Kapitel 2 der

Stand: 12.12.2007

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

Bitte beachten Sie: Für die Richtigkeit der Inhalte und insbesondere der Programme wird keine Garantie übernommen.

Gliederung

0. Vorbemerkungen, Voraussetzungen
1. Objektorientierte Programmierung
2. **Funktionales Programmieren**
3. Maschinennahe (abstrakte) Programme
4. Nebenläufigkeit, S/T-Netze

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

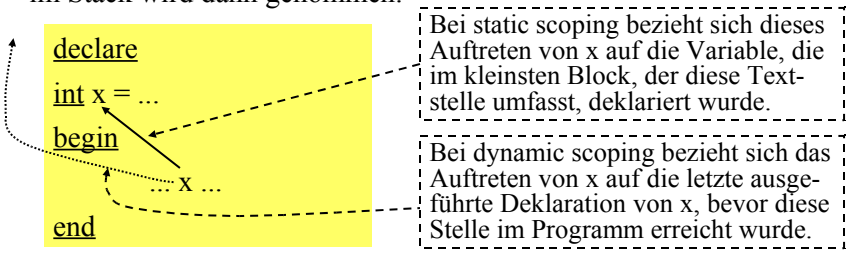
Vorbemerkungen

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.



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.

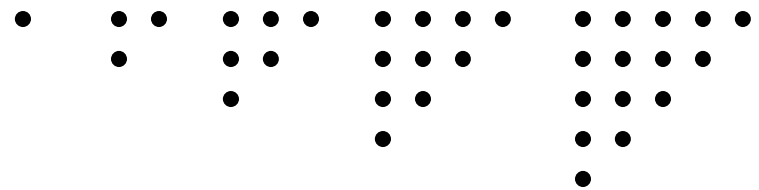
Funktionen definiert man meist durch Ausdrücke. Das zu lösende Problem ist dann ein Ausdruck, der die zuvor definierten Funktionen enthält. Ein Programm in Scheme hat daher die Form

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, $x^2 + y^2$, $D_n^2 + D_m^2$

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



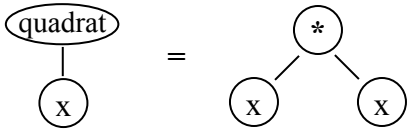
$$\mathbf{D_n} = 1 + 2 + 3 + ... + n = \sum_{i=1}^{n} i = n \cdot (n+1) / 2 = (n^2 + n) / 2$$

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

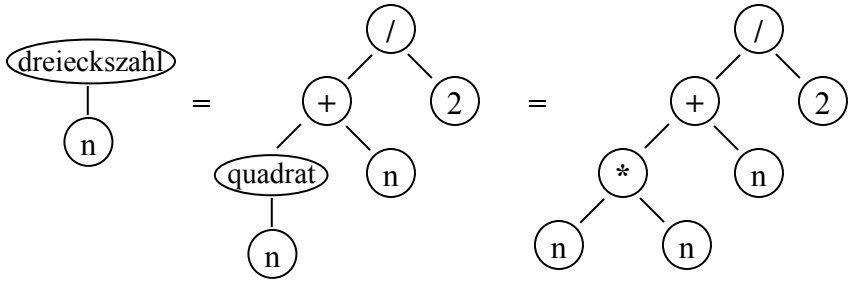
```
(define (quadrat  x) (*  x  x) )
(define (dreieckszahl  n) (/ (+ (quadrat  n)  n) 2) )
(define (quadratsumme  x  y) (+ (quadrat  x)  (quadrat  y) ) )
(define (dquadratsumme  n  m)
        (quadratsumme  (dreieckszahl  n) (dreieckszahl  m) ) )
```

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

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

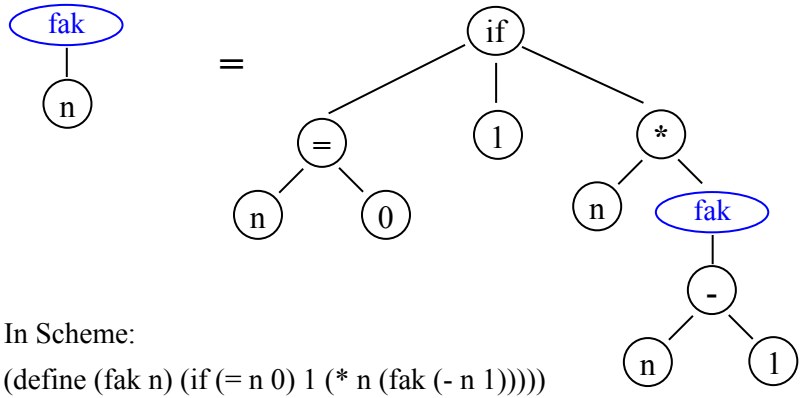


(define (dreieckszahl n) (/ (+ (quadrat n) n) 2))



Algorithmische Mächtigkeit entsteht erst durch die Rekursion.

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

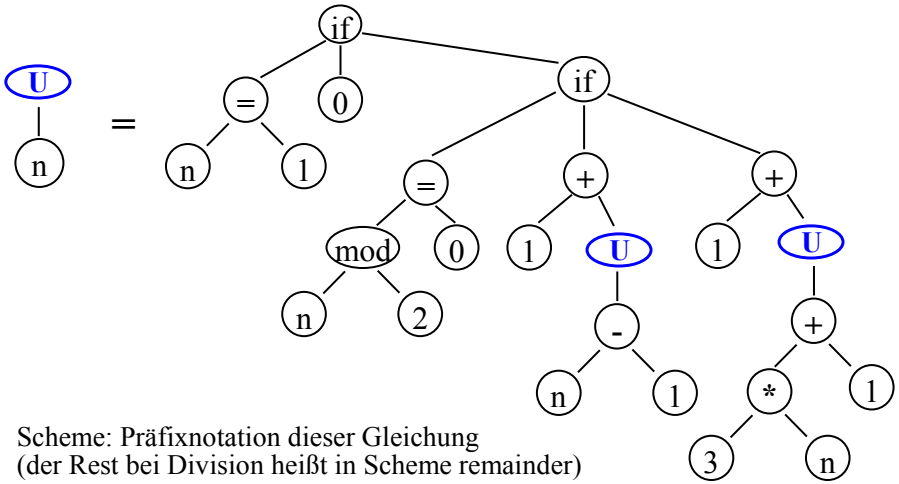


In Scheme:

(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.

Appendices
A Formal semantics
A.1 Background
A.2 Grammar
A.3 Quote
A.4 Multiple values
A.5 Exceptions
A.6 Arithmetic and basic forms
A.7 Lists
A.8 Eqv
A.9 Procedures and application
A.10 Call/cc and dynamic wind
A.11 Letrec
A.12 Underspecification
B Sample definitions for derived forms
C Additional material
D Example
E Language changes
References
Alphabetic index of definitions of concepts, keywords, and procedures

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 ⇔ (x-1) mod 3 = 0 ⇔ (x-2) mod 3 = 2,
x mod 3 = 2 ⇔ (x-1) mod 3 = 1 ⇔ (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)

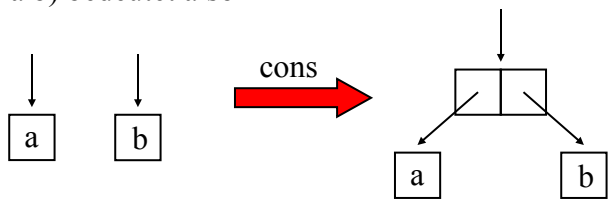
' '(+ 4 5) ==> '(+ 4 5)

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.

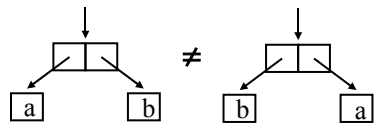
Logisch gesehen fasst der Operator cons (= "constructor") zwei Objekte zu einem kleinst möglichen binären Baum mit einer "inhaltsleeren" Wurzel zusammen.
(cons a b) bedeutet also



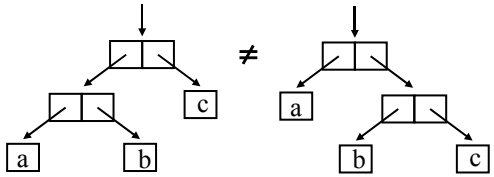
Auf die erste Komponenten kann man mittels car und auf die zweite mittels cdr zugreifen, gesprochen "kahr" und "kudd^{e}r". Die Bezeichnungen stammen von den Assemblerbefehlen der IBM-704 (ein viel benutzter Rechner von 1954 bis 1960), mit denen man die Komponenten erreichte: car = content of address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

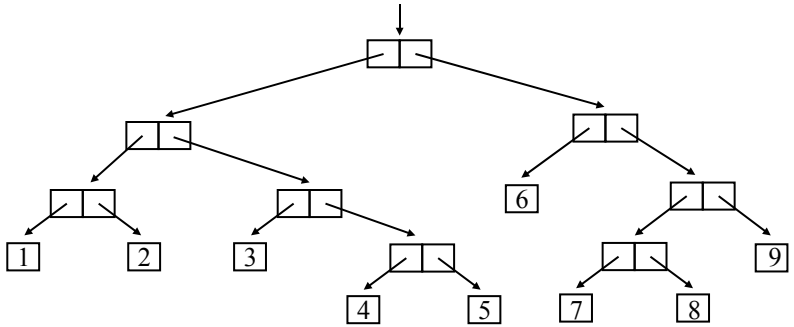


(cons (cons a b) c) ≠
(cons a (cons b c))



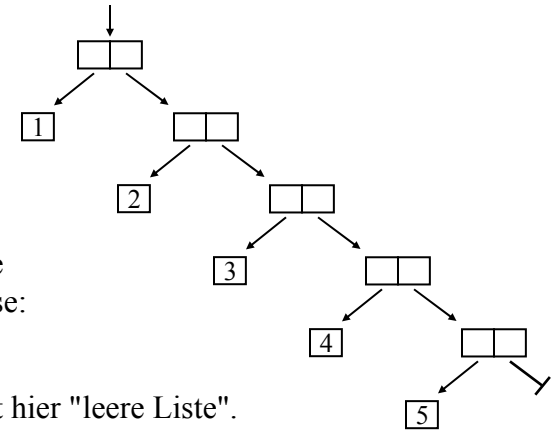
Es ist klar, dass man mit cons binäre Bäume aufbauen kann. Solche Bäume bilden die wichtigste Datenstruktur in Scheme.

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)
```

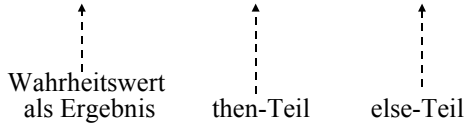
[Hinweis: DrScheme liefert hier die externe Darstellung:
(#<primitive:+> 9 8 7 6 5 4 3 2 1), d.h., "+" ist ein Symbol, kein Zeichen]

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 Zerlegung einer natürlichen Zahl in ihre Primfaktoren
2.2.3 Größter gemeinsamer Teiler (ggT, gcd) und kgV
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 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 nachzuvollziehen. 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)           ==>  ()
```


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?)

2.2.3 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 $T(x) = \{y \mid y \text{ teilt } x\}$ die Menge aller Teiler von x, dann erhält man also:

$ggT(a,b) = Max(T(a) \cap T(b))$ 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 diese 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))
```

then-Teil

else-Teil

;;; 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. Beachte: Wir prüfen nicht, ob die Liste
;;; leer ist, da später stets die Zahl 1 in der Liste sein muss.

```
(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)
(schnitt (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))) )
```

```
(ggT 808038 720720)  (teilermenge 808038)  (teilermenge 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: 16 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.

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
```

2.2.4 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 2.2.3 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 prim?.)

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 a)
        (if (prim? (+ a 1)) (+ a 1) (nextprim (+ a 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 (prim? x)
        (and (not (= x 1)) (= x (kleinster-teiler-i-bis-x  2  x))) )

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


Hinweise zu den Beispielen:

Beispiel 2.2.2 ist unkommentiert; man kann daher kaum nachvollziehen, ob es korrekt arbeitet.

Die Spezifikation in Beispiel 2.2.3 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 ganzen Zahlen genau berechnet, bleibt hier undurchsichtig.

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 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.

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 500·500, also 250.000 Einzelschritte ausführen muss, was mehrere Tage kostet. 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), siehe Mathematik.

```
function newton2 (x: real; m: natural) is
if m=0 then Ergebnis ist x else newton2 (x/2.0+1.0/x, k-1) fi
Man berechne dann z.B. newton2 (2.0, 50).
```


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, siehe Behauptung auf der nächsten Folie).

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.

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 $p_0 > q_0$ 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: a, p_k und q_k erfüllen ebenfalls die Pellsche Gleichung, d. h., es gilt: $p_k^2 - a \cdot q_k^2 = 4$.

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

$$p_{k+1}^2 - a \cdot q_{k+1}^2 = (p_k^2 - 2)^2 - a \cdot (p_k \cdot q_k)^2 = p_k^4 - 4p_k^2 + 4 - a \cdot p_k^2 \cdot q_k^2$$

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

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

= 0, weil a, p_k und q_k die Pellsche Gleichung nach Induktionsannahme erfüllen. ■

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

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

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**, diesmal 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 exakten rationalen Zahl sieht man die Ziffernfolge nicht an, daher stellen wir das Ergebnis "inexakt" nochmals 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 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. Will man eine Dezimaldarstellung als Ausgabe, so 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 ist aber keine Quadratzahl).

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 gegen die Wurzel aus a. Diese Wurzeln 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	2,666666...
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	9
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.

Aufgabe: 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
```


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

75556361324694677031114817108679665456610659127864222100124800 1 /
23893019350069211314452863610358711580181288639111879111704136

57087637366278175705092720001647140561995771825368044249389595539
16001346687858740383097313789890205078575145409019995064960001 /
57087637366278175705092720001647140561995771825368044249389595539
16001346687858740383097313789890205078575145409019995064960

Differenz des Quadrats zu a:  -1 /
57087637366278175705092720001647140561995771825368044249389595539
16001346687858740383097313789890205078575145409019995064960

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 müssten 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:

Programm für die Lösungsmethode 1

```
(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))  )))
```

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:

Programm für die Lösungsmethode 2

```
(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))
```

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.