Nr2, Konto-Stand2), ...* vor.
Zur Modellierung erzeugen wir mit

```
; kl: Zahl --> Liste von Listen (= Paaren aus Zufallszahlen)
(define (kl n)
        (cond
                ((zero? n) empty)
                (else
                (cons (list (random 50) (random 10000)) (kl (- n 1))))))
```

eine Kontoliste, z.B.

```
(list
(list 33 8053) (list 9 3887) (list 39 5975) (list 32 2850) (list 4 1511)
(list 30 3728) (list 35 4472) (list 25 5637) (list 49 6093) (list 16 1673)
(list 42 7345) (list 18 6609) (list 15 662) (list 45 7864) (list 28 3490)
(list 8 5613) (list 44 1018) (list 2 2983) (list 45 7884) (list 35 2699))
```

Mit einer Funktion k-stand

```
; k-stand: Zahl Liste --> Zahl (ermittelt aus Konto-Nr. den Kontostand)
(define (k-stand Nr kontoListe)
   (.....))
```

könnte man jetzt den Kontostand ermitteln, was bei vielen tausend Konten sicher aufwendig ist.
Gelänge es jetzt, diese Liste – die im Gegensatz zu obiger Liste für den Zahlenbaum – keinerlei Baumstruktur erkennen läßt, in einen binären Suchbaum umzuwandeln, wäre unser Problem gelöst.
Dazu definieren wir eine passende Baumstruktur

```
(define-struct bbaum (kn ks lb rb))
```

wobei `kn` = Konto-Nummer und `ks` = Konto-Stand sind.
Wir brauchen also eine Funtion:

```
;; BSB: Liste --> Baum
```

Dies werden wir in mehreren Schritten tun:

1. Zunächst entwickeln wir eine Funktion
   ```
   ; baum-insert: Baum Zahl1 Zahl2 --> Baum
   ```
   folgendermaßen:
   Angenommen, man hat schon einen binären Suchbaum – auch wenn er leer ist –, dann soll `baum-insert` ein neues Konto, bestehend aus Konto-Nr. (= Zahl1) und Konto-Stand (= Zahl2) entsprechend der Konto-Nr. an der richtigen Stelle im vorhandenen Baum einfügen (= insert)
2. Wir entwickeln eine Funktion
   ```
   ; mache-BSB: Baum Liste --> Baum
   ```
   mit der Eigenschaft, daß sie eine Kontoliste wie oben in einen binären Suchbaum – auch wenn er leer ist – mit Hilfe von `baum-insert` umwandelt

Zu 1 ergibt sich:

```
; baum-insert: Baum Zahl1 Zahl2 --> Baum
(define (baum-insert Baum xkn ykn)
        (cond
                ((empty? Baum)
                        (make-bbaum xkn ykn empty empty))
                ((< xkn (bbaum-kn Baum))
                        (make-bbaum
                                (bbaum-kn Baum)
                                (bbaum-ks Baum)
                                (baum-insert (bbaum-lb Baum) xkn ykn)
                                (bbaum-rb Baum)))
                (else
                        (make-bbaum
                                (bbaum-kn Baum)
```

```
                                (bbaum-ks Baum)
                                (bbaum-lb Baum)
                                (baum-insert (bbaum-rb Baum) xkn ykn)))))
```

Zu (2) ergibt sich:

```
; mache-BSB: Baum Liste-->Baum
(define (mache-BSB hBaum liste)
  (cond
    ((empty? liste) hBaum)
    (else
     (mache-BSB
      (baum-insert hBaum
                   (first (first liste)) (first (rest (first liste))))
      (rest liste)))))
```

Um das Ganze zu Testen, benötigen wir zunächst

- eine Kontoliste: `(define einekl (kl 20))`
- einen (leeren) Anfangsbaum: `(define einBaum empty)`

Damit könnten wir jetzt (mache-BSB einBaum einekl)) aufrufen und den Baum im Interaktonsfenster betrachten, bzw. ihm mit

```
(define andererBaum (mache-BSB einBaum einekl))
```

einen Namen geben.

Um nun den Suchaufwand zu vergleichen, können wir
entweder

1. eine Funktion *; suche-im-Baum: Baum Zahl –> Zahl* erstellen, und dann mit
   `(time (suche-im-Baum andererBaum <Konto-Nr>)`
   den Zeitaufwand messen,
   oder

2. die Funktion

   ```
   ;inorder: Baum-->Liste mit
   (define (inorder Baum)
     (cond
       ((empty? Baum) empty)
       (else
        (append
         (inorder (bbaum-lb Baum))
         (list (list (bbaum-kn Baum) (bbaum-ks Baum)))
         (inorder (bbaum-rb Baum))))))
   ```

benutzen, die den Baum von links nach rechts (= inorder (infix)) in eine (geordnete) Liste umwandelt, und mittels (define anderekl (inorder andererBaum)) und (time (k-stand Nr anderekl)) ebenfalls die Zeit ermitteln.
Die nach 1. oder 2. ermittelte Zeit

– für das Suchen in einer ungeordneten Liste (oder Baum)

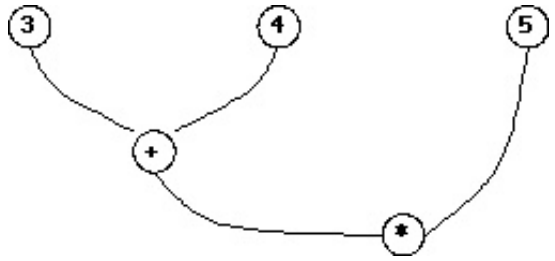
können wir schließlich mit dem Aufwand

– für das Suchen in einer geordneten Liste (oder Baum)

vergleichen, in dem wir `(time (k-stand Nr einekl))` aufrufen.

5.2.2 Rechenbäume

Eine besondere Baumform sind die uns bekannten Rechenbäume, mit denen Rechenterme dargestellt werden können:



1. $(3 + 4)5$ als infix
2. `(* (+ 3 4) 5)` als präfix

Für eine Datenmodellierung – sei es nach (1) oder (2) – beschränken wir uns zunächst der Einfachheit halber auf

- 2 Operanden pro Operator („binärer" Rechenbaum)
- die Grundrechenarten („+", „–", „*" und „/")

So unterscheiden sich Rechenbäume von Zahlenbäumen dadurch, dass die Knoten nicht nur Zahlen, sondern auch Operatoren enthalten, und zwar

- Zahlen, wenn die Knoten Blätter sind
- Operatoren, wenn die Knoten keine Blätter sind

Wenn wir also z.B. nach (1)- infix - einen Rechenbaum durch

```
(define-struct rbaum (lb op rb))
```

festlegen, müssen wir sicherstellen,

- op ist dann ein Operator, wenn lb und rb Zahlen sind
- op ist dann eine Zahl, wenn lb und rb `empty` sind

Daher definieren wir zusätzlich

```
(define-struct zahl (z))
(define-struct operator (o))
```

wobei o vom einfachen Typ symbol, d.h. '+, '-, '* oder '/ sein soll.
Jetzt können wir obigen Baum mit

```
(make-rbaum
 (make-rbaum (make-zahl 3) (make-operator '+) (make-zahl 4))
 (make-operator '*)
 (make-zahl 5))
```

erzeugen.
Allerdings ist hierbei die Eingabe durch das häufige Eintippen von `(make-...` mühevoll, wogegen eine Eingabe von

```
(list (list 3 '+ 4) '* 5)
```

einfacher ist.
Daher definieren wir die Funktion

```
; liste-->verbund: Liste-Baum --> Verbund-Baum
```

die die Infix-Listen-Dartellung des Baumes in die zuerst erwähnte Infix-Verbund-Darstellung des umwandelt; hierbei benötigen wir neben `number?` und `list?` noch die Prädikatorfunktion

```
; rz?: element symbol-liste --> BOOLEsch
```

um zu prüfen, ob eine Listenelement außer einer Zahl oder einer (Teil-)liste auch ein Rechenzeichen aus der Liste

```
(define symbole (list '+ '- '* '/))
```

ist; also

```
; rz?: element symbol-liste --> BOOLEsch
(define (rz? symbol symbol-liste)
  (cond
    ((empty? symbol-liste) false)
    (else
     (or (equal? symbol (first symbol-liste))
```

Jetzt können wir *liste–>verbund* wie folgt definieren:

```
; liste-->verbund: Liste-Baum --> Verbund-Baum
(define (liste-->verbund liste)
  (cond
    ((number? liste) (make-zahl liste))
    ((rz? exp symbole) (make-operator liste))
    ((list? liste) (make-rbaum
                    (liste-->verbund (first liste))
                    (liste-->verbund (first (rest liste)))
                    (liste-->verbund (first (rest (rest liste))))))
    (else
     false)))
```

Definieren wir noch
`(define Baum1 (list (list 3 '+ 4) '* 5))`,
dann liefert

```
(liste-->verbund Baum1)
==>
(make-rbaum
 (make-rbaum (make-zahl 3) (make-operator '+) (make-zahl 4))
 (make-operator '*)
 (make-zahl 5))
```

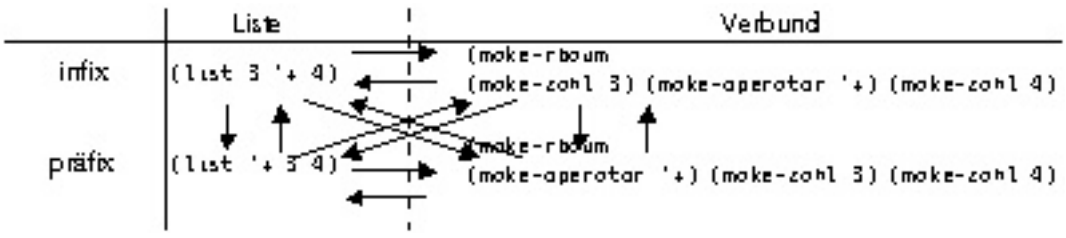
das gewünschte Ergebnis.
Wie sieht nun die Umkehrung aus ?

```
;verbund-->liste: Liste-Baum --> Verbund-Baum
(define (verbund->liste baum)
  (cond ((operator? baum) (operator-o baum))
        ((zahl? baum) (zahl-z baum))
        ((rbaum? baum)
         (list  (verbund->liste (rbaum-lb baum))
                (verbund->liste (rbaum-op baum))
                (verbund->liste (rbaum-rb baum))))
        (else false)))
```

Im bisherigen sind die Rechenbäume jeweils in der infix-Form wiedergegeben, aber wie können wir können wir einen Baum

von der infix-Liste in	vom präfix-Verbund in
- eine präfix-Liste	- eine präfix-Liste
- einen präfix-Verbund	- eine infix-Liste
	- einen infix-Verbund

transformieren ?
Dazu eine tabellarische Übersicht über die 12 möglichen Fälle:



Übungen

1. Entwickle zu jedem der möglichen Übergänge aus obiger Tabelle eine passende Funktion !

5.3 Graphen

5.3.1 Beispiele und Modelle

Reisende im Raum Stuttgart könnten sich beim Betrachten des Diagramms

Abb. 5.3: S-Bahn-Plan (Ausschnitt) von Stuttgart

die Frage stellen

> *Wie komme ich von Marbach nach Vaihingen ?*

oder allgemein

> *Wie komme ich von A nach B ?*

Oder genauer:

- ▷ Gibt es überhaupt einen Weg ?
- ▷ Wenn ja, gibt es mehrere ?
- ▷ Welcher ist der kürzeste ?
- ▷ ...

Dass die oben gestellten Fragen berechtigt sind, liegt offenbar am Weg bzw. an den Wegen von A nach B, genauer an der *Struktur* der Wege: sie bestehen aus

- ◦ *Abschnitten* - von jetzt ab **Kanten** genannt -, an deren

◦ *Schnittstellen* - von jetzt ab **Knoten** genannt - auch Verzweigungen möglich sind

*Eine Struktur aus **Kanten** und **Knoten** heißt **Graph***

Ein Graph ist für viele reale Situationen ein passendes **Modell**

Machen wir uns klar:

▷ die Wahl des Modells hängt von der Fragestellung ab
▷ das Modell vernachlässigt alle für die Fragestellung bzw. Problemlösung bedeutungslosen Aspekte

Dies können wir am folgenden Beispiel sehen:

⇒

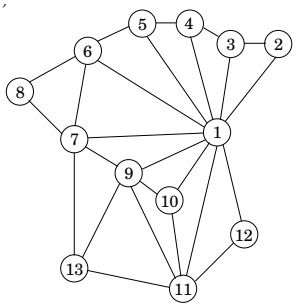


Abb. 5.4: Südamerika: Graph als Modell für mögliche länderverbindende Flugrouten

Offensichtlich hilft dieses Modell bei Fragen wie

▷ Von welchem Land nach welchem anderen Land gibt es eine direkte Flugverbindung ?
▷ Wie komme ich von Uruguay nach Venezuela ?
▷ Wo muss ich umsteigen ?
▷ ...

Dazu braucht man nur die Ländernamen (hier durch Nummern vereinfacht) als Knoten, und die Kanten geben die Verbindungen an, sofern es eine gibt.
Für einen Biologen oder Meterologen dagegen wäre dieses Modell von Südamerika völlig uninteressant.

Man kann Graphen (orthogonal) einteilen in verschiedene Typen:

- *gerichtet*, wenn die Kanten zwischen zwei Knoten nur in eine Richtung durchlaufen werden können
- *gewichtet*, wenn die Länge der Kanten eine Rolle spielt
- *zusammenhängend* (selbsterklärend)
- *zyklisch*, wenn es für einen Knoten (in einem gerichteten Baum) einen Weg über andere Knoten zu sich zurück gibt
- ...

Die gängige *symbolische* Darstellung ist das *Diagramm:*
Dabei werden

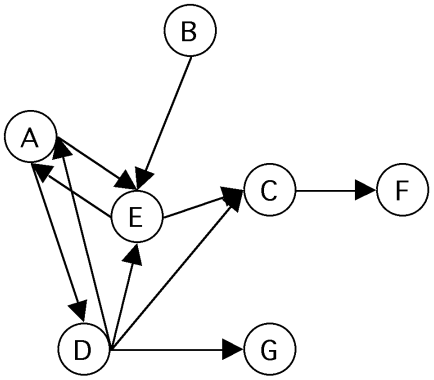


Abb. 5.5: Muster-Graph G_0: *gerichtet, nicht gewichtet, zusammenhängend*

- die *Knoten* durch *Kreise mit Namen* repräsentiert
- die *Kanten* durch *Pfeile* zwischen den Kanten repräsentiert

Da nach Definition ein Graph G aus einer Mengen von Knoten, im folgenden mit V bezeichnet, und aus einer Mengen von Kanten, im folgenden mit E bzeichnet, besteht, können bei unserem Mustergraph G_0 (*Abb. 5.5*) sagen:

$$\begin{aligned} G_0 &= (V, E) \text{ mit} \\ V &= \{A, B, C, D, E, F, G\} \text{ und} \\ E &= \{(A,D), (A,E), (B,E), (C,F), (D,A), (D,C), (D,E), (D,G), (E,A), (E,C)\} \end{aligned}$$

In der formalen Definition ist ein Graph ein Paar, bestehend aus der Knotenmenge V und der Kantenmenge E, also

```
(define-struct graph (knoten kanten))
```

Bei Mengen liegt eine Modellierung als *Listen* nahe, und es ergibt sich für den den Mustergraphen G_0

```
(define g0
  (make-graph
   (list ... <Knoten> ...)
   (list ... <Kanten> ...))
```

Während die Knoten als Symbole `'A 'B ...` repräsentiert werden können, handelt sich bei den Kanten um geordnete *Paare*. Demnach bietet sich für eine Kante der Datentyp `record` an, d.h. wir können

```
(define-struct kante (ak ek))
```

definieren.
Somit ergibt sich für unseren Mustergraphen G_0:

```
(define g0
  (make-graph
   (list 'A 'B 'C 'D 'E 'F 'G)
   (list
    (make-kante 'A 'D) (make-kante 'A 'E) (make-kante 'B 'E)
    (make-kante 'C 'F) (make-kante 'D 'A) (make-kante 'D 'C)
    (make-kante 'D 'E) (make-kante 'D 'G) (make-kante 'E 'A)
    (make-kante 'E 'C))))
```

Alternativ kann man z.B. auch Einzellisten mit den Knoten und ihren Nachfolgern bilden.

Übungen

1. Vergleiche

```
(define g0-alternativ
  (make-graph
   (list 'A 'B 'C 'D 'E 'F 'G)
   (list (list 'A 'D) (list 'A 'E) (list 'B 'E) (list 'C 'F) (list 'D 'A)
         (list 'D 'C) (list 'D 'E) (list 'D 'G) (list 'E 'A) (list 'E 'C))))
```

   mit `g0`.
   a) Entwickle eine Funktion

      ;wandle-graf: graph (Kanten als Verbunde) --> graf (Kanten als Listen)

      die `g0` in `g0-alternativ` umwandelt.
   b) Entwickle die Umkehrfunktion zu `mache-graf`
2. Finde weitere Datenmodelle
3. Überlege, wovon die Wahl des Datenmodells abhängt
4. Bei einem zusammenhängenden Graph enthält E alle Informationen. Um zusätzlich eine Liste aller Knoten V zu erhalten, ist eine Funktion

   ;hole-knoten: graph --> liste

   zu entwickeln.

5.3.2 Erreichbarkeit und Wege

Bevor wir uns dem zentralen Problem

Wie komme ich von A nach B ?

widmen, gehen wir der grundsätzlichen Frage

Kann man vom Knoten A überhaupt Knoten B erreichen ?

nach.
Also suchen wir eine Funktion

;erreichbar?: anfangsknoten zielknoten graph --> BOOLEsch

und erhalten als Gerüst

```
(define (erreichbar? anfangsknoten zielknoten graph)
        ...)
```

Für `g0` erwarten wir z.B.

```
(erreichbar? 'A 'B g0) --> false
(erreichbar? 'A 'E g0) --> true
(erreichbar? 'A 'F g0) --> true
```

Dafür soll im Folgenden anhand des Mustergraphen G_0 eine Lösung erarbeitet werden.

Neben dem trivialen Fall, wenn der Zielknoten der Anfangsknoten ist `((equal? k z) true)` haben wir den einfachen Fall, dass der Zielknoten B ein direkter Nachfolger des Anfangsknotens ist, also z.B. ist E direkter Nachfolger von A, d.h. über eine einzige Kante zu erreichen. Dann sind wir schnell fertig, wenn nicht, müßten wir uns die Nachfolger der Nachfolger anschauen, usw.
Deshalb liegt es nahe, sich die Nachfolger anzuschauen. Dafür führen wir folgenden Begriff ein:

> Die Menge aller Nachfolger eines Knotens $k \in V$, also alle Knoten, die über eine Kante direkt von k erreichbar sind, heißt **Rand** des *Knotens k*

Wir benötigen eine Funktion

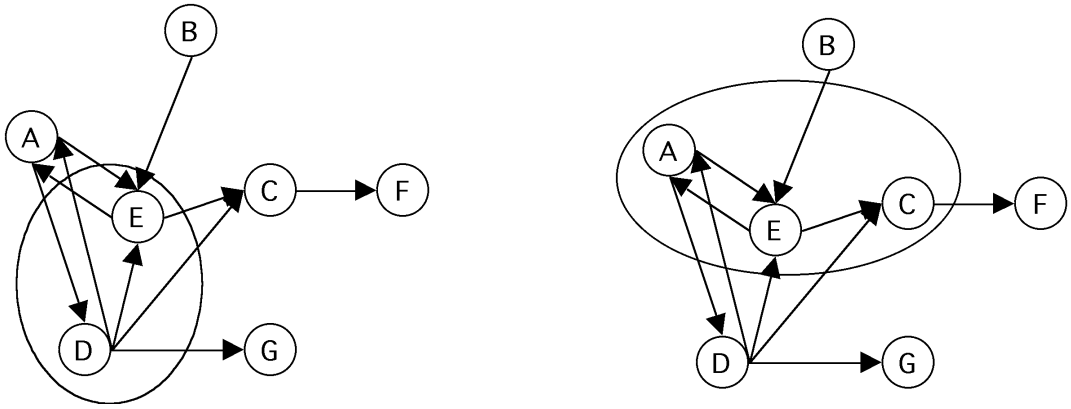


Abb. 5.6: Rand von A (links) und Rand von E (rechts) in G_0

```
;rand: knoten graph --> Liste (der Nachfolger)
```

die uns in einem Graphen alle Nachfolger zu einem gegebenen Knoten liefert, etwa in Listenform, also:

```
(define (rand knoten graph)
  ...)
```

und erwarten z.B. für G_0:

```
> (rand 'A g0)  -->   (list 'E 'D)
> (rand 'B g0)  -->   (list 'E)
> (rand 'C g0)  -->   (list 'F)
> (rand 'D g0)  -->   (list 'A 'C 'E 'G)
> (rand 'E g0)  -->   (list 'A 'C)
> (rand 'F g0)  -->   empty
> (rand 'G g0)  -->   empty
```

Dazu muss `rand` die Liste der Kanten des Graphen durchlaufen, die Nachfolger des Anfangsknotens finden und in eine Liste schreiben.
Da die Funktion lediglich die Kantenliste `(graph-kanten g)` des Graphen konsumieren muss, lassen wir das eine Hilfsfunktion

```
;rand-hilfe: knoten liste --> liste
```

erledigen und erhalten

```
(define (rand knoten graph)
  (rand-hilfe knoten (graph-kanten graph)))
```

Wenn der Graph nicht leer ist, gibt es beim Vergleich des (Ausgangs-)knotens mit den Anfangsknoten `(kante-ak <kante>)` der einzelnen Kanten zwei Fälle: Entweder Gleichheit oder nicht, damit lautet das Gerüst

```
(define (rand-hilfe knoten liste)
  (cond
    ((empty? liste) empty)
    ((equal? knoten (kante-ak (first liste)))
     ...
    (else
    ...)))
```

Bei Gleichheit wird der Endknoten der Kante `(kante-ek (first liste))` in die Randliste aufgenommen und die Funktion ruft sich mit dem Rest Kantenliste selbst auf, bei Ungleichheit ruft sie sich lediglich selbst mit dem Rest auf, also insgesamt

```
(define (rand-hilfe knoten liste)
  (cond
    ((empty? liste) empty)
```

```
((equal? knoten (kante-ak (first liste)))
 (cons (kante-ek (first liste)) (rand-hilfe knoten (rest liste))))
(else
 (rand-hilfe knoten (rest liste)))))
```

Damit können wir die Frage nach einer Strategie für `(erreichbar? 'A 'F g0)` , also für die Erreichbarkeit von F, ausgehend von Knoten A, mit Hilfe von Rändern beantworten: F liegt im Rand von B, B liegt im Rand von von C, C liegt im Rand von E, und E liegt im Rand von A.
Oder anders: Ist der Zielknoten nicht im Rand vom Anfangsknoten, dann fragen wir, ob der Zielknoten in den Rändern der Knoten des ersten Randes liegt, usw. Also müssen wir die Ränder von Rändern von Rändern ... untersuchen usw. usw., bis F auftaucht oder ein Rand leer ist.

Leider hat dieses Vorgehen einen Haken: Wie wir z.B. am Graph G_0, insbesondere an Abb. 5.6 sehen, liegt D im Rand von A, andererseits liegt A im Rand von E, und E liegt wiederum im Rand von D, d.h. es liegt ein *Zyklus* vor! Im diesem Fall gerät das o.a. Verfahren, die Ränder der Ränder usw. zu untersuchen, in eine Endlosschleife.

Was ist zu tun? Damit nicht bereits untersuchte Knoten erneut untersucht werden, müssen sie ausgesondert werden: dies kann mit einem Parameter `schon-besucht` umgesetzt werden, den wir der Hilfsfunktion

```
;erreichbar?-hilfe: anfangsknoten zielknoten graph schon-besucht
```

zuordnen.
Da am Anfang noch kein Knoten besucht ist, wird die Hilfsfunktion mit dem Wert `empty` für `schon-besucht` aufgerufen:

```
(define (erreichbar? anfangsknoten zielknoten graph)
  (erreichbar?-hilfe anfangsknoten zielknoten graph empty))
```

Die Hilfsfunktion muss drei Fälle unterscheiden:

1. wenn ein Knoten schon besucht ist, dann Ende
2. wenn Anfangsknoten = Zielknoten, dann Ende
3. sonst muss der Rand des Knoten näher untersucht werden

Wir erhalten als Gerüst:

```
(define (erreichbar?-hilfe anfangsknoten zielknoten graph schon-besucht)
  (cond
    (...<anfangsknoten schon-besucht> --> false)
    (...<anfangsknoten = zielknoten> --> true)
    (else
     ...<Untersuchung des Randes des Anfangsknotens>...)))
```

Die beiden ersten Fälle sind schnell erledigt; für den dritten Fall muss zunächst der Rand des Knotens ermittelt werden und als Zwischenergebnis gemerkt werden. Dafür machen wir die lokale Definition

```
(local ((define rand-aktuell (rand k g)))
        ...)
```

Bei der Untersuchung von `rand-aktuell` muss wieder eine Fallunterscheidung gemacht werden:

1. der Zielknoten liegt in diesem Rand, dann Ende
2. wenn nicht, dann werden alle Knoten des Randes (eine Liste) untersucht und der aktuelle Knoten ausgesondert, indem er der Liste `schon-besucht` zugefügt wird, wofür wir eine neue Hilfsfunktion benötigen

Die neue Hilfsfunktion unterscheidet sich von `erreichbar?-hilfe` dadurch, dass sie als ersten Parameter nicht einen einzelnen (Anfangs-)Knoten, sondern eine Liste von Knoten konsumiert:

```
;erreichbar?-von-liste: liste knoten graph schon-besucht --> BOOLEsch
```

Somit ergibt sich

```
(define (erreichbar?-hilfe k z g schon-besucht)
  (cond
    ((member k schon-besucht) false)
    ((equal? k z) true)
    (else
     (local ((define rand-aktuell (rand k g)))
       (cond
         ((member z rand-aktuell) true)
         (else
          (erreichbar?-von-liste rand-aktuell z g (cons k schon-besucht))))))))
```

Die Hilfsfunktion `erreichbar?-von-liste` wird daher mit `rand-aktuell` für den Parameter `liste` aufgerufen und `schon-besucht` wird durch `(cons k schon-besucht)` ersetzt.
Da die Funktion eine Liste (von Knoten) verbraucht, liegt eine strukturelle Rekursion vor, und wir haben das Gerüst:

```
(define (erreichbar?-von-liste liste z g schon-besucht)
  (cond
    ((empty? liste) false)
    (........)
    (else
     ........)))
```

In der ersten Ellipse muss geprüft werden, ob der Zielknoten vom ersten Knoten des Randes, also vom ersten Element der Liste, aus erreichbar ist. Das kann mit der schon definierten Funktion `erreichbar?-hilfe` geschehen: `((erreichbar?-hilfe (first liste) z g schon-besucht) true)` .
In der zweiten Ellipse, also der Alternative, ruft sich die Funktion mit dem restlichen Rand auf und sondert den aktuellen wieder aus in `schon-besucht` .
Wir fügen alles zusammen:

```
(define (erreichbar?-von-liste liste z g schon-besucht)
  (cond
    ((empty? liste) false)
    ((erreichbar?-hilfe (first liste) z g schon-besucht) true)
    (else
     (erreichbar?-von-liste (rest liste) z g (cons (first liste) schon-besucht)))))
```

Anmerkung
Es fällt auf, dass im Rumpf der Funktion `erreichbar?-hilfe` die Funktion `erreichbar?-von-liste` aufgerufen wird und umgekehrt: Wir haben zwei Funktionen, die wechselseitig aufrufen, man spricht auch von *mutueller* Rekursion.

Nachdem die Frage nach der Erreichbarkeit in einem Graphen geklärt ist, kehren wir zur Ausgangsfrage

Wie kommt man von A nach B in einem Graphen?

zurück, d.h. man benötigt zur Lösung dieser Frage keine BOOLEsche Funktion, sondern eine Funktion, die eine der Liste der möglichen Wege liefert. Es liegt nahe, einen einzelnen Weg selbst auch als Liste, und zwar der besuchten Knoten, zu modellieren:

```
;alle-wege: start ziel graph --> wegliste
```

Für unseren Mustergraphen
erwarten wir z.B.:

```
(alle-wege  'A 'B g0) --> empty
(alle-wege  'A 'E g0) --> (list (list 'A 'D 'E) (list 'A 'E))
(alle-wege  'A 'F g0) --> (list (list 'A 'D 'E 'C 'F) (list 'A 'E 'C 'F) (list 'A 'D 'C 'F))
```

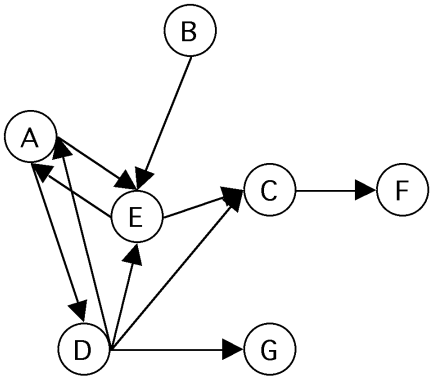


Abb. 5.7: Muster-Graph G_0

Für den Sonderfall *start* = *ziel* genügt z.B. die Ausgabe `(alle-wege 'A 'A g0) --> (list 'A)`. Die Vorgehensweise ähnelt der von `erreichbar?` insofern, dass zunächst der Rand des Anfangsknotens durchsucht wird. Für den Fall, dass ein Randknoten=Zielknoten ist, also ein Weg gefunden wurde, muss sich die Funktion diesen Weg als Liste von Knoten merken, was sinnvoller Weise durch einen zusätzlichen Parameter, etwa durch `zielwege` geschehen kann. Die restlichen Randknoten bilden mit dem Anfangsknoten in Form von Kanten (Teil-)Wege, die noch weiter untersucht werden müssen. Auch diese muss sich die Funktion merken, etwa in Form des Parameters `wegliste`. Für diese Aufgabe eignet sich eine Funktion

```
;alle-wege-hilfe: ziel wegliste graph zielwege --> allewege
```

die von `alle-wege` aufgerufen wird, wobei `zielwege` den Anfangswert `empty` und, da jeder Weg den Anfangsknoten `start` enthält, hat `wegliste` den Anfangswert `(list (list start))`:

```
(define (alle-wege start ziel graph)
  (alle-wege-hilfe ziel (list (list start)) graph empty))
```

Das Gerüst von `alle-wege-hilfe` ergibt sich aus folgender Überlegung: Ist ein Randknoten=Zielknoten, dann

- wird der Weg bis zu diesem Knoten der Liste `zielwege` hinzugefügt
- die restlichen Randknoten, also die restlichen (Teil-)Wege werden untersucht

Ist Randknoten $\neq$ Zielknoten, dann

- wird eine neue `wegliste` gebildet, die sich aus den restlichen, noch nicht untersuchten Wegen und den neuen, längeren Wegen zusammensetzt, die sich daraus ergeben, dass man dem aktuellen Weg jeweils die Kanten zu den Nachfolgern hinzufügt

```
(define (alle-wege-hilfe ziel wegliste graph zielwege)
( ......
  (cond
    ((equal? einknoten ziel)
     (alle-wege-hilfe ziel
                      (rest wegliste)
                      graph
                      <... füge weg zielwege hinzu>...)))
  (else
   (alle-wege-hilfe ziel
                    <...erstelle neue wegliste...>
                    graph
                    zielwege))))
```

Die Terminierung ist dadurch sichergestellt, dass die Wegliste irgendwann leer ist, und zwar deshalb, weil der erste rekursive Aufruf stets mit dem Rest der Wegliste erfolgt und beim zweiten Aufruf irgendwann keine neuen Teilwege mehr hinzugefügt werden können.

Der Zugriff auf den ersten Knoten erfolgt dadurch, dass aus der Wegliste der erste Weg `einweg` und daraus der erste Knoten `einknoten` lokal definiert werden. Somit ergibt sich für die erste Ellipse im Gerüst:

```
........
  (cond
    ((empty? wegliste) zielwege)
    (else
     (local
         ((define einweg (first wegliste))
          (define einknoten (first einweg)))
  ........
```

Das Hinzufügen eines Zielwegs zur Liste der Zielwege geschieht einfach durch `(cons (reverse einweg) zielwege))`, wobei die Liste aus optischen Gründen umgedreht wird.
Das Erstellen der neuen Wegliste beim zweiten Selbstaufruf (dritte Ellipse im Gerüst) gestaltet sich etwas komplexer, da sie sich aus zwei Teilen zusammensetzt: Zum einen aus den noch nicht untersuchten (Teil-)Wegen, also `(rest wegliste)`, zum anderen aus den neuen (Teil-)Wegen, die mit Hilfe der Funktion

```
;neue-wege: einweg einknoten graph --> Liste neuer Wege
```

erhält. Damit lautet die dritte Ellipse

```
(append (rest wegliste) (neue-wege einweg einknoten graph))
```

Die neuen Wege ergeben sich daraus, dass man einen Weg dadurch zu neuen Wegen verlängert, indem jeweils die Kante zu einem Nachfolgerknoten hinzufügt, also benötigt man die schon definierte Funktion `rand` mit dem Aufruf `(rand knoten graph)`.
An dieser Stelle ist Vorsicht geboten: Es muss vermieden werden, dass dabei Knoten berücksichtigt werden, die bereits in einem anderen Weg vorkommen! Das kann durch eine Funktion

```
;nicht-in-liste: liste1 liste2 --> liste
```

sicher gestellt werden, die diejenigen Elemente von `liste1` auflistet, die nicht in `liste2` enthalten sind. Da diese Funktion eine Liste durchläuft, ergibt sich sofort das Gerüst:

```
(define (nicht-in-liste liste1 liste2)
  (cond
    ((empty? liste1) empty)
    ((...<ist nicht in liste2>...)
     (...<füge hinzu>... (nicht-in-liste (rest liste1) liste2)))
    (else
     (nicht-in-liste (rest liste1) liste2))))
```

Für die erste Ellipse kann die vordefinierte Funktion `member` benutzt werden, für die zweite Ellipse `(cons ...)`.
Insgesamt ergibt sich:

```
(define (nicht-in-liste liste1 liste2)
  (cond
    ((empty? liste1) empty)
    ((not (member (first liste1) liste2))
     (cons (first liste1) (nicht-in-liste (rest liste1) liste2)))
    (else
     (nicht-in-liste (rest liste1) liste2))))
```

Im vorliegenden Fall müssen also die Knoten des neuen Randes ausgesondert werden, die bereits im vorliegenden Weg enthalten sind, also durch den Aufruf `(nicht-in-liste (rand knoten graph) weg)`, wobei man die so erhaltene Liste der Übersichtlichkeit halber einer lokalen Variablen zuweisen kann:

```
(define echte-nachbarn (nicht-in-liste (rand knoten graph) weg))
```

Die oben beschriebene Erzeugung der neuen Weg kann man elegant durch

```
(map (lambda (n) (cons n weg)) echte-nachbarn)
```

erreichen und erhält zusammengefasst:

```
(define (neue-wege weg knoten graph)
  (local
      ((define echte-nachbarn (nicht-in-liste (rand knoten graph) weg)))
    (map (lambda (n) (cons n weg)) echte-nachbarn)))
```

Damit sind alle Teile der Funktion `alle-wege` besprochen.

An dem Beispiel-Aufruf

```
(alle-wege 'D 'F g0) --> (list (list 'D 'A 'E 'C 'F) (list 'D 'E 'C 'F) (list 'D 'C 'F))
```

kann man an der Reihenfolge der Wege deutlich die Breitensuche erkennen: Da mittels `(cons (reverse einweg) zielwege))` ein neuer (Ziel-)Weg stets am Anfang der Zielweg-Liste eingefügt wird, werden die Wege mit wachsender Länge gefunden.

Wenn es Wege von A nach B gibt, ist es klar, dass es mindestens einen kürzesten Weg gibt. Spielt die Wahl es kürzesten Weges keine Rolle, kann man sich auf die Suche nach *einem* kürzesten Weg beschränken.

Von der gesuchten Funktion

```
;min-weg: start ziel graph --> weg
```

erwarten wir z.B.

```
(min-weg  'A 'B g0) --> empty
(min-weg  'A 'E g0) --> (list 'A 'E)
(min-weg  'A 'F g0) --> (list 'A 'D 'C 'F)
```

Ein Vergleich mit *`;alle-wege`* hilft uns bei der Konstruktion der gesuchten Funktion:

1. Sobald in *`;alle-wege-hilfe`* durch die Abfrage `(equal? einknoten ziel)` der Fall `start = ziel` eintritt, müssen weder der Rest der Wegliste durchlaufen werden noch dieser Zielweg der Liste `zielwege` hinzugefügt werden; statt dessen wird dieser Weg als Funktionswert ausgegeben und die Funktion terminiert. Somit liefert die Funktion keine Wegliste, also eine Liste von Listen, sondern einen Weg, also eine einfache Liste von Knoten.
2. Dass im Falle 1. ein kürzester Weg gefunden wurde, ist dadurch sichergestellt, dass der zweite Selbstruf nicht mehr stattfindet, der ja ohnehin aufgrund der Funktionsweise von *`;neue-wege`* nur längere Wege liefert.

Insgesamt genügen folgende Änderungen am Quellcode:

- *`;alle-wege`* wird in *`;min-weg`* umbenannt
- *`;alle-wege-hilfe`* wird in *`;min-weg-hilfe`* umbenannt
- der Parameter `zielwege` entfällt
- der Selbstaufruf nach der Abfrage `(equal? einknoten ziel)` wird durch die Ausgabe des kürzesten Weges `(reverse einweg)` (in der richtigen Reihenfolge der Knoten) ersetzt

Im Überblick:

```
;min-weg: ziel wegliste graph --> weg (als Liste)
(define (min-weg-hilfe ziel wegliste graph)
  (cond
    ((empty? wegliste) empty)
    (else
     (local
         ((define einweg (first wegliste))
          (define einknoten (first einweg)))
       (cond
         ((equal? einknoten ziel)
```

```
          (reverse einweg))
         (else
          (min-weg-hilfe ziel
                         (append (rest wegliste)
                                 (neue-wege einweg einknoten graph))
                         graph)))))))

;min-weg: knoten knoten graph --> weg (als Liste)
(define (min-weg start ziel graph)
  (minweg-hilfe ziel (list (list start)) graph))
```

6 Funktionale Modellierung, Teil 2

6.1 Abstraktion von Funktionen

6.1.1 Ähnlichkeit von Funktionen

Bisher haben wir Funktionen, bei denen im Rumpf Konstanten (Zahlen, Symbole) auftraten, verallgemeinert, indem wir stattdessen eine weiteren Funktionsparameter eingeführt haben. Im Folgenden wollen wir einen Schritt weiter gehen; dazu betrachten wir die folgenden beiden Funktionen:

```
;unterhalb: liste zahl --> liste
;Konstr. eine Liste von Zahlen,
;die unterhalb von zahl liegen
(define (unterhalb liste wert)
  (cond
    ((empty? liste) empty)
    (else
     (cond
       ((< (first liste) wert)
        (cons (first liste)
              (unterhalb (rest liste) wert)))
       (else
        (unterhalb (rest liste) wert))))))
```

```
;oberhalb: liste zahl --> liste
;Konstr. eine Liste von Zahlen,
;die oberhalb von zahl liegen
(define (oberhalb liste wert)
  (cond
    ((empty? liste) empty)
    (else
     (cond
       ((> (first liste) wert)
        (cons (first liste)
              (oberhalb (rest liste) wert)))
       (else
        (oberhalb (rest liste) wert))))))
```

Zum Beispiel erhalten für `(define eineListe (list 3 5 14 6 4 23 6 77 31 25 30 2))` folgendes:

```
(unterhalb eineListe 14)        --> (3 5 6 4 6 2)
(oberhalb eineListe 14) --> (23 77 31 25 30)
```

Die beiden Funktionen sind sich ähnlich, sie unterscheiden sich im Rumpf nur durch das Relationszeichen $<$ bzw. $>$. So stellt sich die Frage, ob man sie nicht durch eine einzige ersetzen könnte. Das bedeutet, es wird nicht wie bisher über eine Konstante (Wert) abstrahiert, sondern über einen Operator (Funktion) !
Wir erhalten:

```
1  ;filter1: rel-zeichen liste wert --> liste
2  (define (filter1 rel-zeichen liste wert)
3    (cond
4      ((empty? liste) empty)
5      (else
6       (cond
7         ((rel-zeichen (first liste) wert)
8          (cons (first liste)
9                (filter1 rel-zeichen (rest liste) wert)))
10        (else
11         (filter1 rel-zeichen (rest liste) wert))))))
```

und es ergibt sich in der Tat: `(filter1 < 14 eineListe) -> (3 5 6 4 6 2)`
oder auch

```
(filter1 = eineListe 14)        --> (14)
(filter1 <= eineListe 14)       --> (3 5 14 6 4 6 2)
(filter1 >= eineListe 14)       --> (14 23 77 31 25 30)
(filter1 equal? eineListe 14)   --> (14)
```

Wir haben als über ein Relationszeichen abstrahiert, ja sogar über die vordefinierte Funktion `equal?`.
Die Frage ist nun, ob man über jede beliebige Funktion, die zwei Zahlen als Eingabe und einen BOOLEschen Wert als Ausgabe hat, abstrahieren kann. Wir versuchen es mit

```
(define (quadrat>? zahl1 zahl2)
  (> (* zahl1 zahl1) zahl2))
```

und erhalten tatsächlich

```
(filter1 quadrat>? eineListe 14)==> (5 14 6 4 23 6 77 31 25 30)
```

Wir stellen fest:

> In *Scheme*
>
> - können Funktionen als Parameter von anderen Funktionen dienen, d.h.
> - **sind Funktionen** auch **Daten** !
>
> Man spricht auch „Objekten erster Klasse" (*first class objects)*

Einschränkung:
Dies ist nur bei bestimmten (aufrufenden) Funktionen möglich (siehe Abschnitt „Funktionen höherer Ordnung", nächster Abschnitt). Zwar liefern im funktionalen Konzept von Scheme alle Ausdrücke (<expr>) einen Wert, d.h. sie verhalten sich wie Funktionen, aber nicht alle <expr> (Ausdrücke) sind Objekte erster Klasse !

6.1.2 Funktionen höherer Ordnung

Wir können jetzt `filter1` aus dem letzten Kapitel noch weiter verallgemeinern:
Nehmen wir an, wir haben eine Liste von Zahlen und eine Funktion

`hat-Eigenschaft: zahl --> boolean`,

die prüft, ob eine Zahl eine bestimmte Eigenschaft hat, können wir formulieren:

```
1  ;filter: hat-Eigenschaft liste --> liste
2  (define (filter hat-Eigenschaft liste)
3    (cond
4      ((empty? liste) empty)
5      (else
6       (cond
7         ((hat-Eigenschaft (first liste))
8          (cons (first liste)
9                (filter hat-Eigenschaft (rest liste))))
10        (else
11         (filter hat-Eigenschaft (rest liste)))))))
```

z.B. ergibt sich mit eineListe aus dem letzten Kapitel:

```
(filter even? eineListe)--> (14 6 4 6 30 2)
```

Offensichtlich ist eine solche Funktion, die die Zahlen einer Liste, die einer besonderen Eigenschaft (= „Prädikat") genügen, herausfiltert, besonders nützlich.
Deshalb ist sie auch vordefiniert:

(filter <prädikat> <liste>) -> (Teil-)Liste

> *Zusammenfassung*
> Es ist möglich, Funktionen soweit zu abstrahieren, dass sie neben Konstanten auch (andere) Funktionen als Parameter haben können.
> Solche Funktionen nennt man Funktionen höherer Ordnung (*high-order functions*)

Beispiel

Ist etwa

```
(define (betrag>5 zahl)
  (cond
    ((> (abs zahl) 5) true)
    (else false)))
```

dann liefert
`(filter betrag>5 (list 1 -2 22 4 -5 1 76)) --> (list 22 76)`
Eine ähnliche Situation liegt vor, wenn wir - statt aus einer Liste mit Hilfe eines Prädikats oder allgemein einer Funktion bestimmte Elemente heraus zu filtern - einen Operator bzw. Funktion auf alle Listenelemente anwenden und eine Liste der Funktionswerte erhalten wollen:

```
;fuer-alle: f liste --> liste
(fuer-alle f (list e1 e2 ...)   --> (list (f e1) (f e2) ...)
```

Ähnlich wie bei `filter` ergibt sich

```
(define (fuer-alle op liste)
  (cond
        ((empty? liste) empty)
        (else
                (cons (op (first liste)) (fuer-alle op (rest liste))))))
```

Eine solche Funktion ist in Scheme bereits implementiert:

;map: Funktion liste –> liste
(map <funktion> <liste>)

`map` wendet die Funktion auf jedes Element der Liste an und liefert eine Liste mit diesen Funktionswerten.
Beispiele

```
(map sqr (list 1 -2 22 4 -5 1)) -> (list 1 4 484 16 25 1)
(map abs (list 1 -2 22 4 -5 1)) -> (list 1 2 22 4 5 1)
(map betrag>5 (list 1 -2 22 4 -5 1)) -> (list false false true false false
false)
```

Sehr hilfreich ist auch

;apply: Funktion liste –> wert (hängt von der Funktion ab)
(apply <funktion> <liste>)

`apply` betrachtet die Element der Liste als Operanden für die Funktion und liefert einen Funktionswert vom Typ der Funktion
Beispiele

```
(apply + (list 1 -2 22 4 -5 1 76)) -> 97
(apply * (list 1 -2 22 4 -5 1 76)) -> 66880
(apply append (list (list 'a 'b) (list 'c))) -> (list 'a 'b 'c)
(apply * (map sqr (list 1 -2 22 4 -5 1 76))) -> 4472934400
```

Der Einsatz von Funktionen höherer Ordnung bringt u.a. viele Vereinfachungen, wie z.B.

```
(define (durchschnitt liste)
  (if (empty? liste)
      "kein Durchschnitt"
      (/ (apply + liste) (length liste))))
```

Bei den BOOLEschen Operatoren AND, OR und NOT ist folgendes zu beachten: Während
`(map not (list true false)) ==> (list false true)`
zeigt, dass `not` ein Objekt erster Klasse ist, also eine „echte" Funktion, führt
`(apply and (list true false)) --> false`
zu einer Fehlermeldung, d.h. AND und OR sind nur Operanden bzw. Scheme-Schlüsselbegriffe, aber keine echten Funktionen !

Übungen

1. Entwickle die Funktion `apply` selbst !
2. Die Kantenlängen eines Quaders seien durch `(define quader (list 3 4 5))` festgelegt. Begründe, weshalb man mit

   ```
   (define (diagonale eine-liste)
      (sqrt (apply + (map sqr eine-liste))))
   ```

   die Diagonale berechnen kann.
3. Untersuche

   ```
   (define (a-plus-abs-b a b)
   ((if (>= b 0) + -) a b))
   ```

   Werden + und – als Funktionen genutzt ?
4. Analysiere genau die Aufrufe

   ```
   (list (list 1 2))
   (first (cons list (list 1 2)))
   (apply map (list list (list 1 2 3 4) (list 4 5 6 9)))
   (apply + (list 1 2 3 4))
   (map sqr (list 1 2 3 4))
   (apply + (map sqr (list 1 2 3 4)))
   (map list (list 1 2 3 4) (list 4 5 6 9))
   ```

6.2 Anonyme Funktionen: λ-Operator

In Scheme ist möglich, namenlose (anonyme) Funktionen zu konstruieren. Dazu wird eine spezielle Form, der sog. λ-Operator benötigt (λ = „lambda" ist ein Buchstabe des griechischen Alphabets, aus dem das lateinisch l entstanden ist) mit der Syntax

Konstruktion einer *anonymen Funktion*
`(lambda (<p1> <p2> ...) <Rumpf>)`

Konstruktion einer *anonymen Funktion mit Aufruf*
`((lambda (<p1> <p2> ...) <Rumpf>) <Ausdruck>)`

Beispiele

```
(lambda (x) (+ x 3))    --> #<procedure>

((lambda (x) (+ x 3)) 7)        --> 10

(map
   (lambda (x) (cond ((equal? x #\0) false) (else true)))
   (list #\1 #\1 \# #\0 #\1))   --> (list true true false true)
```

Genau genommen ist die gewohnte Funktionsdefinition lediglich eine Abkürzung von der ausführlichen Definition:

```
;Ausführliche Definition
(define f
  (lambda (x)
    (+ (expt x 2) (* -2 x) 4)))
```

```
;Abkürzung
(define (f x)
  (+ (expt x 2) (* -2 x) 4))
```

Anonyme Konstruktionen sind dort praktisch, wo eine Funktionsname nicht nötig ist.

Angenommen, man hat die Liste

```
(define zahlenfolgen (list (list 1 2 3) (list 1 2) (list 1 2 3 4)))
```

und möchte jede Liste mit 0 beginnen lassen, erreicht man das durch

```
(map (lambda (n) (cons 0 n)) zahlenfolgen)
       -->  (list (list 0 1 2 3) (list 0 1 2) (list 0 1 2 3 4))
```

Vergleiche mit `(map (lambda (n) (append n (list 'A 'B))) zahlenfolgen)`.

Übungen

1. Was liefern folgende Aufrufe ?

   ```
   ((lambda (x y) (* x (+ x y))) 7 13)  140
   ((lambda (f x) (f x x)) + 11)  22
   ((lambda () (+ 3 4)))  7
   ((lambda (x y) (+ x y 5)) 3 4)
   ((lambda (a d g) (* a (+ d g))) 4 8 2)
   ((lambda (a) (square a)) 5)
   ((lambda (radian) (/ (* pi radian) 180)) 30)
   ((lambda (x) (x 3)) (lambda (x) (* x x)))
   ```

2. Begründe, weshalb anonyme Funktionen nicht rekursiv sein können.

3. Man kann `(lambda...)` auch schachteln:

   ```
   (define doppler
      (lambda (f)
        (lambda (x) (f x x))))
   ```

   Damit ist `doppler` eine Funktion mit einem Parameter, der selbst eine Funktion sein muss, die 2 Parameter benötigt. Oder anders gesagt: `doppler` konstruiert nur eine Funktion mit einem Parameter x, der zweimal eingesetzt wird.
   Wenn wir z.B. für f den Operator + nehmen, also etwa `(define doppelt (doppler +))` dann benötigt die Funktion `doppelt` für x eine Zahl:
   `(doppelt 3)  --> 6`

6.3 Funktionen aus der Kryptographie

6.3.1 Überblick

Begriffe

Die *Kryptographie* beschäftigt sich mit der *Verschlüsselung (Chiffrierung)* eines *Klartextes (KT)* in einen *Geheimtext (GT)* sowie deren Umkehrung, also mit der *Entschlüsselung (Dechiffrierung)* eines *Geheimtextes* in einen *Klartext*:

$$\textit{Klartext} \xrightarrow[\text{Schlüssel}]{\text{Verschlüsseln (Chiffrieren)}} \textit{Geheimtext} \qquad \textit{Geheimtext} \xrightarrow[\text{Schlüssel}]{\text{Entschlüsseln (Dechiffrieren)}} \textit{Klartext}$$

Abb. 6.1: Kryptographie

Die *Kryptoanalyse* dagegen beschäftigt sich mit Methoden, aus einem Geheimtext den Klartext ohne Kenntnis des Schlüssels oder des Verfahrens oder beidem zu ermitteln.

Die historisch bekanntesten kryptographischen Verfahren beruhen auf

Substitution d.h. jeder Buchstabe des Klartextes wird durch einen (i.a. anderen) Buchstaben ersetzt. Entscheidend ist dabei der *Schlüssel*, in diesem Fall die Zuordnung zwischen *Klaralphabet (KA)* und *Geheimalphabet (GA)*. Allgemein gesprochen wird das KA durch eine Permutation zum GA. Beispiel:

```
KA: ABCDEFGHIJKLMNOPQRSTUVWXYZ
GA: NGPSBVWLHYKDIXMJFACRTUOZEQ
```

Mit diesem Schlüssel wird das Wort `INFORMATIK` zu `HXVMAIRHK` verschlüsselt.

Transposition Bei diesen Verfahren werden die Buchstaben des Klartextes nicht ersetzt, sondern permutiert (nicht zu Verwechseln mit der Permutation des KA bei Substitutionen!), d.h. sie tauchen an anderen Stellen auf, so dass insgesamt die Häufigkeit eines Buchstabens in Klartext und Geheimtext gleich ist. Beispiel:

```
KT: INFORMATIK
GT: RFOTIAMIKN
```

Zur Entschlüsselung benötigt man bei beiden Verfahren den Schlüssel und das Umkehrverfahren. Da bei Verschlüsselung und Entschlüsselung jeweils der gleiche Schlüssel benutzt wird, spricht man bei Substitution und Transposition von *symmetrischen* Verfahren.
Die zu den vorgestellten Verschlüsselungsverfahren gehörenden Entschlüsselungsverfahren sind z.T. als (derzeit noch) Übungsaufgaben vorgesehen.

Vereinbarung

Da überlicherweise nur Großbuchstaben chiffriert und dechiffriert werden, sollen die zu konstruierenden Funktionen folgendes berücksichtigen:

- der Geheimtext besteht nur aus Großbuchstaben ($65 \leq$ ASCII-Nr. ≤ 90)
- der Klartext kann aus Bequemlichkeitsgründen auch Kleinbuchstaben enthalten, also insgesamt Zeichen mit $65 \leq$ ASCII-Nr. ≤ 90 und $97 \leq$ ASCII-Nr. ≤ 122, versehentlich eingegebene Sonderzeichen werden ignoriert

Sonderzeichen kann man mit Hilfe von

```
;entferne-sonstige: zeichenkette --> zeichenkette
```

aussondern, in dem man die Funktion höherer Ordnung `filter` auf eine anonyme Auswahlfunktion und eine Liste von Zeichen anwendet:

```
(define (entferne-sonstige zeichenkette)
  (list->string
   (filter
    (lambda (zeichen)
      (cond
        ((<= 65 (char->integer (char-upcase zeichen)) 90) true)
        (else false)))
    (string->list zeichenkette))))
```

6.3.2 Monoalphabetische Substitution

Allgemeines Verfahren

Zunächst werden monoalphabetische Substitutionsverfahren, die sog. *Caesar*-Verschlüsselungen untersucht: jeder Buchstabe des KT wird anhand der Zuordnung KA $\rightarrow$ GA ersetzt (substituiert), wobei das GA eine bestimmte oder zufällig erzeugte Permutation des KA ist (im Gegensatz dazu werden bei polyalphabetischen Substitutionen mehrere GAs benutzt).

Beispiel:

```
KA: ABCDEFGHIJKLMNOPQRSTUVWXYZ
GA: GQFALNDZJXPIKMEHWBOTVSRCYU
```

Für `KT: ADAM` bedeutet das: Das A wird durch G, das D durch A, das A wieder durch G und das M durch K ersetzt, wodurch sich `GT: GAGK` ergibt.

Es müssen zwei Fragen geklärt werden:

1. Wie erhalte ich das GA?
2. Wie funktioniert das Substitutionsverfahren?

Offensichtlich unterscheiden sich die Verfahren nur durch 1., während die Ersetzung mittels GA bei allen die gleiche Struktur hat. Deshalb beginnen wir mit 2.:

```
;caesar: klartext GA --> geheimtext
```

Der Klartext wird, nachdem er von Sonderzeichen mittels `entferne-sonstige` befreit worden ist, in eine Liste von Zeichen (`char`) umgewandelt, dann erfolgt die eigentliche Substitution, und anschließend wird so erhaltene Liste von Zeichen wieder in eine Zeichenkette, den GT, transformiert. Das ergibt das Gerüst:

```
(define (caesar klartext GA)
    (list->string
    ...<Substitution mittels GA>...
      (string->list (entferne-sonstige klartext))))
```

Die eigentliche Substitution erfolgt zeichenweise: Zuerst wird mittels `char->integer` die ASCII-Nr. des Buchstabens ermittelt, und zwar angewandt auf `(char-upcase buchstabe)` – damit auch Kleinbuchstaben berücksichtigt werden –, dann wird vom Ergebnis 65 subtrahiert, damit für die Großbuchstaben A, B, ..., Z die Werte = 0, 1, ..., 25 erhalten, was jeweils die um 1 verminderte Position des Buchstabens im KA angibt. Jetzt wird mittels `list-ref` (liefert Ergebnisse beginnend mit 0) in dem als Liste dargestellten GA durch als Zahl angegebene Position der Ersatzbuchstaben geliefert.
Eleganterweise kann man den zugehörigen Term

```
(list-ref (string->list GA)
          (- (char->integer (char-upcase buchstabe)) 65))
```

als anonyme Funktion mittels `map` auf den KT als Liste anwenden und erhält insgesamt:

```
;caesar: zeichenkette zeichenkette --> zeichenkette
;verschlüsselt einen Klartext durch ein Geheimalphabet
(define (caesar klartext GA)
    (list->string
     (map
      (lambda (buchstabe)
           (list-ref (string->list GA)
                     (- (char->integer (char-upcase buchstabe)) 65)))
      (string->list (entferne-sonstige klartext)))))
```

Es bleibt die Frage nach der Gewinnung des GA, der im folgenden für die verschiedenen Substitutionsverfahren nachgegangen wird.

Alphabetischer- oder Verschiebe-Caesar

In diesem Fall ist das GA um k Stellen gegenüber dem KA zyklisch verschoben. Für z.B. $k = 3$ bedeutet das, dass das GA mit dem Buchtaben D beginnt, die fehlenden Buchstaben A, B und C werden am Ende aufgefüllt:

```
KA: ABCDEFGHIJKLMNOPQRSTUVWXYZ
GA: DEFGHIJKLMNOPQRSTUVWXYZABC
```

Für eine mögliche Funktion zur Gewinnung von GA aus KA liegt eine Funktion

```
;GA-macher-verschiebe: zeichenkette zahl --> zeichenkette
;erzeugt aus KA ein zyklisch um den ganzzahligen Wert  von zahl verschobenes GA
```

nahe.
Bei einem Aufruf allerdings, z.B.

```
> (GA-macher-verschiebe "ABCDEFGHIJKLMNOPQRSTUVWXYZ" 3)
> -->  "DEFGHIJKLMNOPQRSTUVWXYZABC"
```

müsste jeweils das komplette KA eingeben werden. Um dies zu vermeiden, wird stattdessen KA lokal definiert, und zwar als Liste von Zeichen:

```
;GA-macher-verschiebe: zahl --> zeichenkette
;erzeugt aus KA ein zyklisch um zahl verschobenes GA
(define (GA-macher-verschiebe verschiebung)
  (local
      ((define KA-liste (string->list "ABCDEFGHIJKLMNOPQRSTUVWXYZ")))
  ..... ))))
```

mit dem vereinfachten Aufruf

```
> (GA-macher-verschiebe  3)
> -->  "DEFGHIJKLMNOPQRSTUVWXYZABC"
```

Damit erwarten wir z.B. folgende Aufrufe und Ergbnisse:

```
> (caesar "Das ist ein Geheimtext" (GA-macher-verschiebe  3))
> --> "GDVLVWHLQJHKHLPWHAW"
> (caesar "DasisteinGeheimtext" (GA-macher-verschiebe  3))
> --> "GDVLVWHLQJHKHLPWHAW"
> (caesar "Dasist$..ei34nGehe ?imtext" (GA-macher-verschiebe  3))
> --> "GDVLVWHLQJHKHLPWHAW"
> (caesar "DASISTEINGEHEIMTEXT" (GA-macher-verschiebe  3))
> --> "GDVLVWHLQJHKHLPWHAW"
```

Der Algorithmus entspricht auch einem Substitutionsverfahren: Zuerst wird mittels `char->integer` die ASCII-Nr. des Buchstabens ermittelt, dann wird vom Ergebnis 65 subtrahiert und zu der Differenz die Verschiebung addiert. Das Ergebnis wird mit Rest durch 26 geteilt, damit bei Überschreitung von 26 der Rest die richtige ASCII-Nr. angibt. Zu dieser wird wieder 65 addiert und mit `integer->char` in den entsprechenen Buchstaben umgewandelt.
Der zugehörige Term

```
(integer->char  (+ (modulo (+ (- (char->integer zeichen) 65) k) 26) 65))
```

wird als anonyme Funktion mittels `map` dann auf `KA-liste` angewendet und das Ergebnis mittels `list->string` zum Funktionswert gemacht:

```
;GA-macher-verschiebe: verschiebung --> zeichenkette
;erzeugt GA aus KA durch Verschiebung
(define (GA-macher-verschiebe verschiebung)
  (local
      ((define KA-iste (string->list "ABCDEFGHIJKLMNOPQRSTUVWXYZ")))
    (list->string
     (map
      (lambda (zeichen)
        (integer->char
         (+ (modulo (+ (- (char->integer zeichen) 65) verschiebung) 26) 65)))
      KA-liste))))
```

Alternative: verkürztes Verfahren

Bei genauerem Hinsehen stellt man fest, dass bei einem Aufruf

```
> (caesar "DASISTEINGEHEIMTEXT" (GA-macher-verschiebe  3))
> --> "GDVLVWHLQJHKHLPWHAW"
```

zweimal substituiert wird, sowohl von `(caesar ...)` als auch von `GA-macher-verschiebe ...)` . Diesen Aufwand kann man reduzieren, wenn nur einmal substituiert wird, in dem die Buchstaben des KT direkt modulo 26 verschoben werden, so dass man die Hilfsfunktion `GA-macher-verschiebe ...)` nicht benötigt, aber dafür ein Substitutionsprogramm `(caesar-verschiebe ...)` erhält, das für andere Verfahren nicht brauchbar ist:

```
;caesar-verschiebe: zeichenkette zahl --> zeichenkette
;erzeugt GT aus KT direkt (ohne GA-Erzeugung)
(define (caesar-verschiebe klartext verschiebung)
    (list->string
     (map
      (lambda (zeichen)
        (cond
          ((<= 65 (char->integer (char-upcase zeichen)) 90)
           (integer->char
            (+ (modulo
                (+ (- (char->integer (char-upcase zeichen)) 65) verschiebung) 26) 65)))
          (else zeichen)))
      (string->list (entferne-sonstige klartext)))))
```

Aufrufe:

```
> (caesar-verschiebe "Das ist ein Geheimtext" 3)
> --> "GDVLVWHLQJHKHLPWHAW"
> (caesar-verschiebe "Sonderzeichen !$%&,,??" 3)
> --> "VRQGHUCHLFKHQ"
```

Entschlüsselung

Grundsätzlich wird bei symmetrischen Verfahren beim Entschlüsseln der gleiche Schlüssel wie beim Verschlüsseln benutzt, allerdings mit anderem Algorithmus. Dagegen ist es bei monoalphabetischen Substitutionen auch möglich – wenn man das Verfahren aufteilt in Gewinnung des GA und die eigentliche Substitution – aus GA ein "Rück-GA" zu ermitteln und das Substitutionsverfahren bei zu behalten:
Bei

```
> (caesar "DASISTEINGEHEIMTEXT" (GA-macher-verschiebe  3))
> --> "GDVLVWHLQJHKHLPWHAW"
```

liegt es auf der Hand, dass das KA in die entgegengesetzte Richtung verschoben werden muss, also muss `(GA-macher-verschiebe verschiebung` mit dem Wert von verschiebung mit entgegengesetztem Vorzeichen aufgerufen werden:

```
> (GA-macher-verschiebe -3)
> --> "XYZABCDEFGHIJKLMNOPQRSTUVW"
```

womit sich

```
> (caesar "GDVLVWHLQJHKHLPWHAW" "XYZABCDEFGHIJKLMNOPQRSTUVW")
> --> "DASISTEINGEHEIMTEXT"
```

oder

```
> (caesar (caesar "GDVLVWHLQJHKHLPWHAW" (GA-macher-verschiebe -3))
> --> "DASISTEINGEHEIMTEXT"
```

ergibt oder kurz

```
> (caesar (caesar "DASISTEINGEHEIMTEXT" (GA-macher-verschiebe 3)) (GA-macher-verschiebe -3))
> --> "DASISTEINGEHEIMTEXT"
```

Bei der abgekürzten Alternative `caesar-verschiebe` liegt der Fall besonders einfach:

```
> (caesar-verschiebe "DASISTEINGEHEIMTEXT" 3)
> --> "GDVLVWHLQJHKHLPWHAW"

> (caesar-verschiebe "GDVLVWHLQJHKHLPWHAW" -3)
> --> "DASISTEINGEHEIMTEXT"
```

Zufällige Permutation von KA: Buchstaben-Caesar

Hierbei erhält man das GA durch beliebige (zufällige) Permutation von KA, etwa

```
GA: GQFALNDZJXPIKMEHWBOTVSRCYU
```

woraus sich beim Aufruf folgender Wert (Geheimtext) ergeben muss:

```
> (caesar "Das ist ein Geheimtext" "GQFALNDZJXPIKMEHWBOTVSRCYU")
> --> "AGOJOTLJMDLZLJKTLCT"
```

Bei der Konstruktion einer geeigneten Funktion, die aus KA ein Zufalls-GA erzeugt, stehen wir wieder wie oben vor der Frage, ob das benötigte KA als Parameter eingegeben wird oder lokal definiert werden soll. Wenn man sich für Letzteres entscheidet, ergibt sich das Gerüst einer parameterlosen Funktion

```
;GA-macher-zufall:   --> zeichenkette
;erzeugt ein zufälliges GA aus KA
(define (GA-macher-zufall )
  (local
      ((define KA "ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
    ....)
```

Hinweise: Parameterlose Funktionen können nur im Sprachlevel „Fortgeschritten" deklariert werden!

Man erwartet z.B.

```
> (GA-macher „ABCDEFGHIJKLMNOPQRSTUVWXYZ")
> --> „GQFALNDZJXPIKMEHWBOTVSRCYU"
> (GA-macher „ABCDEFGHIJKLMNOPQRSTUVWXYZ")
> --> „CYMLJEZHWFGPXNVTUISKDARQOB"
> (GA-macher „ABCDEFGHIJKLMNOPQRSTUVWXYZ")
> --> „OYIXBTLHRGDPNFSJVQCMUKZWAE"
```

Da das Verarbeiten von Listen günstiger ist als von Zeichenketten und das Permutieren einer Liste am elegantesten rekursiv geschieht, wird die dazu notwendige Funktion in

```
(define (GA-macher-zufall )
  (local
      ((define KA "ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
    (list->string (GA-macher-zufall-hilfe (string->list KA)))))
```

als Hilfsfunktion ausgelagert:

```
;Geheim-Alphabet (durch zufällige Permutation von KA) als Liste
;GA-macher-hilfe: liste --> liste
```

Ein möglicher Algorithmus für `GA-macher-hilfe` ist folgender:

1. Wähle einen zufälligen Buchstaben x aus KA aus
2. Konstruiere eine Liste mit x
3. Rufe `GA-macher-Hilfe` mit KA ohne x auf
4. ... bis KA leer ist

Das ergibt folgendes Funktionsschablone:

```
(define (GA-macher-zufall-hilfe KA)
  (cond
    ((empty? KA) empty)
    (else
     .........
       (cons <x> (GA-macher-zufall-hilfe <KA ohne x>))))))
```

Dabei wird x mit Hilfe von `;list-ref: liste n --> element` durch den Aufruf `(n-tes KA (-zufallsstelle 1))` ermittelt, wobei man die Zufallsnummer `zufallsstelle` aus `(+ 1 (random (length KA)))` erhält.
Beim Selbstaufruf darf x in KA nicht mehr enthalten sein, da es schon verbraucht ist. Das kann durch

```
;entferne-el: liste element --> liste
```

erfolgen:

```
(define (entferne-el liste element)
  (cond
    ((empty? liste) empty)
    ((equal? element (first liste)) (rest liste))
    (else
     (cons (first liste) (entferne-el (rest liste) element)))))
```

Insgesamt ergibt sich:

```
;GA-macher-zufall-hilfe: liste --> liste
;Geheim-Alphabet (durch zufällige Permutation von KA) als Liste
(define (GA-macher-zufall-hilfe KA)
  (cond
    ((empty? KA) empty)
    (else
     (local
         ((define zufallsstelle (+ 1 (random (length KA))))
          (define (entferne-el liste element)
            (cond
              ((empty? liste) empty)
              ((equal? element (first liste)) (rest liste))
              (else
               (cons (first liste) (entferne-el (rest liste) element))))))
       (cons
        (list-ref KA (- zufallsstelle 1))
        (GA-macher-zufall-hilfe
         (entferne-el KA (list-ref KA (- zufallsstelle 1)))))))))

;GA-macher-zufall:   --> zeichenkette
;Geheim-Alphabet (durch zufällige Permutation von KA) als zeichenkette
(define (GA-macher-zufall )
  (local
      ((define KA "ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
    (list->string (GA-macher-zufall-hilfe (string->list KA)))))
```

Substitution mit Losungswort (Schlüsselwort)

Bei diesem-Verfahren beginnt das Geheimalphabet mit einem leicht zu merkenden Wort (Länge $<$ 26!), wobei Buchstaben, die sich im Losungswort wiederholen, im weiteren Verlauf entfallen. Der Rest wird mit den fehlenden Buchstaben aufgefüllt.
Beispiel: Mit „Dechiffrieren" als Losungswort ergibt sich:

```
KA: ABCDEFGHIJKLMNOPQRSTUVWXYZ
GA: DECHIFRNABGJKLMOPQSTUVWXYZ
```

Von der gesuchten Funktion

```
;GA-macher-losung: losungswort --> schluessel (GA)
```

erwarten wir z.B.

```
> (GA-macher-losung "DECHIFFRIEREN")
> --> "DECHIFRNABGJKLMOPQSTUVWXYZ"
> (caesar "Informatik" (GA-macher-losung "DECHIFFRIEREN"))
> "ALFMQKDTAG"
```

Der Quellcode von `GA-macher-losung` wird hier ohne Erläuterungen zum intensiven Selbststudium angeben: (s. Übungen)

```
;GA-macher-losung: zeichenkette --> zeichenkette
;erzeugt GA aus Losungswort
(define (GA-macher-losung losungswort)
  (local
      ((define KA "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
       (define (stripper-hilfe liste vorgekommen)
         (cond
           ((empty? liste) empty)
           ((member (first liste) vorgekommen)
            (stripper-hilfe (rest liste) (cons (first liste) vorgekommen)))
           (else
            (cons (first liste)
                  (stripper-hilfe (rest liste) (cons (first liste) vorgekommen))))))
       (define (stripper-rest zeichenkette)
         (local
             ((define GATeil1 (stripper-hilfe (string->list zeichenkette) empty)))
           (list->string
            (filter
             (lambda (element)
               (cond
                 ((member element GATeil1) false)
                 (else
                  true)))
             (string->list KA)))))
       (define (stripper zeichenkette)
         (string-append
          (list->string (stripper-hilfe (string->list zeichenkette) empty))
          (stripper-rest zeichenkette))))
    (stripper losungswort)))
```

Ein letzter Aufruf:

```
> (caesar "Es lohnt sich" (GA-macher-losung "SELBSTSTUDIUM"))
> --> "TPFJIHQPMLI"
```

Übungen

1. Zum *Verschiebe-Caesar*: Modifiziere das Programm so, dass
   a) der Parameter `verschiebung` durch einen Parameter `anfangszeichen` vom `char`, der den ersten Buchstaben des GA angibt, z.B. `#\D`
   b) die anonyme Verschiebungsfunktion als lokale Funktion namens `neuzeichen` definiert ist
2. Zum *Buchstaben-Caesar*: Modifiziere das Programm so, dass der Geheimtext und GA ausgegeben werden
3. Lies und analyse den Quellcode von *Substitution durch Losungswort*
4. Versuche, für *Buchstaben-Caesar* und *Substitution durch Losungswort* ein Entschlüsselungsverfahren zu entwickeln(!)
5. Vergleiche die drei Substitutionsverfahren im Hinblick auf die Anzahl der verschiedenen GAs, die möglich sind

6.4 Suchen in Listen

Eine typische Suchsituation liegt vor, wenn wir z.B. nach einer Telefonnummer suchen, z.B.

- in einem Telefonbuch
- im Internet
- in einem selbsterstellten Verzeichnis

Angenommen, wir haben ein selbsterstelltes Verzeichnis, in dem die Einträge nacheinander so erfolgt sind, wie wir die Nummern erfahren haben, d.h. sie sind ungeordnet. Bei einer kleinen Liste, d.h. bei einer Liste von einigen Dutzend Namen finden wir die zugehörige Nummer mit geringem Aufwand, aber mehrern hundert Einträgen wir die Suche schon aufwendig.
Dagegen geht es recht zügig, wenn wir in einem amtlichen Telefonbuch suchen: Obwohl wir es mit möglicherweisemit tausenden von Einträgen (bei einer Großstadt) zu tun haben, finden wir den gesuchten Teilnehmer recht schnell – die Einträge sind alphabetisch nach den Namen und Vornamen geordnet.
Daher unterscheiden wir zwischen *unsortierten* und *sortierten* Listen.
Für ein geeignetes Suchprogramm können wir auf unsere Datenmodellierung von S. 70 zurückgreifen:

```
(define-struct eintrag (name tel))

(define ein-tverzeichnis
        (list
            (make-eintrag "Lehmann" 4711)
            (make-eintrag "Schneider" 1234)
            (make-eintrag "Müller" 55772)))
            (...
```

Also brauchen wir eine Funktion
;suche-nr: name verzeichnis --> nummer
mit z.B.

```
> (suche-nr "Lehmann" ein-tverzeichnis) --> 4711
```

Da wir – wie in diesem Fall – die Liste nicht nach dem (Such-)*Schlüssel* `name` sortiert ist, müssen wir im ungünstigsten Fall die komplette Liste durchlaufen, um zu den gesuchten Namen zu finden, sofern er vorhanden ist:

```
;suche-nr: name verzeichnis
(define (suche-nr name verzeichnis)
  (cond
    ((empty? verzeichnis) "nicht vorhanden")
    ((equal? name (eintrag-name (first verzeichnis))) (eintrag-tel (first verzeichnis)))
     (else
      (suche-nr name (rest verzeichnis)))))
```

mit z.B.

```
> (suche-nr "Lehmann" ein-tverzeichnis) --> 4711
> (suche-nr "Lorenz" ein-tverzeichnis) --> "nicht vorhanden"
```

Die Suche in einer unsortierten Liste wird auch *lineare Suche* genannt, der Aufwand offensichtlich zu der Anzahl der Telefoneinträge (=Länge der Liste) proportional ist.
Wie oben schon erwähnt, geschieht Suchen in einem amtlichen Telefonbuch wesentlich schneller, und zwar deshalb weil die Einträge schon nach dem Suchschlüssel geordnet bzw. sortiert sind.
Ein passender Algorithmus für diesen Fall heißt *binäres Suchen*, er ist alles auf Listen schlecht zu realisieren, da man auf die einzelnen Elementen keinen sog. *wahlfreien Zugriff* hat, d.h. man kann direkt nur auf das erste Elmenent `(first liste)` zugreifen.
Zur Behebung dieses Problem gibt es zwei Möglichkeiten: Das Telefonverzeichnis wird

1. mit einem indexierten Datentyp (in *Scheme:* `vector`) dargestellt
   oder
2. als *Suchbaum* dargestellt oder von einer Liste in einen solchen umgewandelt, wie wir es auf S.78 ff durchgeführt haben.

Im zweiten Fall greifen wir das Banken-Konto-Beispiel von S. 80 auf und definieren
`(define-struct tel-such-baum (name tel lb rb))`

Entsprechend müssen die zugehörigen Funktionen *;baum-insert: Baum name tel –> Baum* und *;mache-BSB: Baum Liste –> Baum* ändern und die Funktion *;suche-im-Baum: Baum name –> tel-nr* konstruieren ⇒ Übungsaufgabe !

6.5 Sortieren

6.5.1 Einfache Verfahren

Im folgenden werden einige der gängigsten Sortierverfahren vorgestellt, und zwar zunächst auf Zahlenlisten. Angenommen, wir wollen folgende Zahlen aufsteigend sortieren:

22	11	27	33	6	35	45	11	47	17	8	23

Zunächst repräsentieren wir sie in einer Liste

```
(list 22 11 27 33 6 35 45 11 47 17 8 23)
```

Wir wählen die kleinste Zahl, hier 6, und mache sie zum ersten Element einer neuen Liste:

```
(list 6)
```

Aus der alten Liste entfernen wir die kleinste Zahl und erhalten

```
(list 22 11 27 33 35 45 11 47 17 8 23)
```

dann wählen wir wieder die kleinste Zahl, jetzt 8, und hängen sie an die neue Liste an:

```
(list 6 8)
```

Aus der alten Liste entfernen wir die kleinste Zahl und erhalten

```
(list 22 11 27 33 35 45 11 47 17 23)
```

dann wählen wir wieder die kleinste Zahl, jetzt 11, und hängen sie an die neue Liste an:

```
(list 6 8 11)
```

usw.
D.h. die unsortierte Liste wird immer kleiner und die neue, sortierte Liste immer größer - bis die unsortierte Liste leer ist.
Dieses Verfahren heißt **direktes Auswählen**.
Als Scheme-Funktion
;dir-ausw-auf: liste --> liste
können wir das Verfahren fast direkt umsetzen:

```
(define (dir-ausw-auf liste)
  (cond
    ((empty? liste) empty)
    (else
     (cons (kleinstes liste) (dir-ausw-auf (entferne (kleinstes liste) liste))))))
```

Dabei können wir *;kleinstes: liste –> zahl* durch

```
(define (kleinstes liste)
  (apply min liste))
```

und *;entferne: zahl liste –> liste* durch

```
(define (entferne element liste)
  (cond
    ((equal? (first liste) element) (rest liste))
    (else
     (cons (first liste) (entferne element (rest liste))))))
```

oder

```
(filter (lambda (x) (not (= x (apply min liste)))) liste)
```

ersetzen.
Entsprechendes gilt für *absteigendes* Sortieren.
Die Grundidee, schrittweise eine kleiner werdende unsortierte Liste in eine wachsende sortierte Liste zu überführen, findet sich auch bei folgendem Verfahren: Wir nehmen einfach das erste Element der unsortierten Liste

```
(list 22 11 27 33 6 35 45 11 47 17 8 23)
```

und mache sie zum ersten Element einer neuen Liste:

```
(list 22)
```

Aus der alten Liste entfernen wir diese Zahl und erhalten

```
(list 11 27 33 6 35 45 11 47 17 8 23)
```

Jetzt nehmen wir wieder die erste Zahl, nämlich 11, und fügen sie in der neuen Liste an die passende Stelle ein:

```
(list 11 22)
```

Aus der alten Liste entfernen wir diese Zahl und erhalten

```
(list 27 33 6 35 45 11 47 17 8 23)
```

Jetzt nehmen wir wieder die erste Zahl, nämlich 27, und fügen sie in der neuen Liste an die passende Stelle ein:

```
(list 11 22 27)
```

usw., bis die alte Liste leer ist.
Dieses Verfahren heißt **direktes Einfügen**.
Für die Scheme-Funktion
;dir-einf-auf: liste --> liste
ergibt sich:

```
(define (dir-einf-auf liste)
  (cond
    ((empty? liste) empty)
    ((cons? liste)
     (einfuegen (first liste) (dir-einf-auf (rest liste))))))
```

Für *;einfuegen: zahl liste –> liste* können wir

```
(define (einfuegen element liste)
  (cond
    ((empty? liste) (cons element empty))
    ((< element (first liste)) (cons element liste))
    (else
     (cons (first liste) (einfuegen element (rest liste))))))
```

nehmen.

Übungen

1. Erzeuge mithilfe einer geeigneten Funktion Zahlenlisten aus Zufallszahlen und teste `dir-ausw-auf` sowie `dir-einf-auf` damit.
2. Entwickle die Funktionen `dir-ausw-ab`sowie `dir-einf-ab`, um auch absteigend sortieren zu können.
3. Verallgemeinere beide Verfahren so, dass jeweils nur eine Funktion
   `- dir-ausw - dir-einf`
   gibt, um auf- und absteigend sortieren zu können (z.B. durch Einführen eines geeigneten Parameters, Stichwort: „Funktionen höherer Ordnung").
4. Modifiziere die Funktionen so, dass auch Zeichenketten sortiert werden können.
5. Der Realität kommt sicherlich das Sortieren von Assoziationslisten näher als einfache Zahlenlisten. Modifiziere die Funktionen so um, dass z.B. ein Telefonbuch der Form
   ```
   (define-struct person (name nummer))
   (define telefonbuch (<liste von personen>)
   ```
   sortiert werden kann.

6.5.2 *Quicksort*

Einen ganz anderen Weg geht folgendes Verfahren, um eine Liste z.B. aufsteigend zu sortieren:
Man wählt aus der Zahlenliste ein (im Prinzip) beliebiges Element, das sog. *Pivot*-Element aus, in unserem Beispiel etwa das erste, nämlich 22:

22	11	27	33	6	35	45	11	47	17	8	23

Dann zerlegt man die Zahlenliste in 2 Teillisten $T_<$ und $T_>$ mit

$T_<$ = | 11 | 6 | 11 | 17 | 8 |, also alle, die kleiner als p = 22 sind,
$T_>$ = | 27 | 33 | 35 | 45 | 47 | 23 |, also alle, die größer als p = 22 sind,

und fügt $T_<$, p und $T_>$ wieder zusammen, nachdem man - und das ist das Entscheidende - das

11	6	11	17	8	**22**	27	33	35	45	47	23

gleiche Verfahren auf die Teillisten $T_<$ und $T_>$ angewandt hat, usw. bis jede Teilliste entweder leer ist oder nur noch ein Element enthält.
Dieses Verfahren wir Quicksort genannt und legt sofort einen rekursiven Ansatz nahe:

```
;quick-sort: liste --> liste
(define (quick-sort-auf liste)
  (cond
    ((empty? liste) empty)
    ((empty? (rest liste)) liste)
    (else (append
           (quick-sort-auf <T<>))
           (list (first liste))          ;pivot-Element
           (quick-sort-auf <T>>))))))
```

Jetzt brauchen wir noch zwei Funktionen, um $T_<$ und $T_>$ zu erzeugen; der Einfachheit halber erledigen wir das mit einer Funktion höherer Ordnung, in dem einen Parameter z.B. namens `relop` (=„Relations-Operator") einführen, der je nach dem für $<$ oder $>$ steht:

```
;teilliste: liste zahl relop --> liste
(define (teilliste liste pivot relop)
  (cond
    ((empty? liste) empty)
    (else (if (relop (first liste) pivot)
              (cons (first liste) (teilliste (rest liste) pivot relop))
              (teilliste (rest liste) pivot relop)))))
```

Durch Einsetzen von teilliste in quick-sort ergibt sich:

```
1 ;quick-sort-auf: liste --> liste
2 (define (quick-sort-auf liste)
3   (cond
4     ((empty? liste) empty)
5     ((empty? (rest liste)) liste)
6     (else (append
7            (quick-sort-auf (teilliste liste (first liste) <))
8            (list (first liste))
9            (quick-sort-auf (teilliste liste (first liste) >))))))
```

Hinweis
Scheme enthält eine vordefinierte Funktion höherer Ordnung
;quicksort: liste (x x -> BOOLEAN) –> liste. Wende sie auf Beispiele an.

Übungen

1. Beachte, dass `quick-sort-auf` nur funktioniert (warum ?), wenn keine Zahl mehrfach vorkommt. Behebe das Problem !
2. Entwickle die Funktionen `quick-sort-ab`, um auch *absteigend* sortieren zu können.
3. Erzeuge mithilfe einer geeigneten Funktion Zahlenlisten aus Zufallszahlen und teste `quick-sort-auf` damit.

6.5.3 Testen von Sortierverfahren

Welches Verfahren nimmt man nun in der Praxis ? Natürlich das schnellste ! Aber ist ein bestimmtes Verfahren in allen Situation stets das schnellste ? Nebem dem Verfahren könnte die benötigte Zeit auch vom Umfang und der Art der unsortierten Liste abhängen, etwa

- wie groß ist die Anzahl n der Elemente ?
- sind die Elemente
  - völlig unsortiert (Zufallszahlen) ?
  - schon vorsortiert (auf-/absteigend) ?
  - teilsortiert ?
  - ...

Zur Klärung dieser Fragen können wir

- Tests durchführen
- theoretische Überlegungen anstellen

Zur Durchführung praktischer Tests müssen wir die benötigte Zeit messen können, dazu stellt *Scheme* die vordefinierte Funktion

`(time <Funktionsaufruf>)`

zur Verfügung. Z. B. liefert

`(time (quick-sort-auf zliste))`,

wobei `zliste` eine Liste von 500 Zufallszahlen ist – die 3 Zeiten (in ms = Millisekunden)

```
cpu time: 274 real time: 283 gc time: 0
```

Dabei ist *gc time* (*garbage collection* = Speicherbereinigung) für uns uninteressant; und da der Unterschied zwischen *cpu time* und *real time* nirgendswo dokumentiert ist, entscheiden wir uns für *real time* als Meßgröße.

Übungen

1. Mache Dir klar, dass das Einlesen einer Liste in einen binären Suchbaum (vgl. Kapitel „Baumstrukturen", S. II,8 bis II,11) mit anschließendem Auslesen mittels `(inorder <bsb>)` ebenfalls ein Sortierverfahren („Baumsortieren") darstellt.
2. Fertige Messtabellen (–> Tabellenkalkulation) an etwa der Form

	A	B	C	D	E	F	G	H	I	J	K
1		100	200	300	400	500	1000	1500	2000	2500	
2	dir-ausw-auf										
3	dir-ausw-ab										
4	dir-einf-auf										
5	dir-einf-auf										
6	quick-sort-auf										
7	quick-sort-ab										
8	baum-sort-auf										
9	baum-sort-ab										
10											

   und zwar jeweils für
   - unsortierte
   - aufsteigend sortierte
   - absteigend sortierte Listen

   Führe die Tests durch und trage die Zeiten ein. Vergleiche, evtl. Diagramme anfertigen.
3. Führe 2 auch mit Assoziationslisten durch !
4. In *Scheme* gibt es bereits die eingebaute Funktion `quicksort` ! Nimm Sie auch in Deine Messtabellen mit auf. Was fällt auf ? Begründung ?

Näheres zu Sortierverfahren:

- *http://www.matheprisma.uni-wuppertal.de/Module/Sortieren*
- *http://www.informatiktreff.de/materialien/sek_ii/algorithmen/main.htm*

7 Modellierung aus Mathematik & Informatik

7.0.4 Polynome

Den Term einer ganz-rationalen Funktion

$$f(x) = ax_n + a_{n-1}x_{n-1} + ...a_1x_1 + a_0 \text{ mit } a_i \in \mathbb{R}, a_n \neq 0 \tag{7.1}$$

nennt man bekanntlich *Polynom* und die a_i *Koeffizienten*, durch die die Funktion eindeutig bestimmt ist.
Für die Koeffizientenliste liegt offensichtlich eine Modellierung durch eine Liste nahe. Wir betrachten das Beispiel

$$f(x) = 3x_4 - x_3 + x + 2 \tag{7.2}$$

Dann stellt `(define kliste1 (list 3 -1 0 1 2))` die Koeffizientenliste dar.
Ein Ansatz zur Berechnung des Funktionswertes wäre

```
;poly: koeffizientenliste variable --> Funktionswert
(define (poly koeff-liste x)
  (cond
    ((empty? koeff-liste) 0)
    ((= (length koeff-liste) 1) (first koeff-liste))
    (else
     (+ (first koeff-liste) (* x (poly (rest koeff-liste) x))))))
```

Leider ergäbe z.B. der Aufruf der Form `(poly kliste1 2)` ein falsches Ergebnis, da beim Selbstaufruf die Potenz nicht erniedrigt, sondern erhöht werden (Reihenfolge in der Liste), dies korrigieren wir durch

```
(define (f x)
  (poly (reverse kliste1) x))
```

Die Listenstruktur ermöglich uns auch ein *formales* Ableiten:

```
;ableitung: koeff-liste --> koeff-liste
(define (ableitung koeff-liste)
  (cond
    ((= (length koeff-liste) 1) empty)
    (else
     (cons (* (first koeff-liste) (- (length koeff-liste) 1))
        (ableitung (rest koeff-liste))))))
```

Allerdings müssen wir auch die Liste umdrehen und erhalten für die Ableitung an der Stelle x schließlich für f':

```
(define (f-dx x)
 (poly (reverse (ableitung kliste1)) x))
```

Schließlich können wir auch leicht Wertetabellen erstellen fuer f an den Stelleln $x_1 = a, x_2 = a + dx, ..., x_n = a + (n-1)dx$:

```
(define (werte-tabelle f a n dx)
  (cond
    ((= n 1) (list (list a (f a))))
    (else
     (cons (list a (f a)) (werte-tabelle f (+ a dx) (- n 1) dx)))))
```

7.0.5 Binomialkoeffizient

Im allgemeinen binomischen Ausdruck

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k \text{ mit } n \in \mathbf{N} \tag{7.3}$$

wird die Schreibweise

$$\binom{\mathbf{n}}{\mathbf{k}} = \frac{n!}{k!(n-k)!} \text{ mit } n, k \in \mathbf{N} \text{ und } k \leq n \tag{7.4}$$

Binomialkoeffizient genannt – gelesen: „n über k" – und hat u.a. in der Kombinatorik große Bedeutung.
Die praktische Berechnung kann für große n wegen der Fakultäten Probleme bereiten, nicht nur handschriftlich, sondern auch softwaremäßig aufgrund der beschränkten integer-Repräsentation, je nach Sprachimplementation.
Eine Abhilfe schafft folgende Überlegung: Schreibt man die Koeffizienten für aufsteigendes n = 0, 1, 2,... zeilenweise mittensymmetrisch untereinander und ergänzt auf beiden Seiten jeweils eine Null, erhält man folgendes Schema, auch *Pascalsches Dreieck* genannt:

```
n
0                          0     1     0
1                       0     1     1     0
2                    0     1     2     1     0
3                 0     1     3     3     1     0
4              0     1     4     6     4     1     0
5           0     1     5    10     5     5     1     0
...
```

Abb. 7.1: Pascalsches Dreieck

Man erkennt leicht, dass man aufgrund der Beziehung

$$\binom{n}{k} = \binom{n}{k+1} + \binom{n+1}{k+1} \tag{7.5}$$

den k+1-ten Koeffizienten in der n+1-ten Zeile durch Addition der „Nachbarn" k und k+1 der n-ten Zeile erhält, z.B. 1+3=4; d.h., dass man die Koeffizienten für n+1 aus den denen von n unter Ergänzung von Nullen rekursiv erhalten.
Damit liegt eine Modellierung zur Berechnung der Koeffizienten auf der Hand: Wir repräsentieren die Koeffzienten für bestimmtes n als Liste, ergänzen sie einmal links mit 0 und einmal rechts mit 0 und addieren die beiden so erhaltenen gleichlangen Listen zur neuen Koeffizientenlist für n+1: Da wir zwei gleichlange Listen haben, können wir die Addition mit der high-order-Funktion *map*

```
    0  1  3  3  1   (cons 0 liste)
+   1  3  3  1  0   (append liste (list 0))
=   1  4  6  4  1   -> neue Liste
```

erledigen: `(map + (cons 0 liste) (append liste (list 0)))`.
Als Terminierung der Rekursion beginnen wir mit `(define liste (list 0))` für n = 0.
Insgesamt ergibt sich zur Berechnung der Liste der Binomialkoeffzienten für bestimmtes n:

```
;binom-n-liste: n --> Liste
(define (binom-n-liste n)
  (if (zero? n)
      (list 1)
```

```
      (local
          ((define liste (binom-n-liste (- n 1))))
        (map + (cons 0 liste) (append liste (list 0))))))
```

Aufrufe:

```
(binom-n-liste 3)  --> (list 1 3 3 1)
(binom-n-liste 10)  --> (list 1 10 45 120 210 252 210 120 45 10 1)
```

Um auch die Summanden der Binomialverteilung zu erhalten, müssen die Binomialkoeffizienten $\binom{n}{k}$ mit den Wahrscheinlichkeiten $p^{n-k}q^k$ multipliziert werden, wofür wir die Funktion

```
;prob:n p k --> liste
```

benötigen:

```
(define (prob n p k)
  (cond
    ((zero? k)  (expt p n))
    (else
     (* (/ (- 1 p) p) (prob n p (- k 1))))))

;probliste: n p j --> liste der Wahrsch. 1<=j <=n
(define (probliste n p j)
  (cond
    ((zero? j) (list (prob n p j)))
    (else
     (cons (prob n p j) (probliste n p (- j 1))))))

;binomial: n p --> liste der Bn,p;k für bestimmtes k
(define (binomial n p)
  (map * (binkoeff n) (probliste n p n)))
```

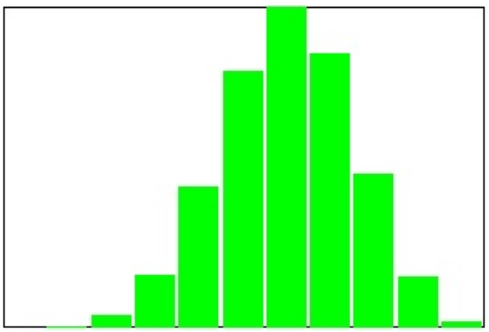
Z.B. liefert der Aufruf

```
(binomial 10 0.6) -->
   (list 0.0001048576 0.001572864 0.010616832 0.042467328 0.111476736
   0.2006581248 0.250822656 0.214990848 0.120932352 0.040310784 0.0060466176)
```

Zur Visualierung der Verteilung können wir auf `balken-diagramm` von S. zurückgreifen und erhalten mit

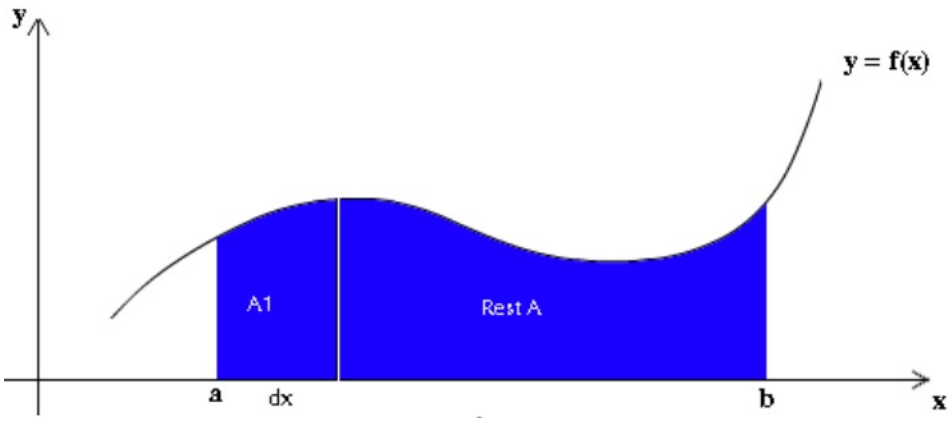
```
;Szene-Größe
(define b 300)
(define h 200)
```

aus dem Aufruf `(balken-diagramm (binomial 10 0.6) 'green)` in der REPL folgende Grafik



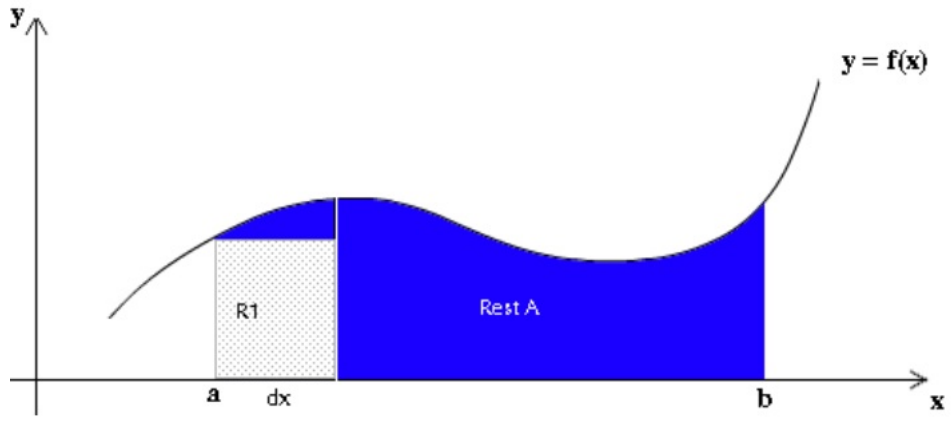
7.0.6 (Numerische) Integration

Um die Fläche A zwischen dem Graph einer einer Funktion f und der x-Achse zu bestimmen, versuchen wir eine induktive Modellierung (vgl. Listen): wir zerlegen sie in



- einen Abschnitt der Breite dx
- den Rest

und erhalten $A = A_1 + Rest\ A_0$, wobei $A_0 = A$.
Ersetzen wir A_1 näherungsweise durch ein Rechteck R_1 der Breite dx und der Höhe f(a)



$$A \approx R_1 + Rest\ A_0 = f(a)dx + Rest\ A_0 \tag{7.6}$$

Jetzt stellen wir die gleiche Überlegung für den Rest an, d.h. wir zerlegen $Rest\ A_1$ in ein Rechteck der gleichen Breite und neuen Rest: $Rest\ A_1 = R_2 + Rest\ A_1 = f(a+dx)dx + Rest\ A_1$:

$$A \approx R_1 + R_2 + Rest\ A_1 = f(a)dx + f(a+dx)dx + Rest\ A_1 \tag{7.7}$$

Dieses Verfahren können wir fortsetzen und A näherungsweise in n Rechtecke zerlegen,
bis $a + (n-1)dx \geq b$ ist.
In Scheme lautet unser Vertrag:

```
; A: f a b dx --> zahl (= Rechtecksumme)
(define (A f a b dx)
        (<Terminierung> ?
        (+ <Rechteck> <Rest>)))
```

Stellen wir uns jetzt vor, dass wir bei jedem Schritt die linke Grenze des Rechtecks als neues a auffassen, d.h. $a+dx$ wird jeweils zum neuen a, ist das Verfahren beendet, wenn $a \geq b$, d.h. es wird nur noch 0 addiert:

```
(define (A f a b dx)
  (if (>= a b) 0
      (+ (* (f a) dx) (A f (+ a dx) b dx))))
```

Es ist offensichtlich, dass die Abweichung vom tatsächlichen Flächeninhalt A um so kleiner wird, je kleiner dx wird. Erfüllt f bestimmte Bedignungen, so nennt man den Grenzwert für $x \to \infty$ *bestimmtes Integral von f über dem Intervall a b*. In diesem Fall haben wir eine Annäherung durch eine sog. „Untersumme".
Beispiele

```
(A sqr 0 1 0.1)          --> 0.285
(A sqr 0 1 0.01)         --> 0.32835
(A sqr 0 1 0.001)        --> 0.3328335
```

Man erkennt: je kleiner dx, desto besser nähert sich A dem exakten Wert

$$\int_0^1 x^2 dx = \frac{1}{3}$$

an.
Wir können den prinzipiellen Fehler, dass durch die Tatsache, dass a bei jedem Aufruf durch $a+dx$ ersetzt wird, aber im allgemeinen bei der Terminierung nicht den Wert b erreicht (außer, wenn $b-a$ durch dx teilbar ist), verkleinern, wenn wir die Anzahl der Teilintervalle n vorgeben, und dx durch $dx = (b-a)/n$ erhalten:

```
; A2: f a b n --> zahl (= Rechtecksumme)
(define (A2 f a b n)
  (local ((define dx (/ (- b a) n))
          (define (summe m)
            (cond
              ((= 0 m) (f a))
              (else
               (+ (* (f (+ a (* m dx))) dx) (summe (- m 1)))))))) (summe (- n 1))))
```

Übungen

1. Entwickle eine Funktion
   ; *ableitung* : $f\ x_0\ h \to zahl$,
   mit der zu einer Funktion f an der Stelle x_0 den Differenzenquotienten für eine Umgebung h ermitteln kann.
2. Statt den Wert des bestimmten Integrals näherungsweise zu bestimmen, kann man auch den Graphen einer Stammfunktion (sofern sie existiert) einer gegebenen Funktion *f'* näherungsweise bestimmen durch das sog. *EULER-CAUCHYsche Streckenzugverfahren*: Man kennt
   - die Ableitung $f'(x)$
   - einen Anfangspunkt $P_0 = (x_0|y_0)$

   Dann kann man aus P_0 mittels
   a) $x_n = x_{n-1} + \Delta x$ mit $\Delta x > 0$ und $n = 1, 2, ...$
   b) $y_n = y_{n-1} + f'(x_{n-1})\Delta x$

   sukzessive die Koordinaten $(x_n|y_n)$ der Punkte $P_1, P_2, P_3, P_4, ...$ des Streckenzuges bestimmen.
   Entwickle eine Lösung und teste sie für $f'(x) = \frac{1}{x}$ mit $\Delta x = 1$, $P_0 = (1|0)$

7.0.7 Matrizenrechnung

Grundsätzlich können wir eine Matrix interpretieren als

1. Datentyp
2. Abbildung

Im Falle (1) können wir eine Matrix als 3 Spalten von 2 Reihen realisieren, z.B. indem wir sie als Liste von Listen modellieren. Sei etwa

```
(define A
  (list (list 1 2 3)
        (list 4 5 6)))
```

Jetzt bieten sich die verschiedenen Operationen an, z.B. Addition, Multiplikation, Inverse bilden, usw. Als Beispiel betrachten wir die Transposition einer Matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} (*)$$

Wie können wir diese Abbildung in Scheme realisieren ? Ein möglichst kurze Variante sei hier zur Analyse angegeben:

```
(define (transponiere matrix)
        (apply map (cons list matrix)))
```

Wie man sieht, kann man mit dem Aufruf `(transponiere A)` das Gewünschte, nämlich $(*)$, sofort erreichen. Dabei zeigt sich die ganze Kraft von Funktionen höherer Ordnung; man versuche, `transponiere` mit Pascal oder Java umzusetzen !

Übungen

1. Analysiere `transponiere` genau. Vorher empfiehlt sich Übung 4, S. IV, 4

7.0.8 Symbolisches Differenzieren (CAS)

Computer-Algebra-Systeme (CAS) können u.a. den Term der Ableitungsfunktion $f'(x)$ einer Funktion $f(x)$ ermitteln, was häufig hilfreich sein kann. Für den Anfang machen wir folgende Annahmen:

▷ Wir beschränken uns auf Addition und Multiplikation, und zwar jeweils nur mit 2 Operanden

▷ die Terme liegen in Präfix vor, und zwar als Listen, wobei die Operatoren und Variablen als Symbole behandelt werden, z.B.
3x + 5 –> `(list '+ (list '* 3 'x) 5)`

Wir suchen also eine Funktion ableitung mit
; ableitung: exp1 var –> exp2
wobei *exp1* den Funktionsterm und *exp2* den Ableitungsterm in o.a. Listenform darstellen; mit *var* können wir entscheiden, nach welcher Variablen wir abeleiten:

```
(define (ableitung exp var)
        (...exp...var))
```

In dieser Funktion müssen also für folgende Regeln berücksichtigt werden:

$$\begin{aligned} \frac{dc}{dx} &= 0 \qquad \text{für eine Konstante c} \\ \frac{dx}{dx} &= 1 \\ \frac{d(u+v)}{dx} &= \frac{du}{dx} + \frac{dv}{dx} \qquad \text{Summenregel} \\ \frac{d(uv)}{dx} &= u\frac{dv}{dx} + v\frac{du}{dx} \qquad \text{Produktregel} \end{aligned}$$

Also gilt für den Rumpf: Falls `fexp`

▷ ein Konstante ist, dann 0
▷ gleich der Variablen ist, nach der abgeleitet wird, dann 1
▷ eine Summe ist, dann Summenregel
▷ ein Produkt ist, dann Produktregel

Dazu benötigen wir einige Prädikatoren, Selektoren und Konstruktoren:

Damit ergibt sich:

Prädikatoren	*Selektoren*	*Konstruktoren*
`(define (konstante? x)` `  (number? x))` `(define (variable? x)` `  (symbol? x))` `(define (v1=v2? v1 v2)` `  (and` `  (variable? v1)` `  (variable? v2)` `  (eq? v1 v2)))` `(define (summe? x)` `  (if (cons? x)` `  (eq? (first x) '+)` ` false))	`(define (summand1 s)` `  (first (rest s)))` `(define (summand2 s)` `  (first (rest (rest s))))` `(define (multiplikand p)` `  (first (rest p)))` `(define (multiplikator p)` `  (first (rest (rest p))))`	`(define (mache-summe s1 s2)` `  (list '+ s1 s2))` `(define (mache-produkt f1 f2)` `  (list '* f1 f2))`

```
; ableitung: exp1 var --> exp2
(define (ableitung exp var)
  (cond
    ((konstante? exp) 0)
    ((variable? exp)
     (if (v1=v2? exp var)
         1
         0))
    ((summe? exp)
     (mache-summe
      (ableitung (summand1 exp) var) (ableitung (summand2 exp) var)))
    ((produkt? exp)
     (..............))))
```

Der Aufruf `(ableitung (list '+ (list '* 3 'x) 5) 'x)` liefert den etwas unhandlichen Ausdruck

```
==> (list '+ (list '+ (list '* 'x 0) (list '* 1 3)) 0)
```

Was man jetzt braucht, ist eine Funktion, die solche Ausdrücke vereinfacht, etwa der Form
; optimiere-ausdruck: exp –> exp
die die Addition mit 0 und die Multiplikation mit 1 verkürzt:

```
(define (optimiere-ausdruck expr)
  (cond
    ((summe? expr) (optimiere-summe
                    (optimiere-ausdruck (summand1 expr))
                    (optimiere-ausdruck (summand2 expr))))
    ((produkt? expr) (optimiere-produkt
                      (optimiere-ausdruck (multiplikand expr))
                      (optimiere-ausdruck (multiplikator expr))))
    (else
     expr)))
```

mit

```
(define (optimiere-summe s1 s2)
  (cond
    ((and (konstante? s1) (konstante? s2)) (+ s1 s2))
    ((eqv? 0 s1) s2)
    ((eqv? 0 s2) s1)
    (else
     (mache-summe s1 s2))))
```

Jetzt liefert `(optimiere-ausdruck (ableitung (list '+ (list '* 3 'x) 5) 'x))` den vereinfachten Ausdruck `==> (list '+ (list '* 'x 0) 3)==> (list '+ (list '* 'x 0) 3)`

Übungen

1. Vervollständige den Rumpf von `(ableitung ...)`
2. Erstelle die für noch fehlende Funktion `(optimiere-produkt ...)`
3. Verallgemeinere das Programm
   a) für mehr als 2 Operanden bei Addition und Subtraktion
   b) auf Differenzen
   c) auf Quotienten

7.0.9 Einfache Schaltnetze

Wir wissen, dass die einfachsten digitalen Schaltung sog. *Schaltnetze* oder *Gatter* sind; sie sind aus den technischen Realisierungen der drei logischen Grundoperationen

UND : `(and ...`
ODER : `(or ...`
NICHT : `(not ...`

zusammengesetzt. Ihre Symbole sind:

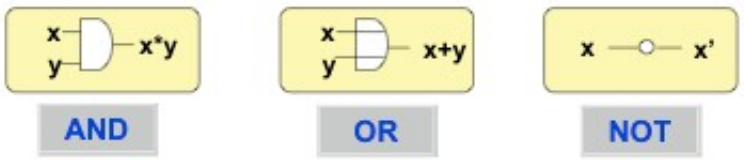


Abb. 7.2: herkömmliche Symbole

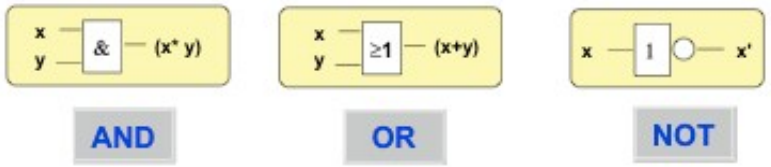
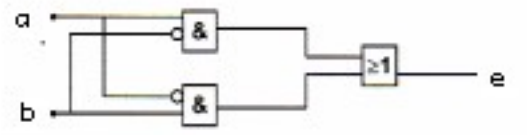


Abb. 7.3: neuerer Standard

Mit diesen drei Grundbausteinen (oder Schaltfunktionen) kann man alle anderen Schaltnetze realisieren, z. B.

1. das XOR (Exklusives ODER), auch Antivalenz genannt:



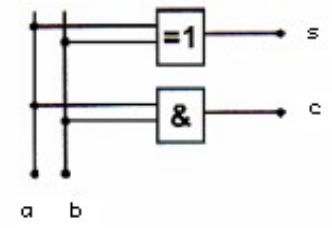
, als Symbol: $\boxed{=1}$

```
(define (XOR a b)
        (or (and a (not b)) (and (not a) b)))
```

2. ein Halbaddierer (HA) addiert 2 einstellige Dualzahlen und liefert den Stellenwert s sowie den Übertrag (*carry*) c:

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

, also für s ein XOR und für c ein UND:



Ein Halbaddierer hat zwei Ausgänge, d.h. zur Simulation benötigen wir entweder

- zwei Funktionen: *;s: a b –> s* und *;c: a b –> c*
  oder
- eine Funktion *;ha: a b –> (s c)*, wobei (s c) z.B. eine Struktur ist der Art `(define-struct aus2 (z1 z2))`

Jetzt können wir eine Funktion für einen Halbaddierer definieren:

```
;halbaddierer
;ha: a b --> aus2
(define (ha a b)
        (make-aus2 (or (and a (not b)) (and (not a) b)) (and a b)))
```

3. Bestätige, dass durch

```
(define (va a b aue)
        (make-aus2
                (aus2-z1 (ha (aus2-z1 (ha a b)) aue))
                (or (aus2-z2 (ha (aus2-z1 (ha a b)) aue)) (aus2-z2 (ha a b)))))
```

ein Volladdierer dargestellt wird.

Übungen

1. Um obige Funktionen testen zu können, müssen wir einstellige Dualzahlen statt mit 0 und 1 mit false und true schreiben:

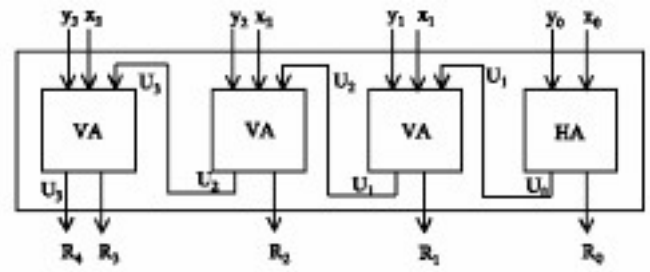
```
(ha false false)        -->     (make-aus2 false false) ;d.h. 0 + 0 = 0 Übertrag 0
(va true true true)     --> ?
```

2. Entsprechend können wir mehrstellige Dualzahlen als Listen von schreiben:

```
(list false true true true)     =  0111
(list false true true false)    =  0110
```

Zwecks Addition werden zwei Zahlen als Dualzahlen in zwei Register geschrieben und dann mittels eines *Paralleladdierers*, der aus einem Halbaddierer und mehreren Volladdierern besteht, addiert:

```
(define regX (list false true true true))
(define regY (list false true true false))
```



a) Entwickle die Funktionen

i.
```
;s: register1 register2 --> register, wobei register = Liste aus true und false
(define (s x y)
        ....)
```
die unter Benutzung von ha und va das Ergebnis als Liste von Stellenwerten (Register) berechnet,

ii.
```
;ue:register1 register2 --> BOOLEAN
(define (ue x y)
        ...)
```
die den letzten Übertrag berechnet

b) Teste a) mittels `(ue regX regY)` und `(s regX regY)`

c) Entwickle
- „Halbsubtrahierer"
- „Vollsubtrahierer"

7.0.10 Register

In der letzten Übung haben wir mehr oder weniger ein ein Register als BOOLEsche Liste modelliert, z.B. für 4 bit: |0|1|1|0| = `(list false true true false)`
Neben Verknüpfungen zweier Register, wie die Addition im letzten Beispiel, können wir auch Operationen auf einem Register realisieren, wie z.B. das „Linksschieben": |0|1|1|0| -> |1|1|0|0|
`(ASL (list false true true false)) -> (list true true false false)`
Dabei können wir als Namen die Bezeichnungen des 6502-Assemblers übernehmen:

ASL = „Arithmetic shift left" = 1-bit-Linksschieben mit Nullauffüllung
LSR = „Logical shift right" = 1-bit-Rechtsschieben mit Nullauffüllung
ROL = „Rotate left" = 1-bit-Linksrotieren
ROR = „Rotate right" = 1-bit-Rechtsrotieren

Die Modellierung von Registern mittels Listen erweist sich hier als ideal:

```
;ASL: register --> register
(define (ASL register)
  (cond
    ((empty? register) empty)
    (else
     (append (rest register) (list false)))))
```

oder

```
;LSR: register --> register
(define (LSR liste)
  (reverse (ASL (reverse liste))))
```

Übungen

1. Entwickle

   ```
   ;ROL: register --> register
   ;ROR: register --> register
   ```

2. Entwickle

   ```
   ;ASLn: register n --> register
   ;LSRn: register n --> register
   ```

   (Verschiebung um n Stellen nach links bzw. rechts)

8 Zustandsmodellierung

8.1 Automaten und Zustände

Die Arbeitsweise eines Getränkeautomaten, der nur Limo für 30 Ct ausgibt und nur 10-Cent und 20-Cent-Münzen annimmt und keine Geldrückgabe kennt, kann man leicht durch folgendes Diagramm veranschaulichen:

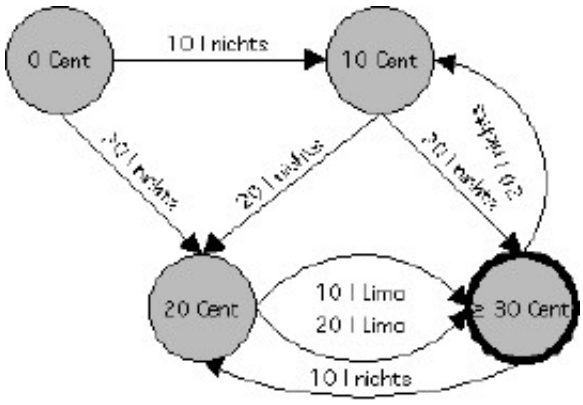


Abb. 8.1: Automatendiagramm

Wir sehen, dass die Ausgabe des Automaten (hier nichts oder Limo) davon abhängt,

1. in welchem Zustand (hier die „Ladung"(= der bisher eingeworfene Geldbetrag): 0 Cent, 10 Cent, 20 Cent oder $\geq$ 30 Cent) er sich befindet
2. welche Eingabe (hier 10 oder 20) erfolgt

Ebenso hat der aktuelle Zustand (die Kreise im Diagramm) des Automaten die gleichen Abhängigkeiten wie die Ausgabe, nämlich

1. der bisherige Zustand
2. die Eingabe

Wie können wir den Automaten simulieren ?

Funktionaler Ansatz

```
;Anfangszustand des Limo-Automaten
(define zustand '0)
;zustands-funktion: zustand eingabe --> folge-zustand
(define (zustands-funktion zustand eingabe)
  (cond
    ((equal? zustand '0)
     (cond
       ((equal? eingabe 10) '10)
       (else '20)))
    ((equal? zustand '10)
     (cond
       ((equal? eingabe 10) '20)
       (else '>=30)))
```

```
    ((equal? zustand '20) '>=30)
    ((equal? zustand '>=30)
     (cond
       ((equal? eingabe 10) ''10)
       (else '20)))))

;ausgabe-funktion: zustand eingabe --> ausgabe
(define (ausgabe-funktion zustand eingabe)
  (cond
    ((equal? zustand '10)
     (cond
       ((equal? eingabe 10) "nichts")
       (else "Limo")))
    ((equal? zustand '20) "Limo")
    (else "nichts")))
```

Mit diesen beiden Funktionen können wir allerdings nur einen „Schritt" des Automaten beschreiben: die Reaktion auf einen einzigen Münzeinwurf, aber in der Realität erfolgen mehrere Einwürfe. Wir können also mit Hilfe der o.a. Funktionen aus einem bestimmten Zustand und einer Eingabe (Münze) den Folgezustand und die Ausgabe bestimmen, aber die Funktionen können sich für die nächste Münze den letzten Zustand nicht merken ! Aber mit welchem Programm/Funktion können wir jetzt unser Modell interaktiv bedienen ? D.h. wie können wir z.B. festellen, ob man bei einer bestimmten Folgen von Eingaben (Münzen) eineLimo bekommt ? Das Programm muss alle Eingaben, d.h. alle Münzen wissen !
Eine (Not-)Lösung stellt die Eingabe aller Münzen in Form einer Liste dar:

```
;limo-automat: eingabe-folge (liste) --> ausgabe
(define (limo-automat zustand eingabe-folge)
  (cond ((empty? eingabe-folge) "nichts eingeworfen")
        ((empty? (rest eingabe-folge)) (ausgabe-funktion zustand (first eingabe-folge)))
        (else
         (limo-automat (zustands-funktion zustand (first eingabe-folge))
         (rest eingabe-folge)))))
;Aufrufe:
(limo-automat '0 (list 10 10 10))       --> "Limo"
(limo-automat '0 (list 20 10))  --> "Limo"
(limo-automat '0 (list 10 20 10))       --> "nichts"  (warum ?)
```

Besser wäre es, wenn sich unser Programm den jeweiligen Zustand merken oder „speichern" könnten, d.h. die hier benutzte „Variable" `zustand` auch wirklich **variabel** und nicht an einen festen Wert gebunden wäre.
Diese Möglichkeit gibt es tatsächlich, wie wir in der nächsten Version sehen werden:

„Imperativer" Ansatz

Wir beginnen wieder mit

```
;Anfangszustand des Limo-Automaten
(define zustand '0)
```

Die Funktion `(zustands-funktion ...)` ersetzen wir durch `(aendere-zustand ...)`

```
;aendere-zustand: eingabe-zustand --> folge-zustand
(define (aendere-zustand eingabe)
  (cond
    ((equal? zustand '0)
     (cond
       ((equal? eingabe 10) (set! zustand '10))
       (else (set! zustand '20))))
    ((equal? zustand '10)
     (cond ...
```

usw.
Die entscheidende Änderung ist hier die Form

`(set! <variable> <wert>)`
Sie weist der Variablen zustand einen (neuen) Wert zu; man spricht von einer *Zuweisung* oder Mutation [1]: Während z.B. in der funktionalen Variante die Variable während kompletten Programmablaufs an den Wert 'z0 gebunden bleibt, kann er mit `set!` immer wieder geändert werden.
Der Begriff „Variable" entspricht jetzt seiner Wortbedeutung. Genau genommen ist *aendere-zustand* gar keine Funktion mehr, da sie keinen Wert liefert, vielmehr löst sie eine Aktion in Form von `(set! ...)` aus, und ist damit für eine Substituion nicht mehr brauchbar.

Unverändert übernehmen wir

```
;ausgabe-funktion: zustand eingabe --> ausgabe
(define (ausgabe-funktion zustand eingabe)
  (...
```

Die Bedienung des Automaten simulieren wir durch das Endlos-Programm(!):

```
;limo-automat: --> ausgabe
(define (limo-automat)
  (cond ((< 3 4)
         (begin
           (local ((define s (read)))
             (begin
               (display (ausgabe-funktion zustand s))
               (aendere-zustand s)
               (newline)))
           (limo-automat)))))
```

Das bedeutet

1. die Tastatureingabe wird mit der Funktion (read) gelesen und an s gebunden
2. mit `(display (ausgabe-funktion zustand s))` wird die Ausgabe im Interaktionsfenster angezeigt
3. durch `(aendere-zustand s)` wird der Wert von `zustand` geändert
4. mit `(newline)` wird einen Zeilenvorschub augelöst

Der Aufruf `(limo-automat)` liefert im Interaktionsfenster einen Dialog der Form

```
10
nichts
20
Limo
```

in der sich Eingabe über die Tastatur durch `(read ...)` und Ausgabe durch `(display ...)` abwechseln.
Die „Funktions"-Aufrufe (2) bis (4) liefern keinen Wert, sondern lösen Aktionen aus. Deshalb werden sie auch Anweisungen oder „*Befehle*" genannt. Damit sie direkt nacheinander ausgeführt werden können, brauchen wir eine neue syntaktische Struktur: Die Form

```
(begin
        <Anweisung1>
        <Anweisung2>
        ...)
```

heißt „*Sequenz*", wodurch mehrere Anweisungen zu einer Anweisung zusammengefasst werden.
Anweisungen und Sequenzen sind typische Merkmale einer **imperativen Programmierung**.

Fazit

Offensichtlich gelingt eine Programmierung des Getränkeautomaten im imperativen Stil besser; das liegt u.a. daran, dass ein System aus Zuständen, die sich ändern können, mit „echten" Variablen, also solchen, deren Wert mit Hilfe einer Zuweisungen verändert werden können, beschrieben werden

[1] *mutare* (lat.) = ändern

können.
Daher verallgemeinern wir unserern Getränkeautomaten: Ein allgemeines System, das aus Zuständen sowie aus Ein- und Ausgaben besteht, wird in der Informatik auch als *Automat* bezeichnet.
Die einfachste Automatentyp ist folgender:

Ein **endlicher Automat** (EA) besteht aus

▷ einem (endlichen, nichtleeren) *Eingabealphabet E*

▷ einem (endlichen, nichtleeren) *Ausgabealphabet A*

▷ einer (endlichen, nichtleeren) *Zustandsmenge Z* = z0, z1, .. zn, in der es einen besonderen Zustand, den *Anfangszustand z0* gibt.

▷ einer *Überführungsfunktion ü: Z x E –> Z*, die jedem Zustand in Abhängigkeit von der Eingabe einen neuen Zustand zuordnet

▷ einer *Ausgabefunktion a: E x Z –> A*, die jeder Eingabe in Abhängigkeit vom aktuellen Zustand eine Ausgabe zuordnet

Abb. 8.2: Endlicher Automat

Darstellen kann man einen EA z.B. durch einen *Automatendiagramm* (s.o.) oder einer Automatentafel, wie im folgenden gezeigt:
Bei unserem Getränkeautomaten sind

- E = {10, 20 }, A = {"nichts", "Limo"}, Z = {'0, '10, '20, '≥30} und z0 = '0
- die Wertetabelle der Überführungsfunktion ist

Zustand	Eingabe 10	Eingabe 20
'0	'10	'20
'10	'20	'≥30
'20	'≥30	'≥30
'≥30	'20	'10

- die Wertetabelle der Ausgabefunktion ist

Zustand	Eingabe 10	Eingabe 20
'0	"nichts"	"nichts"
'10	"nichts"	"Limo"
'20	"Limo"	"Limo"
'≥30	"nichts"	"nichts"

oder man kann Überführungsfunktion und Ausgabefunktion auch in einer Tabelle, der Automatentafel, zusammenfassen:

Zustand	Eingabe 10	Eingabe 20
'0	'10 | "nichts"	'20 | "nichts"
'10	'20 | "nichts"	'30 | "Limo"
'20	'≥30 | "Limo"	'≥30 | "Limo"
'≥30	'20 | "nichts"	'10 | "nichts"

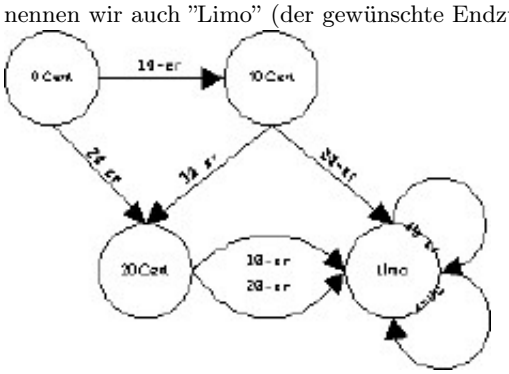
Grundsätzlich gibt es EA in zwei Varianten: das oben beschriebene Modell heißt *Transduktor*; daneben gibt es auch den sog. *Akzeptor*, der sich vom Transduktor wie folgt unterscheidet:

- es gibt keine Ausgabe (also kein Ausgabealphabet und keine Ausgabefunktion)
- statt dessen gibt es sog. *Endzustandsmenge* $ZE = \{ze_1, ze_2, ..ze_n\} \subseteq Z$, d.h. man sagt, dass der Automat eine bestimmte Eingabe akzeptiert, wenn er durch sie in den *Endzustand* gelangt.

Dazu auch ein Beispiel in Übung 1:

Übungen

1. Wir modifizieren unseren Limo-Automaten geringfügig: es interessieren uns nur die Zustände; den Zustand, in dem genug Geld eingegeben wurde, d.h. der Automat ist zur Ausgabe einer Limo bereit, nennen wir auch "Limo" (der gewünschte Endzustand):

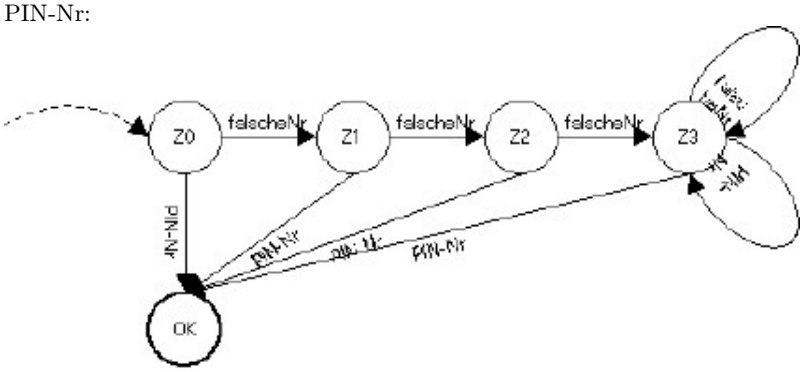


   a) Erläutere die unterschiedliche Funktionsweise im Vergleich zum ursprünglichen Modell.
   Zunächst wollen wir diesen Automaten funktional programmieren; zur Abwechslung wählen wir folgende Datenstruktur:

```
(define-struct uebergang (alter-zustand eingabe neuer-zustand))
(define limo-automat (list
            (make-kante "0 Cent" "10-er" "10 Cent")
            (make-kante "0 Cent" "20-er" "20 Cent")
            (make-kante "10 Cent" "10-er" "20 Cent")
            (make-kante "10 Cent" "20-er" "Limo")
            (make-kante "20 Cent" "10-er" "Limo")
            (make-kante "20 Cent" "20-er" "Limo")
            (make-kante "Limo" "10-er" "20 Cent")
            (make-kante "Limo" "20-er" "10 Cent")))
```

   b) Entwickle eine Funktion *;gib-folge-zustand: automat zustand uebergang –> zustand* z.B. `(gib-folge-zustand ea "10 Cent" "10-er") -> "20 Cent"`
   c) Entwickle eine Funktion *;gib-alle-zustände: automat anfangszustand eingabe-liste –> liste*, wobei *eingabe-liste* eine Liste aller getätigten Eingaben darstellt: z.B. `(gib-alle-zustaende ea "0 Cent" (list "10-er" "20-er")) -> (list "10 Cent" "Limo")`
   d) Entwickle eine Funktion *;limo?: automat eingabe-liste –>BOOLEAN*, die feststellt, ob der Automat bei einer bestimmten Eingabefolge in den Endzustand "Limo" gerät z.B. `(limo? ea (list "10-er" "20-er")) -> true`
   e) Entwickle eine imperative Variante zu diesem Automaten

2. Ein typisches Beispiel für einen Akzeptor ist die Erkennung einer EC-Karte am Geldautomaten mittels PIN-Nr:



a) Erläutere, inwieweit dieses Diagramm der Realität entspricht
b) Gib E, Z, und ZE an.
c) Entwickle ein imperatives Simulationsprogramm.

3. Was ist, wenn die Zeilen
```
(display (ausgabe-funktion zustand s))
(aendere-zustand s)
```
vertauscht werden ?

4. Eine Mausefalle kann man im einfachsten Fall als EA (Akzeptor) auffassen. Zustände: z0 = Falle gespannt (Anfangszustand), z1 = Falle nicht gespannt (Endzustand), Eingaben: mk = Maus kommt, mn = Maus kommt nicht
   a) Fertige ein Automatendiagramm an
   b) Entwickle ein Programm (imperativ)
   c) Modelliere die Mausefalle als Transduktor, indem Du die als Ausgabe mt (Maus tot) und mn (Maus nicht tot) hinzunimmst; erstelle dazu ein Automatendiagramm und ein Programm

8.2 Imperative Programmierung

Fassen wir die Ergebnisse des letzten Abschnitts zusammen:

Eine **imperative Programmierung** ist gekennzeichnet durch

▷ *Variable* sind „echte" Variable: sie können durch Mutatoren oder Zuweisungen wie z.B. `set!` ihren Wert während des Programmablaufs ändern

▷ Es gibt (außer bestimmten syntaktischen Spezialformen wie z.B. `define`) auch vordefinierte Formen, die keine Funktionen sind, d.h. keinen Funktionswert liefern, sondern den Wert von Variablen ändern; dazu gehören z.B. `set!, setfirst!, setrest!, append!` usw.
Es handelt sich um sog. *Zuweisungen*; in Scheme wird ihr Bezeichner durch ein Ausrufezeichen ! gekennzeichnet.
Es können auch „Funktionen" selbst definiert werden, die keinen Funktionswert liefern; im Unterschied zu echten Funktionen werden sie in vielen Programmiersprachen *Prozeduren* genannt. Der Aufruf einer vor- oder selbstdefinierten Prozedur wird im folgenden auch Anweisung genannt. Anweisungen liefern keinen Werte, sondern lösen Aktionen aus, weshalb sie auch „Befehle" (*imperare*, lat. = befehlen) heißen.
In diesem Fall wird beim Funktionsvertrag hinter den Zuweisungspfeil –> häufig *void* geschrieben.

▷ Um mehrere Anweisungen hintereinander ausführen zu können, wird ein syntaktisches Konstrukt benötigt, das die Anweisungen zu einer sog. zusammengesetzten Anweisung oder Anweisungsfolge (Sequenz) bündelt:
```
(begin
<Anweisung1>
<Anweisung2>
...)
```

▷ Ein „imperatives" Programm besteht aus einer Abfolge von Anweisungen, wobei die Reihenfolge i.a. nicht verändert werden darf

Hinweis

Durch Zuweisungen innerhalb einer selbst definierten Prozedur können z.B. die Werte von globalen Variablen geändert werden; dies kann auch unbeabsichtigt geschehen: man spricht von *Seiteneffekten*. Daher ist das Substitutionsprinzip der funktionalen Programmierung bei imperativer Programmierung nicht mehr gültig ! (vgl. Übung 1)

Beispiel

Wir wollen ein einfaches elektronisches Telefonbuch erstellen und verwalten, dessen Einträge jeweils aus dem Vornamen und der Telefonnummer von Personen besteht: `(define mein-telefonbuch (list (list 'Adam 4711) (list 'Eva 4712)))`
Unser Simulationsprogramm soll zwei Möglichkeiten bieten:

1. Suchen der Nummer anhand des Vornamens. etwa durch

```
;nachschauen: liste symbol --> zahl oder false
(define (nachschauen telbuch name)
        (...
```

2. Hinzufügen einer neuen Person mittels Vornamen und Nummer

```
;hinzufuegen: symbol zahl --> void
(define (hinzufuegen name nummer)
        (...
```

Eine mögliche Interaktion könnte dann folgendermaßen verlaufen:

```
> (nachschauen mein-telbuch 'Adam)      --> 4711
> (nachschauen mein-telbuch 'Erna)      --> false
> (hinzufuegen 'Erna 4715)
> (nachschauen mein-telbuch 'Erna)      --> 4715
```

Dabei fällt folgendes auf:
Der Funktionsaufruf `(nachschauen mein-telbuch 'Erna)` liefert beim ersten Mal `false` , beim zweiten Mal `4715` !
Dies widerspricht fundamental unserem bisherigen fuktionalen Programmiergrundsatz, wonach eine Funktion mit gleichen Parametern stets den gleich Wert liefert, also muß in vorliegendem Fall die Variable `mein-telbuch` geändert worden sein, und zwar durch den Aufruf `(hinzufuegen 'Erna 4715)` .
Demnach ist `hinzufuegen` keine echte Funktion, sondern eine Prozedur; in diesem Falle schreiben wir im Funktionsvertrag statt eines Wertetyps den Begriff *void*.
(1) ist leicht realisiert mittels

```
;nachschauen: liste symbol --> zahl oder false
(define (nachschauen telbuch name)
  (cond
    ((empty? telbuch) false)
    (else
     (cond
       ((symbol=? (first (first telbuch)) name) (first (rest (first telbuch))))
       (else
        (nachschauen (rest telbuch) name))))))
```

Für (2) ergibt sich

```
;hinzufuegen: symbol zahl --> void
(define (hinzufuegen name nummer)
        (set! mein-telbuch (cons (list name nummer) mein-telbuch)))
```

Für die praktische Nutzung dieses Programmes muß man im Interaktionsfenster - ähnlich wie anfangs gezeigt - abwechselnd die Aufrufe `(nachschauen telbuch name)` und/oder `(hinzufuegen name nummer` eingeben. Dies ist natürlich sehr unpraktisch, weil reale Anwenderprogramme i.a. kein Interaktionsfenster haben.
Lösung: Programm mit GUI (graphische Benutzeroberfläche), s. nächstes Kapitel

Übungen

1. Es gibt auch typenspezifische Mutatoren wie z.B. set-first! und set-rest! für Listen: Wir betrachten

```
(define a (list 1 2))
(define b a)
a                        --> (list 1 2)
b                        --> (list 1 2)
(set-first! a 10)
a                        --> (list 10 2)
b                        --> (list 10 2)
```

Offensichtlich ist durch `(set-first! a 10)` auch der Wert von b verändert worden, d.h. wir haben ein Beispiel für (i.a. unerwünschten) Seiteneffekt. Wir definieren `(define x (list 'a 'b))`. Wie wird x durch

- `(set-first! (rest x) (list 1 2))`
- `(set-rest! (rest x) (list 1))`

jeweils geändert ?

2. Wir definieren

```
(define max-geschwindigkeit 75)
(define toleranz-faktor 1.1)
(define meine-fahr-geschwindigkeit (* toleranz-faktor max-geschwindigkeit))
```

und erhalten `meine-fahr-geschwindigkeit -> 82.5`
Was erhalten wir nach `(set! max-geschwindigkeit 75)` für `meine-fahr-geschwindigkeit` ?

3. Sei `(define meine-liste (list 1 2 3))`
Gib nach jeder der folgenden Zuweisungen den Wert von `meine-liste` an:

```
(set-first! meine-liste 10)
(set-rest! meine-liste (list 2 4))
(set-rest! (rest meine-liste) meine-liste)
```

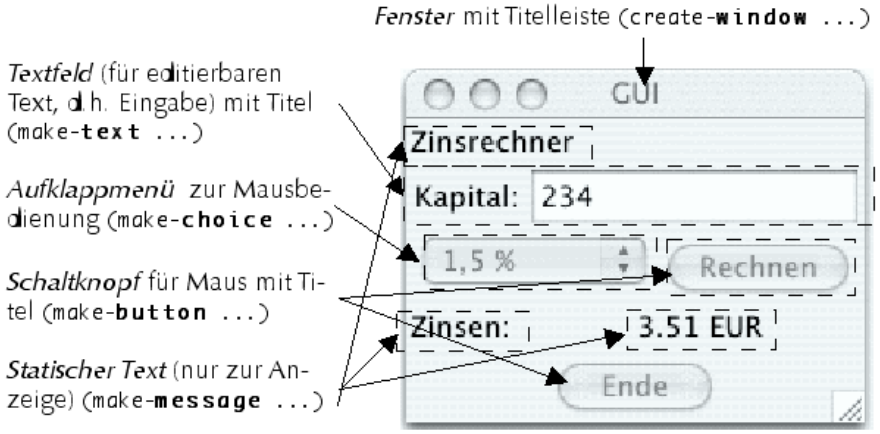
9 Modellierung mit Interaktion - `gui.ss`

9.1 Elemente einer graphischen Benutzeroberfläche (GUI)

Das, was wir heutzutage auf dem Bildschirm eines Computers sehen, sind nicht mehr - wie in den Anfangsjahren – 24 Zeilen mit durchgängigem (linearem) Text oder Zahlen, sondern besteht aus „Fenstern", die ihrerseits strukturiert sind durch graphische Elemente, die möglichst selbsterklärend sein sollen: Felder zum Eingeben über die Tastatur, Felder zum Anzeigen von Text oder Zahlen, Symbole und Bildchen zum Anklicken mit der Maus usw.: All diese graphischen Objekte bilden zusammen mit der sog. Ereignissteuerung die sog. Graphischen Benutzeroberfläche (GUI = *Graphical User Interface*).
Mit *DrScheme* können solche Programme mit GUI erstellt werden, und zwar in verschiedenen Schwierigkeitsgraden. Für Einsteiger ist das Teachpack `gui.ss` vorgesehen, das zwar im Hinblick auf die Vielfalt der GUI-Objekte stark eingeschränkt ist, dafür aber umso leichter zu handhaben ist.
Wir untersuchen die Möglichkeiten von gui.ss an einem einfachen Beispiel:

Abb. 9.1: GUI-Beispiel

In diesem Beispiel kommen alle fünf möglichen GUI-Objekte von `gui.ss` kommen hier zum Einsatz:



Am Beispiel dieses „Zinsrechner"-Programms schauen wir uns die Erstellung eines GUI-Programms im Detail an, es gliedert sich in 3 Abschnitte:

1. Erstellung
   - der GUI-Objekte, *Zeilen 5 bis 13 und 22 bis 33*
   - der Event-Handlern: BOOLEsche Funktionen, die auf Ereignisse reagieren (z. B. Mausklick), *Zeilen 14 bis 21*
2. Definition sonstiger Funkionen: das „eigentliche" Programm , *Zeilen 36 bis 40*
3. Erzeugung des Programmfensters und Anordnung der GUI-Objekte, *Zeilen 42 bis 50*

```
1  ;Zinsrechner V. 1 - 16.10.2005
2  ;gui.ss und level: Mittelstufe
3  ;---------------------------------
4  ;*************************************************************
5  ; Erstellung der GUI-Objekte, die im Fenster erscheinen sollen:
6  (define titel-m (make-message "Zinsrechner"))
7  ; --> statischer Text (message), nicht editierbar, aber mit (draw-message ...) veraenderbar
8  ; --> dient hier als Titel
9  (define kapital-t (make-text "Kapital:"))
10 ; --> Textfeld = editierbarer (ueber die Tastatur) Text (text)
11 ; --> dient zur Eingabe
12 (define zinssatz-c (make-choice (list "1 %""1,5 %" "2 %" "2,5 %")))
13 ; --> Aufklappmen?, um ein Option (item) auszuw?hlen
14 (define (antwort ereignis)
15   (local
16       ((define zs (+ 1 (* 0.5 (choice-index zinssatz-c))))
17        (define kap (string->number (text-contents kapital-t))))
18     (draw-message zinsen-m (string-append (number->string
19                                            (zins-berechnung kap zs)) " EUR"))))
20 ; --> Ereignis-Handler (call-back-Funktion) = BOOLEsche Funktion
21 ; --> dient als Reaktion auf einen Mausklick auf den rechnen-b
22 (define rechnen-b (make-button "Rechnen" antwort))
23 ; --> Schaltknopf (button), reagiert auf Mausklick, genauer: Loslassen
24 ; --> l?st eine Reaktion aus, die durch einen Ereignis-Handler beschrieben wird
25 (define zins-m (make-message "Zinsen:"))
26 ; --> statischer Text (message), nicht editierbar, aber mit (draw-message ...) veraenderbar
27 ; --> dient hier als Titel
28 (define zinsen-m (make-message " "))
29 ; --> statischer Text (message), nicht editierbar, aber mit (draw-message ...) veraenderbar
30 ; --> dient hier zur Anzeige/Ausgabe des Ergebnisses
31 (define ende-b (make-button "Ende"(lambda (ereignis) (hide-window fenster))))
32 ; --> Schaltknopf (button), reagiert auf Mausklick, genauer: Loslassen
33 ; --> l?st eine Reaktion aus, die durch einen Ereignis-Handler beschrieben wird
34
35 ;*************************************************************
36 ; evt. sonstige Funktionen und Aufrufe:
37 ;eigentliche Zinsberechnung:
38 ; zins-berechnung: kapital sazt --> zinsen
39 (define (zins-berechnung kapital satz)
40   (/ (round (* 100 (exact->inexact (* kapital (/ satz 100))))) 100))
41 ;*************************************************************
42 ; Erstellung des Programmfensters und geometrische Anordnung der GUI-Objekte:
43 (define fenster
44   (create-window
45    (list
46     (list titel-m)
47     (list kapital-t)
48     (list zinssatz-c rechnen-b)
49     (list zins-m zinsen-m)
50     (list ende-b))))
```

Die geometrische Anordnung der GUI-Objekte im Programmfenster ist durch eine Liste von unsichtbaren vertikalen Zeilen gegeben, wobei jede Zeile wieder eine Liste von horizontal angeordneten GUI-Objekten ist.

`rectangle` : *Number Number Mode Color -> Image*
zeichnet ein Rechteck mit Breite, Höhe, Zeichenmodus und Farbe

`show-window` : *-> true*
Zeigt das Fenster

`hide-window` : *-> true*
Versteckt das Fenster

`create-window` : *(listof (listof gui-item)) -> true*
Fügt eine Liste von GUI-Objekten dem Fenster hinzu und macht es sichtbar

`make-button` : *str (event% -> boolean) -> gui-item*
Erstellt einen Knopf als Schaltfläche mit Beschriftung und Ereignisbehandlung

`make-message` : *str -> gui-item*
Erzeugt ein Objekt für statischen Text („messsage"-Objekt)

`draw-message` : *gui-item[message%] stri -> true*
Zeigt eine Zeichenkette in einem message-Objekt an

`make-text` : *str -> gui-item*
Erzeugt ein Textfeld (für editierbaren Text) mit Beschriften

`text-contents` : *gui-item[text%] -> str*
Liefert den Inhalt (Zeichenkette) eines Textfeldes

`make-choice` : *(listof str) -> gui-item*
Erzeugt aus einer Liste von Zeichenketten (als Optionen) ein aufklappbares Menü (Mehrfach-Auswahl), aus der man eine Option auswählen kann

`choice-index` : *gui-item[choice%] -> num* Liefert die ausgewählte Option einer Auswahl als Index-Nr. (bei 0 beginnend)

Hinweise zu Ein-/Ausgabe

Alles, was über die Tastatur eingegeben wird, muß in ein Textfeld eingegeben werden und wird dort als Zeichenkette (string) behandelt. Der Zugriff erfolgt über die Funktion `(text-contents ...)`, die wiederum eine Zeichenkette; das gilt auch für Zahlen, die eingegeben werden. Deshalb wird man öfters die Funktion `(string->number ...)` benötigen.
Umgekehrt wird alles als Zeichenkette mittels der Funktion `(draw-message ...)` in Feld für statischen Text, sog. message geschrieben; Zahlenwerte müssen also mit `(number->string ...)` umgewandelt werden. Will man in der Anzeige keine echten Brüche sehen, sondern Dezimalzahlen, sollten sie vorher mit `(exact->inexact ...)` umgewandelt werden.

Übungen

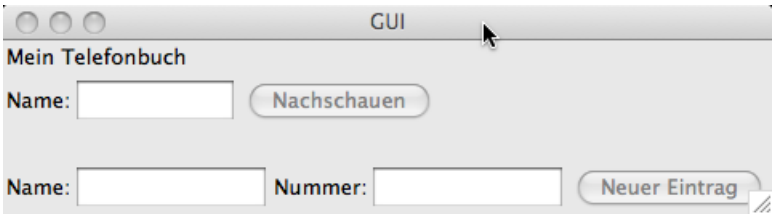
Vorbemerkung
Umfangreiches Aufgaben- und Übungsmaterial zur GUI-Erstellung erhält schon dadurch, dass man die Beispiele und Übungen der vorangegangenen Kapitel mit einer GUI versieht (natürlich nur, nachdem man geprüft hat, ob eine solche jeweils sinnvoll ist).
Von daher sind einige der folgenden Aufgaben lediglich Varianten bekannter Aufgaben, jetzt halt mit GUI

1. Gegeben ist ein Programmfenster:

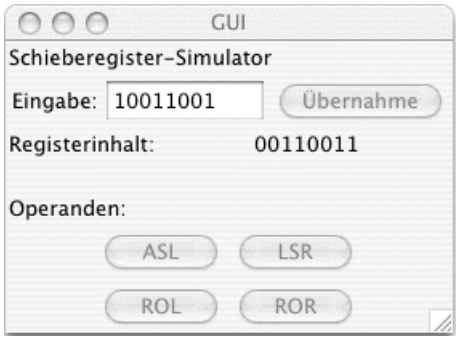
a) Erkläre die (vermutete) Funktionsweise des Programms
b) Gib Anzahl und Art der GUI-Objekte an
c) Erstelle einen passenden Quellcode

2. Entwickle ein passendes Programm:

3. Modelliere eine GUI für die Frontseite (= Bedien-Seite)
   a) eines Getränkeautomaten
   b) eines Fahrkartenautomaten
4. Stelle das Beispiel „Telefonbuch" aus dem Kapitel „imperative Programmierung" als GUI-Programm dar:



5. Wir greifen das Beispiel „Register" auf und versehen es mit einer GUI:



Beachte, dass wir u.a. zusätzlich die Umwandlungs-Funktionen
`;string->wliste: bitstring -> wliste (string->wliste "1011") -> (list true false true true)`
und
`;wliste->string bl -> string (wliste->string (list true false true true) -> "1011"`
benötigen (warum ?)

6. Wir greifen das Beispiel „Binomialkoeffizienten" von früher auf und versehen es mit einer GUI, um das bekannte Pascalsche Dreieck zu erhalten:


```
1  ;-----Modell
2  ;binkoeff: n --> Liste der Binomialkoeffizienten Bn;k fuer k=0,1,2.....,n
3  (define (binkoeff n)
4    (cond
5      ((zero? n) (list 1))
6      (else
7       (local
8           ((define liste (binkoeff (-  n 1))))
9         (map + (cons 0 liste) (append liste (list 0)))))))
10
11 ;VIEW: Fenster fuer Koeffizientenliste inkl. GUI-Objekte-----------
12 ;item-liste: zahlenliste --> Liste von messages
13 (define (item-liste l)
14   (cond
15     ((empty? l) (list (make-message " "))) ;wegen opt. Darstellung (linker Rand)
16     (else
17      (cons (make-message (number->string (first l))) (item-liste (rest l))))))
18
19 ;gesamtliste
20 (define (gesamtliste m)
21   (local ((define liste (binkoeff m)))
22     (cond
23       ((zero? m) (list (list (make-message " ") (make-message "1"))))
24       (else
25        (cons (reverse (item-liste liste)) (gesamtliste (- m 1)))))))
26
27 ;koeff-aktion: e  --> void (BOOLEsch als callback-Funktion fuer start-b-Knopf)
28 (define (koeff-aktion e)
29   (local
30       ((define n (string->number (text-contents anzahl-t))))
31     (begin
32       (create-window
33        (cons (list (make-message
34                     (string-append "Pascal-Dreieck fuer n = " (number->string n) ":")))
35              (reverse (gesamtliste n))))
36       true
37       )))
38
39 ;VIEW: Hauptfenster------------
40 (define titel-m (make-message "      Pascal-Dreieck        "))
41 (define anzahl-t (make-text "fuer n: "))
42 (define start-b (make-button "Erstellen" koeff-aktion))
43
44 ;Hauptfenster gestalten:
45 (define fenster (create-window
46                  (list
47                   (list titel-m)
48                   (list anzahl-t)
49                   (list start-b))))
```

7. Erstelle zum Programm zur grafischen Darstellung der Binomialverteilung (7.0.5) eine GUI, mit der die Paramter n und p sowie die Farbe der Balken geregelt werden können.

9.2 Interaktion & Grafik

Für die interaktive Steuerung von Programmen mit zeichnerischen Anwendungen, z.B. Kurvenplotter, ist die Kombination eines Zeichenfensters (canvas) mit einer interaktiven Steuerfensters (GUI) gut geeignet; neben dem Teachpack `gui.ss` benötigen wir entweder `draw.ss`(imperativ) oder `world.ss`(funktional).

Ausführliches Beispiel: Einfacher Funktionsplotter

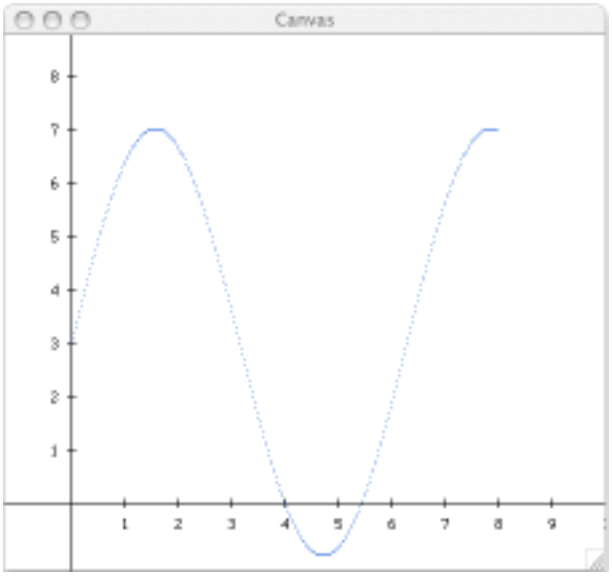
Forderungen:

1. Eingabe des Funktionsterms in Präfix (wie in *DrScheme*)
2. Angabe des Intervalls [a; b]
3. Durch Mausklick wird das Zeichnen in ein Zeichenfenster (canvas) gestartet
4. Der Inhalt des Zeichenfenster kann durch Mausklick gelöscht werden.

   Das "Steuerungsfenster" (GUI) könnte damit folgendermaßen aussehen:

   Der Einfachheit halber machen wir zusätzliche Annahmen:

5. Zoomen ist nicht möglich; es gilt - x-Achse: $0 \leq a \leq x \leq b \leq 10$ - y-Achse: gleicher Maßstab wie x-Achse, aber durch Fenster beschränkt und erwarten etwa folgende Darstellung im Zeichenfenster (canvas):



Wie sieht die Realisierung aus ?
Zunächst machen wir uns klar, dass der Zeichen-Button 2 Dinge auslösen soll:
- Zeichnen des Koordinatensystem
- Zeichnen der Kurve

Also etwa:

```
...
(define (antwort1 e)
  (zeichnen e))
(define zeichne-button (make-button "Zeichnen" antwort1))
...
```

mit

```
...
(define (zeichnen x)
  (begin
   (koord-zeichnen x)
   (kurve-zeichnen x)))
...
```

Spätestens jetzt wird es Zeit für globale Variablen für unsere Abmessungen, da u.a. die Kurve in Bezug um Koordinatensystem steht, etwa

```
...
;globale Variablen f. canvas:
(define cbreite 450)     ; Breite der canvas
(define choehe 400)      ; Höhe der canvas
(define x0 50)           ; (x0 | y0) = Koordinatenursprung
(define y0 (- choehe x0))
(define m 40) ;Auflösungsfaktor in x- und y-Richtung
...
```

Den Faktor m brauchen wir, weil uns für 10 Einheiten auf der x-Achse 400 Pixel auf dem Bildschirm zu Verfügung stehen, also m = 40 Pixel pro Einheit.
Das Zeichnen des Koordinatensystems besteht aus

- Zeichnen der beiden Achsen
- Skalenstriche auf beiden Achsen
- Skalenbeschriftung beider Achsen

also etwa

```
(define (koord-zeichnen x)
  (local
      ((define (xskala n)
         (cond
           ((<= n 0)  true)
           (else
            (begin
             (draw-solid-line
              (make-posn (+ x0 (* n m)) (- y0 3)) (make-posn (+ x0 (* n m)) (+ y0 3)))
             (draw-solid-string
              (make-posn (+ x0 (* n m) -2) (+ y0 20)) (number->string n))
             (xskala (- n 1))))))
       (define (yskala n)
      ....
      ....
    (begin
     (draw-solid-line (make-posn 0 y0) (make-posn cbreite y0))  ;x-Achse
     (draw-solid-line (make-posn x0 0) (make-posn x0 choehe))  ;y-Achse
     (xskala 10)
     (yskala 9))))
```

Der kniffligste Teil ist das Zeichnen der Kurve:

```
(define (kurve-zeichnen z)
  (local ((define term (text-contents term-feld))       ;Funktionsterm als string !
          (define a (string->number (text-contents a-feld)))
          (define b (string->number (text-contents b-feld)))
          (define (plotten stelle)                              ;plottet ab dem x-Wert "Stelle"
       ...
                ...
                ...))
    (and
     (cond
       ((< a 0) (draw-message fehler-feld "a >= 0 nötig!"))
       ((> b 10) (draw-message fehler-feld "b <= 10 nötig!"))
       ((> a b) (draw-message fehler-feld "a <= b nötig!"))
       (else
        (begin
         (draw-message fehler-feld " ")
         (plotten a)))))))
```

wobei die Prozedur `(plotten stelle)` das Kernstück darstellt. Das Problem dabei ist, dass der Funktionsterm, etwa `(+ 3 (* 4 (sin x)))` als String vorliegt ! Eine Auswertung ist nur mit der Funktion `(eval-string <string>)` möglich, wenn es sich um einen regulären Scheme-Ausdruck handelt und die Variable vorher global definiert wurde, z.B.

```
(define x 0.5)
(eval-string "(+ 3 (* 4 (sin x)))") --> 4.917702154416812
```

Leider ist diese Funktion in HtDP-Lernsprachenlevels nicht verfügbar, deshalb

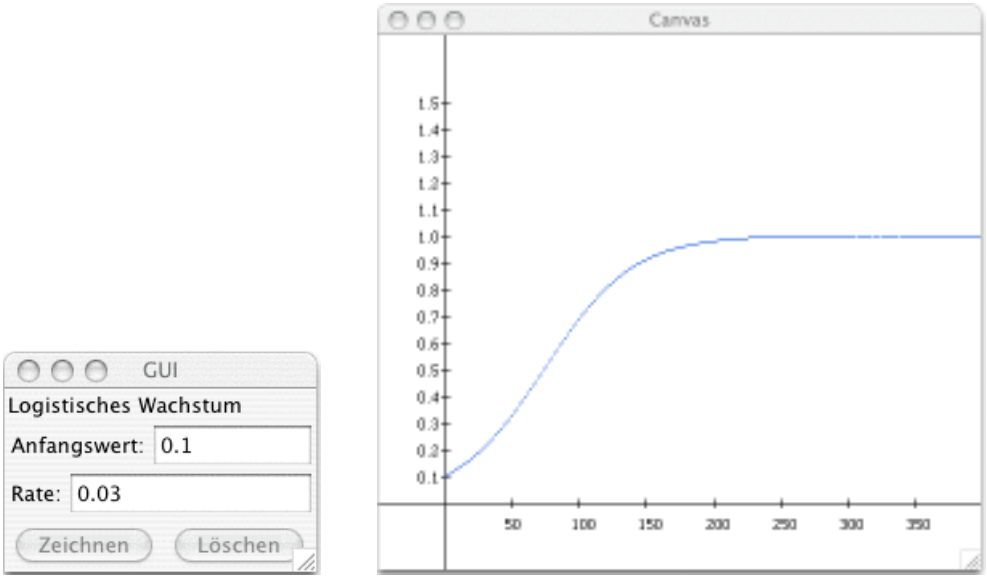
Wichtiger Hinweis
Bei diesem Beispiel muß wegen der Benutzung `(eval-string ...)` ausnahmsweise auf das PLT-Sprachlevel "Pretty Big" (in älteren Version von *DrScheme*) bzw. „Kombo" (neuere Versionen) gewechselt werden !

In unserem Fall muss also die Variable x vor der Definition von `(kurve-zeichnen ...)` global definiert und zunächst an einen beliebigen Zahlenwert gebunden; in der Prozedur `plotten` wird dann x stets gändert:

```
(define (plotten stelle)
  (begin
    (set! x stelle)
    (cond
      ((>= x b) true)
      (else
       (begin
         (draw-solid-line
          (make-posn (+ x0 (* x m)) (- choehe x0 (* (eval-string term) m)))
          (make-posn (+ x0 (* x m)) (- choehe x0 (* (eval-string term) m))) 'blue)
         (plotten (+ stelle (/ 1 m)))))))))
```

Übungen

1. (zu obigem Beispiel:)
   a) Ergänze den fehlenden Programmcode und bringe das Programm zum Laufen
   b) Man mache sich insbesondere die Prozedur `plotten` klar ! (Koordinatentransformation und Auflösungsfaktor bei `(draw-solid-line ...)`, Rekursionsaufruf und -abbruch, usw.)
   c) Finde Erweiterungen und Verbesserungen
2. Wir erinnern uns an das *ogistische Wachstum* (Verhulst-Modell); auch hier bietet sich das Arbeiten mit zwei Fenstern an:



a) Entwickle ein passendes Programm (Auf Sprachlevel „Mittelstufe" machbar). Es empfiehlt sich Endrekursion für $p_{n+1} = p_n + rp_n(1 - p_n)$

b) Untersuche die Wachstumskurve für verschiedene Wachstumsraten $0,02 < r < 3$. Informiere Dich zum Thema „Deterministisches Chaos"

c) ...

3. Simuliere den Geldautomaten mit EC-Karten-Bedienung (aus dem vorigen Absschnitt „Automaten und Zustände") durch ein Programm mit GUI. Dabei sollen die Zustände durch Farben repräsentiert werden: Geldautomat als Rechteck in einem canvas-Fenster, wobei sich die Farbe des Rechtecks je nach Zustand ändert (z.B. orange für Anfangszustand, grün für Endzustand, rot für Fehlerzustand usw.:

Literaturverzeichnis

[1] *Felleisen, M.*, u.a., **How to Design Programs** - An Introduction to Programming and Computing, MIIT-Press, 2001, ISBN-Nr. 0-262-06218-6, online-Version (engl.): *http://www.htdp.org*

[2] *Abelson, H.*, u.a., **Struktur und Interpretation von Computerprogrammen**, Springer-Verlag Heidelberg, 1991, ISBN-Nr. 3-540-52043-0, Neuauflage 1996, ISBN-Nr. 3-540-63898-9, online-Version (engl.): *http://mitpress.mit.edu/sicp/full-text/book/book.html*

[3] *Klaeren, Herbert* und *Sperber, Michael*, **Vom Problem zum Programm** – Architektur und Bedeutung von Computerprogrammen –, 3. Auflage, Teubner, 2001, ISBN-Nr.0-519-22242-6

[4] *Wagenknecht, Christian*, **Programmmierparadigmen** – Eine Einführung auf der Grundlage von Scheme –, Teubner, 2004, ISBN-Nr.0-519-00512-3

Sonstige Quellen

Zu Scheme

- *http://www-pu.informatik.uni-tuebingen.de/info-i-y2k/scheme/*
- *http://www.schemers.org/*

Unterrichtsprojekte

- *http://www.deinprogramm.de/index.html* (Deutschland)
- *http://www.teach-scheme.org/* (USA)

Software

- Entwicklungswerkzeug *DrScheme* (kostenlos, für alle Betriebssysteme): *http://www.drscheme.org/*

Stichwortverzeichnis