Kapitel 2 der Vorlesung

Stand: 22.1.2008

Einführung in die Informatik III

(für Studierende des 3. Fachsemesters)

Pflichtvorlesung für die Diplomstudiengänge "Informatik" und "Automatisierungstechnik in der Produktion" sowie wählbare Vorlesung für weitere Studiengänge, z. B. für Mathematik, Softwaretechnik, Lehramt Informatik usw.

Universität Stuttgart, Wintersemester 2007/2008

Dozent: Volker Claus

2. Funktionales Programmieren

Vorbemerkungen

2.1 Einführung in Scheme

2.2 Beispielprogramme

2.3 Funktionen höherer Ordnung

2.4 Listen

2.5 Prinzipien der funktionalen Programmierung und zugehörige Programmiersprachen

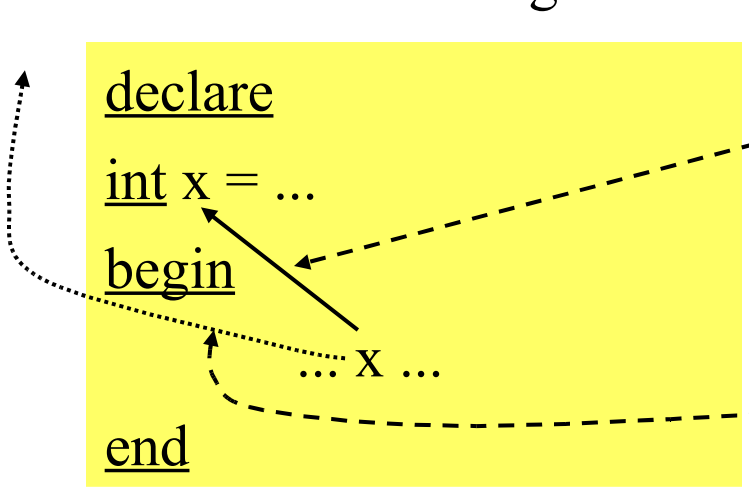
Vorbemerkungen

Bevor wir in die Programmierung mit Scheme einsteigen, sollten Sie sich einige grundlegende Begriffe nochmals klar machen wie zum Beispiel

- Bezeichner, Name, Variable
- Aufrufmechanismen (call by value / reference / name)
- statischer oder dynamischer Sichtbarkeitsbereich
- lokal, global
- Lebensdauer
- statische oder dynamische Bindung
- Typisierung
- Ausdruck (Term)
- Auswertungsreihenfolge von Ausdrücken

Der Gültigkeitsbereich (oder Sichtbarkeitsbereich, engl. scope) einer Variablen oder eines Bezeichners ist der Bereich in einem Programm, in dem die jeweilige Variable oder die Größe, die durch den Bezeichner benannt ist, direkt verwendet werden kann. In imperativen Sprachen (auch in Ada) beginnt der Gültigkeitsbereich genau mit der Deklaration und endet mit dem "end" des zugehörigen Blocks; hierbei werden aber alle die Programmbereiche im Inneren dieses Blockes ausgenommen, in denen die Variable bzw. der Bezeichner umdeklariert werden (dort ist die Größe nicht sichtbar, "lebt" aber weiter).
Die Variable ist hierbei stets lokal zu dem Block, indem sie deklariert wurde, und global in allen Unter-Blöcken, in denen sie sichtbar ist (in denen sie also nicht neu deklariert wurde).
Die Lebensdauer ist der Programmbereich ab der Deklaration bis zum "end" des zugehörigen Blocks.
Es gibt Sprachen, in denen man eine Größe nach Erreichen des "end" weiter leben lässt, sodass sie beim erneuten Eintritt in diesen Block oder diese Programmeinheit wiederum ihren alten Wert besitzt. Man kläre bei jeder Programmiersprache daher genau, welche Sichtbarkeit bzw. Lebensdauer mit eienr Deklaration verbunden ist.

Unter dem "static scoping" (oder "lexical scoping") versteht man, dass der Gültigkeitsbereich einer Variablen oder eines Bezeichners sich genau auf den Text des definierenden Programmbereichs bezieht; das heißt, die Größe ist höchstens dann sichtbar, wenn sich das Programm im Bereich des Programmtextes zwischen der Deklaration und dem zugehörigem "end" befindet und jede Verwendung des zugehörigen Bezeichners bezieht sich genau auf diese Deklaration.
Unter einem "dynamic scoping" versteht man, dass der Wert einer Variablen sich auf die Deklaration bezieht, die zur Laufzeit als letzte ausgeführt (und noch nicht beendet) wurde. Ein Bezeichner führt hierbei einen Stack an Referenzen auf seine zuletzt erfolgten Deklarationen mit sich; die oberste Deklaration (bzw. deren aktueller Wert) im Stack wird dann genommen.



Bei static scoping bezieht sich dieses Auftreten von x auf die Variable, die im kleinsten Block, der diese Textstelle umfasst, deklariert wurde.

Bei dynamic scoping bezieht sich das Auftreten von x auf die letzte ausgeführte Deklaration von x, bevor diese Stelle im Programm erreicht wurde.

Beispiel (Java-ähnlich formuliert):

```
int a = 1;
void g() {print (a);}
void h() {int a = 2; g();}
```

Static scoping liefert die Ausgabe 1 beim Aufruf h(), während dynamic scoping die Ausgabe 2 ergeben würde.

Java verwendet für Variablen static scoping, sodass folgendes Java-Programm (mit overloading des Bezeichners h) den Wert 1 liefert:

```
public class test {
    static int h = 1;
    static void g() {System.out.println(h);}
    static void h() {int h = 2; g();}
    public static void main (String[ ]  args) {
        h();
    }
}
```

Den Vorgang, die Referenz einer Variablen zu setzen, bezeichnet man als "binding" oder Bindung. Man spricht entsprechend von statischer oder dynamischer Bindung. Statische Bindung erfolgt also bereits, bevor ein Programm gestartet wird, während dynamische Bindung erst zur Laufzeit durchgeführt werden kann. Wichtig wird dies bei der Entscheidung, welche von mehreren gleich benannten Methoden ausgeführt werden soll, die in einer Klassenhierachie liegen. Dann ist die Methode zu nehmen, auf die man von der Klasse, zu der das Objekt aktuell gehört, beim Aufwärtsschreiten als erste stößt. Diese Methode kann nur "dynamisch" zur Laufzeit ermittelt werden.

Nochmals: Dynamische Bindung ist erforderlich, wenn der Typ einer Variablen (bzw. deren zugehörige Methode) zur Übersetzungszeit nicht feststeht. In Ada erfolgt diese nur in dem Fall, dass eine Variable an ein Objekt gebunden ist, dessen Lage in der Klassenhierarchie erst zur Laufzeit feststeht; meist handelt es sich um die genaue Auswahl einer überladenen Methode. In Java ist es ähnlich. Beispiel hierzu:

```
class Europaeer {
    void meinLand () {System.out.println("Europa"); }
}
 class Deutscher extends Europaeer {
    void meinLand () {System.out.println("Deutschland"); }
}
class Test {
   public static void main (String [ ] args) {
        for (int i=0; i<10; i++) {
            Europaeer Mensch;
            if (Math.random() > 0.5) Mensch = new Europaeer ();
            else Mensch = new Deutscher ();
            Mensch.meinLand();
        }
    }
}
```

Hier erfolgt die dynamische Bindung.

Als Ausgabe erhält man je nach gezogenen Zufallszahlen irgendeine Folge aus "Deutschland" und "Europa".

Unterschiedliche Verwendung des Begriffs "Variable".

In der Mathematik ist eine Variable ein Platzhalter, der durch einen aktuellen Wert überall in gleicher Weise ersetzt wird. Dabei unterscheidet man in Formeln zwischen freien und gebundenen Variablen, je nachdem ob die Variable quantifiziert ist oder nicht (also im Bereich eines Quantors ∃ oder ∀ steht).

In der Informatik bezeichnet eine Variable einen Behälter oder einen Speicherbereich, dessen Inhalt durch Wertzuweisungen verändert werden kann. Standardbeispiel (Ada ähnlich):

```
X, Y: Integer := 0;
function f return Integer is begin X := X+1; return X; end;
```

Der Ausdruck X + f + f liefert den Wert 3, sofern ein Ausdruck stets von links nach rechts ausgewertet wird.
Der Ausdruck f + f + X liefert den Wert 5.
Im Ausdruck X + f + f + X steht X einmal für den Wert 0 und einmal für den Wert 2.

In der funktionalen Programmierung orientiert man sich zunächst mehr an der mathematischen Verwendung von Variablen und versucht, Seiteneffekte (wie in obigem Beispiel durch eine globale Variable) zu vermeiden.

Die Grundidee besteht darin, ein Problem durch ein System von Funktionen zu beschreiben. Wählt man die Funktionen so, dass man sie effektiv ausrechnen kann, dann kann man einen Berechnungsprozess starten, der eine Lösung liefert, sofern er terminiert.

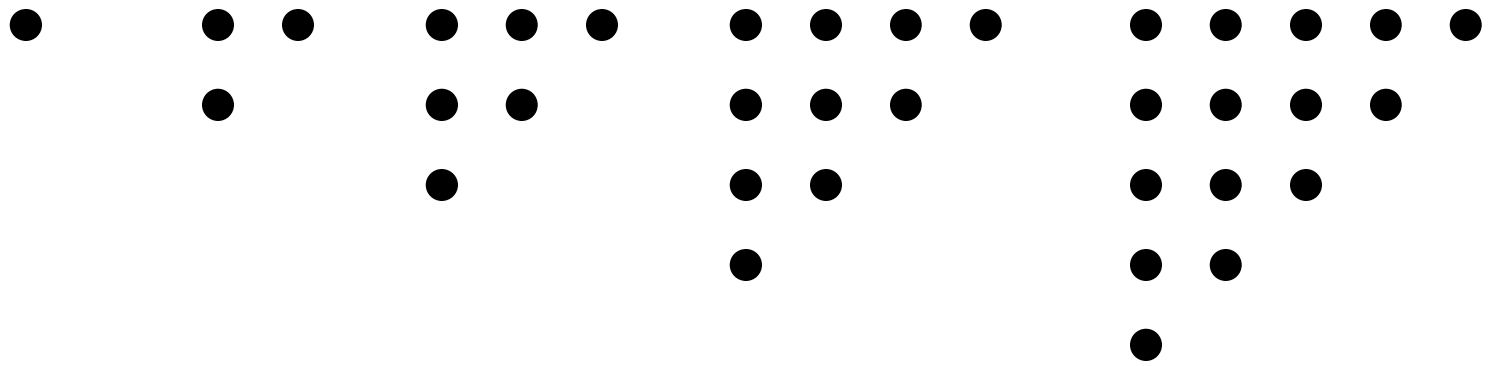
Funktionen definiert man meist durch Ausdrücke. Das zu lösende Problem ist dann ein Ausdruck, der die zuvor definierten Funktionen enthält. Ein Programm in Scheme hat daher die Form

definiere $f_1(x_1,x_2,...)$ = <Ausdruck 1>;
definiere $f_2(x_1,x_2,...)$ = <Ausdruck 2>;
...
definiere $f_m(x_1,x_2,...)$ = <Ausdruck m>;

<Ausdruck>

Beispiel: n-te Dreieckszahl D_n, $x^2 + y^2$, $D_n^2 + D_m^2$

$D_1 = 1$ $D_2 = 3$ $D_3 = 6$ $D_4 = 10$ $D_5 = 15$



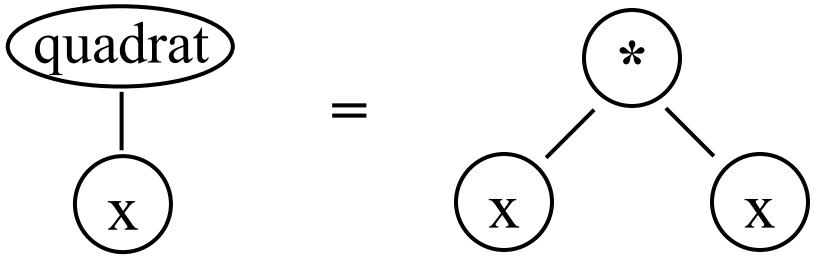
$$\mathbf{D_n} = 1 + 2 + 3 + \ldots + n = \sum_{i=1}^{n} i = n \cdot (n+1) / 2 = (n^2 + n) / 2$$

Darstellung in Präfixform, also z. B. (f x y) statt f(x,y) :

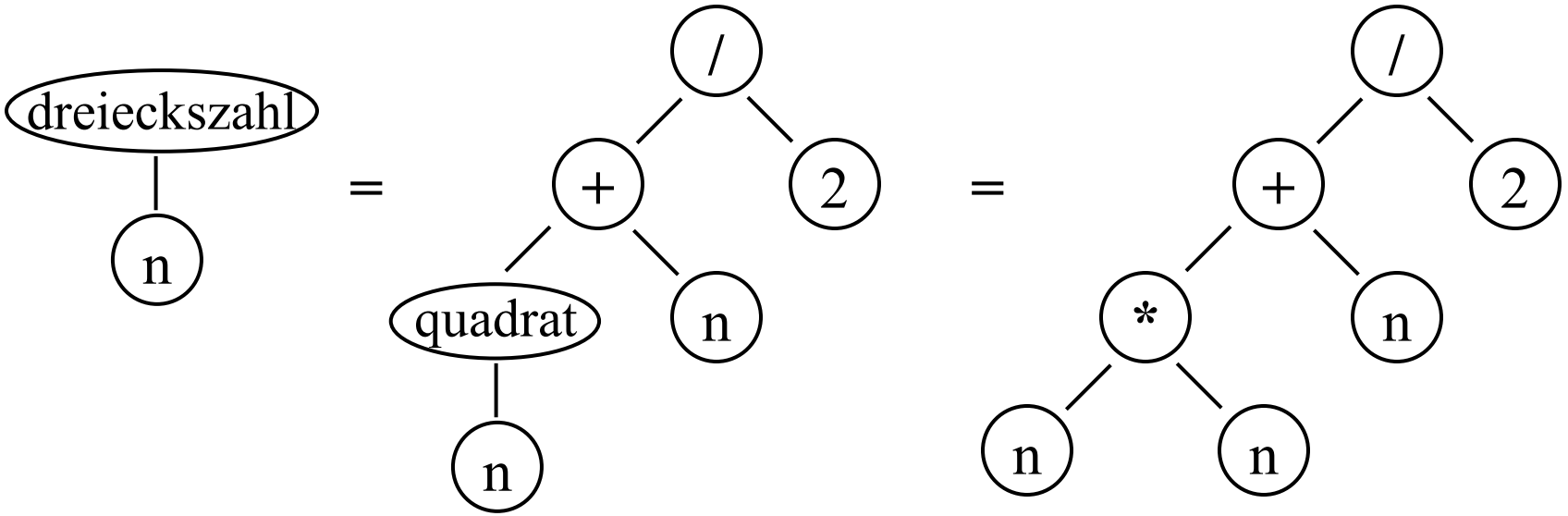
```
(define (quadrat  x) (*  x  x) )
(define (dreieckszahl  n) (/ (+ (quadrat  n)  n) 2) )
(define (quadratsumme  x  y) (+ (quadrat  x)  (quadrat  y) ) )
(define (dquadratsumme  n  m)
        (quadratsumme  (dreieckszahl  n) (dreieckszahl  m) ) )
```

Diese Funktionen gehören zu einfachen Rechenbäumen, z. B.:

(define (quadrat x) (* x x))

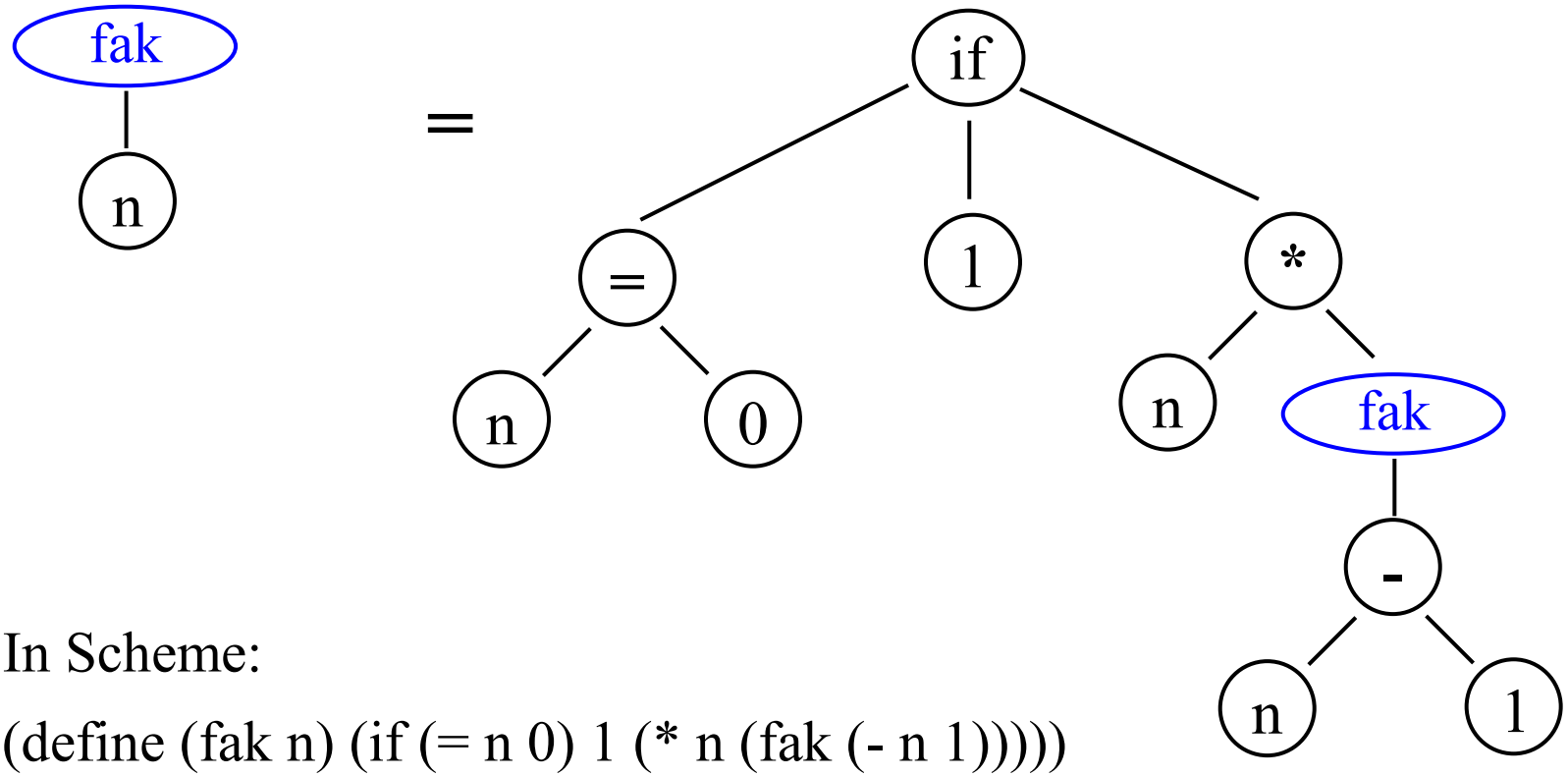


(define (dreieckszahl n) (/ (+ (quadrat n) n) 2))



Algorithmische Mächtigkeit entsteht erst durch die Rekursion.

fak(n) = if n=0 then 1 else n*fak(n-1) fi gehört zum Rechenbaum

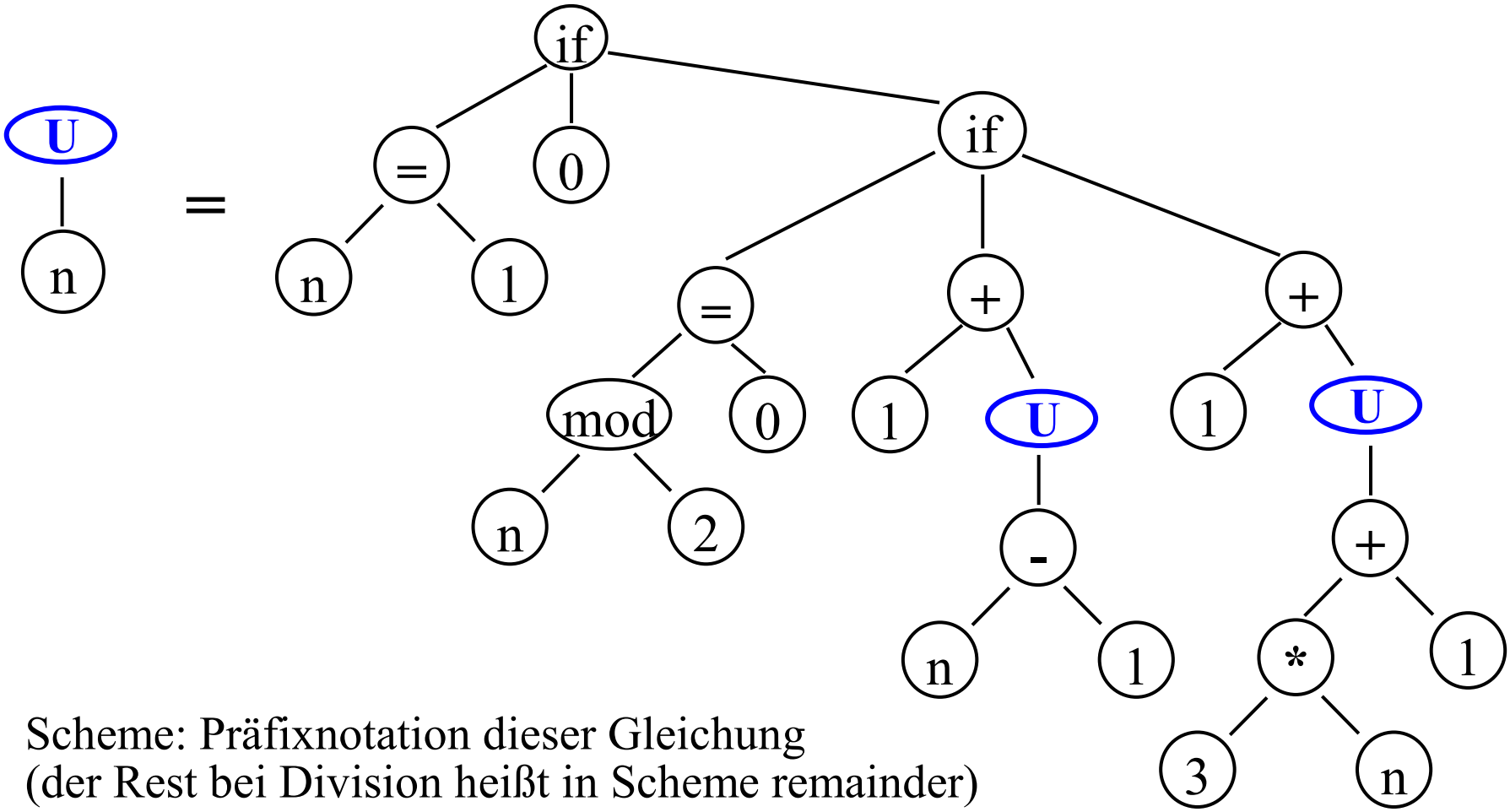


In Scheme:

(define (fak n) (if (= n 0) 1 (* n (fak (- n 1)))))

Beispiel: Ulam-Collatz-Funktion

U(n) = if n=1 then 0 else if n mod 2 = 0 then 1+U(n/2) else 1+U(3*n+1) fi fi



Scheme: Präfixnotation dieser Gleichung
(der Rest bei Division heißt in Scheme remainder)

```
(define (U n) (if (= n 1)  0
        (if (= (remainder n 2) 0) (+ 1 (U (/ n 2))) (+ 1 (U (+ (* 3 n) 1 ))))))
```

2.1 Einführung in Scheme

Hinweise zu Scheme und zu DrScheme

2.1.1 Einführendes Beispiel

2.1.2 (Elementare) Datentypen

2.1.3 Ausdrücke

2.1.4 Auswertung und die Funktion quote

2.1.5 Datenstrukturen

2.1.6 Die Funktionen apply und eval

2.1.7 Alternativen

2.1.8 Konditionen (Conditionals)

2.1 Einführung in Scheme

Für Übungen und Berechnungen benötigen Sie ein Programmsystem mit Entwicklungsumgebung.

An der Rice University in Houston, Texas, arbeitet das "Programming Language Team" (PLT) seit etwa 15 Jahren unter anderem an dem Projekt **DrScheme**. Dieses wird mittlerweile weltweit an sehr vielen Hochschulen für die Ausbildung in funktionaler Programmierung bzw. in Scheme eingesetzt. Das System, das derzeit noch auf dem Revised Report[5] basiert, können Sie sich aus dem Netz herunterladen:

download: http://download.plt-scheme.org/drscheme/

Zugleich finden Sie dort Tutorien zum Erlernen von Scheme und zur Nutzung des Systems.

Eine Kurzübersicht zu Scheme und die Versionen findet sich unter Wikipedia. (Besser ist natürlich ein Lehrbuch.)

Hinweis: Die Sprache Scheme wurde als ein einfacher LISP-Dialekt ("properly tail-recursive") von G.L.Steele jr. und G.J.Sussmann am MIT entwickelt und implementiert.

Die Sprache Scheme ist 1998 mehrfach standardisiert worden. Den aktuellen

Revised^6 Report on the Algorithmic Language Scheme

vom 26.9.2007 finden Sie im Netz z. B. über die Adresse

http://www.r6rs.org/final/r6rs.pdf

Alle folgenden Ausführungen können bzgl. Syntax und deren Bedeutung in diesem Report nachvollzogen werden. Zugleich finden Sie dort weitere Sprachelemente, die wir in unserer Vorlesung nicht betrachten. Was dort alles behandelt wird, können Sie der folgenden Gliederung des Revised^6 Reports entnehmen:

Introduction
Description of the language
1 Overview of Scheme
1.1 Basic types
1.2 Expressions
1.3 Variables and binding
1.4 Definitions
1.5 Forms
1.6 Procedures
1.7 Proc. calls & synt. keywords
1.8 Assignment
1.9 Derived forms and macros
1.10 Synt. data & datum values
1.11 Continuations
1.12 Libraries
1.13 Top-level programs
2 Requirement levels
3 Numbers
3.1 Numerical tower
3.2 Exactness
3.3 Fixnums and flonums
3.4 Implementation requirements
3.5 Infinities and NaNs
3.6 Distinguished -0.0
4 Lexical syntax & datum syntax
4.1 Notation
4.2 Lexical syntax
4.3 Datum syntax

5 Semantic concepts
5.1 Programs and libraries
5.2 Variables, keywords, regions
5.3 Exceptional situations
5.4 Argument checking
5.5 Syntax violations
5.6 Safety
5.7 Boolean values
5.8 Multiple return values
5.9 Unspecified behavior
5.10 Storage model
5.11 Proper tail recursion
5.12 ... the dynamic environment
6 Entry format
6.1 Syntax entries
6.2 Procedure entries
6.3 Implement. responsibilities
6.4 Other kinds of entries
6.5 Equivalent entries
6.6 Evaluation examples
6.7 Naming conventions
7 Libraries
7.1 Library form
7.2 Import and export levels
7.3 Examples
8 Top-level programs
8.1 Top-level program syntax
8.2 Top-level program semantics

9 Primitive syntax
9.1 Primitive expression types
9.2 Macros
10 Expansion process
11 Base library
11.1 Base types
11.2 Definitions
11.3 Bodies
11.4 Expressions
11.5 Equivalence predicates
11.6 Procedure predicate
11.7 Arithmetic
11.8 Booleans
11.9 Pairs and lists
11.10 Symbols
11.11 Characters
11.12 Strings
11.13 Vectors
11.14 Errors and violations
11.15 Control features
11.16 Iteration
11.17 Quasiquotation
11.18 Binding constructs for syntactic keywords
11.19 Macro transformers
11.20 Tail calls and tail contexts

Hinzu kommen mehrere Anhänge:

Insgesamt umfasst der Report 90 Seiten, wobei jede dortige Seite etwa 1,5 normalen Seiten entspricht. Wenn Ihnen etwas unklar ist, lesen Sie sich bitte in diesen Report ein. Durch unsere Vorlesung erhalten Sie das nötige Wissen, um sich dort rasch zurecht zu finden.

2.1.1 Einführendes Beispiel

Wir wollen feststellen, ob der Wert von 27*37+91*11-83*55 durch 3 teilbar ist.

Zunächst müssen wir klären, was "durch 3 teilbar" bedeutet. Hierfür verwenden wir den Rest, der bei der Division bleibt. Im Falle "3" bedeutet dies: Für x = 0, 1, 2 ist x der Rest, der bei der Division durch 3 bleibt. Weiterhin gilt:
x mod 3 = 1 $\Leftrightarrow$ (x-1) mod 3 = 0 $\Leftrightarrow$ (x-2) mod 3 = 2,
x mod 3 = 2 $\Leftrightarrow$ (x-1) mod 3 = 1 $\Leftrightarrow$ (x-2) mod 3 = 0.

Hieraus folgen Rekursionsformeln für die drei Funktionen Rest0, Rest1 und Rest2, die den Wahrheitswert liefern, ob der Rest bei der Division durch 3 gleich 0, 1 oder 2 ist. Also:

Rest0(x) = if x = 0 then true else Rest2(x-1) fi;
Rest1(x) = if x = 0 then false else Rest0(x-1) fi;
Rest2(x) = if x = 0 then false else Rest1(x-1) fi;

Rest0(x) = if x = 0 then true else Rest2(x-1) fi;
Rest1(x) = if x = 0 then false else Rest0(x-1) fi;
Rest2(x) = if x = 0 then false else Rest1(x-1) fi;

Dies liefert bereits das Programm in der Sprache Scheme:

```
(define (Rest0 x) (if (= x 0) "ja" (Rest2 (- x 1))))

(define (Rest1 x) (if (= x 0) "nein" (Rest0 (- x 1))))

(define (Rest2 x) (if (= x 0) "nein" (Rest1 (- x 1))))

(Rest0 ( - (+ (* 27 37) (* 91 11)) (* 13 55)))
```

Was fällt auf?

Preorder Darstellung der Operatoren; viele Klammern; rekursive Definitionen erlaubt; Liste von Objekten, wobei das erste Objekt als Operator aufgefasst wird, der auf die restlichen Objekte angewendet wird.

2.1.2 (Elementare) Datentypen

[Erinnerung: Elementare Datentypen in Ada sind Boolean, Character, Integer, Float und Aufzählungstypen, wobei zum Wertebereich stets auch die zulässigen Operationen gehören. Der Typ einer (Informatik-) Variablen wird bei der Deklaration festgelegt, er braucht daher später in Ada nicht abgefragt zu werden.]

In der Sprache Scheme werden die Identifikatoren/Bezeichner/ Variablen zunächst im mathematischen Sinne (also als "formale Parameter") aufgefasst. Sie stehen für einen Wert. Der Typ dieses Wertes liegt nicht von vornherein fest. Daher muss man im Programm abfragen können, von welchem Typ der Wert eines Bezeichners ist. Dies erfolgt durch Typ-Prädikate.

In Scheme werden vor allem Listen und die folgenden Typen verwendet.

Höchstens genau einer der folgenden Typen ist in Scheme für ein Objekt (= eine durch einen Bezeichner identifizierte Einheit) zutreffend.

Typprädikat	*Datentyp*
number?	eine Zahl (hier wird nicht zwischen reellen und ganzen Zahlen unterschieden)
boolean?	Boolescher Wert [#f für "false", #t für "true"}
char?	alphanumerisches Zeichen [mit #\ beginnend]
symbol?	Objekt mit einem Namen/Bezeichner zur Identifikation
pair?	Paar [zwei Objekte, wie man sie mittels cons bildet, auch eine Liste ist ein Paar]
string?	Zeichenkette [kann auch leer sein]
vector?	Vektor [= eindimensionales Feld, mit #(beginnend]
procedure?	Prozedur
null?	leere Liste [die leere Liste notieren wir auch als '()]

Wie sehen die Konstanten in Scheme aus?

Zahlen (number): Man gibt eine Folge von Ziffern an, wenn man eine natürliche Zahl meint; diese kann ein Vorzeichen besitzen, wenn man eine ganze Zahl meint; diese kann zusätzlich einen gebrochenen Anteil besitzen (abgetrennt durch einen Punkt), wenn man eine reelle Zahl meint. Beispiele: 34, -651, 54.1204, -23.65. In Scheme kann man auch rationale und komplexe Zahlen verwenden.

Wahrheitswerte (boolean): Für false schreibt man #f und für true #t. (Aber auch jeder andere Scheme-Wert außer #f gilt in einer Bedingung als Warheitswert true.)

Zeichen (char) werden in der Form #\a (Kreuz,Backslash,Zeichen) dargestellt. (Eine Folge von Zeichen ist ein string, s. u.)

Symbole: Dies sind Objekte, die keine Konstanten sind, im Programm verwendet werden und durch einen Bezeichner identifiziert werden können.

Als Bezeichner sind alle nichtleeren Folgen von Buchstaben, Ziffern und folgenden Zeichen

! $ & % + - * / . : < = > ? @ ^ _ ~

zugelassen, deren erstes Zeichen nicht mit dem Anfang einer Zahl verwechselt werden kann. Beispiele für Bezeichner:

x ?a2 =<_als%er v123-aber-nicht-negativ x-->y

Mittels (define x <Ausdruck>) wird dem Bezeichner x ein Wert zugeordnet.

Prozeduren (= ausrechenbare Funktionen) bestehen aus einem Namen, einer Folge formaler Parameter und einem definierenden Ausdruck (dem Rumpf). Wir führen Prozeduren in einem Programm in der Regel ein mittels (s. u.)

(define <Name und Parameter> <Prozedurrumpf>)

λ-Schreibweise: Allgemein erhält man Funktionen aus Ausdrücken durch den Lambda-Operator. Meint man z.B. nicht den Ausdruck (/(+ (* x x) x) 2), sondern die Funktion $f(x) = (x^2+x)/2$ *(x-te Dreieckszahl)*, so schreibt man

```
(lambda (x) (/ (+ (* x x) x) 2))
```

Allgemein:

```
(lambda <Formalteil> <Rumpf>)
```

Im einfachsten Fall ist der Formalteil die Liste der formalen Parameter (eventuell leer) und der Rumpf ist ein Ausdruck. Will man der Funktion einen Namen geben, so muss man dies wie oben mittels "define" tun:

```
(define d (lambda (x) (/ (+ (* x x) x) 2)))
```

Der Prozeduraufruf (procedure call) liefert dann z.B.:

```
(d  5)   ==>  15
```

Anstelle von
(define <Bezeichner> (lambda <Formalteil> <Rumpf>))

schreibt man kürzer

(define <Bezeichner und Formalteil> <Rumpf>)

Beispiel: Statt

```
(define d (lambda (x) (/ (+ (* x x) x) 2)))
```

kann man daher auch schreiben

```
(define (d x) (/ (+ (* x x) x) 2))
```

(d 5) liefert erneut den Wert 15.

2.1.3 Ausdrücke

Ausdrücke werden aus Konstanten, Variablen (Bezeichnern), Operatoren und Funktionsaufrufen (procedure calls) in preorder Darstellung gebildet. Durch Klammern wird die für die Auswertung wichtige Baumstruktur festgelegt. Dies klingt zunächst unsinnig, weil die Preorder-Darstellung ja die eindeutige Reihenfolge festlegt, jedoch ist in Scheme die Anzahl der Argumente einer Funktion oft nicht fest, und zugleich dient die Klammerung der Zusammenfassung zusammengehöriger Teile des Ausdrucks. Weiterhin ist der Begriff "Ausdruck" nicht auf Zahlbereiche beschränkt (s. u.).

Ausdrücke sind einzelne Variablen oder Konstante oder sie besitzen eine Listenstruktur.

Beispiel: Ganze Zahlen mit den Operationen succ, pred, +, -, *, /, sign, =, <, > , /=, <=, >=. Hiermit kann man unmittelbar Ausdrücke (Terme) bilden: (23+18)*(4-2), jedoch muss man in Scheme die preorder-Darstellung verwenden, also

(* (+ 23 18) (- 4 2))

Dieser Term lässt sich als eine vierstellige Funktion fk auffassen: (define (fk a b c d) (* (+ a b) (- c d))).
Auch diese Liste ist ein Ausdruck, nämlich ein Ausdruck, der den Bezeichner fk definiert und der seine Stelligkeit und die Berechnungsvorschrift festlegt.

Der Begriff "Ausdruck" (expression) wird also nicht nur für arithmetische und boolesche Ausdrücke benutzt, sondern für alle Darstellungen, also auch für Kontrollelemente, Funktionen und Programme. Letztlich ist ein Scheme-Programm eine Menge von Definitions-Ausdrücken mit einem oder mehreren Ausdrücken, die zum Ergebnis des Programms ausgewertet werden können.

2.1.4 Auswertung und die Funktion quote

Ausdrücke werden ausgewertet, indem man sie ausrechnet.
Das Ergebnis der Auswertung notieren wir durch "==>".
Beispiel: (+ 4 5) ==> 9
Der Pfeil ==> ist also zu lesen als *"wird ausgewertet zu"*.

Konstanten werden immer zu sich selbst ausgewertet:

```
27      ==>  27
#f      ==>  #f
```

Ein Bezeichner wird zu dem ihm zugeordneten Wert ausgewertet:

```
(define x 6)  x    ==> 6
(define (f x) (* x x))  f  ==> #<procedure:f>
(define (q x) (if (#t + -))  q  ==> +
```

Hat ein Bezeichner keinen Wert, so liegt ein Fehler vor.

Ein Lambda-Ausdruck kann ausgewertet werden, indem er auf so viele Argumente, wie die Liste seiner formalen Parameter angibt, angewendet wird:

(**(**lambda (x) (+ (* x x) x)**)** 5) ==> 30

Allgemein bildet man also eine Liste aus dem Lambda-Ausdruck (oder dem Namen einer Funktion), der k formale Parameter besitzen möge, und k aktuellen Parametern <aktparam$_i$>:

(<Lambda-Ausdruck> <aktparam$_1$> ... <aktparam$_k$>)

Beispiel für eine Funktion zum Prüfen, ob $x^2+y^2=z^2$ ist:

```
(define (Pyth? x y z) (= (+ (* x x) (* y y)) (* z z)))
(Pyth? 3 4 5)      ==>  #t
(Pyth? 2 7 8)      ==>  #f
(Pyth? -3 -4 5)    ==>  #t
```

Generell muss man stets zwischen den Objekten und dem, was sie bedeuten (also dem Ergebnis einer Auswertung), unterscheiden. Das Objekt selbst erhält man mittels quote.

Meint man zum Beispiel den Ausdruck (+ 4 5) selbst und nicht das Ergebnis 9, so schreibt man

(quote (+ 4 5)) ==> (+ 4 5)

Statt (quote <Objekt>) schreibt man kurz '<Objekt>

'(+ 4 5) ==> (+ 4 5)

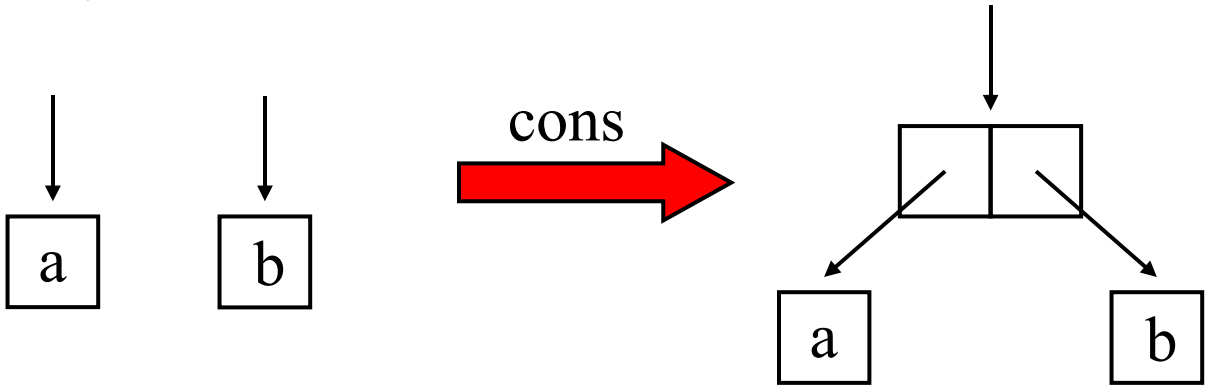
' '(+ 4 5) ==> '(+ 4 5)

2.1.5 Datenstrukturen erhalten wir, indem wir auf Datentypen Konstruktoren anwenden. In Ada haben wir array, record, access und die Unterbereichsbildung a..b betrachtet.

In Scheme gibt es im Wesentlichen nur einen Konstruktor: die Paarbildung. Hieraus werden Listen aufgebaut. In ihnen werden Elemente sequentiell aneinander gereiht und mit der leeren Liste abgeschlossen. Da die Elemente einer Liste selbst wieder Listen sein können, ergibt sich eine große Vielfalt an Baumstrukturen.

Paar: Zusammenfassung zweier Objekte mit Hilfe des Operators cons: (cons <objekt1> <objekt2>). Paare bezeichnet man auch als "dotted pair"; im Ausdruck erscheint zwischen den beiden Objekten dann auch ein Punkt. cons bildet also ein Zwei-Tupel; die Komponenten kann man mittels car und cdr wieder erreichen.

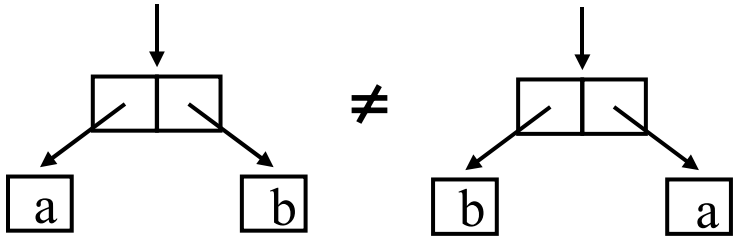
Logisch gesehen fasst der Operator cons (= "constructor") zwei Objekte zu einem kleinst möglichen binären Baum mit einer "inhaltsleeren" Wurzel zusammen.
(cons a b) bedeutet also



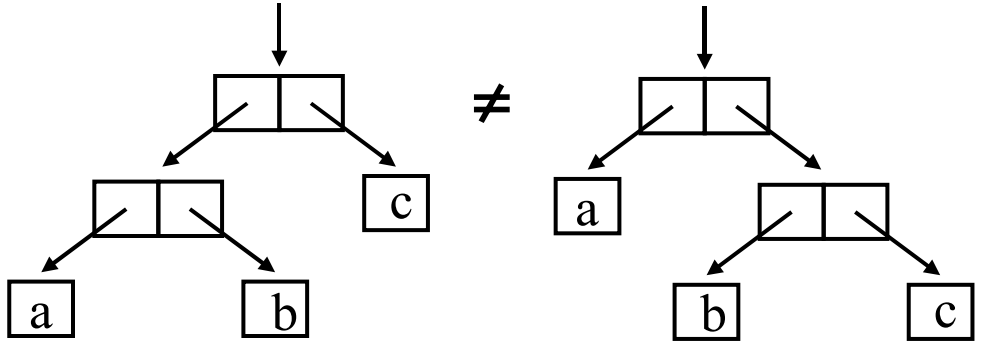
Auf die erste Komponenten kann man mittels car und auf die zweite mittels cdr zugreifen, gesprochen "kahr" und "kudd^{e}r". Die Bezeichnungen stammen von den Assemblerbefehlen der IBM-704 (ein viel benutzter Rechner von 1954 bis 1960), mit denen man die Komponenten erreichte: car = content of address register und cdr = content of decrement register.

Die externe Darstellung, also die Ausgabe, von (cons a b) ist (a . b) . Wie bei binären Bäumen ist cons weder kommutativ noch assoziativ, d. h.:

(cons a b) ≠ (cons b a)

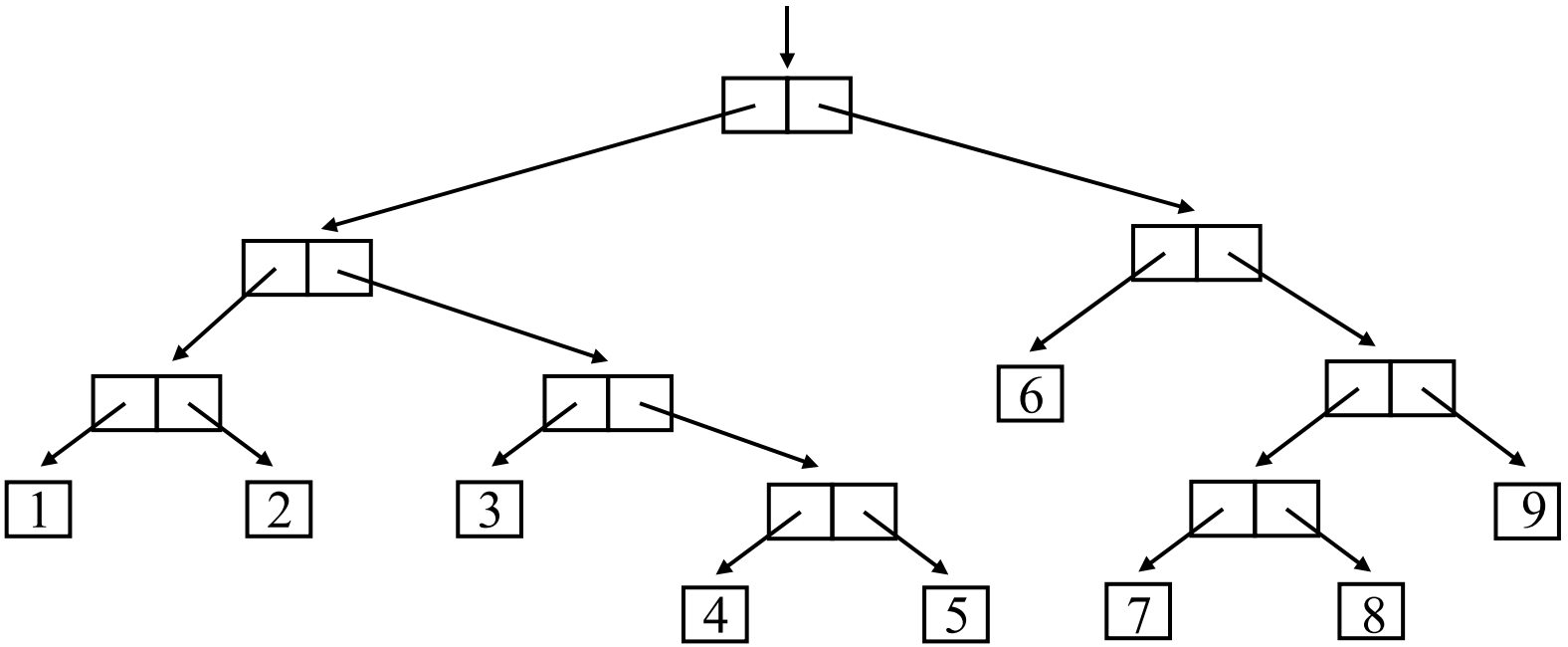


(cons (cons a b) c) ≠
(cons a (cons b c))



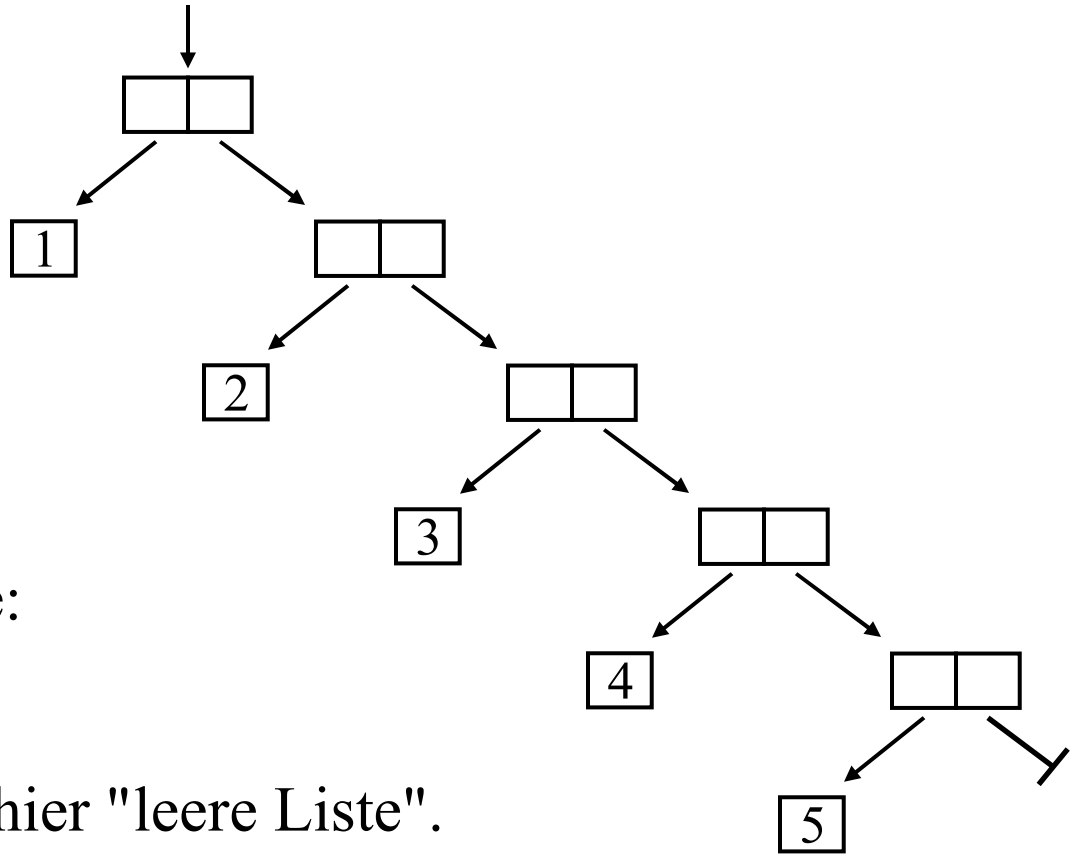
Es ist klar, dass man mit cons binäre Bäume aufbauen kann. Solche Bäume bilden die wichtigste Datenstruktur in Scheme.

Beispiel:
(cons (cons (cons 1 2) (cons 3 (cons 4 5))) (cons 6 (cons (cons 7 8) 9)))
Dieser Ausdruck liefert die Darstellung: (((1 . 2) 3 4 . 5) 6 (7 . 8) . 9)
Diese Darstellung setzt keinen Punkt, wenn es nach rechts "listenartig" (siehe unten) weitergeht. Es liegt folgender binärer Baum vor:



Ein Spezialfall der binären Bäume sind lineare Listen, siehe Grundvorlesung Informatik, Abschnitt 3.5.

(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 '())))))



Abkürzende Schreibweise:
(1 2 3 4 5)

↘ bedeutet hier "leere Liste".

Listen sind Folgen von Objekten, in runde Klammern eingeschlossen und mit der leeren Liste beendet. Man kann sie mittels (list <Folge von Objekten>) erzeugen, z.B.:

```
(list '2 '3 '4)                  ==>  (2 3 4)
(list '2 (- 5 2) '4)             ==>  (2 3 4)
(list (list '5 'a) '7 (list 'b))  ==>  ((5 a) 7 (b))
```

Man kann Listen auch explizit hinschreiben mittels ' :

```
'(2 3 4)      ==>  (2 3 4)
```

Es gibt diverse Operationen für Listen, z. B. (reverse L) dreht die Reihenfolge der Objekte in der Liste L um.

Ein Spezialfall ist die leere Liste ( ).

Formale Definition: Eine Liste ist entweder die leere Liste oder (list e_1 e_2 e_3 ... e_n) ist gleichbedeutend mit der n-maligen wiederholten Anwendung des cons-Operators in der Form:

(cons e_1 (cons e_2 (cons e_3 ... (cons e_n '()) ...))).

Wegen dieser Definition gehört eine nichtleere Liste auch zum Typ "Paar", d. h.

(pair? (list 'a)) ==> #t

Dagegen ist die leere Liste kein Paar: (pair? '( )) ==> #f.
Daher führt man eine eigene Abfrage auf die leere Liste ein:

null? <Objekt> Ist <Objekt> die leere Liste?

Die leere Liste ist in Scheme eine spezielle Konstante, mit der jede Liste abgeschlossen wird (siehe formale Definition).

Man kann auch abfragen, ob eine Liste vorliegt:

list? <Objekt> Ist <Objekt> eine Liste?

Ein Paar ist in der Regel keine Liste, weil es nicht mit der leeren Liste abschließen muss.

Auf Listen sind die Operatoren car und cdr ebenfalls erlaubt:

(car L) liefert das erste Objekt der Liste L,

(cdr L) liefert die Liste L ohne das erste Element.

In Scheme werden Ausdrücke (außer einzelnen Konstanten und Bezeichnern) als Listen beschrieben. Eine Liste L

$(a_1\ a_2\ a_3\ a_4)$

wird als Ausdruck ausgewertet, indem man das erste Objekt a_1 als eine Prozedur (Funktion oder Operator) auffasst, die auf die weiteren Objekte $a_2\ a_3\ a_4$ angewendet wird; a_2, a_3, a_4 sind also die aktuellen Parameter für den Operator a_1.

Eine Liste L wird als Ausdruck folglich ausgewertet als

(car L) anwenden auf (cdr L).

Listen werden in Abschnitt 2.4 genauer behandelt.

Zeichenketten (string) = eine Folge von beliebigen Tastatur-Zeichen, die in Anführungsstriche eingeschlossen ist, also "es", "Heute ist Dienstag", "" (für den leeren String). Der Backslash \ hat hierbei die Bedeutung einer Unterbrechung. Dies benutzt man, um \ und " in eine Zeichenkette einzufügen, z.B.: den Text

"Mengendifferenz" wird mit "M\N" bezeichnet.

stellt man als string dar durch

"\"Mengendifferenz\" wird mit \"M\\N\" bezeichnet."

Vektoren: Eine Folge von Elementen, die fortlaufend mit den Indizes 0, 1, 2, ... nummeriert sind. Darstellung in der Form #(<Folge von Elementen>)

2.1.6 Die Funktionen apply und eval

(apply <proc> <Ausdruck$_1$> ... <Ausdruck$_m$> <Liste>)
wendet die Funktion <proc> auf die Liste an, die entsteht, indem man <Ausdruck$_1$>, ..., <Ausdruck$_m$> an den Anfang der Liste <Liste> setzt (m = 0 ist erlaubt).

(apply - '(9 4 3)) ==> 2

(apply + 4 (- 6 1) '(3 1))
Dies wird in (apply + '(4 5 3 1)) umgewandelt und dann als (+ 4 5 3 1) ausgewertet, also ergibt sich ==> 13

(apply (if (= 7 8) + *) 4 (- 6 1) '(3 1)) ==> 60
(apply (if (= 7 7) + *) 4 (- 6 1) '(3 1)) ==> 13

(eval <Ausdruck>) wertet den <Ausdruck> aus.

```
(+ 2 3)                    ==>  5
'(+ 2 3)                   ==>  (+ 2 3)
(eval (+ 2 3))             ==>  5
(eval '(+ 2 3))            ==>  5
(eval (eval '(+ 2 3)))     ==>  5
(eval ' '(+ 2 3))          ==>  (+ 2 3)
(eval (eval ' '(+ 2 3)))   ==>  5
(eval ' ' '(+ 2 3))        ==>  '(+ 2 3)
(eval (eval ' ' '(+ 2 3))) ==>  (+ 2 3)
(eval ' ' ' '(+ 2 3))      ==>  ' '(+ 2 3)
```

```
(define (von-1-bis-x  x)
      (if (= 0 x) ( ) (cons x (von-1-bis-x (- x 1)))))

(define (summenformel a) (cons + (von-1-bis-x a)))

(summenformel 9)                    ==> (+ 9 8 7 6 5 4 3 2 1)
```

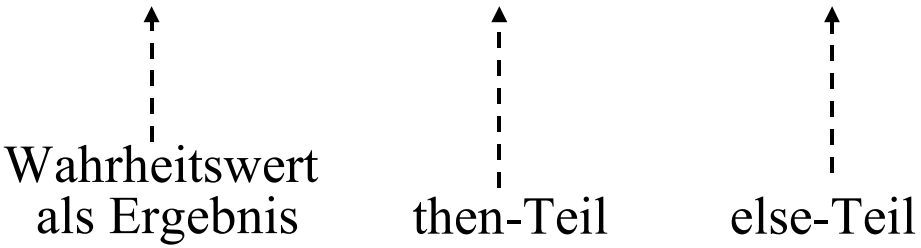
[Hinweis: DrScheme liefert hier die externe Darstellung:
(#<primitive:+> 9 8 7 6 5 4 3 2 1), d.h., "+" ist ein Symbol, kein Zeichen]

Dies ergibt nur die Liste. Nun möchte man die Summe auch wirklich ausrechnen, z. B. mittels apply oder eval:

```
(apply + (von-1-bis-x  0))          ==> 0
(apply + (von-1-bis-x  10))         ==> 55
(eval (summenformel 2))             ==> 3
(eval (summenformel 9999))          ==> 49995000
```

2.1.7 Alternativen

Syntax: (if <Ausdruck> <Ausdruck> <Ausdruck>)



Beispiele:

```
(if (< 3 7) 'x 'y)                    ==>  x
((if (= 3 5) + -) ('6 '2))            ==>  4
((if (= 3 3) + -) ('6 '2))            ==>  8
```

Man kann also auch Funktionen auf diese Weise auswählen.

```
(define (fak n) (if (<= n 0) 1 (* n (fak (- n 1)))))
(fak 8)                               ==>   40320
```

2.1.8 Konditionen (Conditionals)

Syntax:

(cond <Klausel$_1$> ... <Klausel$_k$>)

Jede Klausel ist von der Form:

(<Bedingung> <Ausdruck$_1$> ...)

oder von der Form

(<Bedingung> => <Ausdruck>).

Die letzte Klausel darf von folgender Form sein:

(else <Ausdruck$_1$> ... <Ausdruck$_m$>)

Bedeutung: Die Klauseln werden der Reihe nach abgearbeitet. Sobald die Bedingung in einer Klausel den Wert true ergibt, wird die Folge der zugehörigen Ausdrücke ausgewertet; danach ist die Kondition beendet. Trifft man auf else, so werden die auf das else folgenden Ausdrücke ausgewertet und danach die Kondition beendet.

2.1.9 Hinweis: Probieren Sie nun alles aus

Manches im Revised Report on Scheme ist beim ersten Lesen nicht recht verständlich. Probieren Sie daher die Bedeutung elementarer Funktionen und insbesondere der Abfragen mit Hilfe des DrScheme-Systems (siehe erste Folie des Abschnitts 2.1) aus. Auf der folgenden Folie sind einige Beispiele abgedruckt einschließlich der Antworten des Systems.

Dabei finden und erlernen Sie zugleich die "Standard-Ein-und Ausgabe". (Prozeduren display, newline, read usw.)

Beachten Sie: Sie können neben den define-Bereichen am Ende mehrere Ausdrücke auflisten, diese werden nacheinander ausgewertet.

Beispielprogramm	zugehörige Ausgabe
(display "list? pair?") (newline) (define q '(7 2 3 6)) q (list? q) (pair? q) (define q (cons 8 4)) q (list? q) (pair? q) (display "boolean?") (newline) 'x (boolean? 'x) #t (boolean? #t)	list? pair? (7 2 3 6) #t #t (8 . 4) #f #t boolean? x #f #t #t
(display "char?") (newline) 4 (char? 4) 'a (char? 'a) '4 (char? '4) #\a (char? #\a) #\4 (char? #\4)	char? 4 #f a #f 4 #f #\a #t #\4 #t
#\space (char? #\space) (display "symbol?") (newline) (define x 'S) x (char? x) (symbol? x) (define x #\a) x (char? x) (symbol? x) 'z (symbol? 'z) (define z 'y) z (symbol? z)	#\space #t symbol? S #f #t #\a #t #f z #t y #t
(display "string? char?") (newline) (define x "Die \"Informatik\" am 11.12.07 (+- eine Woche)") (display x) (newline) (string? x) (char? x) (define y (string-ref x 27)) y (string? y) (char? y)	string? char? Die "Informatik" am 11.12.07 (+- eine Woche) #t #f #\6 #f #t

2.2 Beispielprogramme

2.2.1 Einstiegsprogramme

2.2.2 Größter gemeinsamer Teiler (ggT, gcd) und kgV

2.2.3 Zerlegung einer natürlichen Zahl in ihre Primfaktoren

2.2.4 Die n-te Primzahl

2.2.5 Die Wurzel aus einer natürlichen Zahl

2.2.6 Zeichnen

2.2.7 m gültige Ziffern (dezimal)

2.2 Beispielprogramme

2.2.1 Einstiegsprogramme

Ergebnis

```
(define x1 (if (null? '4) 4 (= 4 5)))
x1
```

==> #f

```
(define (hochvier z) (* z (* z z) z))
(hochvier 5)
```

==> 625

In Scheme wird * nacheinander auf beliebig viele Zahlen angewandt, evtl. keinmal (Ergebnis 1) oder nur einmal (Ergebnis z).

```
(define (kubik z) (* z (* z z)))
(define (summe z) (if (null? z)  0
                      (+ (car z) (summe (cdr z)))))
(summe (list (kubik 1)
             (kubik 2) (kubik 3) (kubik 4)))
```

==> 100

An dem Programm

```
(define x1 (if (null? '4) 4 (= 4 5)))
x1
```

erkennt man die dynamische Bindung von Werten an Variablen. Je nach Auswertung eines Booleschen Ausdrucks kann hier x1 entweder an eine Zahl oder an einen Wahrheitswert gebunden werden.

Dies kann man mit den Typprädikaten aus Abschnitt 2.1.2 abfragen, z. B. liefert das Programm:

```
(define x1 (if (null? '4) 4 (= 4 5)))
(cond   ((number? x1) "Zahl")
        ((boolean? x1) "Wahrheitswert")
        (else "Weiss nicht"))
```

das Ergebnis "Wahrheitswert".

2.2.2 Größter gemeinsamer Teiler (ggT, gcd) und kgV

Notation: $\mathbb{N} = \{1, 2, 3, ...\}$, $\mathbb{N}_0 = \{0, 1, 2, 3, ...\}$.

Definition: Der größte gemeinsame Teiler d zweier natürlicher Zahlen a und b ist die größte natürliche Zahl, die a und b teilt.

Formale Definition von "teilt": d teilt a $\Leftrightarrow \exists\, k \in \mathbb{N}: k \cdot d = a$.

Sei T(x) = {y | y teilt x} die Menge aller Teiler von x, dann erhält man also:

$$ggT(a,b) = Max\,(T(a) \cap T(b)) \quad \text{für alle } a, b \in \mathbb{N}.$$

Dies ist die Spezifikation, die zum Nachweis der Korrektheit gebraucht wird. Sie wird als Erstes implementiert.

Bevor man beginnt, muss man sich überzeugen, dass die Definition sinnvoll und widerspruchsfrei (also "wohldefiniert") ist.

1. Für eine natürliche Zahl a (ohne die Null) ist T(a) eine endliche Menge. Die Zahl 1 liegt stets in T(a).
2. Es folgt: Insbesondere ist $T(a) \cap T(b)$ eine endliche nichtleere Menge natürlicher Zahlen. Eine solche Menge besitzt stets genau ein Maximum.

Der ggT(a,b) ist somit wohldefiniert.

Wir benötigen die Hilfsfunktion "d teilt a". Diese berechnen wir mittels: d teilt a $\Leftrightarrow$ a mod d = 0.

Hierdurch verlassen wir den bisherigen Zahlbereich $\mathbb{N}$ und arbeiten ab jetzt in $\mathbb{N}_0$, wobei aber die Eingabezahlen a und b nicht Null sein dürfen, da wir den ggT nur auf $\mathbb{N}$ definiert haben. Auch die zwischenzeitlich berechneten Teiler dürfen nicht Null sein.

Die Abfrage, ob a mod = 0 ist, wird in Scheme durch

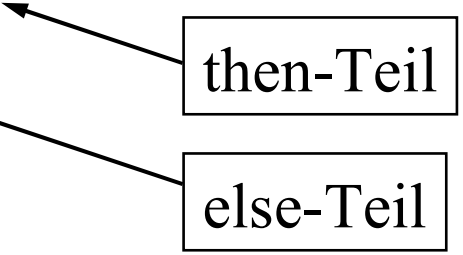
(= 0 (remainder a d))

dargestellt.

```
;;; Zuerst berechnen wir zu einer Zahl a die Teilermenge.
;;; Hierzu ermitteln wir deren Teiler von einer Zahl i bis a und
;;; setzen die Teilermenge von a auf die Teiler von 1 bis a.
;;; Diese Menge speichern wir als Liste.

(define (teilermenge-i-bis-x i x)
    (if (> i x) ( )    ; leere Menge, falls i > x
        (if (= 0 (remainder x i))
                    ; falls "i teilt x", dann i in die Liste aufnehmen
                    ; in jedem Fall danach mit i+1 weitermachen
            (cons i (teilermenge-i-bis-x (+ i 1) x))
            (teilermenge-i-bis-x (+ i 1) x) ) ) )

(define (teilermenge a) (teilermenge-i-bis-x 1 a))
```



```
;;; Den Durchschnitt zweier Listen erhält man, indem man für jedes
;;; Element der ersten Liste prüft, ob es in der zweiten enthalten ist.
;;; Wir betrachten daher zunächst die Funktion element?.


(define (element? a L)
    (if (null? L) #f
        (if (= a (car L)) #t (element? a (cdr L)) ) ) )


(define (durchschnitt L1 L2)
    (if (null? L1) ( )
        (if (element? (car L1) L2)
            (cons (car L1) (durchschnitt (cdr L1) L2))
            (durchschnitt (cdr L1) L2) ) ) )
```

```
;;; Nun müssen wir das Maximum einer Liste bestimmen.
;;; Das Maximum einer einelementigen Liste ist das einzige Element
;;; der Liste, anderenfalls vergleiche das erste Element mit dem
;;; Maximum der Restliste. Hinweis: Wir prüfen später nicht, ob die
;;; Liste leer ist, da hier stets die Zahl 1 in der Liste vorkommt.

(define  (eins? L)  (and (not (null? L)) (null? (cdr L))) )

(define  (maximum L)
    (if (eins? L) (car L)
        (if (< (car L) (maximum (cdr L)))
            (maximum (cdr L))
            (car L)) ) )
```

```
;;; Nun haben wir alle notwendigen Begriffe eingeführt.
;;; Der ggT lässt sich unmittelbar in Scheme "spezifizieren".

(define  (ggT a b)
(maximum (durchschnitt (teilermenge a) (teilermenge b))) )

;;; Nun kann man Werte berechnen lassen, zum Beispiel:
(ggT  12  66)  (ggT  527  961)  (ggT  782673  969494)
(teilermenge 782673)
(teilermenge 969494)
(durchschnitt (teilermenge 782673)  (teilermenge 969494))
(cons + (teilermenge 12))              ; Liste aus "+" und teilermenge
(eval (cons + (teilermenge 12)) )      ; Summe aller Teiler von 12
(eval (cons + (teilermenge 720720)))
```

"Gesamte Spezifikation"

```
(define (teilermenge-i-bis-x i x)
    (if (> i x) ( )
        (if (= 0 (remainder x i))
            (cons i (teilermenge-i-bis-x (+ i 1) x))
            (teilermenge-i-bis-x (+ i 1) x) ) ) )
(define (teilermenge a) (teilermenge-i-bis-x 1 a))
(define (element? a L)
    (if (null? L) #f  (if (= a (car L)) #t (element? a (cdr L)) ) ) )
(define (durchschnitt L1 L2)
    (if (null? L1) ( )
        (if (element? (car L1) L2)
           (cons (car L1) (durchschnitt (cdr L1) L2))
           (durchschnitt (cdr L1) L2) ) ) )
(define (eins? L)  (and (not (null? L)) (null? (cdr L))) )
(define (maximum L)
    (if (eins? L) (car L)
        (if (< (car L) (maximum (cdr L))) (maximum (cdr L))  (car L)) ) )
(define  (ggT a b)
    (maximum (durchschnitt (teilermenge a) (teilermenge b))) )
```

```
(teilermenge 808038)  (teilermenge 720720) (ggT 808038 720720)
```

Den ggT kennen wir bereits gut:

Der ggT wurde in der Grundvorlesung in Abschnitt 1.6.3 (5) eingeführt, unter 1.7 zweites Beispiel rekursiv definiert und in Beispiel 2.4.3 ausführlich erläutert. In 2.4.4 wurde bewiesen, dass der euklidische Algorithmus den ggT korrekt ermittelt, und in 6.5.2 wurde nachgewiesen, dass die uniforme Zeitkomplexität O(n) und die Zeitkomplexität für die ziffernweise Bearbeitung $O(n^3)$ betragen (letzteres bei Verwendung der "Schulmethode" für die Berechnung des Restes).

Der euklidische Algorithmus euklid in seiner rekursiven Form kann direkt nach Scheme übertragen werden:

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

Experimentieren Sie nun:

Vergleichen Sie die Laufzeiten der beiden Programme ggT und euklid. Stellen Sie fest, für welche der einzelnen Funktionen die meiste Zeit verbraucht (überrascht?).

Beachten Sie: euklid ist für alle ganzen Zahlen definiert. Geben Sie daher auch negative Zahlen ein. Machen Sie sich die Funktion "remainder" für ganze Zahlen klar.

Geben Sie dann die Funktion euklid als Funktion auf ganzen Zahlen genau an. Zum Beispiel gilt:
(euklid -1 1) ==> 1, (euklid -1 1) ==> -1,
(euklid 4 -4) ==> -4, (euklid 4 -8) ==> 4,
(euklid 4 -5) ==> -1, (euklid 4 -7) ==> 1 usw.

Zeitmessungen mit einem (älteren) Laptop:

(teilermenge 808038) (teilermenge 720720) (Zeit: < 1 Sekunde)

(durchschnitt (teilermenge 808038) (teilermenge 720720)) (Zeit: < 1 Sekunde)

(ggT 808038 720720) (Zeit: 12 Sekunden)

Ergebnis: Um das Maximum der 24-elementigen Schnittmenge zu ermitteln, braucht das Programm wesentlich länger als zur Bildung der beiden Teilermengen und ihres Durchschnitts.
Woran liegt das?

Analysieren Sie die Funktion, die das Maximum berechnet.

Geben Sie eine schneller arbeitende Funktion zur Berechnung des Maximums des Durchschnitts an, die ausnutzt, dass die Teilermengen geordnet sind.
(Eine andere Lösung finden Sie am Ende von 2.2.4.)

kgV = kleinstes gemeinsames Vielfaches

Das kleinste gemeinsame Vielfache zweier natürlicher Zahlen a und b ist die kleinste natürliche Zahl v, für die gilt:

a teilt v und b teilt v.

Es gilt offensichtlich: $kgV(a,b) = v \leq a \cdot b$.

v muss alle Primfaktoren von a und von b enthalten, wobei deren Vielfachheit durch das Maximum der Vielfachheit in a oder b bestimmt wird. Da der ggT genau das Produkt der entsprechenden Minima der Vielfachheiten ist, erhält man

$$a \cdot b = ggT(a,b) \cdot kgV(a,b).$$

```
(define (kgV a b) (* (/ a (ggT a b)) b) )

(kgV  340  720)                ==>  12240
```

2.2.3 Zerlegung einer natürlichen Zahl in ihre Primfaktoren

Auch wenn Scheme mit wenigen Konzepten auskommt, erfordert es doch einige Übung, um Scheme-Programme zu lesen. Ohne Erläuterungen ist kaum etwas zu verstehen.

Wir wollen diesen schlechten Stil, Programme einfach ohne Erläuterungen hinzuschreiben, hier einmal weiter pflegen und präsentieren auf der nächsten Folie nur das Ergebnis.

Versuchen Sie, die Korrektheit dieses Programms nachzu-vollziehen. Beachten Sie, dass 1 keine Primzahl ist.

(reverse L) liefert die umgekehrte Reihenfolge einer Liste L.

```
(define  (Primfaktoren a Faktor Faktorliste)
    ( if (> (* Faktor Faktor) a) (cons a Faktorliste)
        ( if (= 0 (remainder a Faktor))
            (Primfaktoren (/ a Faktor) Faktor (cons Faktor
Faktorliste))
            (Primfaktoren a (+ Faktor 1) Faktorliste) ) ) )
```

```
(reverse (Primfaktoren 8192  2  '( )))
```

= Funktionskopf

= then-Teil

= if-Bedingungsteil

= else-Teil

Was ist für die Zahl 1? Obiges Programm gibt (1) aus, das Ergebnis muss jedoch die leere Liste sein. So erhält man:

Zerlegung einer Zahl in Primfaktoren

```
(define  (Primfaktoren a Faktor Faktorliste)
   ( if (> (* Faktor Faktor) a) (cons a Faktorliste)
      ( if (= 0 (remainder a Faktor))
         (Primfaktoren (/ a Faktor) Faktor (cons Faktor Faktorliste))
            (Primfaktoren a (+ Faktor 1) Faktorliste) ) ) )
(define (Primfaktorliste-von a)
   (if (<= a 1) () (reverse (Primfaktoren a 2 ()))))
```

```
(Primfaktorliste-von 84)          ==>  (2 2 3 7)
(Primfaktorliste-von 8192)        ==>  (2 2 2 2 2 2 2 2 2 2 2 2 2)
(Primfaktorliste-von 4331253)     ==>  (3 103 107 131)
(Primfaktorliste-von 101)         ==>  (101)
(Primfaktorliste-von 1)           ==>  ()
```

Erläuterung:

Primfaktorliste-von angewandt auf a soll die Faktorliste, also die Liste der Primfaktoren von a liefern. Die Faktorliste erhalten wir wie folgt:

Wir prüfen für die Variable Faktor nach, ob a durch Faktor teilbar ist, d.h., ob (= 0 (remainder a Faktor)) gilt. Hierbei beginnen wir mit der kleinsten Primzahl 2 = Faktor.

Falls ja, so füge man Faktor zur Faktorliste hinzu. Die weitere Prüfung muss sich nun auf a/Faktor beziehen, wobei wir nicht erneut mit 2 beginnen müssen (die Zahlen kleiner als Faktor wurden ja schon getestet), sondern ab dem Wert Faktor weiter prüfen können.

Falls nein, so prüfen wir, ob a durch (Faktor+1) teilbar ist, usw.

Das Verfahren bricht ab, sobald Faktor größer als die Wurzel von a, d.h., sobald Faktor*Faktor > a geworden ist. In diesem Fall ist a der letzte Primfaktor, der zur Faktorliste noch hinzuzufügen ist.

Der Sonderfall a=1 muss die leere Liste liefern.

Aufgabe 1: Ersetzen Sie in dem obigen Programm zur Primfaktorzerlegung das ">"-Zeichen durch ">=". Ist das Programm dann noch richtig (Beweis?) oder können Sie Beispiele angeben, für die das Programm dann fehlerhaft arbeitet?

Aufgabe 2: Hin und wieder benötigt man die Funktion

pr(n) = n-te Primzahl

also pr(1) = 2, pr(2) = 3, pr(3) = 5, ..., pr(10) = 29, ...

Schreiben Sie ein Scheme-Programm, das diese Funktion berechnet. Berechnen Sie hiermit mindestens 5 Werte, davon mindestens zwei mit n > 1000.
Wie lautet die 3000-ste Primzahl?
(Messen Sie die Laufzeit Ihres Programms für verschiedene n. Können Sie eine Abhängigkeit erkennen?
Vergleichen Sie Ihre Lösung mit den folgenden Folien.)

2.2.4 Die n-te Primzahl

Aufgabe: Programmiere die Funktion f(n) = n-te Primzahl.

Vorgehen: rekursiv nach der Vorschrift:

a) Die erste Primzahl ist 2.

b) Wenn p_n die n-te Primzahl ist, dann ist p_{n+1} die kleinste Primzahl, die größer als p_n ist.

Hilfsfunktionen, die hier offenbar vorkommen:

(prim? x) möge den Wahrheitswert, ob x eine Primzahl ist, liefern.

(nextprim x) möge die kleinste Primzahl, die größer als x ist, liefern.

Hilfsfunktion (prim? x)
Beschreibung: x ist genau dann eine Primzahl, wenn x>1 ist und x durch keine der Zahlen 2, 3, ..., x-1 teilbar ist.

Lösungsansatz 1: prim1?
x ist genau dann eine Primzahl, wenn die Teilermenge von x zweielementig ist, wobei (teilermenge x) in Beispiel 2.2.2 definiert wurde. Siehe nächste Folie.

Lösungsansatz 2: prim2?
Für x>1 muss mindestens eine der Zahlen 2, 3, ..., x die Zahl x teilen. Genau dann, wenn x die kleinste dieser Zahlen ist, ist x eine Primzahl. Siehe übernächste Folie.

Lösungsansatz 3: prim3?
Wie Ansatz 2, jedoch nur für 2, 3, ..., Wurzel(x) testen. Details selbst ausführen!

Lösungsvorschlag 1

```
(define (teilermenge-i-bis-x i x)
        (if (> i x)  ( )
                    (if (= 0 (remainder x i))
                        (cons i (teilermenge-i-bis-x (+ i 1) x))
                        (teilermenge-i-bis-x (+ i 1) x) ) ) )
(define (teilermenge a) (teilermenge-i-bis-x 1 a))

(define (zwei? L)
    (cond
        ((null? L) #f)
        ((null? (cdr L)) #f)
        ((null? (cdr (cdr L))) #t)
        (else #f) ) )
(define (prim1? x) (zwei? (teilermenge x)))

(prim1? 1) (prim1? 13) (prim1? 1337) (prim1? 494761)
(teilermenge 362880) (prim1? 362880)
```

Lösungsvorschlag 2

```
(define (kleinster-teiler-i-bis-x  i  x)
        (cond   ((= i x) x)
                ((= 0 (remainder x i)) i)
                (else (kleinster-teiler-i-bis-x  (+ i 1) x))   ) )
(define (prim2? x)
        (and (not (= x 1)) (= x (kleinster-teiler-i-bis-x  2  x))) )
```

;;; Eine Testreihe könnte sein:

```
(prim1? 0)  (prim1? -15)  (prim1? 1)  (prim1? 13)
(prim1? 1337)  (prim1? 494761)  (prim1? 100000001)

(prim2? 0)  (prim2? -15)  (prim2? 1)  (prim2? 13)
(prim2? 1337)  (prim2? 494761)  (prim2? 100000001)
```

;;; Ergebnis: #f #f #f #t #f #t #f #f #f #f #t #f #t #f

(Die Berechnung von (prim1? 100000001) dauert recht lange. Die Berechnung von prim2? ist deutlich schneller als die von prim1?.)

Hilfsfunktion (nextprim x)
berechnet die kleinste Primzahl, die größer als x ist.

Lösungsansatz: nextprim: $\mathbb{N} \rightarrow \mathbb{N}$ ist definiert durch
(nextprim x) = x+1, falls x+1 eine Primzahl ist
(nextprim x) = (nextprim (+ x 1)), sonst.

Hieraus folgt:
Die n-te Primzahl p_n erhält man, indem man n-mal nextprim auf die Zahl 1 anwendet.

In der Grundvorlesung (dort 2.5.5) hatten wir solche Funktionen die "Iterierte" genannt. Die Funktion $n \rightarrow p_n$ ist also die Iterierte von nextprim.

x	(nextprim x)
1	2
2	3
3	5
4	5
5	7
6	7
7	11
8	11
9	11

Wie beschreibt man in Scheme die Iterierte einer Funktion?

Formal ist die Iterierte von f die Funktion

(f-x-iter 0 x) = x (die 0-te Iterierte ist die Identität) und

(f-x-iter m x) = $\underbrace{(f\,(f\,(f\,(\,\ldots\,f}_{\text{m-mal}}(x)\,\ldots)\,)\,)$ für m > 0.

Übertragung nach Scheme:

```
(define  (f-x-iter m x)
         (if (= m 0)  x  (f (f-x-iter (- m 1) x)) ) )
```

Hinweis: Dies lässt sich aus Sicht der funktionalen Programmierung noch eleganter formulieren, siehe f-iter in Abschnitt 2.2.5.

Hieraus erhält man die gesuchte Funktion (wir verwenden *prim2?*):

```
(define (n-te-Primzahl n) (nextprim-iter n 1))

(define (nextprim-iter m x)
        (if (= m 0)  x  (nextprim (nextprim-iter (- m 1) x)) )  )

(define (nextprim x)
        (if (prim2? (+ x 1)) (+ x 1) (nextprim (+ x 1))) )

 (define (kleinster-teiler-i-bis-x  i  x)
        (cond   ((= i x) x)
                ((= 0 (remainder x i)) i)
                (else (kleinster-teiler-i-bis-x  (+ i 1) x))   ) )

(define (prim2? x)
        (and (not (= x 1)) (= x (kleinster-teiler-i-bis-x  2  x))) )

(n-te-Primzahl 25)     ;;; Ergebnis ist 97
```

Hinweis zur Effizienz: Indem man keine geraden Zahlen testet (also bei nextprim in Zweierschritten vorangeht und beim Modulo-Bilden nur ungerade Zahlen zulässt) und die Modulo-Bildung nur bis zur Wurzel von n durchführt (Test auf $i^2 > x$ statt $i = x$), lässt sich das Programm deutlich beschleunigen, so dass z. B. die 1.000.000-ste Primzahl in etwa 10 Minuten (2,3 GHz-Prozessor) ermittelt werden kann:

```
(define (n-te-Primzahl n) (if (<= n 1) 2 (nextprim-iter (- n 1) 1) ) )
(define (nextprim-iter m x)
          (if (= m 0)  x  (nextprim (nextprim-iter (- m 1) x)) )  )
(define (nextprim x)
          (if (prim2? (+ x 2)) (+ x 2) (nextprim (+ x 2))) )
 (define (kleinster-teiler-i-bis-x  i  x)
          (cond    ((> (* i i) x) x)
                   ((= 0 (remainder x i)) i)
                   (else (kleinster-teiler-i-bis-x  (+ i 2) x))   ) )
(define (prim2? x)
          (and (not (= x 1)) (= x (kleinster-teiler-i-bis-x  3  x))) )
(n-te-Primzahl 1000000)     ;;; Ergebnis ist 15.485.863
```

Hinweise zu den bisherigen Beispielen und das Schlüsselwort **let**:

ggT: Die Spezifikation in Beispiel 2.2.2 ist aufwendig zu programmieren, während der schnelle euklidische Algorithmus sehr einfach ist. Dass beide Verfahren auf den natürlichen Zahlen (ohne 0) die gleiche Funktion berechnen, hatten wir bereits in der Grundvorlesung bewiesen. Was euklid auf den negativen Zahlen genau berechnet, bleibt hier noch undurchsichtig.

Beispiel 2.2.3 war zunächst unkommentiert; man konnte daher kaum nachvollziehen, ob das Scheme-Programm korrekt arbeitet.

Die Spezifikation in 2.2.4 ist einfach (sofern man Primzahlen kennt). Das Programm zur Berechnung ist dagegen deutlich aufwendiger.

Die Zuverlässigkeit ist immer gefährdet, wenn man den vorgegebenen Wertebereich verlässt und dies auch für Eingabewerte erlaubt.

Der Zeitaufwand beim ggT ist zunächst nicht unmittelbar verständlich. Der Grund liegt in der "Baum-Rekursion" der Funktion maximum. Hier werden bereits berechnete Werte immer wieder neu berechnet. Mit Hilfe einer lokalen Zwischenspeicherung ("let") kann man dies vermeiden.

Syntax für solche Zuordnungen:

```
(let
      (   (name-1 "ausdruck-1")
          (name-2 "ausdruck-2")
           ...
          (name-k "ausdruck-k")
       )
      Rumpf des let-Ausdrucks
 )
```

Bedeutung: Den Namen "name-1" bis "name-k" wird der Wert des Ausdrucks "ausdruck-1", ... bzw. "ausdruck-k" zugeordnet. Hiermit wird dann der Rumpf ausgewertet.

"name-1" bis "name-k" sind also lokale Variable und außerhalb des let-Ausdrucks nicht sichtbar.

Beispiel: Indem man (maximum (cdr L)) an eine Variable m bindet, lässt sich die Maximum-Funktion aus 2.2.2 schreiben als:

```
(define  (eins? L)  (and (not (null? L)) (null? (cdr L))) )

(define  (maximum L)
    (if (eins? L) (car L)
        (let (( m (maximum (cdr L)) ))
            (if (< (car L) m)  m  (car L))
        )
    )
)
```

Mit dieser Änderung erfolgt die Maximumsbildung sehr schnell. (Ausprobieren!)

2.2.5 Die Wurzel aus einer natürlichen Zahl a

Ausgangspunkt mag folgender Limerick sein, den man sich mehrfach im Internet ergoogeln kann:

> An algebra teacher named Drew
> tried to find the square root of two.
> He found it between
> one fourth and fourteen,
> but couldn't get closer. Can you?

Dann wollen wir es einmal versuchen!

2.2.5.1: Betrachte a = 2. Weil Wurzel(2) die Länge der Diagonalen im Einheitsquadrat ist, kennen viele noch die Näherung Wurzel(2) ≈ 1,41421. In mathematischen Tafeln findet man 1,414213562373.
Wie und auf wie viele Ziffern kann man eine solche Zahl per Hand berechnen? Da man das Ergebnis z irgendwann verifizieren muss (sprich: es ist $z \cdot z \approx a$ nachzuprüfen), wofür man die Schulmethode der Multiplikation verwenden wird, gibt es bei etwa 500 Ziffern eine natürliche Grenze, da man hierfür $500 \cdot 500$, also 250.000 Einzelschritte ausführen muss, was mehrere Tage dauert. Mit Computern kommt man jedoch viel weiter.

Die folgenden Ausführungen finden sich im Buch von Nievergelt, Farrar und Reingold, "Computer Approaches to Mathematical Problems", Prentice Hall aus dem Jahre 1974, und später in manchen anderen Büchern.

Methode 1: Intervallschachtelung

function wurz2 (L, R: real) is
lokale Variable M: real := (L+R)/2.0;
if *genau genug angenähert* then *Ergebnis ist* M
else if M*M>2 then wurz2 (L, M) else wurz2 (M, R) fi fi
Man berechne dann wurz2 (1.0, 2.0).

Methode 2: Newtonsche Iteration

Starte mit $x_0 = 2.0$ und bilde (bis zu einem x_m) die Folge

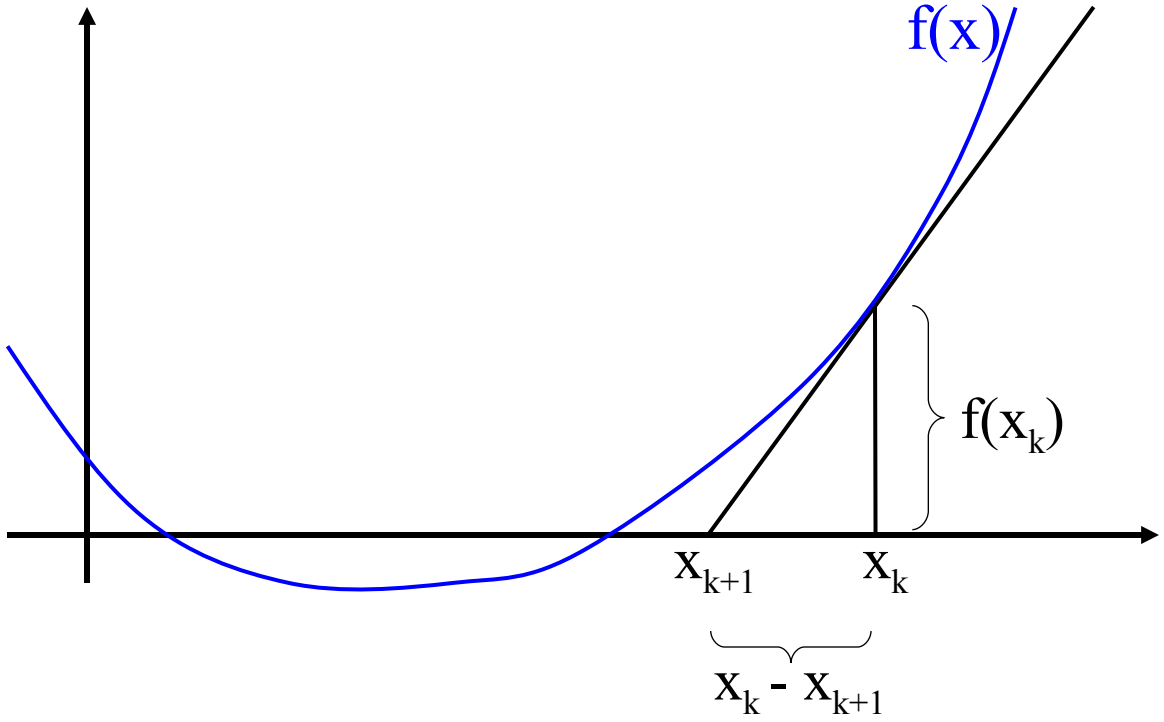
$$x_{k+1} = x_k/2.0 + 1.0/x_k$$

Diese Folge konvergiert gegen Wurzel(2).

function newton2 (x: real; m: natural) is
if *m=0* then *Ergebnis ist* x else newton2 (x/2.0+1.0/x, m-1) fi

Man berechne dann z. B. newton2 (2.0, 15).

Erinnerung: Herleitung der Newtonschen Iteration



$f'(x_k)$ ist die Steigung

im Punkt $(x_k, f(x_k))$.

$f'(x_k) = f(x_k) / (x_k - x_{k+1})$, also $x_{k+1} = x_k - f(x_k) / f'(x_k)$.

Setzt man $f(x) = x^2 - a$ mit $a = 2$ hier ein, so erhält man obige "Methode 2", die sich schrittweise der Nullstelle $f(x) = 0$ nähert.

2.2.5.2 Methode 3: mit Hilfe der Pellschen Gleichung

Es gibt den folgenden zahlentheoretischen Satz:

Wenn die natürliche Zahl a keine Quadratzahl ist, dann hat die Gleichung $p^2 - a \cdot q^2 = 4$ ("Pellsche Gleichung") unendlich viele Lösungen in natürlichen Zahlen p und q.

(Der Beweis garantiert, dass man, notfalls durch systematisches Probieren, ein p und ein q algorithmisch finden kann. Wenn es eine Lösung gibt, so gibt es auch unendlich viele Lösungen, wie auf der nächsten Folie bewiesen wird).

Der Satz klingt nicht sehr aufregend, jedoch bildet er den Ausgangspunkt für eine schnelle exakte Berechnung der Wurzel aus a mit Hilfe von Computern. "exakt" soll hier bedeuten, dass die Näherung in Form einer rationalen Zahl erfolgt, also durch Angabe eines Zählers und eines Nenners.

Zum theoretischen Hintergrund siehe Erläuterungen in 2.2.5.8.

Gegeben seien a, p_0 und q_0, so dass $p_0^2 - a \cdot q_0^2 = 4$ erfüllt ist.
Da $a \geq 2$ ist, muss $p_0 \geq 3$ und $p_0 > q_0 \geq 1$ gelten.

Bilde nun die Folge von natürlichen Zahlen p_k und q_k:

$$p_{k+1} = p_k^2 - 2 \quad \text{und} \quad q_{k+1} = p_k \cdot q_k$$

Behauptung: Alle p_i und q_i erfüllen dann ebenfalls die Pellsche Gleichung für dieses a, d. h., es gilt für alle i: $p_i^2 - a \cdot q_i^2 = 4$.

Beweis durch Nachrechnen: Für i = 0 ist dies nach Voraussetzung richtig. Sei die Behauptung bis zu einem i bewiesen. Betrachte dann

$$p_{i+1}^2 - a \cdot q_{i+1}^2 = (p_i^2 - 2)^2 - a \cdot (p_i \cdot q_i)^2 = p_i^4 - 4p_i^2 + 4 - a \cdot p_i^2 \cdot q_i^2$$

$$= p_i^2 \underbrace{(p_i^2 - 4 - a \cdot q_i^2)}_{= 0} + 4 = 4,$$

also erfüllen auch a, p_{i+1} und q_{i+1} die Pellsche Gleichung.

= 0, weil a, p_i und q_i die Pellsche Gleichung nach Induktionsannahme erfüllen.

■

Aus den Anfangsbedingungen $p_0 \geq 3$ und $p_0 > q_0 \geq 1$ und aus

$$p_{k+1} = p_k^2 - 2 \qquad \text{und} \qquad q_{k+1} = p_k \cdot q_k$$

erkennt man: Die Folge der Zahlen p_k und q_k wächst sehr stark, und zwar jedes Mal ungefähr auf die doppelte Länge.

Wir formen $p_k^2 - a \cdot q_k^2 = 4$ um: $\dfrac{p_k^2}{q_k^2} = a + \dfrac{4}{q_k^2}$

Hieraus folgt:

$$\frac{p_k}{q_k} \xrightarrow[k \to \infty]{} \sqrt{a}$$

und die Konvergenzgeschwindigkeit ist hoch, da sich die Zahlen beim Übergang von k nach k+1 in der Länge fast verdoppeln.

2.2.5.3 Der Fall a = 2.

Für a = 2, $p_0 = 6$ und $q_0 = 4$ gilt $p_0^2 - a \cdot q_0^2 = 4$. Bilde nun die Zahlen p_k und q_k gemäß $p_{k+1} = p_k^2 - 2$ und $q_{k+1} = p_k \cdot q_k$:

k	p_k	q_k
0	6	4
1	34	24
2	1154	816
3	1331714	941664
4	1773462177794	1254027132096

Für k = 3 stimmt $p_k/q_k \approx 1.41421356237469$ schon auf 11 Stellen nach dem Komma mit Wurzel(2) überein; für k = 4 sind es bereits mehr als 20 Stellen.

Nun konstruieren wir das zugehörige Scheme-Programm.
Die Funktion nextzn (= "nächster zähler-nenner") berechnet aus der Zweier-Liste (p, q) die Liste $(p^2 - 2, p \cdot q)$.
Dies müssen wir iterieren; wir bilden also **f-iter**, aber etwas allgemeiner als in 2.2.4. Am Ende muss man den Zähler p durch den Nenner q teilen, also "/" auf (p, q) anwenden. Der rationalen Zahl sieht man die Ziffernfolge der Dezimal-darstellung nicht an, daher stellen wir das Ergebnis "inexakt" auch durch "wu" dar. Die Anzahl der Iterationen sei k; wir definieren k in einem eigenen define-Ausdruck am Anfang.

```
;;; Betrachte folgende Definition für f-iter
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda  (x)  ((f-iter f (- m 1)) (f x)) )) )
```

```
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda  (x)  ((f-iter f (- m 1)) (f x)) )) )
```

Hier wird direkt die Funktion f-iter definiert - und nicht die Funktion f-x-iter aus 2.2.4 zusammen mit ihrem Argument x.

Wenn m = 0 ist, so wendet man die Identität auf den dann gegebenen aktuellen Parameter x an. Die Identität ist gegeben durch (lambda (x) x).

Anderenfalls wird die um eins verringerte Iteration auf die Funktion f angewendet.

f-iter erhält also das Argument x für die Funktion f nicht als Parameter und liefert somit am Ende keinen Wert, sondern: f-iter ist eine Funktion!

Die Funktion nextzn ist offensichtlich:

```
(define (nextzn x)
        (list (- (* (car x) (car x)) 2)  (* (car x) (car (cdr x)) ) ) )
```

Der Ausdruck

```
        (f-iter nextzn k)
```

liefert die k-fach iterierte Funktion von nextzn. Die Wurzel aus 2 erhält man, indem man diese Funktion auf die Anfangsliste '(6 4) ansetzt und auf das Ergebnis die Division anwendet:

```
(define (wurzel-exakt) (apply / ((f-iter nextzn k) '(6 4))))
```

Das Ergebnis ist hierbei eine rationale Zahl bestehend aus Zähler und Nenner. Um die Dezimaldarstellung auszugeben, muss man sich auf eine Ziffernfolge als Ausgabe beschränken, die nur eine Näherung ist. In Scheme schreibt man hierfür:

```
(define (wu) (exact->inexact (apply / ((f-iter nextzn k) '(6 4)))))
```

```
;;; Somit erhalten wir ein Scheme-Programm Wurzel(2)
(define k 4)
(define (nextzn x)
        (list (- (* (car x) (car x)) 2) (* (car x) (car (cdr x))) ) ) )
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda (x) ((f-iter f (- m 1)) (f x)) )) )
(define (wurzel-exakt) (apply / ((f-iter nextzn k) '(6 4))))
(define (wu) (exact->inexact (apply / ((f-iter nextzn k) '(6 4)))))
```

(wurzel-exakt) (wu) ==> *Das Ergebnis lautet für k = 3:*

1 195025 / 470832

1.4142135623746899

(wurzel-exakt) (wu) ==> *Das Ergebnis lautet für k = 4:*

1 259717522849 / 627013566048

1.4142135623730951

2.2.5.4 Der allgemeine Fall (a darf keine Quadratzahl sein!).

Für $a = 3$, $p_0 = 4$ und $q_0 = 2$ gilt $p_0^2 - a \cdot q_0^2 = 4$. Bilde nun die Zahlen p_k und q_k gemäß $p_{k+1} = p_k^2 - 2$ und $q_{k+1} = p_k \cdot q_k$:

k	p_k	q_k
0	4	2
1	14	8
2	194	112
3	37634	21728
4	1416317954	817711552

Für $k = 3$ stimmt $p_k/q_k \approx 1.7320508100147276$ auf 7 Stellen nach dem Komma mit Wurzel(3) überein; für $k = 4$ sind es mehr als 13 Stellen.

Man muss also nur die Anfangswerte p_0 und q_0 verändern, um die Wurzel einer anderen Zahl zu berechnen. Da es für $p \geq 3$ zu jedem q mit $p > q > 0$ ein rationales a mit $p^2 - a \cdot q^2 = 4$ gibt, kann man mit einem (p, q) starten und konvergiert dann stets gegen die Wurzel aus a. Die Zahl a liest man aus der Pellschen Gleichung ab: $a = (p^2-4)/q^2$.

Anfangswerte p und q		Quotient konvergiert gegen die Wurzel aus
3	1	5
3	2	1,25
4	1	12
4	2	3
4	3	1,333333...
5	1	21
5	2	5,25
5	3	2,333333...
5	4	1,3125
6	1	32
6	2	8
6	3	3,555555....

Anfangswerte p und q		Quotient konvergiert gegen die Wurzel aus	
6	4	2	
6	5	1,28	
7	1	45	
7	2	11,25	
7	3	5	
7	4	2,8125	
7	5	1,8	
7	6	1,25	
8	1	60	
8	2	15	
8	3	6,666666...	
8	4	3,75	***usw.***

2.2.5.5 Anfangswerte durch Ausprobieren finden

Wie findet man Anfangswerte p_0 und q_0 zu einer natürlichen Zahl a, die nicht Quadratzahl ist?
Der zahlentheoretische Satz besagt, dass es solche Werte stets gibt. Also probieren wir einfach alle Werte für p und q durch. Wir beginnen mit p=3 und q=1. Falls $p^2-a\cdot q^2-4=0$ ist, haben wir ein geeignetes Paar (p, q) gefunden. Ist dieser Wert dagegen kleiner als 0, so muss p erhöht werden, anderenfalls muss q erhöht werden. Dies führt direkt zur Funktion such, welche zu einem Tripel (p q a) das (bzgl. der Zahl p_0 kleinste) Tripel (p_0 q_0 a) ermittelt, das die Pellsche Gleichung erfüllt:

```
(define (such p q a)
  (let ((z (- (- (* p p) (* a q q)) 4)))
    (cond ((= z 0) (list p q a))
          ((> z 0) (such p (+ q 1) a))
          (else (such (+ p 1) q a)))))
```

Die kleinsten Startwerte für eine Zahl a erhalten wir als zweielementige Liste nun durch:

```
(define a ...)
(define (such p q b)
  (let ((z (- (- (* p p) (* b q q)) 4)))
    (cond ((= z 0) (list p q b))
          ((> z 0) (such p (+ q 1) b))
          (else (such (+ p 1) q b)))))
(define (startwerte x)
  (let ((y (such  3 1 x)))
    (list (car y) (car (cdr y)))))
(startwerte a)
```

Mit diesem Programm erhält man (6 4) für a = 2, (4 2) für a = 3, (48 10) für a = 23 und (1860498 83204) für a = 500.

[Berechnen Sie Startwerte für a = 13, 19, 46, 73 und 97.]

2.2.5.6 Scheme-Programm zur Berechnung von Wurzeln

```
;;; Zu definierende Funktionen für die Wurzelberechnung
(define (such p q b)
        (let ((z (- (- (* p p) (* b q q)) 4)))
                (cond ((= z 0) (list p q b))
                ((> z 0) (such p (+ q 1) b))
                (else (such (+ p 1) q b)))))
(define (startwerte x)
        (let ((y (such  3 1 x)))
                (list (car y) (car (cdr y)))))
(define (nextzn x)
        (list (- (* (car x) (car x)) 2) (* (car x) (car (cdr x)) ) ) )
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda (x) ((f-iter f (- m 1)) (f x)) )) )
```

```
;;; Nun kommen die Ein- und Ausgabefunktionen, z.B. für die
;;; Wurzel aus 11, berechnet mit 2 Iterationen. Mehrfach benötigte
;;; Zwischenwerte werden mittels define Variablen zu gewiesen:
;;; we für "wurzel-exakt" und wi für "16-stellige Wurzel".
(define k 2) (define a 11)
(define sw (startwerte a))
(define we (apply / ((f-iter nextzn k) sw)))
(define wi (exact->inexact (apply / ((f-iter nextzn k) sw))))
(define text "Differenz des Quadrats zu a: ")
(display "a = ") (display a) (display ",  Startwerte = ")
      (display (car sw)) (display " und ") (display (car(cdr sw)))
      (display ",  Iterationen = ") (display k) (newline)
(display we) (newline) (display (* we we)) (newline)
      (display text) (display (- a (* we we))) (newline)
(display wi) (newline) (display (* wi wi)) (newline)
      (display text) (display (- a (* wi wi)))
```

Berechung der Wurzel von 11:

;;; Ausgabe für a = 11 und k = 2
;;; Die zweite Zeile gibt die Näherung der Wurzel von a durch
;;; die rationale Zahl we an, die dritte Zeile ist we*we, also
;;; ungefähr die Zahl a. Es folgt die Abweichung, also die
;;; Differenz a-we*we. In den drei folgenden Zeilen wird dies
;;; wiederholt für die reellwertige "inexact"-Näherung wi.

a = 11, Startwerte = 20 und 6, Iterationen = 2
79201/23880
6272798401/570254400
Differenz des Quadrats zu a: -1/570254400
3.3166247906197657
11.000000001753605
Differenz des Quadrats zu a: -1.753605261001212e-009

Hinweis: Setzt man k=16, so erhält man nach einigen Sekunden eine rationale Zahl mit je rund 85000 Stellen für Zähler und Nenner, deren Quadrat sich nur um ein 170.000-stel von 11 unterscheidet.

Berechung der Wurzel von 1000:

;;; Ausgabe für a = 1000 und k = 3, lange Laufzeit für die Startwerte!

a = 1000, Startwerte = 78960998 und 2496966, Iterationen = 3

755563613246946770311148171086796654566106591278642221001248001 /
2389301935006921131445286361035871158018128863911187911704136

570876373662781757050927200016471405619957718253680442493895955
916001346687858740383097313789890205078575145409019995064960001 /
570876373662781757050927200016471405619957718253680442493895955
916001346687858740383097313789890205078575145409019995064960001

Differenz des Quadrats zu a: -1 /
570876373662781757050927200016471405619957718253680442493895955
916001346687858740383097313789890205078575145409019995064960001

31.622776601683793

1000.0

Differenz des Quadrats zu a: 0.0

2.2.5.7 Diskussion der Effizienz

Das Programm ist nun relativ einfach, da es im Wesentlichen nur aus einer Suchfunktion für die Startwerte und dann einer iterierten Auswertung eines Ausdrucks besteht.

Ist es für die Praxis tauglich? Leider nein.

Der Grund liegt in der Berechnung der Startwerte. Das Ausprobieren ist nicht effizient, weil die Werte sehr groß sein können. Standardbeispiel hierfür ist die Zahl a = 97, deren kleinste Startwerte in Dezimaldarstellung neun- bzw. achtstellig sind. (Es gibt mathematische Methoden zur schnelleren Berechnung der Startwerte, die auf der Theorie der Kettenbrüche basieren.)

Wenn man jedoch die Startwerte kennt, dann ist das Verfahren sehr effizient. Für die dezimale Ziffernfolge müssen am Ende die Zahlen p_k und q_k allerdings noch dividiert werden, doch das ist kein allzu schweres Problem - versuchen Sie es einmal!

Für den Hausgebrauch reicht Lösungsmethode 1 aus, auch wenn die ständige Multiplikation M*M bei wachsender Genauigkeit zum entscheidenden Zeitfaktor wird:

```
(define eps 1.0e-300)
(define (wurzel L R a)
  (let ((M (/ (+ L R) 2)))
      (define diff (- a (* M M)))
      (if (> eps (abs diff))
          M
          (if (> diff 0) (wurzel M R a) (wurzel L M a))  )))
```

Programm für die Lösungsmethode 1

Mit dem Ausdruck (wurzel 1 a a) wird die Wurzel einer Zahl a sehr schnell auf dreihundert Stellen genau berechnet.
Hinweis: In DrScheme wird 1.0e-324 gleich 0.0 gesetzt. Für größere Genauigkeit müssen Sie dann die "exakte Darstellung" mit #e anfordern, also z. B. (define eps #e1.0e-800) schreiben.

Schneller als Lösungsmethode 1 arbeitet das Newtonverfahren (Lösungsmethode 2). Für eine Zahl a > 1 beginnt man mit $x_0 = a$ und bildet die Folge: $x_{k+1} = x_k/2 + a/(2x_k)$, die gegen Wurzel(a) konvergiert. Dies liefert das Programm:

```
(define k 19) (define a 2)
(define (next x) (+ (/ x 2) (/ a (* x 2))))
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda (x) ((f-iter f (- m 1)) (f x)) )) )
(define wurzel ((f-iter next k) a))
wurzel (newline) (- a (* wurzel wurzel))
```

Programm für die Lösungsmethode 2

Mit diesem Programm wird Wurzel(2) leicht auf 400.000 Stellen genau berechnet. (Wir haben also dem Algebralehrer Drew, siehe Anfangsfolie von 2.2.5, bestens helfen können.)

Konkrete Messung: Um die Wurzel von 2 auf 100.000 Stellen genau zu berechnen, benötigte die Lösungsmethode 1 (Intervallschachtelung) rund 5 Minuten, während auf dem selben Rechner die Lösungsmethoden 2 (Newton-Verfahren) und 3 (Pellsche Gleichung) für die gleiche Genauigkeit je 4 bis 6 Sekunden brauchten.

Lösungsmethode 2 brauchte 35 Sekunden für die Genauigkeit von 400.000 Stellen. Lösungsmethode 3 benötigte hierfür nur 16 Sekunden (allerdings ohne Ausgabe, die bei Methode 3 deutlich mehr Zeit benötigt als bei Methode 2, dies liegt wohl an der unterschiedlichen internen Darstellung).
Lösungsmethode 1 arbeitet deshalb so langsam, weil sich bei jeder Iteration die Zahl der gültigen Stellen nur um konstant viele Stellen erhöht, während sie sich bei den Methoden 2 und 3 fast verdoppelt.

2.2.5.8 Theoretische Ergänzungen

Wenn die natürliche Zahl a keine Quadratzahl ist, so ist
sqrt(a) = die Wurzel aus a
keine rationale Zahl. Dies war den Griechen schon vor 2300 Jahren bekannt.

[*Beweis durch Widerspruch:* Wäre sqrt(a) eine rationale Zahl, so wäre sqrt(a) = r/s, d. h., $a = r^2/s^2$ für natürliche Zahlen r und s. Ohne Beschränkung der Allgemeinheit kann man erstens r und s als teilerfremd annehmen, d.h., ggT(r,s) = 1, und zweitens a als eine Zahl, die keine Quadratzahl als Teiler hat. (Anderenfalls wäre $a = a' \cdot k^2$ und man könnte $a' = r^2/(s \cdot k)^2$ an Stelle von $a = r^2/s^2$ betrachten.) Also kann man $a = b \cdot c$ annehmen mit einer Primzahl b, die c nicht teilt. Dann muss b aber wegen $a \cdot s^2 = r^2$ ein Teiler von r sein. Beim Kürzen durch b bleibt ein b rechts stehen, woraus folgt, dass b auch ein Teiler von s sein muss. Widerspruch, da wir ggT(r,s) = 1 annehmen konnten.]

Wenn a keine Quadratzahl ist, dann ist also

$$a = r^2/s^2 \quad \text{bzw.} \quad r^2 - a \cdot s^2 = 0$$

nicht lösbar in natürlichen Zahlen. Aber eine Näherung für sqrt(a) würde man erhalten, wenn

$$r^2 - a \cdot s^2 = 1 \quad \text{oder} \quad r^2 - a \cdot s^2 = -1$$

lösbar wäre; denn dann wäre

$$a = (r^2 \pm 1)/s^2 = r^2/s^2 \pm 1/r^2 \text{ , also } \text{sqrt}(a) \approx r/s,$$

und dies umso genauer, je größer s ist.

Gleichungen der Form $\mathbf{x^2 - a \cdot y^2 = \pm 1}$ sind daher bereits im alten Griechenland untersucht worden. Sie heißen heute Pellsche Gleichungen (nach J. Pell, engl. Mathematiker, 1611-1685, obwohl dieser keine substanziellen Beiträge zur Lösung dieser Gleichung beigetragen hat).

Wenn a eine Quadratzahl ist, also $a=d^2$, dann hat ($t=d\cdot s$)

$$r^2 - a\cdot s^2 = \pm 1 = r^2 - t^2 = (r+t)\cdot(r-t)$$

offensichtlich keine Lösung in natürlichen Zahlen $\mathbb{N} = \{1, 2, 3, ...\}$. Also muss man sich auf solche Zahlen a beschränken, die keine Quadratzahlen sind. Man kann allerdings den Wertebereich für a auf rationale Zahlen (und sogar noch weiter) ausdehnen, wie wir es in 2.2.5.4 getan haben.

Wenn $x^2 - a\cdot y^2 = 1$ eine Lösung hat, dann hat natürlich auch $(2x)^2 - a\cdot(2y)^2 = 4$ eine Lösung, weshalb wir unsere Gleichung $p^2 - a\cdot q^2 = 4$ ebenfalls als Pellsche Gleichung bezeichnet haben (mit $p = 2x$ und $q = 2y$).

Betrachten wir nun also die Pellsche Gleichung $x^2 - a \cdot y^2 = 1$.
Es sei wiederum d = sqrt(a) die Wurzel von a, also $d^2 = a$.

Wir nehmen an, wir hätten eine Lösung (x_1, y_1) dieser Gleichung gefunden. Dann gilt also

$$x_1^2 - a \cdot y_1^2 = (x_1 + d \cdot y_1) \cdot (x_1 - d \cdot y_1) = 1.$$

Quadriere beide Seiten:

$(x_1 + d \cdot y_1)^2 \cdot (x_1 - d \cdot y_1)^2 = 1$, das heißt:

$$\left((x_1^2 + a \cdot y_1^2) + 2d \cdot x_1 y_1\right) \cdot \left((x_1^2 + a \cdot y_1^2) - 2d \cdot x_1 y_1\right)$$
$$= (x_1^2 + a \cdot y_1^2)^2 - d \cdot (2 \cdot x_1 y_1)^2 = 1.$$

Folglich erfüllen auch die Zahlen

$$x_2 = x_1^2 + a \cdot y_1^2 \quad \text{und} \quad y_2 = 2 \cdot x_1 y_1$$

die Pellsche Gleichung. Das Gleiche gilt dann für $x_4 = x_2^2 + a \cdot y_2^2$ und $y_4 = 2 \cdot x_2 y_2$ usw. Man kann dies noch ein wenig vereinfachen, indem man Folgendes ausnutzt:

x_1 und y_1 erfüllen die Pellsche Gleichung; es gilt also $x_1^2 - a \cdot y_1^2 = 1$ und somit $a \cdot y_1^2 = x_1^2 - 1$.

Folglich ist $x_2 = x_1^2 + a \cdot y_1^2 = 2x_1^2 - 1$ und $y_2 = 2 \cdot x_1 y_1$.

[Setzt man $x_1 = p_k/2$, $y_1 = q_k/2$, $x_2 = p_{k+1}/2$ und $y_2 = q_{k+1}/2$ hier ein, so erhält man genau die Rekursionsformeln $p_{k+1} = p_k^2 - 2$ und $q_{k+1} = p_k \cdot q_k$ aus Abschnitt 2.2.5.2.]

Allgemein kann man beide Seiten statt zu Quadrieren auch "hoch n" nehmen für jede natürliche Zahl $n \geq 1$. Man beachte, dass es stets Zahlen x_n und y_n gibt, so dass gilt:

$(x_1 + d \cdot y_1)^n = x_n + d \cdot y_n$ [und zugleich: $(x_1 - d \cdot y_1)^n = x_n - d \cdot y_n$]; den Fall $n = 2$ haben wir oben behandelt.

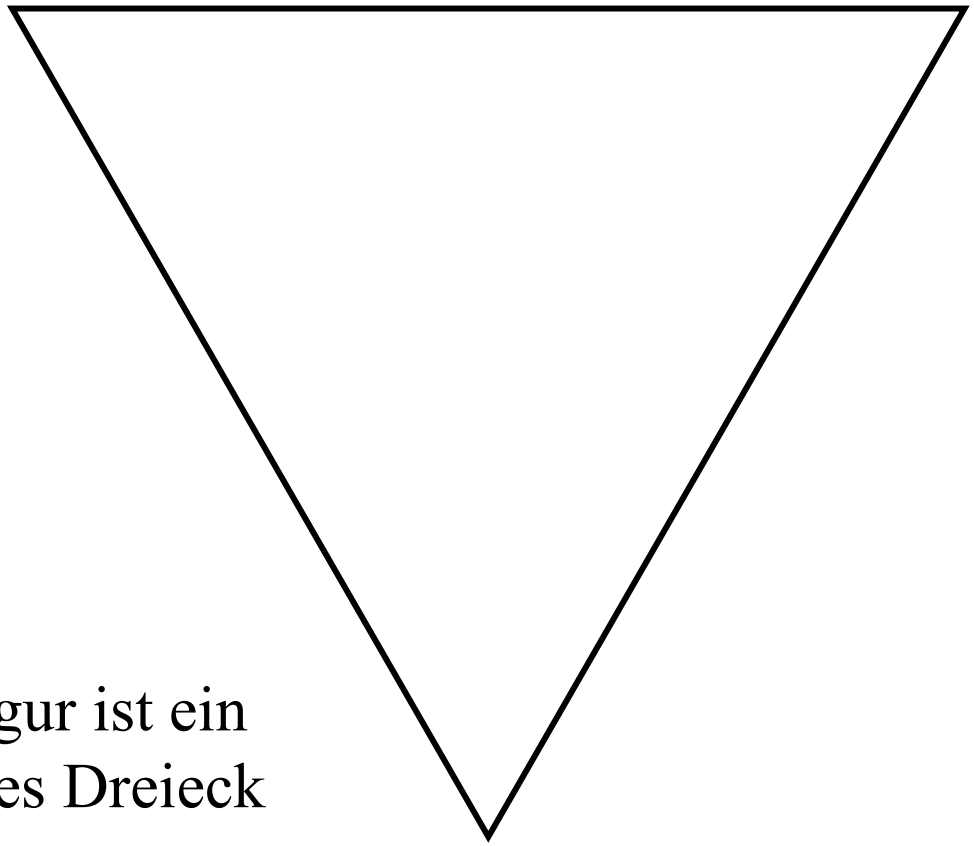
Jedes solche Paar (x_n, y_n) erweist sich dann als Lösung der Pellschen Gleichung.

In der Zahlentheorie wird bewiesen, dass es für jedes a, das nicht Quadratzahl ist, eine Lösung (x_1, y_1) gibt, so dass die genannten Paare (x_n, y_n) alle natürlich-zahligen Lösungen der Pellschen Gleichen sind. (x_1, y_1) ist das Paar mit den kleinsten Lösungs-Zahlen und wird "Fundamentallösung" genannt.
Die Fundamentallösung kann man aus der Kettenbruchdarstellung von d = sqrt(a) konstruieren. Hierzu verweisen wir auf die (oftmals recht alte) Literatur über Kettenbrüche.

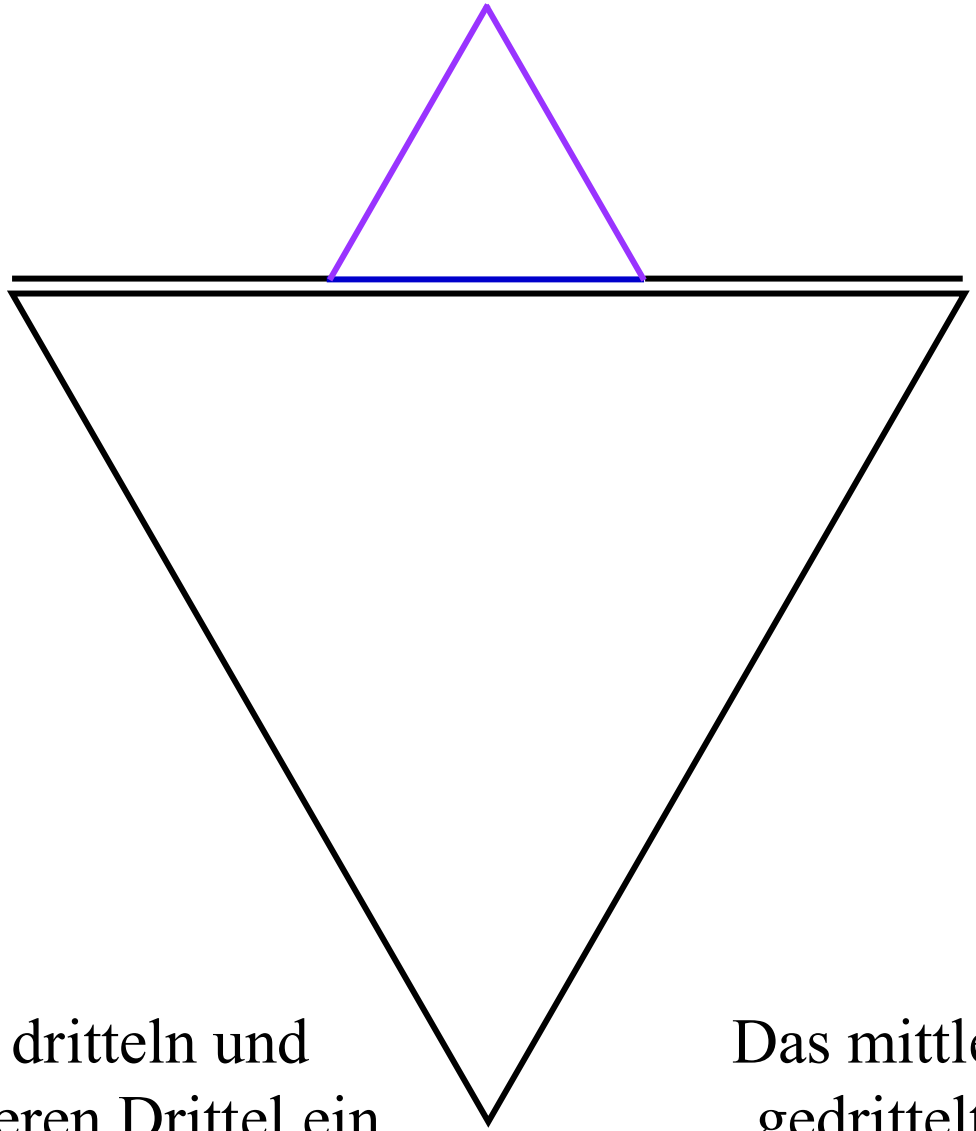
Ein kurioses Beispiel zur Verwendung der Pellschen Gleichung ist das Rinderproblem des Archimedes, siehe Artikel von F. Schwarz in "mathPAD, Band 10, August 2001, Seite 42-47", als pdf im Internet:
http://www.mupad.com/mathpad/2001_1/mathPAD_Vol10_art11.pdf

2.2.6 Zeichnen

Kochsche Seitendrittelungen (Schneeflocken)

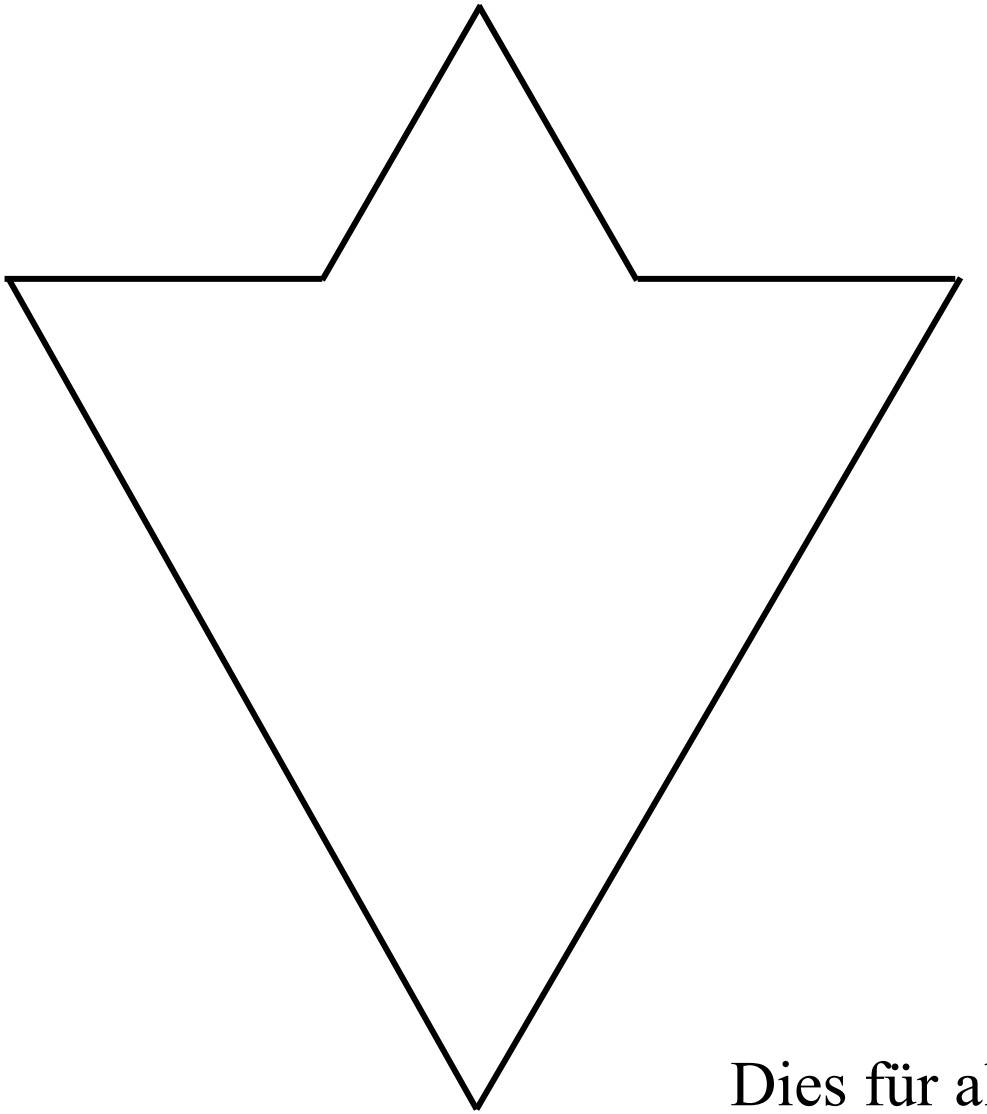


Ausgangsfigur ist ein
gleichseitiges Dreieck

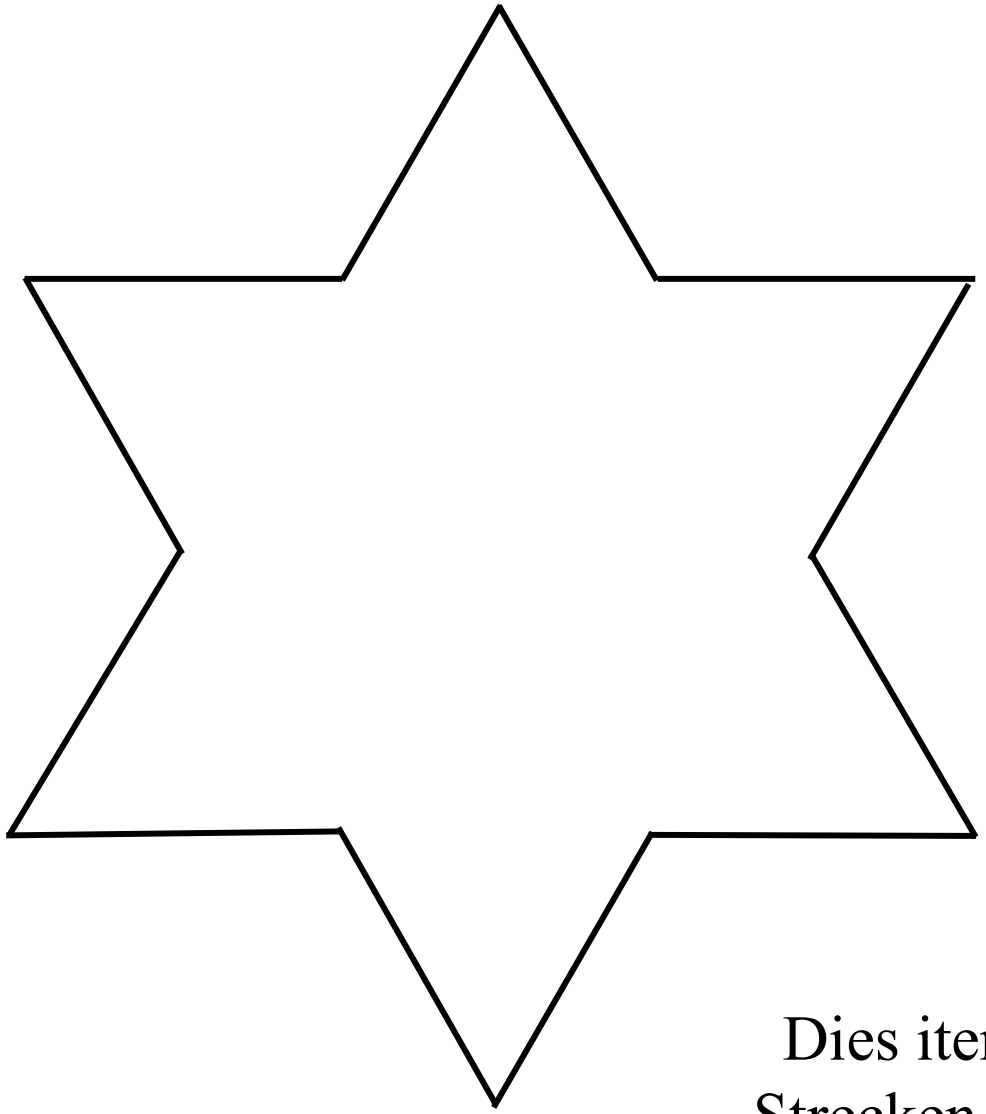


Eine Seite dritteln und auf dem mittleren Drittel ein gleichseitiges Dreieck aufsetzen

Das mittlere Drittel der gedrittelten Seite nun weglassen. Dies liefert:

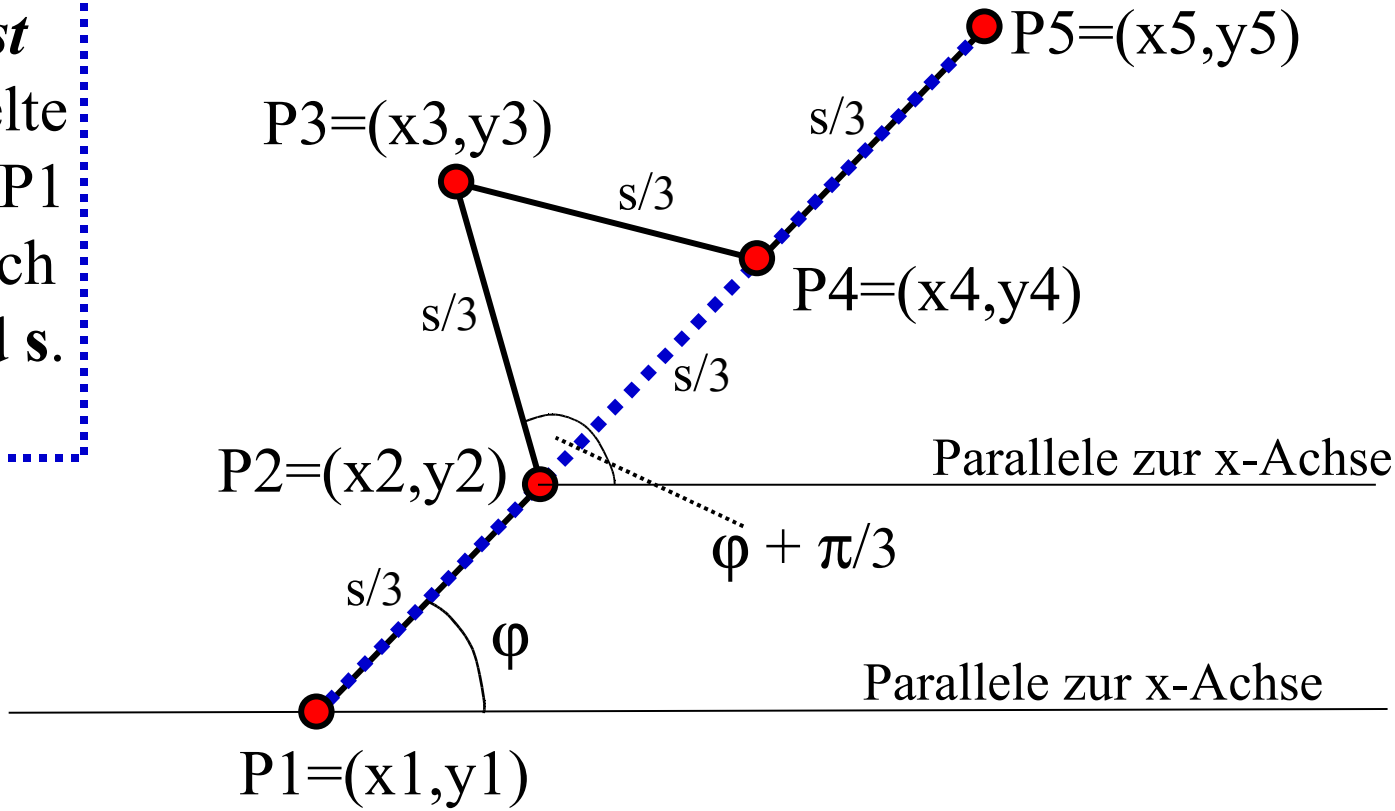


Dies für alle drei Seiten
des Dreiecks durchführen:



Dies iterativ für alle Strecken wiederholen.

Gegeben ist
die gestrichelte Strecke von P1 nach P5 durch **x1, y1, φ** und **s**.



s = Länge der Strecke von P1 nach P5
s/3 = Entfernung P1 nach P2 und P4 nach P5
φ = Winkel der Strecke mit der x-Achse (im Bogenmaß!)

Nun müssen wir für den allgemeinen Fall die vier Strecken P1-P2, P2-P3, P3-P4 und P4-P5 durch die Werte x1, y1, φ, s beschreiben.

Strecke P1-P2: x1, y1, φ, s/3.

Strecke P2-P3: x2, y2, φ+π/3, s/3

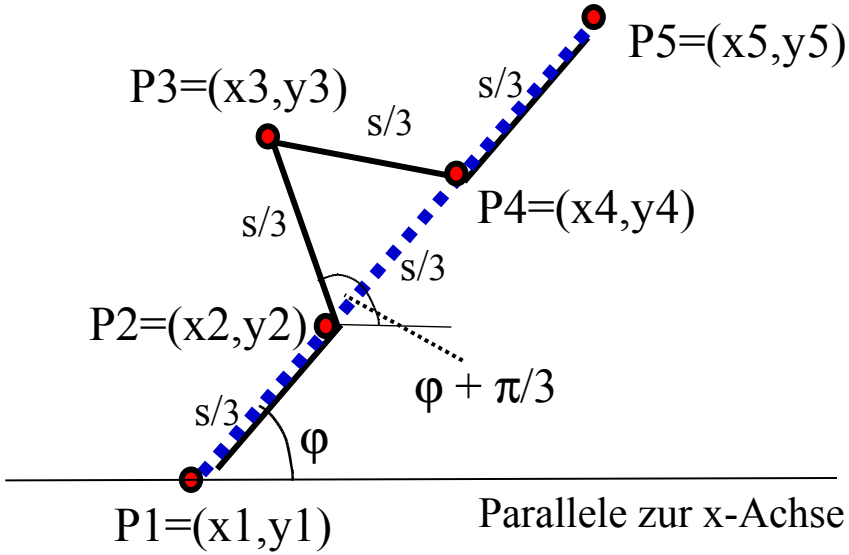
mit x2 = x1 + (s/3)·cos(φ)

und y2 = y1 + (s/3)·sin(φ).

Strecke P3-P4: x3, y3, φ-π/3, s/3

mit x3 = x2 + (s/3)·cos(φ+π/3)

und y3 = y2 + (s/3)·sin(φ+π/3).



Strecke P4-P5: x4, y4, φ, s/3

mit x4 = x1 + (2s/3)·cos(φ) und y4 = y1 + (2s/3)·sin(φ).

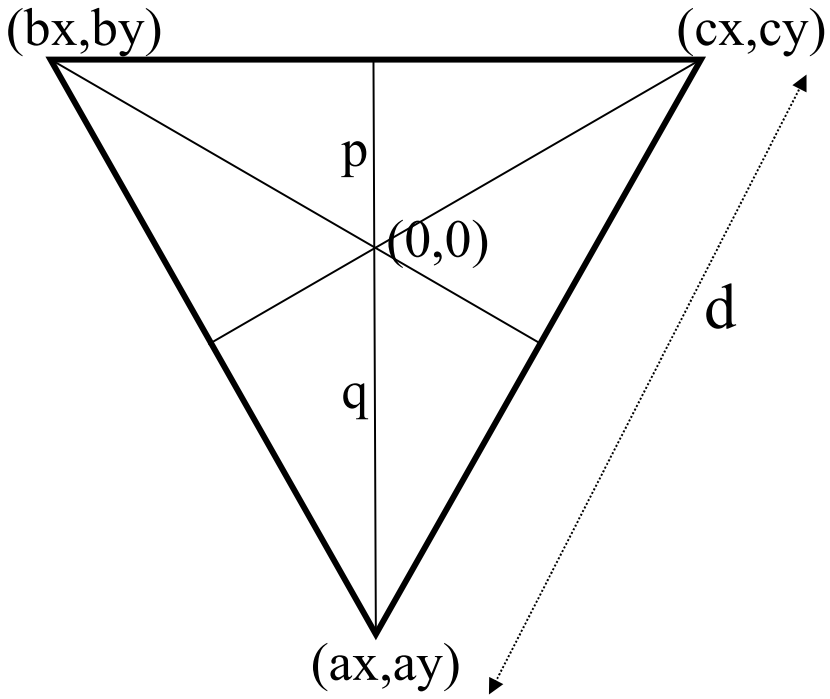
Dies iterieren wir für jede Strecke, bis die Länge einer Strecke wegen der Drittelungen unterhalb einer vorzugebenden Größe gesunken ist; in diesem Fall zeichnen wir die Strecke P1-P5 und beenden die jeweilige Rekursion.

Nun brauchen wir nur noch die "Zeichnen"-Ausgabefunktion. Wir legen ohne Rücksicht auf die Realität fest: (draw-line x1 y1 x2 y2). Die Einheit ist cm und der Koordinatenursprung liegt in der Mitte des Bildschirms.

Wir bezeichnen den Schritt, ein Dreieck auf eine Strecke aufzusetzen, hier mit "dorn", da das Ergebnis eine Art Dorn ist.

```
(define (dorn x1 y1 phi s Min)
    (if (< s Min)
        (draw-line x1 y1 (+ x1 (* s (cos phi))) (+ y1 (* s (sin phi))))
        (let* ((s-drittel (/ s 3.0))  (zwei-s-drittel (+ s-drittel s-
drittel))
                    (x2 (+ x1 (* s-drittel (cos phi))))
                    (y2 (+ y1 (* s-drittel (sin phi))))
                    (x3 (+ x2 (* s-drittel (cos (+ phi pi-drittel)))))
                    (y3 (+ y2 (* s-drittel (sin (+ phi pi-drittel)))))
                    (x4 (+ x1 (* zwei-s-drittel (cos phi))))
                    (y4 (+ y1 (* zwei-s-drittel (sin phi)))) )
            (dorn x1 y1 phi s-drittel Min)
            (dorn x2 y2 (+ phi pi-drittel) s-drittel Min)
            (dorn x3 y3 (- phi pi-drittel) s-drittel Min)
            (dorn x4 y4 phi s-drittel Min)  ) ) )
```

Dies war die wörtliche Übertragung der Herleitungen nach Scheme. Die Ausgangspunkte A=(Ax,Ay), B=(Bx,By) und C=(Cx,Cy) sind durch den Koordinatenursprung im Schwerpunkt des gleichseitigen Dreiecks und die Seitenlänge "Strecke" d gegeben. Aus der Skizze und dem Pythagoras ermittelt man leicht folgende Gleichungen:



$p+q = h$ (Höhe),
$p^2+(d/2)^2 = q^2$,
$h^2 + (d/2)^2 = d^2$,
$ax = 0,\ ay = -q$,
$bx = -d/2,\ by = p$,
$cx = d/2,\ cy = p$.
Es folgt:
$p = (d/2)/\text{Wurzel}(3)$,
$q = d/\text{Wurzel}(3) = 2p$.

```
(define pi 3.14159265358979323846)
(define pi-drittel (/ pi 3.0))
(define (Schneeflocke d Min)
(let*  ((Ax 0) (Ay (/ (- d) (sqrt 3.0)))
         (Bx (/ (- d) 2)) (By (/ Ay 2)) (Cx (- Bx)) (Cy By))
    (dorn Ax Ay (* 2 pi-drittel) d Min)
    (dorn Bx By 0 d Min)
    (dorn Cx Cy (+ pi pi-drittel) d Min) ) )
(Schneeflocke 300 10)
```

Wegen eines Softwarefehlers der DrScheme-Version konnte dieses Programm nicht getestet werden. Dies sei daher den Leser(inne)n überlassen. (Hinweise: Man verwende im Teachpack draw.ss. den Datentyp "posn" für Positionen; Funktion make-posn erzeugt eine Position aus zwei Zahlen, (draw-solid-line P1 P2 'blue) zieht eine blaue Linie von posn P1 nach posn P2, (start a b) liefert eine Zeichenfläche von a mal b Pixeln. Weiterhin liegt der Koordinatenursprung unten links.)

Gesamtes Programm (mit dem Teachpack draw.ss und Verschieben des Ursprungs):

```
(define (dorn x1 y1 phi s Min)
      (if (< s Min)
            (draw-solid-line (make-posn x1 y1) (make posn (+ x1 (* s (cos phi))) (+ y1 (* s (sin phi)))))
            (let* ((s-drittel (/ s 3.0))  (zwei-s-drittel (+ s-drittel s-drittel))
                        (x2 (+ x1 (* s-drittel (cos phi))))
                        (y2 (+ y1 (* s-drittel (sin phi))))
                        (x3 (+ x2 (* s-drittel (cos (+ phi pi-drittel)))))
                        (y3 (+ y2 (* s-drittel (sin (+ phi pi-drittel)))))
                        (x4 (+ x1 (* zwei-s-drittel (cos phi))))
                        (y4 (+ y1 (* zwei-s-drittel (sin phi)))) )
             (dorn x1 y1 phi s-drittel Min)
             (dorn x2 y2 (+ phi pi-drittel) s-drittel Min)
             (dorn x3 y3 (- phi pi-drittel) s-drittel Min)
             (dorn x4 y4 phi s-drittel Min)  ) ) )
(define pi 3.14159265358979323846)
(define pi-drittel (/ pi 3.0))
(start 400 400)  -- Koordinatenursprung liegt zunächst unten links, d.h., nach (200,200) verschieben
(define (Schneeflocke d Min)
(let*((Ax 200) (Ay (+ (/ (- d) (sqrt 3.0)) 200))
            (Bx (+(/ (- d) 2) 200)) (By (+ (/ Ay 2) 200)) (Cx (- 200 Bx)) (Cy By))
      (dorn Ax Ay (* 2 pi-drittel) d Min)
      (dorn Bx By 0 d Min)
      (dorn Cx Cy (+ pi pi-drittel) d Min) ) )
(Schneeflocke 300 10)
```

2.2.7 m gültige Ziffern (dezimal)

Das eigentliche Ziel lautet: Drucke eine Tabelle der Funktion f.
Vom Benutzer sind später vorzugeben:
- Funktion f
- Genauigkeit der Berechnung als Zahl der Ziffern m
- kleinster x-Wert minx
- größter x-Wert maxx
- Schrittweite deltax.

Weiterhin muss die Anzahl maxz-pro-z der Zeichen, die pro Zeile maximal gedruckt werden dürfen, und die Anzahl z-pro-z der Zeichen, die pro Zeile gedruckt werden sollten, festgelegt werden.

Wir betrachten nur folgendes Teilproblem: Gegeben sind eine Zahl $x > 0$ und eine natürliche Zahl m. Ermittle zu x und m (= Genauigkeit) die Darstellung $x = g \cdot 10^e$ mit einer ganzen Zahl g, die aus genau m Dezimalziffern besteht und einer ganzen Zahl e.
Es gilt also: $10^{m-1} \leq |g| < 10^m$, wodurch die Darstellung eindeutig ist.
Imperativer Algorithmus: Setze e auf Null.
Falls $|g| \geq 10^m$ ist, betrachte g := g/10 und e := e+1,
falls $|g| < 10^{m-1}$ ist, betrachte g := g*10 und e := e-1.
So erhält man schließlich die gewünschte Darstellung $x = g \cdot 10^e$.
Daraus folgt für die Berechnung von g und e in Scheme (hier muss man noch zwischen exakter und inexakter Darstellung unterscheiden):

```
(define (berechne-g  x  m)
    (cond  ((> (expt 10 (- m 1)) x) (berechne-g (* x 10) m))
           ((>= x (expt 10 m)) (berechne-g (/ x 10) m))
           (else x)  ) )
(define (berechne-e x m)
    (inexact->exact (round (log10 (/ x (berechne-g x m))))))
(define (log10 x) (/ (log x) (log 10)))
```

```
(define (berechne-g  x  m)
    (cond  ((> (expt 10 (- m 1)) x) (berechne-g (* x 10) m))
           ((>= x (expt 10 m)) (berechne-g (/ x 10) m))
           (else x)  ) )
(define (berechne-e x m)
    (inexact->exact (round (log10 (/ x (berechne-g x m))))))
(define (log10 x) (/ (log x) (log 10)))
;; x muss positiv sein und m eine natürliche Zahl
(define  x  0.123456789) (define  m  2)
(display x) (newline) (display (exact->inexact x)) (newline)
(display m) (newline) (newline)
(display (berechne-g x m)) (newline)
(display (exact->inexact (berechne-g x m))) (newline)
(display (inexact->exact (berechne-e x m))) (newline)
(display (inexact->exact (round (exact->inexact (/ (berechne-g x (+ m 1)) 10 )))))
(display " mal 10 hoch ") (display (berechne-e x m)) (newline)
```

Programm zur dezimalen Ausgabe einer positiven Zahl x mit genau m vorgegebenen gültigen Ziffern

Ausgabe des obigen Programms zur dezimalen Ausgabe von positiven Zahlen x mit m vorgegebenen gültigen Ziffern

```
;Ergebnis für obige Werte:
0.123456789
0.123456789
4

1234.5678899999998
1234.5678899999998
-4
1235 mal 10 hoch -4
```

```
;Ergebnis für x=317/7 und m=7:
317/7
45.285714285714285
7

31700000/7
4528571.428571428
-5
4528571 mal 10 hoch -5
```

```
;Ergebnis für x=0.006543, m=6:
0.006543
0.006543
6

654300.0000000001
654300.0000000001
-8
654300 mal 10 hoch -8
```

Was geschieht für m=0?

2.3 Funktionen höherer Ordnung

2.3.1 Erläuterung

Aus der Mathematik sind wir "Meta-Ebenen" gewohnt: Wir führen Elemente ein, bilden hieraus eine Menge, bilden aus den Mengen eine Menge von Mengen (z.B. die Potenzmenge), formen hieraus eine Menge von Mengen von Mengen usw. Dies übertragen wir auch auf Funktionen: Wir ordnen durch eine Funktion Elementen andere Elemente zu, und diese Elemente können wiederum Funktionen sein.

Auch in der Informatik gehen wir ständig mit Algorithmen um, die nicht auf "normale Daten", sondern auf Algorithmen angesetzt werden und als Ergebnis wieder Algorithmen liefern. Am bekanntesten sind Compiler; auch Anpassungen und die Programmierung von Benutzungsoberflächen gehören hierzu.

Funktionen, deren Argumente oder deren Ergebnis Funktionen sind, bezeichnet man als Funktionen höherer Ordnung.

Standardbeispiele aus der Mathematik sind die Integration und die Differentiation. Das unbestimmte Integral, das einer Funktion seine Stammfunktion zuordnet, ist z. B. eine Funktion höherer Ordnung.

Bereits behandelt hatten wir die Iteration einer Funktion f:

```
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda  (x)  ((f-iter f (- m 1)) (f x)) )) )
```

Ein anderes Beispiel ist die Bearbeitung aller Elemente einer Liste mit der gleichen Funktion:

```
(define (summe L) (if (null? L) 0 (+ (car L) (summe (cdr L)))))

(summe '(3 4 5))          ==> 12
(summe (list (summe '(1 2 3))  (summe '(4 5)) (summe '(6 7 8 9))))
                          ==> 45

(define (nochmal f L startwert)
    (if (null? L) startwert (f (car L) (nochmal f (cdr L) startwert))))
(nochmal + '(3 4 5) 0)          ==> 12
(nochmal * '(3 4 5) 1)          ==> 60
(nochmal / '(3 4 5) 1)          ==> 3.75
```

2.3.2 Die map-Funktion

Eine Funktion $f: A^k \rightarrow B^k$ lässt sich auf $F: (A^k)^m \rightarrow (B^k)^m$ erweitern durch

$$F(a_1, a_2, ..., a_m) = (f(a_{11}), f(a_2), ..., f(a_k)).$$

Wenn $(a_1\ a_2\ ...\ a_k)$ eine Liste ist, so bezeichnet man in Scheme die Funktion F durch F = map f . Das Ergebnis von map ist stets eine Liste.

Syntax von map: (map <proc> <list$_1$> <list$_2$> ... <list$_k$>)

Jede Liste <list$_j$> muss gleich viele Elemente, z.B. m Stück, besitzen. <proc> ist eine Funktion mit k Argumenten. <proc> wird auf das i-te Element aller k Listen angewendet; dieses Ergebnis sei e_i. Das Gesamtergebnis ist dann die Liste der e_i (i = 1, 2, ..., m). Die Reihenfolge, in der <proc> auf die Listen angewendet wird, ist nicht festgelegt. (Beispiel siehe drei Folien weiter.)

Beispiele:

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

(map + '( 9  4 1) '(2 -3))
    ==> Fehler, weil die Listen nicht die gleiche Länge haben

(map + '( 9  4 1) '(2 -3 0) '(5 1 -1))
    ==> (16 2 0)

(map (lambda (x) (/ (* x (+ x 1)) 2)) '(1 2 3 4 5 6 7 8))
    ==> (1 3 6 10 15 21 28 36)

(map car '((a b c) (d e) (f g h i j k l) (m) (n o p) (q r s)) )
    ==>  (a d f m n q)
```

map kann man z. B. für Projektionen und für Tabellierungen verwenden.

Hinweis:

Man kann ein Analogon zu map auf Listen leicht definieren:

```
(define (parallel f)
    (lambda (x) (if (null? x) '( )
        (cons (f (car x)) ((parallel f) (cdr x))))))
```

Z. B. für (define quadrat (lambda (x) (* x x))) erhält man aus ((parallel quadrat) '(1 2 3 4 5 6 7 8 9)) das Resultat

(1 4 9 16 25 36 49 64 81)

map f bzw. (parallel f) sind also Funktionen höherer Ordnung, da sie aus einer Funktion f: $A \rightarrow B$ eine Funktion von A^k nach B^k (bzw. allgemein von A* nach B*) generieren.

Gefahrenhinweis: In Scheme sind Seiteneffekte wie in imperativen Sprachen möglich. Mit "set!" realisiert man Wertzuweisungen:

(set! <Variable> <Ausdr>)

setzt die <Variable> auf den Wert von <Ausdr>.

Folgendes Beispiel für einen Seiteneffekt: Die Funktion zählt, wie oft sie aufgerufen wird:

```
(let ((z 0))
      (map (lambda (x) (set! z (+ z 1)) z) '(7 8 9)) )
```

Auswertung: Die Funktion (lambda (x) (set! z (+ z 1)) z) wird dreimal angewendet, nämlich auf 7, 8 und 9. Da diese Reihenfolge aber nicht festgelegt ist, kann sie also zunächst auf 7, dann auf 8 und dann 9 angewendet werden, wobei das Ergebnis (1 2 3) entsteht; die Funktion kann jedoch auch in der Reihenfolge 8, 9, 7 angewendet werden, was das Ergebnis (3 1 2) liefert. Somit ist jede Permutation von (1 2 3) als Ergebnis möglich; das Ergebnis hängt also von der Implementierung ab! Dies eröffnet undurchschaubare Fehlerquellen. In "rein funktionalen Sprachen" (zu denen Scheme nicht gehört!) sind derartige Seiteneffekte daher gar nicht erst möglich.

Beispiel zu map: Tabelle der Ulam-Collatz-Funktion

(siehe Grundvorlesung 7.5.1 mit Veranschaulichungen)

Ulam(x) = if x = 1 then 0 else
if x gerade then Ulam(x/2)+1 else Ulam(3*x+1)+1 fi

```
(define a 30)
(define (Ulam x)
          (if (<= x 1)  0
              (if (= (remainder x 2) 0) (+ (Ulam (/ x 2)) 1)
                                        (+ (Ulam (+ (* 3 x) 1)) 1))))
(define (ZL x) (if (<= x 0) ( ) (cons x (ZL (- x 1))) ) )
(define Zahlenliste (reverse (ZL a)))
(map cons Zahlenliste (map Ulam Zahlenliste))
```

==> ((1 . 0) (2 . 1) (3 . 7) (4 . 2) (5 . 5) (6 . 8) (7 . 16) (8 . 3) (9 . 19) (10 . 6) (11 . 14) (12 . 9) (13 . 9) (14 . 17) (15 . 17) (16 . 4) (17 . 12) (18 . 20) (19 . 20) (20 . 7) (21 . 7) (22 . 15) (23 . 15) (24 . 10) (25 . 23) (26 . 10) (27 . 111) (28 . 18) (29 . 18) (30 . 18))

2.3.3 Was geschieht hier? (Undurchschaubarkeit)

Beispiel 1:

```
(define id (lambda (x) x))
(define quadrat (lambda (x) (* x  x)))
(display ((id quadrat) 6)) (newline)
(display (quadrat (id 6)))
```

Beispiel 2:

```
(define (zweimal f) (lambda (x) (f (f x))))
(define nachf (lambda (x) (+ x 1)))
(define n-hoch-n (lambda (x) (expt  x  x)))
((zweimal nachf) 6)
((zweimal n-hoch-n) 2)
((zweimal n-hoch-n) 6)
;;; überrascht vom Ergebnis??
```

Beispiel 3:

```
(define (ap f g) (lambda (x y) (f (g x y) (g y x))))
(display ((ap - -) 8 3)) (newline) (display ((ap * +) 8 3))
```

Beispiel 4:

```
(define zweimal (lambda (x y) (x (x y))))
(define kub (lambda (x) (* x x x)))
(display (zweimal kub (zweimal kub 3)))
```

Beispiel 5:

```
(define F1 (lambda (x)
    (if (> (F2 (- x 1)) (* x x)) (F2 (+ x 1) ) (F1 (- x 1)))))
(define F2 (lambda (x)
    (if (> (remainder x 6)(remainder x 5)) (* x x) (* 2 (F1 (- x 2))))))
(display (F1 2))
```

Beispiel 6:

```
(define F1 (lambda (x)
    (if (> (F2 (- x 1)) (* x x)) (F2 (- x 1) ) (F1 (- x 1)))))
(define F2 (lambda (x)
    (if (or (< x 0)(> (remainder x 6)(remainder x 5)))
        (* x x) (* 2 (F1 (- x 1))))))
(display (F1 1))    ==>  8
```

Beispiel 7:

```
(define F1 (lambda (x)
    (if (> (F2 (- x 1)) (* x x)) (F2 (- x 1) ) (F1 (- x 1)))))
(define F2 (lambda (x)
    (if (or (< x 0) (> (remainder x 6)(remainder x 5)))
        (* x x) (+ (F2 (- x 1)) (F1 (- x 1))))))
(display (F1 14))      ==>  551
```

Beispiel 8 (vgl. Grundvorlesung 7.5.2):

```
(define (MC x) (if (> x 100)
                   (- x 10)
                   (MC (MC (+ x 11)))))
(display (MC 150)) (newline) (display (MC 100)) (newline)
(display (MC 15)) (newline) (display (MC -3))
```

Beispiel 9:

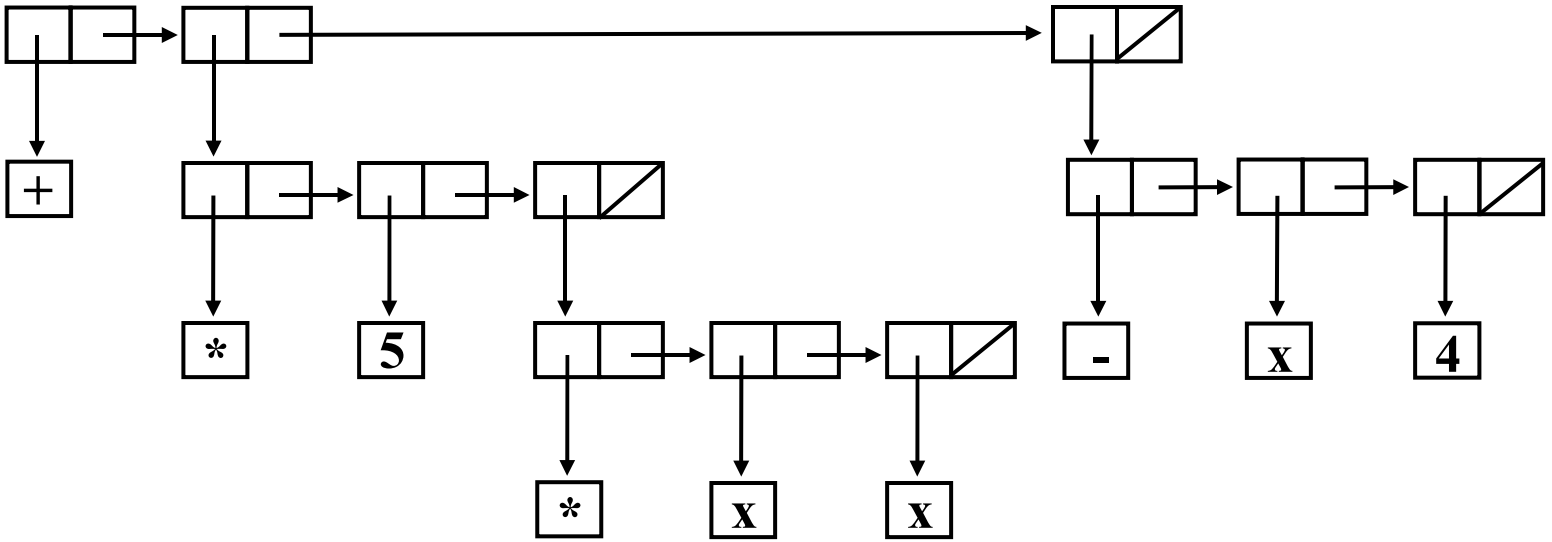
```
(define (mittel f)
  (lambda (x) (if (= x 0)  0
                  (+ (* x (f x)) ((mittel f) (- x 1))))))
(define (h i) (if (= i 0) 0 (+ (/ 1 i) (h (- i 1)))))
(display ((mittel (mittel h)) 2)) (newline)
(display ((mittel (mittel h)) 3)) (newline)
(display ((mittel (mittel h)) 4))
```

2.3.4 Beispiel Differenzieren

Zu den Standardbeispielen des funktionalen Programmierens zählt das symbolische Differenzieren.

Wir gehen aus von einem Ausdruck, der eine Funktion (hier ein Polynom) definiert, z.B.: $5x^2 + x - 4$, also:

(+ (* 5 (* x x)) (- x 4))). Ziel: die Ableitung dieser Funktion.



Die Ableitungsregeln nach einer Variablen x lauten:

$$\frac{d}{dx}(c) = 0, \quad \text{für jede Konstante c oder jede Variable } c \neq x$$

$$\frac{d}{dx}(x^r) = r * x^{r-1}, \quad \text{für alle Zahlen r, speziell:} \quad \frac{dx}{dx} = 1$$

$$\frac{d}{dx}(f + g) = \frac{d}{dx}(f) + \frac{d}{dx}(g), \qquad \text{analog für "-"}$$

$$\frac{d}{dx}(f * g) = f * \frac{d}{dx}(g) + g * \frac{d}{dx}(f)$$

Übliches Vorgehen:

- Welche Objekte müssen wir identifizieren?
- Wie greifen wir auf sie zu?
- Wie konstruieren wir das Ergebnis?

Dies unterteilen wir in
- die Spezifikation (Prozedurkopf) und
- die Implementierung (zugehörige Scheme-Prozedur).

Danach demonstrieren wir das Gesamtprogramm an Beispielen (hinzufügen von Ein- und Ausgabe, auch: Berechnung von Funktionswerten).

Spezifikation. Es seien A und B Ausdrücke.

(1) Identifiziere Objekte:

(Konstante? A)	Ist A eine Konstante (bzw. Variable $\neq$ x)?
(Variable_x? A)	Ist A die Variable x?
(Summe? A)	Ist A eine Summe (der Form (+ ...)) ?
(Differenz? A)	Ist A eine Differenz (der Form (- ...)) ?
(Produkt? A)	Ist A ein Produkt (der Form (* ...)) ?

(2) Zugriff auf Objekte:

(Erster A)	Erster Term von A
(Zweiter A)	Zweiter Term von A

(3) Konstruktion des Ergebnisses

(BildeSumme A B)	Bilde den Summen-Ausdruck A+B
(BildeDifferenz A B)	Bilde den Differenz-Ausdruck A-B
(BildeProdukt A B)	Bilde den Produkt-Ausdruck A*B

Implementierung.
(Var bezeichnet die Variable, nach der abgeleitet wird.)

```
(define (Konstante? A)
   (or (number? A) (and (symbol? A) (not (epv? A Var))) ) )
(define (Variable_x? A) (and (symbol? A) (epv? A Var) ) )
(define (Summe? A) (and (list? A) (epv? (car A) '+) ) )
(define (Differenz? A) (and (list? A) (epv? (car A) '-) ) )
(define (Produkt? A) (and (list? A) (epv? (car A) '*) ) )

(define (Erster A) (cadr A))
(define (Zweiter A) (caddr A))

(define (BildeSumme A B) (list '+ A B))
(define (BildeDifferenz A B) (list '- A B))
(define (BildeProdukt A B) (list '* A B))
```

Ableitung bilden laut Ableitungsregeln:

```
(define (Ableitung A x)
(cond
   ((Konstante? A) 0)
   ((Variable_x? A) 1)
   ((Summe? A)
      (BildeSumme (Ableitung (Erster A) x) (Ableitung (Zweiter A) x)))
   ((Differenz? A)
      (BildeDifferenz (Ableitung (Erster A) x) (Ableitung (Zweiter A) x)))
   ((Produkt? A)
      (BildeSumme
         (BildeProdukt (Erster A) (Ableitung (Zweiter A) x))
         (BildeProdukt (Zweiter A) (Ableitung (Erster A) x)) ))
))
```

Beispiele:

```
(define Var 'x)
(define Beispiel1 '(+ x 2) )
(Ableitung Beispiel1 Var)
(define Beispiel2 '(+ (* x x) (* 6 x)) )
(Ableitung Beispiel2 Var)
(define Beispiel3 '(+ (* 5 (* x x)) (- x 4)) )
(Ableitung Beispiel3 Var)
```

Zugehörige Ausgabe des obigen Scheme-Programms:

```
(+ 1 0)

(+ (+ (* x 1) (* x 1)) (+ (* 6 1) (* x 0)))

(+ (+ (* 5 (+ (* x 1) (* x 1))) (* (* x x) 0)) (- 1 0))
```

Beispiel: Berechnung der Ableitung eines Polynoms an einem Wert x

Ersetze ("replace") in der Liste (Ableitung Beispiel3 Var) die Variable Var durch die Zahl b. Füge also hinzu:

```
(define (replace L b)
(cond ((null? L) '() )   ((Konstante? L) L)
   ((epv? L Var) b)   ((epv? L Var) b)
   (else (list (car L) (replace (Erster L) b) (replace (Zweiter L) b)))))
```

Berechne dann (b=2 hier; evaluiere die Ergebnisliste)

```
(define Beispiel3 '(+ (* 5 (* x x)) (- x 4)) )
(Ableitung Beispiel3 Var)
(replace (Ableitung Beispiel3 Var) 2)
(eval (replace (Ableitung Beispiel3 Var) 2))
```

Zugehörige Ausgabe:

```
(+ (+ (* 5 (+ (* x 1) (* x 1))) (* (* x x) 0)) (- 1 0))
(+ (+ (* 5 (+ (* 2 1) (* 2 1))) (* (* 2 2) 0)) (- 1 0))
21
```

[Das heißt: Die Ableitung von $5x^2 + x - 4$ an der Stelle x=2 ist 21.]

Obiges Programm lässt sich nun ausbauen (selbst probieren!):

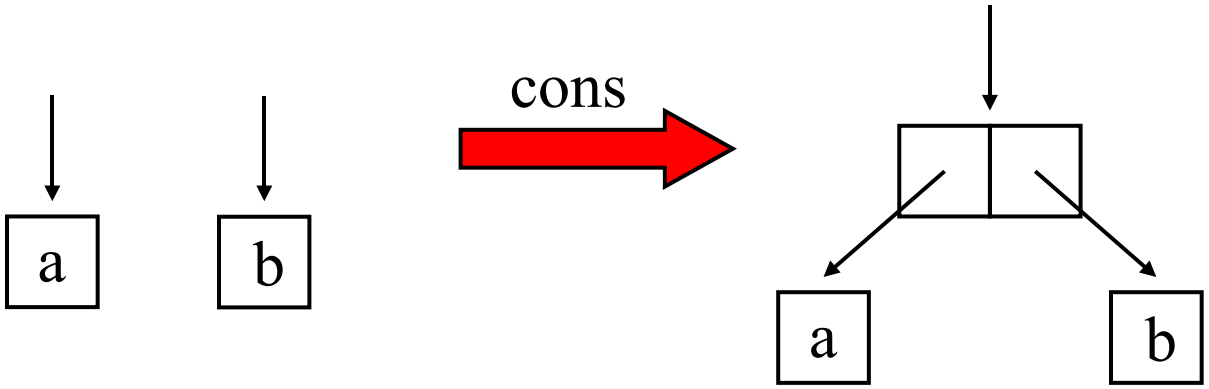
- Berücksichtige Exponenten, d.h. leite x^r explizit ab.
- Multipliziere alle Terme aus, sodass eine Polynomdarstellung als Summe von Termen $a_i x^i$ entsteht.
- Sortiere die Terme in der Liste und addiere Konstanten, die zu gleichen x-Potenzen gehören, sodass die gewohnte Darstellung $a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ entsteht.

Computer-Algebra-Systeme enthalten derartige funktionale Rechenvorgänge.

2.4 Listen

2.4.1 Paare

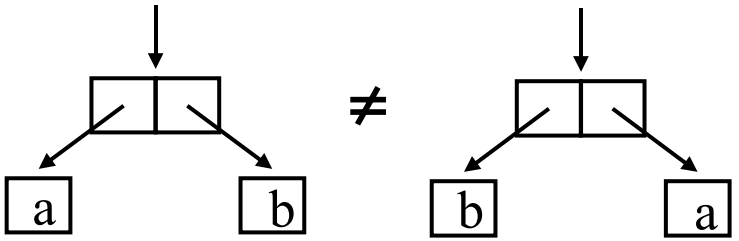
Die zentrale Datenstruktur in Scheme (wie auch in LISP) sind binäre Bäume. Der Operator cons (= "constructor") fasst zwei Objekte zu einem binären Baum mit einer "inhaltsleeren" Wurzel zusammen ("dotted pair"): (cons a b) bedeutet



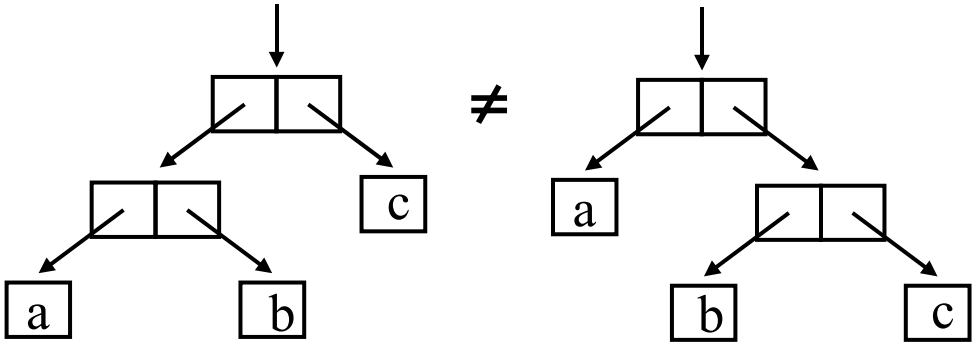
Auf die Komponenten kann man mittels car und cdr zugreifen, gesprochen "kahr" und "kudd^{e}r". Die Bezeichnungen stammen von den Assemblerbefehlen der IBM-704 (ein viel benutzter Rechner aus den Jahren von 1954 bis 1960) car = content of address register und cdr = content of decrement register.

Die externe Darstellung von (cons a b) ist (a . b) . Ein durch cons gebildetes Objekt ist ein "Paar" ("pair"). Wie bei binären Bäumen ist cons weder kommutativ noch assoziativ, d. h.:

(cons a b) ≠ (cons b a)



(cons (cons a b) c) ≠
(cons a (cons b c))

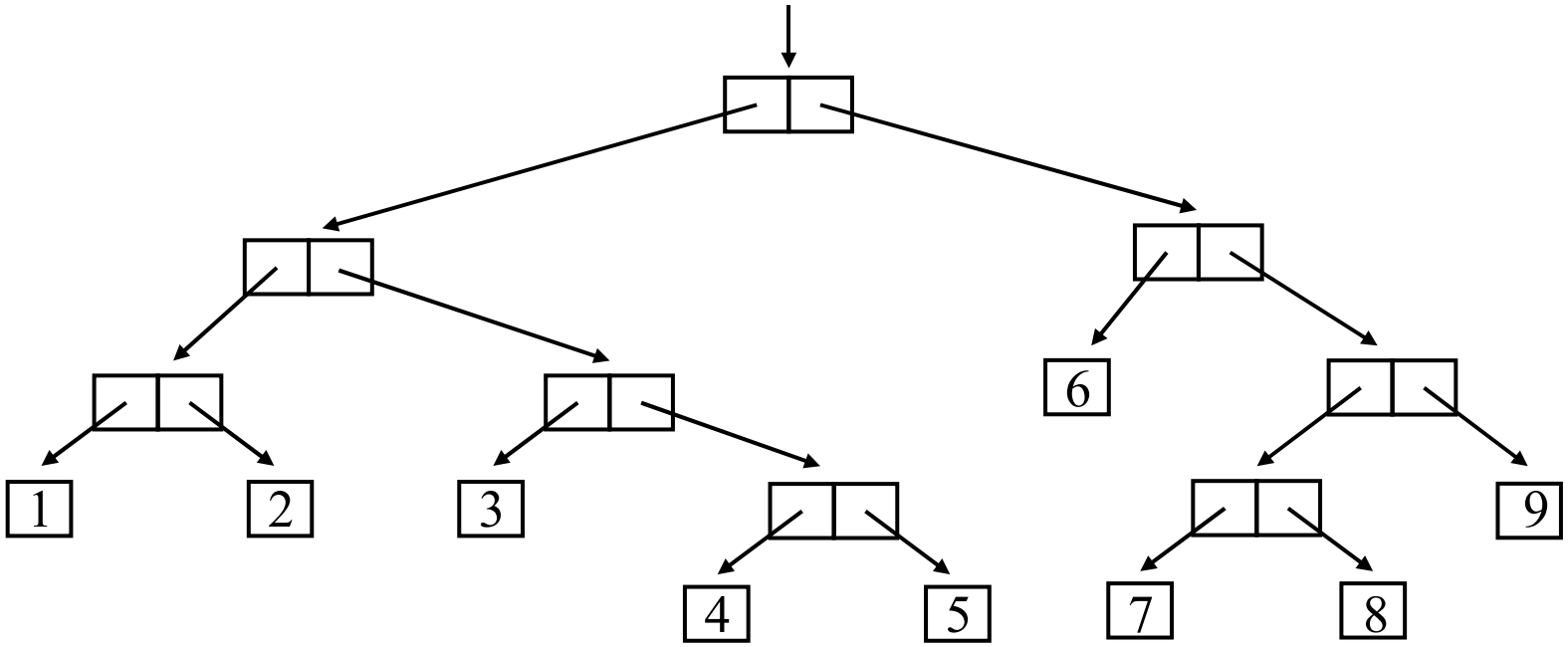


Beispiel:

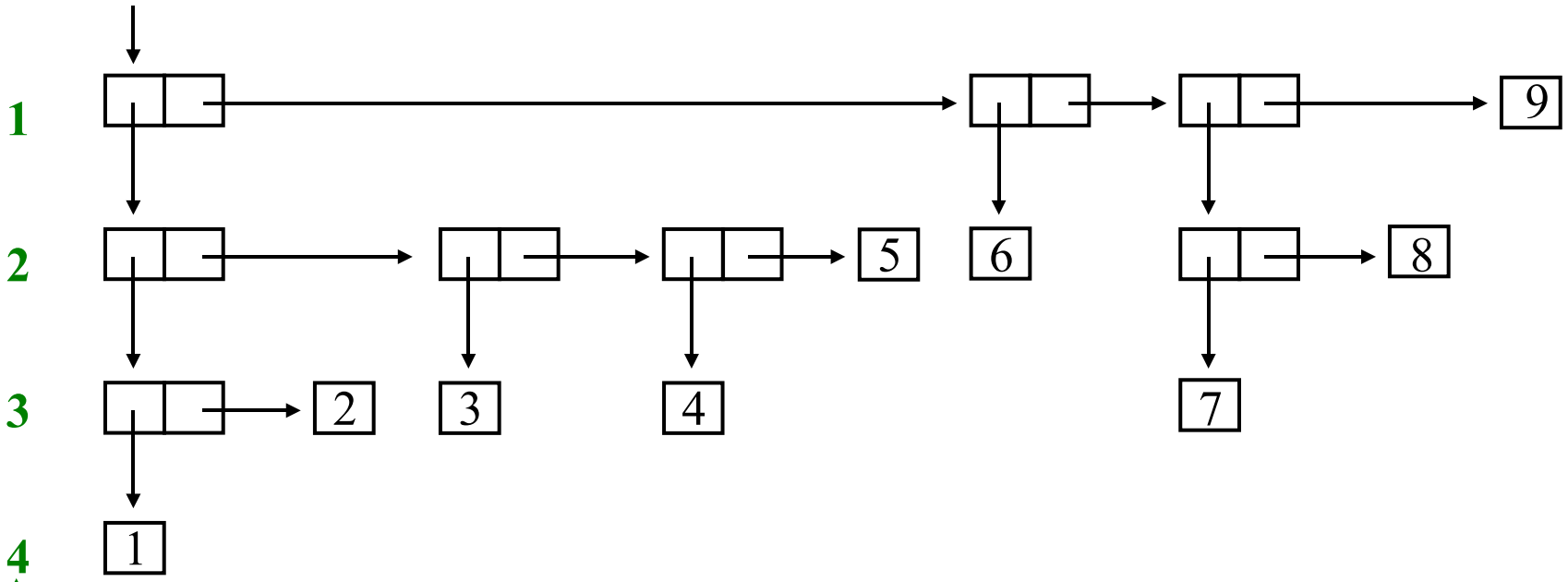
(cons (cons (cons 1 2) (cons 3 (cons 4 5))) (cons 6 (cons (cons 7 8) 9)))

Dieser Ausdruck liefert die Darstellung: (((1 . 2) 3 4 . 5) 6 (7 . 8) . 9)

Als Baum geschrieben liegt folgende Struktur in Scheme vor:



In der Regel stellt man die linken Verweise durch senkrechte und die rechten Verweise durch waagerechte Linien dar. So werden den Elementen zugleich Ebenen (Schichten, Levels) zugewiesen. Für unser Beispiel (cons (cons (cons 1 2) (cons 3 (cons 4 5))) (cons 6 (cons (cons 7 8) 9))) :



Ebene = Level in den Listen

Da man jeden geordneten Baum in einen binären Baum umwandeln kann (Binarisierung, Grundvorlesung 8.2.6), lassen sich mit Listen also auch allgemeine Bäume darstellen. Man kann zugleich gerichtete Graphen hiermit beschreiben (s. u.).

car liefert somit den linken und cdr den rechten Unterbaum eines Knotens. Es gilt also:
(car (cons a b)) = a und (cdr (cons a b)) = b

Die Operationen car und cdr werden verallgemeinert, im Revised Report aber nur bis zu 4 Buchstaben "a" und "d":

(car (car L)) = (caar L), (car (cdr L)) = (cadr L),
(cdr (car L)) = (cdar L), (cdr (cdr L)) = (cddr L),
(car (car (car L))) = caaar L), (car (car (cdr L))) = (caadr L), ...
(cdr (cdr (cdr (car L))))) = (cdddar L), ...

Mittels set-car! und set-cdr! kann man die Werte der Unterbäume wie bei einer Wertzuweisung verändern. Zum Beispiel:
(define L (cons 'A (cons 'B 'C))) L
(set-car! L 'D) L (set-cdr! L 'E) L
==> (A . (B . C)) (D . (B . C)) (D . E)
[Statt (A . (B . C)) wird in DrScheme (A B . C) ausgegeben usw.]

2.4.2 Listen werden rekursiv definiert (vgl. 2.1.5):

> Die leere Liste ( ) ist eine Liste.
>
> Wenn L eine Liste ist, dann ist auch (cons A L) eine Liste (für jedes Objekt A).

Die leere Liste in Scheme ist kein Paar, sondern ein spezielles Objekt mit eigenem Typ.
Eine Liste wird in der Form

$(A_1\ A_2\ ...A_k)$

geschrieben. Hiermit ist stets das Objekt

$(\text{cons } A_1\ (\text{cons } A_2\ (\text{cons } A_3\ ...\ (\text{cons } A_{k-1}\ (\text{cons } A_k\ (\ )\ ))\ ...\ )))$

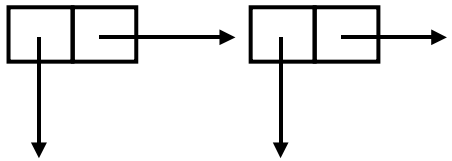
gemeint. Beispiel:

(cons 'A1 (cons 'A2 (cons 'A3 (cons 'A4 (cons 'A5 ( ))))))

liefert die Ausgabe (A1 A2 A3 A4 A5) .
[Auch die Ausgabe

(A1 . (A2 . (A3 . (A4. (A5 . ( )))))) ist korrekt.]

Listen lassen sich somit stets durch Bäume darstellen. In der Baumdarstellung sind "Listen-Teile" durch die Teilstruktur



zu erkennen. In der externen Darstellung werden diese Anteile dann in Listenschreibweise (...), also durch Zwischenraum getrennte Aufzählung ohne Punkte, notiert. Zum Beispiel liefert

(cons (cons 'A 'B) (cons 'C 'D))

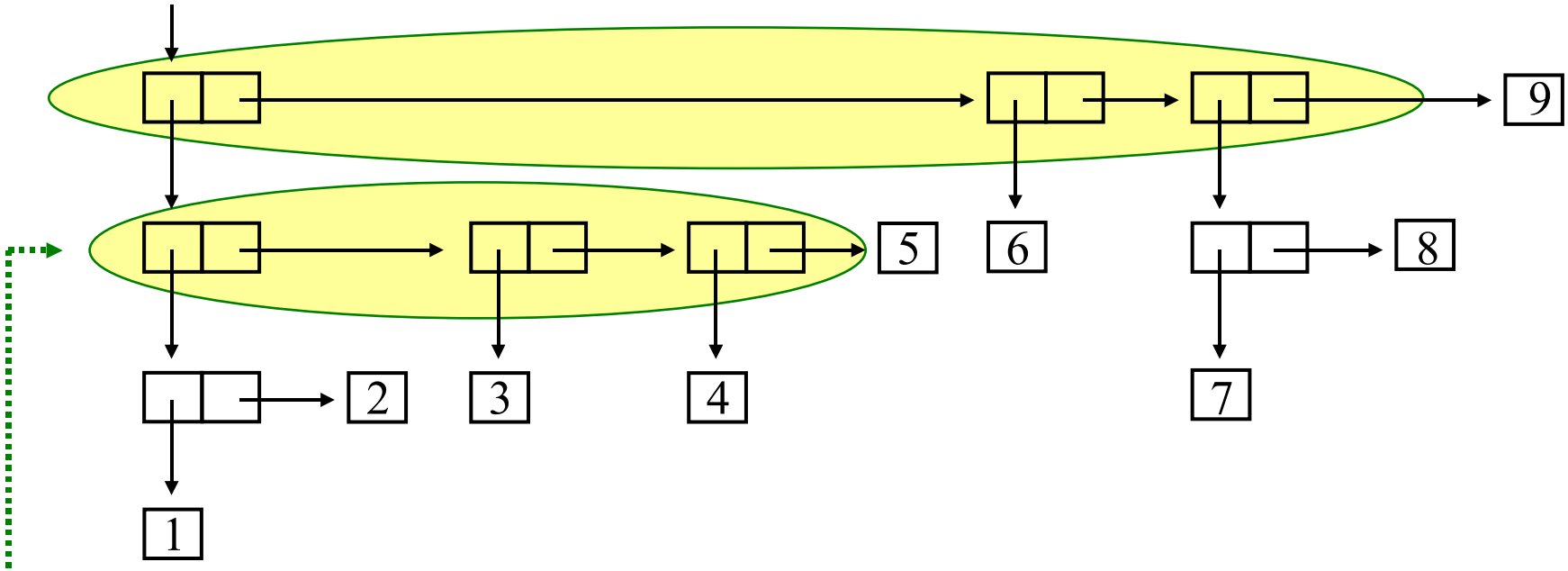
die Ausgabe

((A . B) C . D)

weil die Form (cons X (cons ...) vorliegt.

In unserem früheren Beispiel

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



liegen die umrundeten Teile als Listenstrukturen vor,
weshalb die externe Darstellung (Ausgabe in Scheme) lautet:

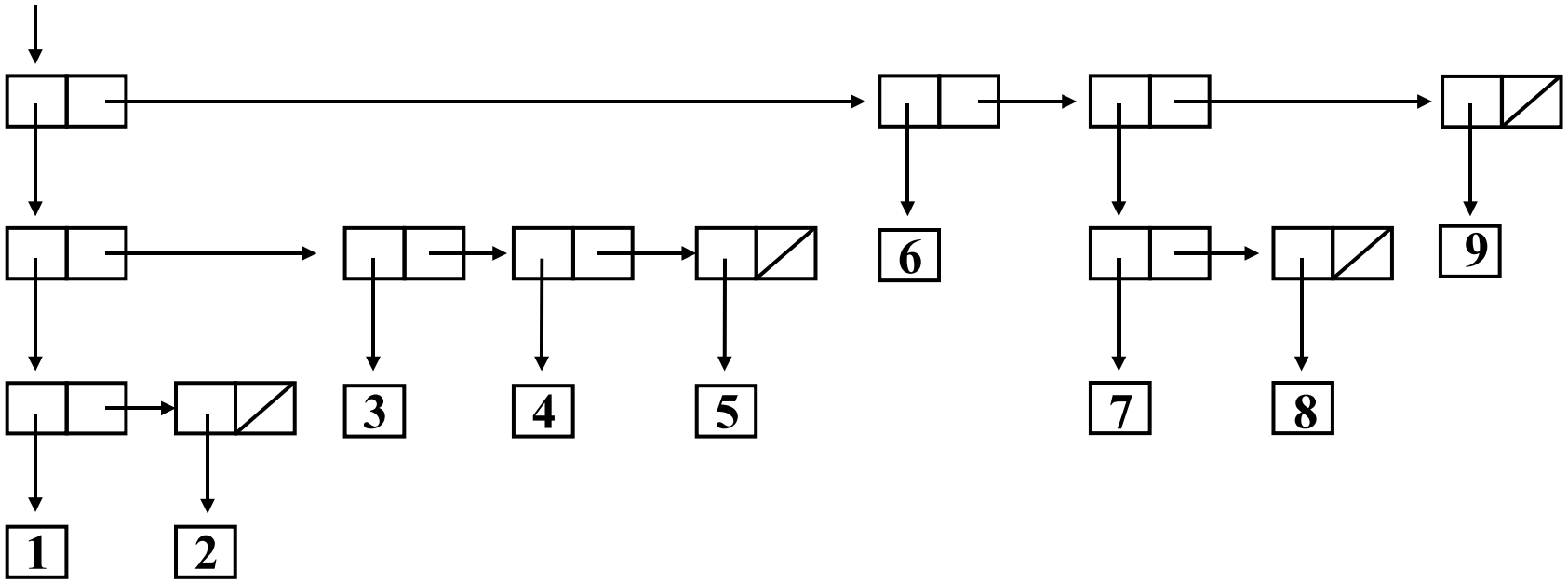
(((1 . 2) 3 4 . 5) 6 (7 . 8) . 9)

Listenstruktur mit 3 Elementen

Dieses Beispiel wandeln wir so um, dass nur noch Listen vorliegen:

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

Dies liefert folgende Baumdarstellung, wobei wir die leere Liste durch ein Kästchen mit Nebendiagonale notieren:



Die Ausgabe lautet daher: (((1 2) 3 4 5) 6 (7 8) 9)

Insbesondere wird jeweils die letzte leere Liste nicht dargestellt.

Betrachte speziell die Listen:

```
(cons ( ) ( ))                    ==>  (( ))
(cons ( ) (cons ( ) ( )))         ==>  (( ) ( ))
(cons (cons ( ) ( )) ( ) )        ==>  ((( )))
```

Eine Folge von Paaren, die nicht mit einer leeren Liste endet, wird manchmal auch als "unvollständige Liste" ("improper list") bezeichnet. Dies sind die Listenteilstrukturen, die wir oben angesprochen hatten und die in der Ausgabe berücksichtigt werden, z. B. (A B C . D) anstelle von (A . (B . (C . D))) .

Wichtiger Hinweis: Die Elemente einer Liste werden in Scheme als nulltes, erstes, zweites ... Element durchnummeriert. Die Zählung beginnt also mit 0 (und nicht wie gewohnt mit 1).

2.4.3 Operationen (Funktionen) auf Listen

cons, car, cdr, caar, cadr, set-car!, set-cdr! ... siehe oben.

pair? prüft, ob das Objekt ein Paar ist, also von der Form (cons A B). [Die leere Liste ist kein Paar.]

list? prüft, ob eine Liste vorliegt, d. h., ein Objekt, das bei wiederholtem cdr schließlich die leere Liste liefert.

null? prüft, ob das Objekt die leere Liste ist.

(list A_1 ... A_k) liefert eine Liste mit den Objekten A_1, ..., A_k in dieser Reihenfolge. Speziell liefert (list) die leere Liste.

(length <Liste>) liefert die Länge der Liste (im 1. Level)
(length '(A (B C D) E (F G) H)) ==> 5

(append <Liste$_1$> <Liste$_2$> ... <Liste$_k$>) hängt die Listen zu einer Liste aneinander (auf dem 1. Level).
(append '(A (B C D)) '(E) '((F G) H)) ==> (A (B C D) E (F G) H)

(reverse <Liste>) dreht die Reihenfolge der Elemente der Liste um.

(list-tail <Liste> k) liefert die liste ohne die k ersten Elemente. Hier werden die k ersten Elemente entfernt; k = 0 liefert die Liste selbst. (list-tail '(A B C) 1) ==> (B C)

(list-ref <Liste> k) liefert das k-te Element der Liste. Man beachte, dass Listen ab 0 durchnummeriert werden, dass sich das k also auf diese Nummerierung bezieht. (list-ref '(A B C) 2) ==> C

(memv X <Liste>) liefert die längste Teilliste (1. Level!), die mit dem Objekt X beginnt (falls X nicht existiert, wird #f und nicht etwa die leere Liste geliefert). Beispiel:
(memv 'A '((A B) B (A) A (B A) (B B))) ==> (A (B A) (B B))

Einschub: Gleichheit von Objekten (siehe Report R^{5}RS)

Erinnerung an imperatives Programmieren: Wann sind zwei Objekte gleich? (vgl. Grundvorlesung 1.13, 2.12, 3.4.7, 3.6.2)

In Scheme gibt es drei Formen der Gleichheit:

(eqv? <Objekt1> <Objekt2>)

(eq? <Objekt1> <Objekt2>) Dies ist im Wesentlichen dasselbe wie eqv?, liefert jedoch manchmal implementierungsabhängig bei Zeichen und Zahlen eventuell #f, wenn eqv? #t geliefert hätte.

(equal? <Objekt1> <Objekt2>) Dies liefert true, wenn beide Objekte die gleiche Auswertung haben, d. h., wenn sie die gleiche Ausgabe erzeugen.

Wir gehen im Folgenden nur auf eqv? ein.

Einschub: Gleichheit von Objekten (siehe Report R^{5}RS)

Es gilt:

(eqv? <Objekt1> <Objekt2>) liefert genau dann #t, wenn

entweder beide Objekte zu Atomen ausgewertet werden können
und diese sich ergebenden Konstanten gleich sind

oder beide Objekte zusammengesetzt sind und das identisch gleiche Objekt sind (also nicht eine Kopie voneinander, sondern das an gleicher Stelle im Speicher stehende Objekt).

Hinweis: Atome sind alle nicht zusammengesetzten Objekte,
d. h., alle Objekte, für die eines der Prädikate
number?, boolean?, char? symbol?, null?
den Wert #t ergibt.

Einschub: Gleichheit von Objekten (siehe Report R^{5}RS)

```
(eqv? 'A 'A)                          ==>  #t
(define a 2) (eqv? a 2)               ==>  #t
(eqv? (/ 3 15) (/ 2 10))              ==>  #t
(eqv? (/ 1 2) 0.5)                    ==>  #f   -- unterschiedliche interne Darstellung
(eqv? (/ 1 2) (inexact->exact 0.5))        ==>  #t
(eqv? #t #t)                          ==>  #t
(let (( z 7)) (eqv? z z))             ==>  #t
(eqv? '(1 2) '(1 2))                  ==>  #f
(define q (lambda (x) (* x x))) (eqv? 4 (q 2))                ==>  #t
(define w (lambda (x) (cdr '(1 2)))) (eqv? (w 5) (w 5))       ==>  #t
(define w (lambda (x) (cdr '(1 2)))) (eqv? (w 4) (w 5))       ==>  #t
(define w (lambda (x) (cdr '(1 2)))) (eqv? '(1 2) (w 2))      ==>  #f
(eqv? (cons 'a 'b) (cons 'a 'b))           ==>  #f
(eqv? "heute" "heute")                     ==>  #f
(eqv? (lambda (x) x) (lambda (x) x))       ==>  #f
(eqv? (eqv? ( ) '1) (eqv? ( ) '1))         ==>  #t
```

(memq X <Liste>) muss also das erste Element in <Liste> finden, welches gleich X ist. Erfolgt dieser Vergleich mit eq?, so nimmt man die Funktion memq, soll eqv? benutzt werden, so verwende man memv, und im Falle von equal? nehme man member.

(assv <Objekt> <Liste>)

<Liste> muss eine Liste von Paaren sein. Zurückgeliefert wird das erste Paar in der Liste, dessen car gleich <Objekt> ist. Kommt dagegen <Objekt> nicht als linker Teil eines Paars vor, so wird #f zurückgegeben. Beispiel:

```
(define z '((1 5) (2 6) (1 7))) (assv 1 z) (assv 2 z) (assv 3 z)
liefert    (1 5)  (2 6)  #f
```

assq und assoc verwendet man, wenn die Gleichheit statt mit eqv? mittels eq? bzw. equal? ermittelt wird.

In der Regel lassen sich diese Funktionen leicht in Scheme selbst beschreiben.

Beispiel: Definition von (reverse L) in Scheme.

```
(define (spiegeln L)
    (if (null? L)  ( ) (append (spiegeln (cdr L)) (list (car L))) ))
```

Beispiel: Definition von (memv X <Liste>) in Scheme.

```
(define (abdann X L)
    (cond ((null? L)  #f)
              ((eqv? X (car L)) L)
              (else (abdann X (cdr L))) ) )
; Beispielanwendung:
(define c '((A B) B (A) A (B A) (B B)))
(memv 'A c)  (memv '(A A) c)  (abdann 'A c)  (abdann '(A A) c)
```

2.4.4 Beispiel: *Sortieren einer Liste durch Einfügen*

Hierzu siehe Grundvorlesung: Anfang von Abschnitt 10.3.

```
(define (einfuege-sortieren vergleich Liste)
    (define (einfuegen s L)
        (cond ((null? L)  (list s))
                ((vergleich s (car L)) (cons s L))
                (else (cons (car L) (einfuegen s (cdr L)) ) ) ) )
    (if (null? (cdr Liste))  Liste
        (einfuegen (car Liste) (einfuege-sortieren vergleich (cdr Liste)))))
(einfuege-sortieren  >  '(24 5 31 7 3 10 6 24))
(einfuege-sortieren  <=  '(24 5 31 7 3 10 6 24))
```

==> (31 24 24 10 7 6 5 3)
(3 5 6 7 10 24 24 31)

Gibt man die leere Liste ein, so bricht die Berechnung mit einem Fehler ab. Warum? Korrigieren Sie das Programm!

Experimente:

Sie können nun die Laufzeit untersuchen, indem Sie das Programm auf große Listen ansetzen. Diese erhält man, indem man zufällig viele Zahlen erzeugt.
Für eine natürliche Zahl b liefert

```
(random b)
```

irgendeine natürliche Zahl von 0 bis b-1.

Folgendermaßen kann man daher eine Liste von 50000 natürlichen Zahlen zwischen 0 und 399999 erzeugen.

```
(define (zliste a b)
    (if (> a 0) (cons (random b) (zliste (- a 1) b)) ( )))
(define Zahlenliste (zliste 50000 400000))
```

Mein 4 Jahre alter Laptop benötigte für das Sortieren von 10000 Zahlen (ohne Ausgabe) mittels der Funktion "einfuege-sortieren" rund 23 Sekunden, bei 50000 Zahlen waren es 650 Sekunden. Dies bestätigt grob die quadratische Abhängigkeit von der Zahl der Zahlen.
(Über die Ungenauigkeit der Messungen per Stoppuhr oder per Systembeobachtung siehe Veranstaltungen zu Betriebssystemen.)

Hinweis auf Fehlerfallen mit Klammern:

Lässt man in obigem Programm ein Klammerpaar in der Zeile, die mit "cond" beginnt, weg, so erhält man ein lauffähiges Programm, welches aber etwas anderes als "einfuege-sortieren" liefert:

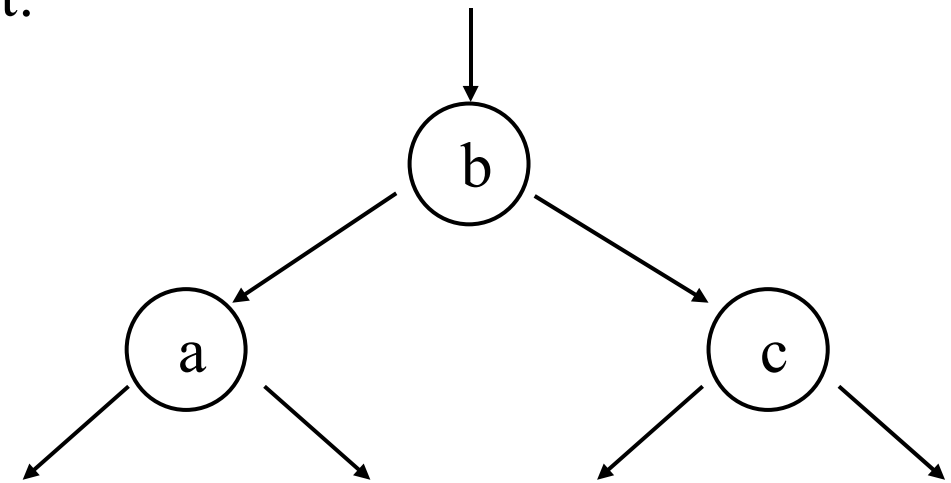
```
(define (einfuege-sortieren? vergleich Liste)
    (define (einfuegen s L)
        (cond (null? L)  (list s)
                ((vergleich s (car L)) (cons s L))
                (else (cons (car L) (einfuegen s (cdr L)) ) ) ) ) )
    (if (null? (cdr Liste))  Liste
        (einfuegen (car Liste) (einfuege-sortieren? vergleich (cdr Liste)))))
(einfuege-sortieren?  >  '(24 5 31 7 3 10 6 24))
```

Welches Ergebnis erhält man nun? Erklären Sie sich diese Änderung!

2.4.5 Sortieren mit Bäumen

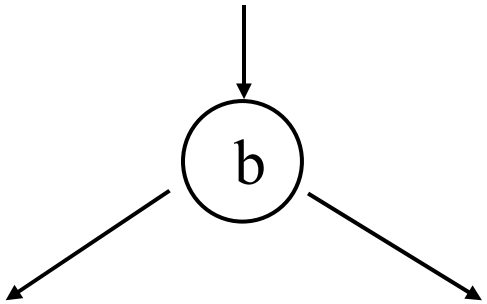
Hierzu siehe Grundvorlesung Abschnitt 3.7.7.

Wie in 2.4.1 angegeben kann man einen binären Baum mittels (cons X Y) aufbauen. Dann besitzen die Knoten jedoch noch keinen Inhalt. Diesen erhält man zum Beispiel, indem man noch mindestens ein Paar hinzufügt. Die gewohnte Darstellung lautet:

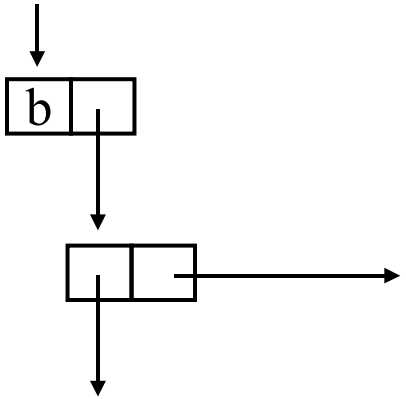


2.4.5.1 *Übliche Darstellung von Bäumen*

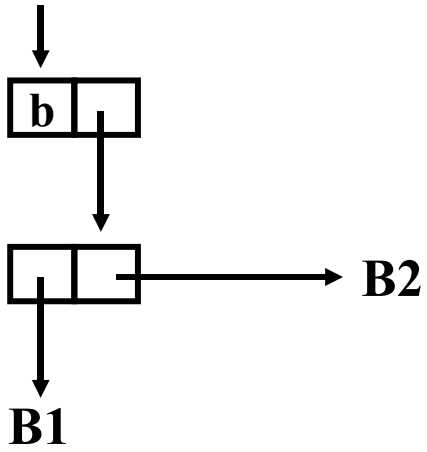
Einen Baumknoten K



stellen wir dar in der Form



Also:
Inhalt des Knotens K = (car K)
linker Unterbaum = (cadr K)
rechter Unterbaum = (cddr K)




```
(define (make-treenode x B1 B2)
        (cons x (cons B1 B2) ) )
(define (links K) (cadr K))
(define (rechts K) (cddr K))
(define (inhalt K) (car K))
```

```
; Füge einen Schlüssel s in einen sortierten Baum B ein (insert-tree)
(define (insert-tree vergleich s B)
          (cond   ((null? B) (make-treenode s ( ) ( )))
                  ((vergleich s (inhalt B))
(make-treenode (inhalt B) (insert-tree vergleich s (links B)) (rechts B)))
                  (else
(make-treenode (inhalt B) (links B) (insert-tree vergleich s (rechts B)) ))))
```

```
; Aufbau (build-tree L) eines sortierten Baums B aus einer Liste L
(define (build-tree L)
              (if (null? L) ( )
                  (insert-tree < (car L) (build-tree (cdr L))) ) )
```

Nun ist die Liste in einen geordneten binären Baum (einen "Suchbaum") übertragen worden. Es fehlt zum Baumsortieren nur noch der abschließende inorder-Durchlauf durch den Baum:

```
; den Baum B inorder in eine Liste auslesen
(define (inorder B)
  (if (null? B) ( )
     (append (inorder (links B)) (list (inhalt B)) (inorder (rechts B)))))
```

Dieses Programm können wir dann auf eine Liste anwenden.

Das gesamte Programm findet sich auf der nächsten Folie.

Gesamtprogramm zum Baumsortieren:

```
(define (make-treenode x B1 B2) (cons x (cons B1 B2) ) )
(define (links K) (cadr K))
(define (rechts K) (cddr K))
(define (inhalt K) (car K))
(define (insert-tree vergleich s B)
    (cond ((null? B) (make-treenode s ( ) ( )))
          ((vergleich s (inhalt B))
          (make-treenode (inhalt B) (insert-tree vergleich s (links B)) (rechts B)))
          (else
          (make-treenode (inhalt B) (links B) (insert-tree vergleich s (rechts B)) ))))
(define (build-tree vergleich L)
          (if (null? L) ( )
          (insert-tree vergleich (car L) (build-tree vergleich (cdr L))) ) )
(define (inorder B)
  (if (null? B) ( )
    (append (inorder (links B)) (list (inhalt B)) (inorder (rechts B)))))
```

```
(inorder (build-tree > '(45 67 8 9 12 4 90 68 65 22 34 3 8 1 6 7 19 22 3 5 7 8 10)))
==> (90 68 67 65 45 34 22 22 19 12 10 9 8 8 8 7 7 6 5 4 3 3 1)
```

Wendet man dieses Programm auf die Liste mit 50.000 Elementen aus 2.4.4 an, d.h. fügt man die dortigen Ausdrücke

```
(define (zliste a b)
    (if (> a 0) (cons (random b) (zliste (- a 1) b)) ( )))
(define Zahlenliste (zliste 50000 400000))
(inorder (build-tree < Zahlenliste))
```

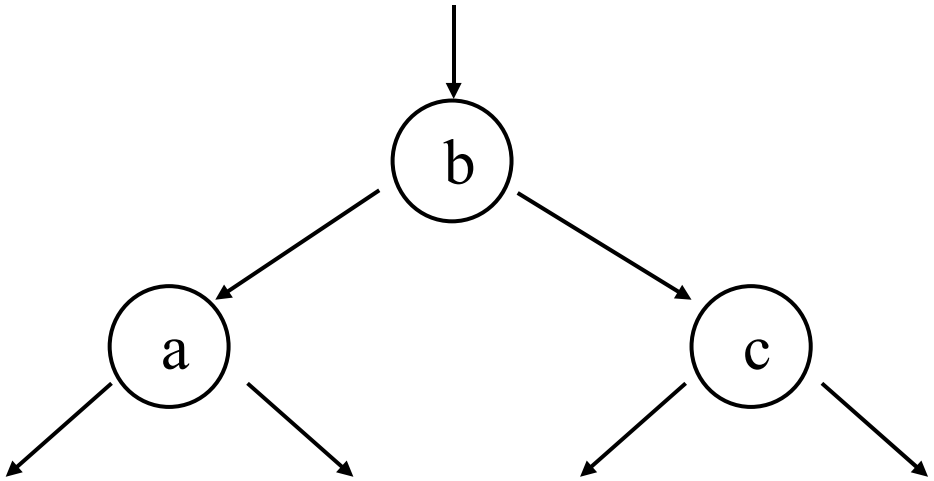
an das Programm an, so ist diese Liste (ohne Ausgabe) nach 3,4 Sekunden sortiert - anstelle der 650 Sekunden mit dem Sortieren durch Einfügen. Auch eine Liste mit 500000 Elementen benötigt nur 36 Sekunden zum Sortieren.

(Probieren Sie dies selbst aus.)

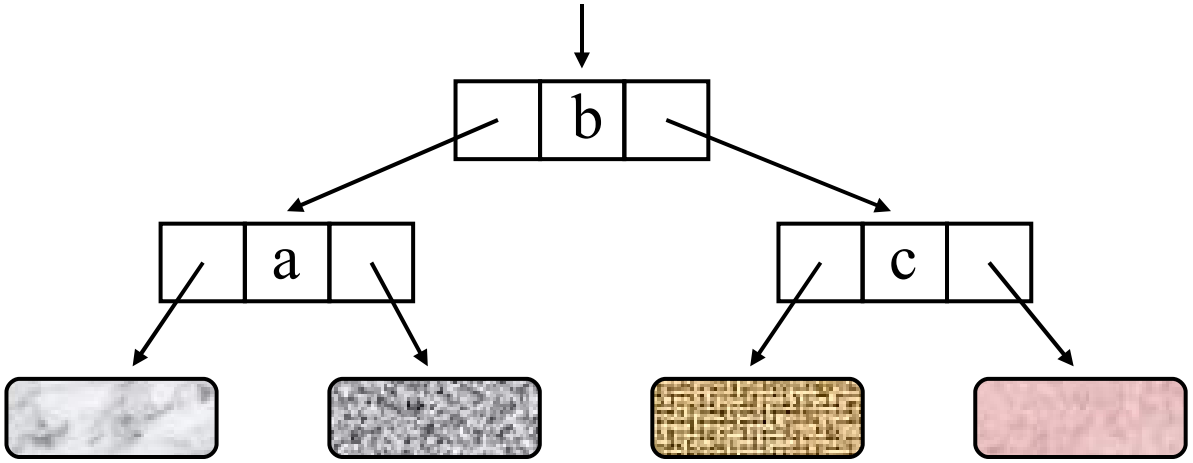
2.4.5.2 Beispiel:
Sortieren mit Bäumen, andere Baumdarstellung

Die in 2.4.5.1 gewählte Darstellung für einen Baum ist nicht die einzig denkbare Möglichkeit. Aus der Vielzahl der Möglichkeiten greifen wir folgende heraus.

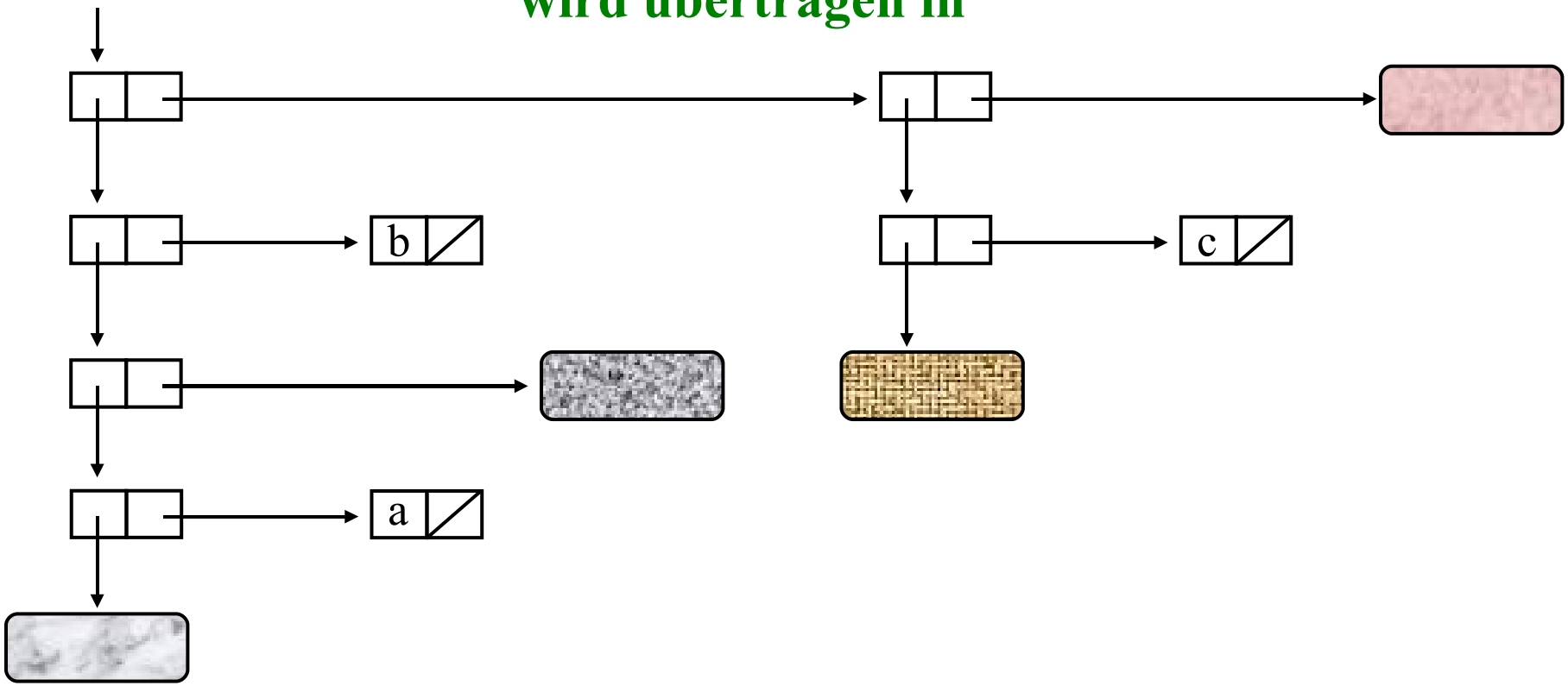
Die gewohnte Darstellung



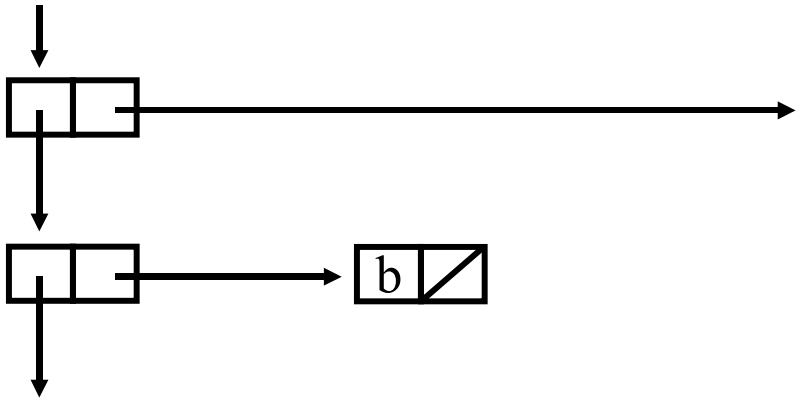
wird nun wie folgt nach Scheme übertragen:



wird übertragen in



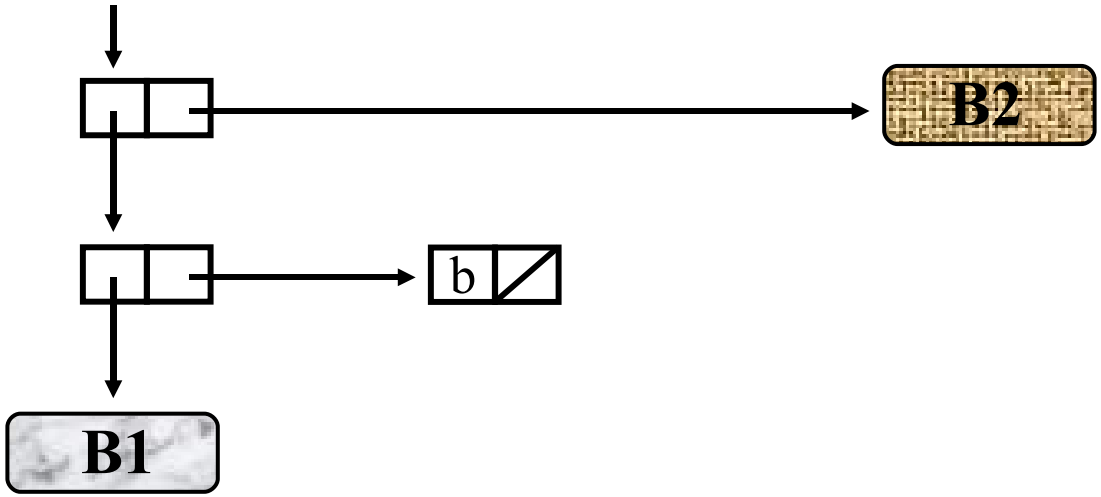
Der Knoten K eines Baums wird nun also dargestellt durch



Auf den Inhalt des Knotens K greift man zu mittels (cadar K), den linken Nachfolgeknoten erhält man durch (caar K) und den rechten Nachfolger durch (cdr K). Die Hilfsfunktionen

make-treenode ...	=	bilde einen Knoten mit Inhalt ...
(links K)	=	Wurzel des linken Unterbaums
(rechts K)	=	Wurzel des rechten Unterbaums

müssen nun entsprechend definiert werden.




```
(define (make-treenode x B1 B2)
        (cons (cons B1 (cons x ( ))) B2) )

(define (links K) (caar K))

(define (rechts K) (cdr K))

(define (inhalt K) (cadar K))
```

Mit diesen Abänderungen für die Hilfsfunktionen kann man das Programm aus 2.4.5.1 wörtlich übernehmen und erhält:

Gesamtprogramm zum Baumsortieren (zweite Darstellung):

```
(define (make-treenode x B1 B2) (cons (cons B1 (cons x ( ))) B2) )
(define (links K) (caar K))
(define (rechts K) (cdr K))
(define (inhalt K) (cadar K))  -- nur dieser umrandete Teil ist neu
(define (insert-tree vergleich s B)
    (cond ((null? B) (make-treenode s ( ) ( )))
          ((vergleich s (inhalt B))
          (make-treenode (inhalt B) (insert-tree vergleich s (links B)) (rechts B)))
          (else
          (make-treenode (inhalt B) (links B) (insert-tree vergleich s (rechts B)) ))))
(define (build-tree vergleich L)
          (if (null? L) ( )
          (insert-tree vergleich (car L) (build-tree vergleich (cdr L))) ) )
(define (inorder B)
  (if (null? B) ( )
    (append (inorder (links B)) (list (inhalt B)) (inorder (rechts B)))))
(inorder (build-tree < '(45 67 8 9 12 4 90 68 65 22 34 3 8 1 6 7 19 22 3 5 7 8 10)))
==> (1 3 3 4 5 6 7 7 8 8 8 9 10 12 19 22 22 34 45 65 67 68 90)
```

Wendet man dieses Programm auf die Liste mit 50.000 Elementen aus 2.4.4 an, d.h. fügt man die dortigen Ausdrücke

```
(define (zliste a b)
    (if (> a 0) (cons (random b) (zliste (- a 1) b)) ( )))
(define Zahlenliste (zliste 50000 400000))
(inorder (build-tree < Zahlenliste))
```

an das Programm an, so ist diese Liste (ohne Ausgabe) nach knapp 4 Sekunden sortiert. Für 500000 Elemente braucht dieses Programm 45 Sekunden.

Diese Darstellung benutzt je Knoten ein Paar mehr als die Darstellung in 2.4.5.1 und erfordert somit mehr Speicherplatz und Zugriffszeit. Diese Darstellung ist aber gut geeignet, wenn der Inhalt jedes Knotens zum Beispiel ein Paar ist mit (cadar K) als Schlüssel und (cddar K) als Dokument.

2.4.5.3 "Nebenprodukte" dieses Abschnitts

Baumdurchläufe:

```
(define (inorder B)
  (if (null? B) ( )
     (append (inorder (links B)) (list (inhalt B)) (inorder (rechts B)))))
(define (preorder B)
  (if (null? B) ( )
     (append (list (inhalt B)) (preorder (links B)) (preorder (rechts B)))))
(define (postorder B)
  (if (null? B) ( )
     (append (postorder (links B)) (postorder (rechts B)) (list (inhalt B)))))
```

Wiederverwendung von Programmteilen mittels Polymorphie

Abstrakter Datentyp (Geheimniskonzept / hidden information)

2.4.6 Potenzmengen

Rekursive Definition der Potenzmenge P(M) einer Menge
$M = \{m\} \cup R$: $P(\emptyset) = \{\emptyset\}$ und $P(M) = P(R) \cup \{\{m\} \cup Q \mid Q \in P(R)\}$.
M als Liste darstellen. Setze m = (car M), R = (cdr M), T = P(R).
Dies lässt sich gut mit der map-Funktion beschreiben.

```
(define (Potenzmenge M)
    (if (null? M)
        (list '( ))
        (let ((T (Potenzmenge (cdr M)) ))
            (append T (map (lambda (Q) (cons (car M) Q)) T)) )))
```

```
(Potenzmenge '( ) )
(Potenzmenge '( 1) )
(Potenzmenge '( 1 2 3 4))
```

==> (( ))
(( ) (1))
(( ) (4) (3) (3 4) (2) (2 4) (2 3) (2 3 4) (1) (1 4) (1 3) (1 3 4) (1 2) (1 2 4) (1 2 3) (1 2 3 4))

Man kann die Liste der Teilmengen auch nach der Anzahl der Elemente sortieren, indem man das Baumsortieren einsetzt und für den "vergleich" die Boolesche Funktion

```
(define (mehr? K L) (<= (length K) (length L)))
```

verwendet. Man füge also zur Definition Potenzmenge das "Gesamtprogramm zum Baumsortieren" aus 2.4.5.1 hinzu sowie die Funktion mehr? und den Ausdruck

(inorder (build-tree mehr? (Potenzmenge '(1 2 3 4))))

So erhält man folgende Liste aller Teilmengen von {1, 2, 3, 4}:

(() (4) (3) (2) (1) (3 4) (2 4) (2 3) (1 4) (1 3) (1 2) (2 3 4) (1 3 4) (1 2 4) (1 2 3) (1 2 3 4))

Will man nun noch die Teilmengen mit gleicher Elementzahl nach ihren Elementen anordnen, so .... (Hier genügt eine einzige zusätzliche reverse-Operation, aber wo? Selbst überlegen!).

2.4.7 Anmerkungen zu Zeichenketten (strings)

Eine Folge von Zeichen ist ein Wort. Der zugehörige Typ heißt in Scheme "string". Die Konstanten werden in "..." eingeschlossen, Sondersymbole durch \ abgetrennt, siehe 2.1.5.

Die Länge eines strings (length) ist die Anzahl der Zeichen im String, also eine nichtnegative ganze Zahl. Der leere String "" hat die Länge 0. Die einzelnen Zeichen eines Strings haben einen Index, nämlich ihre Nummer im String minus 1. (Die Zeichen sind also ab 0 durchnummeriert.)

(string? x) stellt fest, ob das Objekt x ein String ist.

(make-string k a) erzeugt einen String der Länge k bestehend aus k-mal dem Zeichen, das zu a gehört.

(string a b ....) erzeugt aus den Zeichen a, b, ... einen String.

(string-length s) liefert die Länge des Strings s.

(string-ref s k) liefert das (k-1)-te Zeichen des Strings s.

(substring s i j) liefert den Teilstring von s ab Index i bis einschließlich Index (j-1), für $0 \leq i \leq j \leq$ (length s).

(string->list s) liefert die Liste der Zeichen des Strings s.

(list->string L) wandelt eine Liste von Zeichen L in einen String um.

(string-copy s) liefert eine Kopie des Strings s.

2.5 Prinzipien der funktionalen Programmierung und zugehörige Programmiersprachen

2.5.1 Prinzipien (mit Varianten)

Charakteristika der imperativen Programmierung:
Das Programm ist eine Folge von Anweisungen; deren Ablaufreihenfolge wird durch bedingte Sprünge und Wiederholungen gesteuert. Variablen sind Behälter, deren Inhalte durch die Anweisungen verändert werden (die wichtigste Anweisung ist daher die Wertzuweisung). Die Menge dieser Inhalte zusammen mit der Position im Programm bilden den aktuellen Zustand des Programms. Man hat in der Regel direkten Einfluss auf die Speicherstruktur. Es gibt Konstrukte zum Strukturieren von Programmteilen (Block, Paket, Ausnahme, benannte Klammerungen, ...); wenn Teile zu eigenen aufrufbaren Programmteilen (Prozeduren) zusammengefasst werden können, spricht man auch von *prozeduraler Programmierung*. Daten werden in Datenstrukturen organisiert, wobei Verweise (Referenzen) auf Daten (d. h. auf Speicherbereiche) genutzt werden können. Letztlich iteriert ein imperatives Programm eine Abbildung auf dem Zustandsraum. Gedanklich steht im Hintergrund eine "komfortable" von-Neumann-Maschine.

Charakteristika der funktionalen Programmierung:
Das Programm ist eine Folge von Ausdrücken und Funktionen; Variablen dienen zu ihrer Benennung; Berechnungen erfolgen durch Auswertung von Ausdrücken, insbesondere durch Anwenden von Funktionen auf Argumente. Die auszuwertenden Objekte und deren Ergebnisse können wiederum Funktionen sein (Funktionen höherer Ordnung). Im Idealfall sind Seiteneffekte nicht erlaubt.

Die einzelnen Programmiersprachen eines Programmierstils unterscheiden sich zum Beispiel durch:
- zulässige Objekte, Benennung und Verwendbarkeit
- Strukturierungsmöglichkeiten, Makros, Abkürzungen
- Ausnahmebehandlung
- Seiteneffekte (vor allem Zustandsänderungen globaler Variablen)
- Typisierung (strikt, bedingt, untypisiert), Typsystem
- Polymorphie (parametrisieren, Generizität, überladen)
- Auswertungsstrategien (call by value/name/ref., lazy eval, ...)
- Abweichung von der "reinen Lehre"

Typsysteme in der Programmierung

(siehe Vorbemerkungen zu diesem Kapitel!)

Die Zuordnung Bezeichner $\leftrightarrow$ Objekt heißt "Bindung".

Diese Zuordnung wird in der Regel durch eine Deklaration hergestellt und gilt im gesamten zugehörigen Programmteil (Block, Lebensdauer des Bezeichners). Umdefinitionen in Unterbereichen sind möglich (lokale und globale Bezeichner): Es wird stets die zuletzt eingeführte Definition verwendet, wobei diese "statisch geschachtelt" im Sinne der Schachtelung des Programmtextes oder "dynamisch geschachtelt" im Sinne der Ausführungsreihenfolge während des Programmablaufs erfolgen kann.

Statische Bindung: Unveränderliche Festlegung durch die Deklaration.

Dynamische Bindung: Eindeutige Festlegung erst während der Programmausführung (z. B. bei rekursiven Prozeduren).

(Hier gibt es noch weitere Varianten.)

Eine Bindung wird meist mit Zusatzbedingungen versehen.
Die wichtigste ist der Typ. Eine *Programmiersprache* heißt

- statisch typisiert, wenn jedes Objekt (Konstante, Variable, ...) einen festen Typ während seiner Lebensdauer besitzt und dieser Typ vor der Programmausführung ermittelt werden kann,
- dynamisch typisiert, wenn sich der Typ eines Objekts durch Zuweisungen während des Programmlaufs ändern kann, aber zwischen solchen Zeitpunkten unverändert bleibt,
- typfrei, wenn der Typ eines Objekts erst zu dem Zeitpunkt festgestellt wird, zu dem dieses Objekt für eine Operation oder Funktion gebraucht wird (erst aus der aktuellen Interpretation der Operation ergibt sich der Typ des Objekts).

Eine *Typbindung* heißt

- stark, wenn mit dem Objekt nur Operationen, die für dessen Typ zulässig sind, durchgeführt werden,
- schwach, wenn gewisse Ausnahmen hiervon zugelassen sind.

Ein Datentyp legt die Menge der Werte und die auf ihnen zulässigen Operationen fest. (Ein abstrakter Datentyp legt nur Eigenschaften der Wertemenge und ihrer Operationen fest, er definiert im mathematischen Sinne also eine Algebra.)

Ein Typsystem dient mehreren Zwecken:

- Lesbarkeit, Verständlichkeit von Problemlösungen
- Fehlervermeidung
- Effizienz (z. B.: Typkontrolle zur Übersetzungszeit statt zur Laufzeit)
- Flexibilität, einfachere Verwendung im Falle des Überladens

Es gibt natürlich auch Folge-Fragen: Wann sind zwei Typen gleich, wann haben zwei Bezeichner den gleichen Typ, wie erfolgt die Typanpassung in Ausdrücken, ...?

Auswertungsstrategien

call by name (textuelles Einfügen und anschließendes Auswerten; hierbei können/sollen Variablen lokal umdefiniert werden: statische oder dynamische Umgebung von Argumenten)

call by value (Argument auswerten und den Wert übergeben)

lazy evaluation (verzögere die Auswertung von Argumenten bis zu dem Zeitpunkt, zu dem sie wirklich gebraucht werden; man wird dann z. B. beim Aufruf von

```
(define (f A B) (if (= 3 4) A B ) )
```

das Argument A niemals auswerten. Diese Bedarfsauswertung wird mittlerweile bei vielen funktionalen Sprachen zur Auswertung benutzt. (Vorsicht bei undefinierten Argumenten, siehe Vorlesung über Formale Semantik.)

2.5.2 Funktionale Programmiersprachen

Wie schon gesagt: Es gibt für jeden Programmierstil (imperativ, funktional, prädikativ, objektorientiert, ...) eine Vielzahl von Programmiersprachen. Die bekanntesten funktionalen Programmiersprachen sind zurzeit (2007):

λ-Kalkül (= der von Church 1936 entwickelte grundlegende Kalkül)
Lisp (von McCarthy 1960), Inter-Lisp, Emacs-Lisp
Common Lisp, Scheme, Dylan
Mathematica, Erlang
(APL), FP
Joy
Sisal
ML, lazy-ML, SML, Hope
Caml, Caml Light
Beta
OCAML
Gofer, Miranda, Haskell