Funktionales Programmieren im Gymnasialunterricht

Josef Paintner, Martin Ruckert, Helmut Schwichtenberg

Mathematisches Institut der Universität München
Sommersemester 2000

Inhaltsverzeichnis

1. Einführung

Warum Programmieren im Mathematikunterricht? Die algorithmischen Aspekte der Mathmatik sollen erfahrbar gemacht werden.

Warum die Programmsprache Scheme? Sie ist einfach, also ohne viel Aufwand nebenbei zu lernen. Sie erlaubt das Programmieren in einer Vielzahl von Programmierstilen, nicht nur das funktionale Programmieren. Natürlich ist die Zeit im Mathematikunterricht so knapp bemessen, daß man keinesfalls systematisch eine zusätzliche komplexe Programmiersprache so nebenbei einführen kann. Damit die Schüler tatsächlich selbstständig programmieren, sollten sie die zugehörige Software auf ihrem heimischen Rechner zur Verfügung haben. Falls es dann gelingt, sie mit der einfachen Syntax vertraut zu machen, können vielleicht einige dazu ermuntert werden, das Programm zur Problemlösung gelegentlich selber einzubeziehen. Um die Schüler mit dem Programm vertraut zu machen, kann es durchaus sinnvoll sein, bereits in der Unterstufe, etwa bei Termen, ein solches Programm einzusetzen. Am Beispiel von Scheme soll erläutert werden, wie man hier etwa vorgehen kann.

Der Einsatz von Scheme im Unterricht sollte so erfolgen, daß immer deutlich das Problem im Vordergrund steht und Probleme, die sich aus der Syntax der Sprache ergeben, von untergeordneter Bedeutung bleiben. Der Einsatz des Rechners sollte den Schülern primär helfen, mathematischen Begriffe und Vorgehensweisen besser zu verstehen. Vor allem sollte deutlich herausgestellt werden, daß Algorithmen fast wörtlich in ein Programm übersetzt werden können.

Beispielsweise der Termbegriff, das Einsetzen von Termen in andere Terme, das Erkennen der Art eines Termes und Termgliederungen kann in den Klassen 5 und 6 bereits sehr sinnvoll mit Hilfe dieses Programmes erläutert werden.

Beim Buchstabenrechnen (in Klasse 7) sind ständig die Schwierigkeiten mit den Vorzeichenregeln beim Auflösen von Klammern zu beobachten. Ein Überprüfen der Ergebnisse bei Termumformungen unterbleibt natürlich in der Regel, da der Rechenaufwand zu groß wäre. Gerade hier kann ein selbständiges Verwenden des Programms zu Hause wesentlich zum Verständnis beitragen. Die Behandlung von Ungleichungen und Gleichungen mit Beträgen bietet auch ein weites Anwendungsfeld. Für die Behandlung dieser Aufgabenfelder reichen etwa ein halbes Dutzend Sprachkonventionen völlig aus. Diesen Sprachumfang kann man den Schülern im Unterricht „nebenbei“ nahebringen. Diese Grundelemente der Sprache, versehen mit einigen passenden Beispielen, sollen der Inhalt der ersten beiden Vorlesungen sein.

Wichtige Gesichtspunkte: Betonung auf symbolischem Rechnen. Modellierung der Wirklichkeit in einem mathematischen Modell, das auch ausführbar ist. Nachvollziehen von Algorithmen, die der Schüler gelernt hat, auf dem Rechner, möglichst in derselben Form wie sie vom Schüler auch beim Rechnen mit „Papier und Bleistift“ benutzt werden. Dadurch gründliche Vertiefung von Lernstoff. Hervorhebung des algorithmischen Aspektes der Mathematik. Lernkontrolle: nur wenn ein Algorithmus genau verstanden ist kann er implementiert werden.

Für das Arbeiten mit Schmem im Gymnasialunterricht eignet sich besonders die Implementierung „DrScheme“, die an der Rice University speziell auch für die Arbeit mit Schülern entwickelt wurde. Die Installation auf Windows-, Linux- und Unix-Rechnern ist kostenlos und wird sehr gut unterstützt. Eine Anleitung zur Installation von DrScheme ist zu finden unter

http://www.mathematik.uni-muenchen.de/˜ruckert/scheme.html

Ebenfalls gut geeignet ist eine am Mathematischen Institut der LMU (von Otto Forater) entwickelte Implementierung LMUScheme, die ebenfalls kostenlos zu erhalten ist unter

http://www.mathematik.uni-muenchen.de/˜forster/

Scheme-Tutorials erhält man unter

http://www-edlab.cs.umass.edu/cs287/index.html

und

http://www.cs.rice.edu/~dorai/t-y-scheme/t-y-scheme.html

Zur Thematik der Vorlesung gibt es ein interessantes neues Buch „How to Design Programs“ [5] von Felleisen, Findler, Matthew und Krishnamurthi, das bei MIT Press erscheinen wird. Es ist auch im Internet abrufbar unter

http://www.cs.rice.edu/CS/PLT/Teaching/Lectures/Released/

2. Prozeduren und numerische Daten

2.1 Zahlen

Es ist wichtig, folgende Unterscheidungen zu treffen:

– mathematische Zahlen,
– Scheme-Zahlen, die sie modellieren sollen,
– die Maschinen-Repräsentation von Zahlen, und
– die verwendete Bezeichnungsweise für Zahlen.

Mathematisch sind die Zahlen in einer aufsteigenden Liste von Teiltypen angeordnet:

komplex
reell
rational
ganz

Zum Beispiel ist 27 eine ganze, also auch eine rationale, also auch eine reelle, also auch eine komplexe Zahl.

Eine weitere wichtige Unterscheidung betrifft die Maschinen-Repräsentation von Zahlen: eine Zahl kann *exakt* oder *inexakt* sein. Zum Beispiel müssen Indizes von Datenstrukturen immer exakt gegeben sein, und andererseits sind Ergebnisse von Messungen von Natur aus inexakt. Scheme unterscheidet explizit zwischen exakten und inexakten Zahlen; diese Unterscheidung ist orthogonal zu der nach Typen.

Wenn immer möglich arbeitet Scheme mit exakten Zahlen. Inexaktheit ist eine ansteckende Eigenschaft von Zahlen: wann immer in einem Ausdruck ein Anteil inexakt ist, wird es auch das Ergebnis sein. Beispiel für Operationen, die aus exakten Zahlen ein exaktes Ergebnis machen, sind

`+`	`-`	`*`
`quotient`	`remainder`	`modulo`
`max`	`min`	`abs`
`numerator`	`denominator`	`gcd`
`lcm`	`expt`	`/`

Beispiel 2.1.1. `(/ 6 18) ==> 1/3`
`(/ 6.0 18)) ==> 0.3333333333333333`
`(exact->inexact (/ 6 18)) ==> 0.3333333333333333`

Für reelle und komplexe Zahlen gibt es (u.a.) die eingebauten Prozeduren

`exp`	`log`	`sin`
`cos`	`sqrt`	`expt`

Beispiel 2.1.2. `(exp 1) ==> 2.7182818284590455`
`(sqrt -1) ==> 0+1i`

2.2 Prozeduraufrufe

Die Syntax von Prozeduraufrufen ist

```
(<operator> <operand1> ...)
```
(2.1)

Die Reihenfolge der Auswertung ist nicht festgelegt. Ausdrücke der Form (2.1) heißen *Kombination* oder *Verbund*.

Beispiel 2.2.1. `(+ 3 4) ==> 7`
```
(- 27 19) ==> 8
(* 13 6) ==> 78
(/ 27 9) ==> 3
(/ 10 6) ==> 5/3
(+ 2.7 10) ==> 12.7
```

Man beachte, daß wir die sogenannte *polnische Schreibweise* verwenden, in der der Operator links steht. Dies hat 2 wesentliche Vorteile:

1. eine beliebige Anzahl von Operanden ist möglich, und
2. geschachtelte Ausdrücke können besonders übersichtlich dargestellt werden.

Beispiel 2.2.2. `(+ 3 4 2 10) ==> 19`
```
(* 13 6 77) ==> 6006
```

Beispiel 2.2.3. Klasse 5: Gliedern von Termen. Im Arithmetikunterricht lernt man auch das Gliedern von Termen. Betrachten wir etwa

$$2(34-17)(15+25)-65/13.$$

Der Term ist eine Differenz. Der Minuend ist ein Produkt, dessen erster Faktor 2, dessen zweiter Faktor die Differenz der Zahlen 34 und 17 und dessen dritter Faktor die Summe der Zahlen 25 und 15 ist. Der Subtrahend ist der Quotient der Zahlen 65 und 13.

Diese Gliederung läßt sich nun fast wortgetreu in einen Scheme-Ausdruck übersetzen. Auf diese Weise kann man die polnische Notation sehr schön motivieren.

In Scheme schreibt man (mit Unterstützung des Editors) diesen Ausdruck in der Form

```
(- (* 2
      (- 34 17)
      (+ 15 25))
   (/ 65 13))
```

Das Ausrechnen von innen nach außen kann man in DrScheme mit dem dort vorhandenen „Stepper" besonders anschaulich machen.

Alternativ kann diese Gliederung auch mit einem Baumdiagramm veranschaulicht werden, was aber im wesentlichen dasselbe ist.

In einer Unterrichtsstunde lassen sich so bereits mehrere Terme gliedern, wobei man darauf achten sollte, daß die Termwerte auch parallel von Hand berechnet werden. Das Programm sollte als Kontrollinstanz verwendet werden. Der Reiz des Neuen führt hier zu einer guten Motivation.

Beispiel 2.2.4. Klasse 6: Taschenrechner I. Wenn man einen Editor hinreichend beherrscht, kann man jetzt seinen Rechner auch wirklich als solchen benutzen und die Aufgaben eines Taschenrechners von ihm erledigen lassen.

```
(- (* 2
      (- 34 17)
      (+ 15 25))
   (/ 65 13))    ==> 1355
```

Der Interpreter arbeitet immer in einem gewissen Basiszyklus, nämlich

lesen-auswerten-drucken

(*read-eval-print* loop). Insbesondere ist es nicht nötig, den Interpreter explizit zum Ausdrucken des Wertes aufzufordern. Dies macht das Arbeiten mit Scheme besonders einfach: man kann sofort „loslegen“.

2.3 Zuweisungen

Man kann einem Namen (einer „Variablen“) einen Wert (z.B. eine Zahl) zuweisen.

Beispiel 2.3.1. `(define laenge 2)`

```
laenge ==> 2
(* 5 laenge) ==> 10
(+ (* 5 laenge) (* laenge laenge)) ==> 14
```

Beispiel 2.3.2. Klasse 7: Kreisumfang.

```
(define pi 3.14159)
(define radius 10)
(* pi * radius radius) ==> 314.159
(define kreisumfang (* 2 pi radius))
kreisumfang ==> 62.8318
```

Man könnte stattdessen (wie es die Numeriker machen) auch die Zahl π definieren durch

```
(define pi (* 4 (atan 1)))
```

Beispiel 2.3.3. Klasse 6: Taschenrechner II. Es ist hilfreich, bei Rechenaufgaben, für die man einen Taschenrechner benutzen möchte, die Möglichkeit der Benennung von Hilfsgrößen zu haben. Dies wurde schon im vorangehenden Beispiel deutlich

```
(define zins (/ 3 100))
(define kapital 763)
(+ kapital (* zins kapital)) ==> 78589/100
(exact->inexact (+ kapital (* zins kapital))) ==> 785.89
```

2.4 Benutzerdefinierte Prozeduren

Mit der Multiplikation `*` kann man die Quadratfunktion definieren:

Beispiel 2.4.1. Klasse 6: Quadrieren.

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

Man erhält

```
(quadrat 3) ==> 9
(quadrat (+ 2 5)) ==> 49
(quadrat (quadrat 3)) ==> 81
```

Wir können `quadrat` benutzen, um daraus weitere Prozeduren zu definieren, etwa

```
(define (summe-von-quadraten x y) (+ (quadrat x) (quadrat y)))
```

Man erhält

```
(summe-von-quadraten 3 4) ==> 25
```

Beispiel 2.4.2. Klasse 6: Umrechnen I. Häufig sind in Lehrbüchern auch Übungsaufgaben zu finden, bei denen zu Termen mit Variablen diese für bestimmte Einsetzungen auszuwerten sind (meist bei eingekleideten Fragestellungen). $T(a) := 1.95583 * a$ (Umrechnung von Euro in DM). Diese Umrechnung erfolgt in Scheme dann wie folgt:

```
(define (euro->dm a) (* 1.95583 a))
```

Solche Beispiele lassen sich dann auch zu Fragestellungen ausbauen, bei denen das Einsetzen von Termen in Terme notwendig wird.

Beispiel 2.4.3. Klasse 6: Umrechnen II. Firma Zisch&Co zahlt allen Beschäftigten 12 Euro pro Stunde. Die Arbeitszeit beträgt mindestens 20 und höchstens 65 Stunden pro Woche. Entwickle ein Programm, das den Wochenlohn eines Arbeiters in Abhängigkeit von der Wochenarbeitszeit berechnet. Das zu erstellende Programm liest eine Größe, die Anzahl `n` der Stunden ein und gibt eine andere Größe, den Wochenlohn `w` in Euro, aus. Der relevante Term für die Variable ist somit `12*n`. Nun ist es leicht, mit unseren bisherigen Kenntnissen über Scheme das Problem zu lösen. `(define (wochenlohn n) (* 12 n))`. Welche Angaben in obigem Beispiel waren für die gestellte Aufgabe völlig belanglos?

Beispiel 2.4.4. Klasse 6: Einsetzen von Termen in Terme. Eine Firma, die Rechner repariert, erstellt ihre Rechnung in Euro. Für die Anfahrt werden 0.25 Euro pro km berechnet, für die Arbeitszeit 50 Euro pro Stunde. Für diese Kosten ergibt sich also eine Formel mit den Variablen s für die Wegstrecke und t für die Arbeitszeit. Kosten $:= s * 0.25 + t * 50$. Die Umsetzung in Scheme sieht dann so aus:

```
(define (kosten s t) (+ (* s 0.25) (* t 50)))
```

Bei 56km Anfahrtsstrecke und 3 Stunden Arbeit ergibt sich somit der Rechnungsbetrag in Euro als `(kosten 56 3)`. Um nun die Kosten in DM zu erhalten, müssen wir nur noch unsere erste Umrechnung auf diese Funktion Kosten anwenden:

```
(define (kostenmark s t) (euro->dm (kosten s t)))

(kostenmark 56 3) ==> 320.75612
```

Beispiel 2.4.5. Klasse 10: Kreisring. Im Unterricht lernt man, die Abhängigkeit zwischen Größen mit Hilfe von Variablen auszudrücken. Z.B. $V(l, b, h) = l * b * h$; $A(r) = 3.14 * r^2$. Ein Quader mit $l = 4$, $b = 5$ und $c = 7$ hat dann das Volumen $V(4, 5, 7) = 4 * 5 * 7$; ein Kreis mit Radius 5 hat die Fläche $A(5) = 3.14 * 5^2$. Allgemein gesagt sind Ausdrücke, welche Variable enthalten, Regeln, wie man eine Zahl (Größe) errechnet, wenn Werte für die Variablen gegeben sind. Ein Programm (eine Prozedur) ist so eine Regel. Es ist eine Regel, die uns und dem Computer erklärt, wie man aus irgendwelchen Daten neue Daten erzeugt. Große Programme bestehen aus vielen kleinen Programmen, die in bestimmter Weise miteinander zusammenarbeiten. Um die Übersicht zu behalten, ist es wichtig, die Regeln (Programme) geeignet zu benennen. Ein passender Name für unseren Ausdruck $3.14 * r^2$ ist Kreisfläche. Verwenden wir diesen Namen, so sieht die Regel zur Berechnung der Kreisfläche folgendermaßen aus:

```
(define (kreisflaeche r) (* 3.14 r r))
```

Diese zwei Zeilen besagen, daß `kreisflaeche` eine Regel ist, welche die eine Eingabe (input), genannt `r`, benötigt, und die das Resultat (output) `(* 3.14 r r)` erzeugt, wenn der Wert für `r` bekannt ist.

```
(kreisflaeche 10) --> 314
```

Das soeben entworfene Programm können wir nun verwenden, um für die Fläche eines Kreisringes ebenfalls ein Programm zu erstellen. Bei einem Kreisring ist aus einem Kreis mit Radius `ra` in der Mitte ein Kreis mit Radius `ri` herausgestanzt.

```
(define (kreisring ra ri)
  (- (kreisflaeche ra) (kreisflaeche ri)))
```

Da wir gerade ein Programm zur Berechnung der Kreisfläche geschrieben hatten, lag es nahe, dieses als Modul für das neue Problem zu verwenden. Hätten wir allerdings ganz von vorne begonnen, so hätten wir möglicherweise das Programm wie folgt geschrieben:

```
(define (kreisring ra ri)
  (- (* 3.14 ra ra) (* 3.14 ri ri)))
```

Für ein kleines Problem ist der Unterschied zwischen diesen beiden Methoden gering. Bei langen Programmen ist die erste Methode meist vorzuziehen. Sogar bei kleineren Programmen sollten wir uns bemühen, dieses in mehrere kleinere Programme zu zerlegen, die dann geeignet zusammengesetzt werden.

Beispiel 2.4.6. Klasse 7: Auflösen von Klammern. Falls die Schüler in Klasse 7 mit der Syntax vertraut sind, kann man darangehen, auch bei der Auflösung von Klammern in komplexeren Termen den Rechner zu verwenden. Durch geeignete Einsetzungen läßt sich dann das Ergebnis der Umformung unmittelbar ermitteln. Bei solchen Beispielen zeigt sich deutlich die Notwendigkeit, für einige Einsetzungen das Ergebnis auch von Hand zu ermitteln, da sehr schnell eine Klammer falsch gesetzt wird. Das „Programm" läuft dann zwar, liefert aber falsche Ergebnisse. Beispiel:

$$-(20d + 10a - 40c) - (-(7a - 14c) - (-(15d - 4c) - (27a - 5d)))$$

```
(define (T a d c)
  (- (- (+ (* 20 d) (* 10 a) (* (- 40) c)))
     (- (- (- (* 7 a) (* 14 c)))
        (- (- (- (* 15 d) (* 4 c)))
           (- (* 27 a) (* 5 d))))))

(T 1 1 1) ==> -30
```

Die Koeffizienten von a, d und c kann man errechnen durch

```
(T 1 0 0) ==> -30
(T 0 1 0) ==> -30
(T 0 0 1) ==> 30
```

Als Ergebnis erhält man also

$$-30a - 30d + 30c.$$

Beispiel 2.4.7. Klasse 9: Abhängige Größen I. Stellen Sie sich vor, der Besitzer eines Kinos kann die Eintrittspreise völlig frei festlegen. Je höher der Eintrittspreis ist, desto weniger Leute werden die Vorstellung besuchen. In einem zurückliegenden Experiment hat der Besitzer einen präzisen Zusammenhang zwischen Billettpreis und durchschnittlicher Zuschauerzahl ermittelt. Bei 5 Euro Eintrittspreis besuchen 120 Leute die Vorstellung. Vermindert man den Preis jeweils um 10C, so steigt die Zuschauerzahl um je 15 an. Unglücklicherweise verursacht eine größere Zuschauerzahl auch höhere Kosten, und zwar um 4C je Zuschauer. Bei 120 Zuschauern betragen die Kosten insgesamt 1.60 Euro je Zuschauer. Für jeden Zuschauer weniger als 120 darf er 4C Kosten pro Zuschauer abziehen. Der Besitzer will wissen, bei welchem Eintrittspreis er den größten Profit macht. Die Aufgabenstellung ist klar, aber es ist keineswegs offensichtlich, wie man vorgehen muß. Alles was wir sagen können ist, daß einige Größen jeweils voneinander abhängen:

– Die Kosten des Besitzers hängen von der Zuschauerzahl (zusch) ab: Kosten = 1.60 + 0.04* (zusch-120)
– Die Zuschauerzahl hängt vom Eintrittspreis (preis) ab: zusch(preis) = ... usf.

Versuchen wir, ein Programm zu entwickeln, das den Profit bei gegebenem Eintrittspreis ermittelt. Wenn wir mit einer solchen Situation konfrontiert werden, ist es am besten, die einzelnen Abhängikeiten getrennt zu untersuchen. In unserem hypothetischen Beispiel erzielt man alle Einnahmen aus dem Verkauf der Eintrittskarten. Die Einnahme hängt nur ab von der Zuschauerzahl (zusch) und dem Eintrittspreis (preis). Wir formulieren zunächst ein Programm, das den zugehörigen Zusammenhang festhält:

```
(define (einnahme zusch preis) (* zusch preis))
```

Als nächstes versuchen wir, die Kosten (kost) zu verstehen. Die Grundkosten sind 1.60*zusch. Für jeden Zuschauer über 120 erhöhen sich die Kosten um 0.04 Euro, für jeden Zuschauer weniger verringern sich die Kosten entsprechend. Deshalb müssen wir zu den Grundkosten den Term 0.04*(zusch -120) addieren. Wir können nun ein Scheme-Programm formulieren, das den Zusammenhang zwischen Kosten (kost) und der Zuschauerzahl (zusch) festhält:

```
define (kost zusch)
 (+ (* zusch 1.60) ; Grundkosten
    (* 0.04 (- zusch 120)))) ; Zusatzkosten fuer mehr Zuschauer
```

Schließlich müssen wir noch den Zusammenhang zwischen Zuschauerzahl und Eintrittspreis untersuchen. Bei einem Eintrittspreis von 4.90 Euro kommen $120 + 15$ Zuschauer. Bei 4.50 Euro Eintritt kommen $120 + 5 * 15$ Zuschauer. Übersetzt in ein Scheme-Programm erhält man also:

```
(define (zusch preis)
  (+ (* (/ 15 0.10) (- 5.00 preis)) 120))
```

Den Profit erhält man nun, wenn man von den Einnahmen die Kosten abzieht. Das fertige Programm sieht dann folgendermaßen aus:

```
;Programm Theaterbesitzer mit Teilprogrammen
(define (einnahme zusch preis) (* zusch preis))

(define (kost zusch)
  (+ (* zusch 1.60) (* 0.04 (- zusch 120))))

(define (zusch preis) (+ (* (/ 15 0.10) (- 5.00 preis)) 120))

(define (profit preis)
  (- (einnahme (zusch preis) preis)
     (kost (zusch preis))))

;Programm Theaterbesitzer in einem Stueck

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

Leicht läßt sich überprüfen, daß beide Programme die gleichen Ergebnisse liefern. Das erste Programm zeigt unmittelbar, welche Gedankengänge zur Lösung geführt haben. Das zweite Programm hingegen kann kaum mehr „mit Verständnis“ gelesen werden. Falls wir einen Aspekt modifizieren wollen, etwa den Zusammenhang zwischen Zuschauerzahl und Preis, so kann dies in der ersten Form sehr leicht erfolgen. Bei der Formulierung in der zweiten macht diese Modifikation bereits einige Mühe. Leitlinie: Formuliere Hilfsprogramme, wenn die Programmdefinition einen langen Term erfordert. Das Wort „lang“ wird natürlich für jeden Programmierer eine unterschiedliche Bedeutung haben.

Beispiel 2.4.8. Klasse 9: Abhängige Größen II. Ermitteln Sie möglichst genau den “optimalen” Eintrittspreis. Der Besitzer erreicht durch Rationalisierungsmaßnahmen, daß die Kosten pro Zuschauer einfach 1.50 Euro betragen. Modifizieren Sie beide Programmformulierungen entsprechend und ermitteln Sie nun unter der neuen Vorraussetzung den optimalen Eintrittspreis.

Programme und ihre Fehlerfreiheit

Wenn wir Scheme-Programme schreiben, müssen wir einige sorgfältig ausgewählte Regeln befolgen, welche einen Kompromiß zwischen der Aufnahmefähigkeit eines Computers und der gewöhnlichen menschlichen Ausdrucksweise darstellen. (Dies gilt für alle Programiersprachen).

Nicht alle Klammerausdrücke sind auch Scheme-Ausdrücke. Etwa akzeptiert Scheme `(10)` nicht als legalen Ausdruck, da für einzelne Zahlen keine Klammer vorgesehen ist. Auch ein Ausdruck der Form `(10 + 20)` ist nicht korrekt („ill-formed“), da Scheme immer zuerst den Operator erwartet. Schließlich sind auch folgende zwei Definitionen falsch formuliert:

```
(define (P x) (+ (x) 10))
(define (P x) + x 10)
```

Bei der ersten ist die zusätzliche Klammer um den atomaren Ausdruck `x` nicht erlaubt, bei der zweiten ist der Verbund `+ x 10` nicht von Klammern eingeschlossen.

Immer, wenn wir den „execute-button“ drücken, prüft Scheme, ob der eingegebene Ausdruck sprachlich korrekt ist. Falls Fehler auftreten, signalisiert dies Scheme und hebt außerdem (in DrScheme) die problematischen Teile deutlich hervor. (Language Beginner)

Beispiel 2.4.9. Klasse 10: Fehlermeldungen I. Werte folgende Ausdrücke einzeln aus. Versuche dabei, die Fehlermeldungen zu verstehen. `(+ (10) 20)`, `(10 + 20)`, `(+ +)`. Gib folgende Sätze, jeden für sich, im Definitionsfenster ein und klicke dann auf execute. Verbessere dann die Definitionen mit Hilfe der Fehlermeldungen.

```
(define (f 1) (+ x 10)),
(define (g x) + x 10),
(define h(x) (+ x 10)).
```

Nicht alle sprachlich korrekten Ausdrücke lassen sich auch auswerten. Beispielsweise ist die Division durch Null nicht zulässig. Auch können wir bei `(define (f n) (+ (/ n 3) 2))` Scheme nicht den Ausdruck `(f 5 8)` auswerten lassen.

Niemand kann *logische Fehler* völlig vermeiden. Selbst erfahrenen Programmierern unterlaufen Syntax-Fehler, oder sie machen Fehler, die zu run-time Problemen führen. Eine gute Programmierumgebung hilft hier bei der Fehlersuche. Bei logischen Fehlern läuft das Programm einwandfrei, liefert aber falsche Resultate. Das Vermeiden von logischen Fehlern ist das Ziel einer systematischen Programmentwicklung, allerdings auch die bei weitem schwerste Aufgabe.

Beispiel 2.4.10. Klasse 10: Fehlermeldungen II. Werte folgende, sprachlich korrekte Ausdrücke nacheinander aus: `(+ 5 (/ 1 0))`, `(sin 10 20)`, `(somef 10)`. Geben Sie den sprachlich korrekten Ausdruck `(define (somef x) (sin x x))` im Definitionsfenster ein. Versuchen Sie dann im dann, `(somef 10 20)` bzw. `(somef 10)` im unteren Fenster zu evaluieren. Beachten Sie hier besonders die auftretenden Fehlermeldungen.

Rezept zum Erstellen einfacher Programme

Am Beispiel des Programms zur Berechnung der Fläche eines Kreisrings machen wir uns nochmals klar, wie man bei der Gestaltung eines Programms vorgehen muß. Diese Vorgehensweise muß nicht eingehalten werden, erleichtert aber meist den Weg zum Programm erheblich. Zunächst muß man den Zweck des Programms verstehen. Beim Erstellen eines Programms ist das Ziel in der Regel, einen Mechanismus zu erzeugen, der Daten einliest und hieraus neue Daten erzeugt. Dazu wählen wir zunächst einen aussagekräftigen Namen und klären, welche Art von Informationen das Programm verarbeitet bzw. erzeugt. Wir nennen dies „Kontrakt”“.

```
; ringflaeche: Zahl Zahl -> Zahl
```

Wenn der Kontrakt steht, können wir den Programmkopf anfügen. Dieser legt den Programmnamen nochmals, evtl. modifiziert fest und gibt allen Parametern unterscheidbare Namen.

```
(define (ringflaeche ra ri) ...)
```

Schließlich formulieren wir nun den Zweck des Programms, das ist z.B. ein kurzer Kommentar, was das Programm eigentlich berechnet.

```
; Berechne die Flaeche eines Kreisringes, bei dem der aeussere Radius ra und
; der Radius des Loches ri ist.
```

Der Anfang des Beispiels sieht dann insgesamt so aus:

```
; ringflaeche: Zahl Zahl -> Zahl
; Berechne die Flaeche eines Kreisringes, bei dem der aeussere Radius ra und
; der Radius des Loches ri ist.
(define (ringflaeche ra ri) ...)
```

Falls die Problemstellung eine mathematische Formel verwendet, gibt die Anzahl der unterschiedlichen Variablen einen Hinweis auf die Anzahl der Eingaben.

Um ein besseres Verständnis dafür zu gewinnen, was das Programm eigentlich tun soll, legen wir Beispiele für die Eingaben fest und untersuchen, was das Programm hier ausgeben soll. Das Programm Ringfläche sollte für die Eingaben 5 und 3 das Ergebnis 50,24 liefern.

```
; ringflaeche: Zahl Zahl -> Zahl
; Berechne die Flaeche eines Kreisringes, bei dem der aeussere Radius ra und
; der Radius des Loches ri ist.
; Beispiel: (ringflaeche 5 3) = 50.24
(define (ringflaeche ra ri)  ...)
```

Der Programmkörper besteht aus einem Scheme-Ausdruck, der aus den Parametern, aus primitiven Scheme-Operationen und aus Programmen, die bereits definiert sind oder noch definiert werden, besteht. Wir können den Programmkörper nur formulieren, wenn wir verstanden haben, wie das Programm aus den Eingaben die Ausgaben berechnet. In unserem Beispiel ist der Programmkörper eine mathematische Formel, welche auf ein früher definiertes Programm zurückgreift.

```
(define (ringflaeche ra ri) (- (kreisflaeche ra) (kreisflaeche ri)))
```

Nachdem das Programm fertiggestellt ist, müssen wir es noch testen. Hierbei sollten zumindest die Ergebnisse des Beispiels überprüft werden. Ein Test kann nie die Korrektheit eines Programms beweisen, wohl aber helfen, ein fehlerhaft arbeitendes Programm als solches zu erkennen.

Um das Programm eventuell in einem anderen Zusammenhang wieder verwenden zu können, sollten wir die Definitionen unter dem Programmnamen abspeichern. Wir müssen später dann nicht mehr wissen, wie das Programm arbeitet, sondern uns nur daran erinnern, was es tut. Eine deutliche Kommentierung ist hier sehr hilfreich.

```
; Kontrakt: ringflaeche: Zahl Zahl -> Zahl
; Zweck: Berechne die Flaeche eines Kreisringes, bei dem
; der aeussere Radius ra und der Radius des Loches ri ist.
; Beispiel: (ringflaeche 5 3) = 50.24

; Definition:
(define (ringflaeche ra ri) (- (kreisflaeche ra) (kreisflaeche ri)))

; Tests: (ringflaeche 5 3) (ringflaeche 20 10)
```

2.5 Bedingte Ausdrücke

Bisher haben wir noch keine Möglichkeit, Fallunterscheidungen auszudrücken, wie etwa in

$$\mathsf{abs}(x) = \begin{cases} x & \text{falls } x > 0, \\ 0 & \text{falls } x = 0, \\ -x & \text{falls } x < 0. \end{cases}$$

Auch ein Spielprogramm muß beispielsweise entscheiden, ob sich ein Objekt in einem bestimmten Bereich des Bildschirms befindet oder nicht.

In Scheme gibt es zur Formulierung von Fallunterscheidungen die spezielle Form `cond`.

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

oder auch

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

Statt `cond` kann man auch `if` benutzen:

```
(define (abs x) (if (< x 0) (- x) x))
```

Ein Scheme-Ausdruck, der Zahlen vergleicht, hat natürlich auch ein Resultat. Dieses ist bei Bedingungen wie `(> x 0)` immer ein *boolescher Wert*, d.h. einer der speziellen Daten `#f` (`f` steht für „false“) oder `#t` (`t` steht für „true“).

Allgemein spielt offenbar eine wesentliche Rolle, welche Objekte von `if` und `cond` als wahr oder falsch angesehen werden. In Scheme gilt die Konvention, daß *nur* das boolesche Objekt `#f` als falsch gilt, und jedes andere Objekt als wahr gilt.

Die allgemeine Form eines `if`-Ausdrucks ist

(`if` *Test Ja-Ausdruck Nein-Ausdruck*).

`if` ist keine normale Scheme-Prozedur, die immer zuerst alle Argumente auswertet. Stattdessen wertet `if` zuerst nur *Test* aus. Ergibt dies den Wert `#f` (`f` steht für „false“), so wird dann *Nein-Ausdruck* ausgewertet und das Resultat als Wert zurückgegeben. Jeder andere Wert von *Test* ist – wie schon erwähnt – so gut wie `#t` und führt zur Auswertung von *Ja-Ausdruck*.

Entsprechendes gilt für `cond`. Die allgemeine Form eines `cond`-Ausdrucks ist:

```
(cond (frage1 antwort1) (frage2 antwort2) ...  (fragek antwortk))
```

oder

```
(cond (frage1 antwort1) (frage2 antwort2) ...  (else antwortk))
```

Geprüft wird der Reihe nach, ob eine `frage` erfüllt ist. Falls ja, wird die zugehörige `antwort` ausgegeben und das Programm beendet. Ist eine weitere Bedingung zusätzlich erfüllt, bleibt dies unberücksichtigt. Beispiel:

```
(define (bedingung n)
  (cond ((> n 10) 20)
        ((> n 20) 0)
        (else 1)))

(bedingung 27) ==> 20
```

Man nennt deshalb `cond` und `if` *spezielle Formen* (im Gegensatz zu Prozeduren).

Beispiel 2.5.1. Klasse 10: spezielle Formen Man beschreibe den Unterschied zwischen `if` und der wie folgt definierten Prozedur `new-if`:

```
(define (new-if test consequent alternative)
  (cond (test consequent)
        (else alternative)))
```

Zum Rechnen mit booleschen Werten gibt es die Prozedur `not` sowie die speziellen Formen

`and`, `or`.

Zum Beispiel (`or b1` ... `bk`) wertet der Reihe nach `b1` ... `bk` aus und gibt den Wert des ersten `bi` zurück, das als wahr gilt, also verschieden von `#f` ist.

Beispiel 2.5.2. Klasse 10: boolesche Kombinationen. Werte folgende Scheme-Ausdrücke aus: `(= 4 5)`, `(< 7 9)`, `(and (> 4 3) (>= 10 100))`, `(or (> 4 3) (= 10 1000))`, `(and #f #t)`, `(or #f #t)`, `(not #f)`.

Beispiel 2.5.3. Klasse 10: Bedingungen mit Variablen. Was sind die Resultate von

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

für $x = 2$, $x = 4$, $x = 7/2$. Gehe hier zum Beispiel wie folgt vor:

```
(define (bed x) (and (< 4 x) (> x 3)))
(bed 2)
(bed 4)
(bed 7/2)
```

Weitere Beispiele:

```
; ist5 : Zahl -> boolescher Wert
; Stellt fest, ob n gleich 5 ist
(define (ist5 n) (= n 5))
```

Das Programm liefert für $n = 5$ den Wert `#t`, sonst den Wert `#f`. Im Kontrakt erscheint hier der neue Begriff *boolescher Wert* (auch *Wahrheitswert* genannt) .

Zweites Beispiel:

```
; ist-zwischen-19-35 : Zahl -> boolescher Wert
; Stellt fest, ob n zwischen 19 und 35 liegt

(define (ist-zwischen-19-35 n) (and (< 19 n) (< n 35)))
```

Das Programm ist gut zu verstehen, wenn man sagt, daß es auf der Zahlengerade das Intervall $]19;35[$ beschreibt.

Beispiel 2.5.4. Klasse 8: Intervalle I. Beschreibe folgende Intervalle auf der Zahlengerade durch Scheme-Programme: $]3;7]$, $[5;11]$, $[4;7[$, $]1;3[$, $]7;11]$, $[2;5]$.

Beispiel 2.5.5. Klasse 8: Intervalle II. „Übersetze" folgende vier Scheme-Programme in Intervalle auf der Zahlengeraden. Formuliere auch die Statements für Kontrakt und Programmzweck.

```
(define (intervall1 x) (and (< -3 x) (< x 0)))
(define (intervall2 x) (or (< x 1) (> x 2)))
(define (intervall3 x) (not (and (<= 1 x) (<= x 5))))

(define (intervall4 x)
  (or (and (<= 1 x) (<= x 5))
      (and (<= 9 x) (<= x 12))))
```

Prüfe die Ergebnisse sowohl mit dem Rechner als auch mit der Hand.

Mathematische Gleichungen in einer Variablen sind Forderungen an eine unbekannte Zahl. Betrachtet man etwa $x^2 + 2x + 1 = 0$, so ist diese Forderung für $x = -1$ erfüllt, für $x = 1$ aber nicht, wie man durch Einsetzen leicht sieht. Eine Zahl, welche die Gleichung erfüllt, heißt Lösung der Gleichung. Wir können zum Testen, ob x Lösung ist, leicht ein kleines Scheme-Programm schreiben:

```
; gleichung : Zahl -> boolescher Wert
; Testet, ob eine Zahl x die Gleichung loest .
(define (gleichung x)
  (= (+ (* x x) (* 2 x) 1)
     0))
```

`(gleichung -1)` liefert dann `#t`, `(gleichung 4)` hingegen `#f` als Ergebnis.

Beispiel 2.5.6. Klasse 8: Gleichungen. Übersetze folgende Gleichungen in Scheme Programme und teste, ob 10, 12 oder 14 Lösungen dieser Gleichungen sind. $4n + 2 = 62$, $2n^2 = 102$, $4n^2 + 6n + 2 = 462$.

Beispiel 2.5.7. Klasse 8: Mausklick. Der Manager der Firma Ywindow&Co benötigt ein Programm, das feststellt, ob ein Mausklick innerhalb eines Rechteckes erfolgt oder nicht. Der Mausklick wird durch zwei Zahlen x, y repräsentiert. Vorgegeben sind Zahlen `nord`, `sued`, `west` und `ost`, die die Ränder des Rechtecks festlegen.

```
(define nord 100)
(define sued 350)
(define west 30)
(define ost 220)
```

Schreiben Sie das Programm `innen`, das die Koordinaten x, y des Mausklicks einliest und den Wert `#f` liefert, falls der Klick außerhalb des Rechtecks erfolgt.

Ratespiele

Beim Arbeiten mit Schülern ist es wichtig, auch den Spaß, Spiele und Bilder nicht zu kurz kommen zu lassen. DrScheme bietet hierzu einige Hilfen; nebenbei lernt der Schüler, mit bedingten Ausdrücken umzugehen und hat die Befriedigung, die wesentliche Prozedur eines Ratespiels (wie z.B. Mastermind) selber geschrieben zu haben (natürlich kann man dabei auch Fehler machen, die sich dann unmittelbar beim Spiel feststellen lassen).

Systematisch greifen wir hier ein wenig vor und verwenden *Symbole*, die als Daten erst im nächsten Kapitel eingeführt werden. Ein Symbol (zum Beispiel `zu-klein`) ist für Scheme ein unzerlegbares Datenobjekt. Ein Versuch der Auswertung führt zu einer Fehlermeldung:

```
zu-klein ==> Error: variable zu-klein is not bound.
```

Eine *Quotierung* ist notwendig, um diesen Fehler zu vermeiden:

```
'zu-klein ==> zu-klein
```

Beispiel 2.5.8. Entwickle eine Prozedur `check-guess`, welche prüft, ob die geratene Zahl mit dem Ziel übereinstimmt. Das Programm liest die zwei Zahlen `rate` und `ziel` ein und teilt dem Spieler mit, ob die geratene Zahl stimmt, zu klein oder zu groß ist. Die Prozedur implementiert den wesentlichen Teil eines Zahlenratespiels. Die Rechenmaschine ermittelt eine Zahl zwischen 0 und 99999. Der Spieler soll möglichst schnell die gesuchte Zahl ermitteln. Der Rest des Spieles ist in `guess-lib.ss` implementiert. Um das Spiel zu spielen, setzen Sie die library auf `guess-lib` und evaluieren dann den Ausdruck `(repl check-guess)`, nachdem Sie die Prozedur vollständig geschrieben haben.

Beispiel 2.5.9. Entwickeln Sie die Prozedur `check-guess3`, die drei Ziffern und eine Zahl, genannt `ziel` einliest. Die erste Ziffer entspricht der Einer-, die zweite Ziffer der Zehner- und die dritte Ziffer die Hunderterstelle. Abhängig von der Eingabe liefert die Prozedur die Ergebnise `zu-klein`, `phantastisch` und `zu-gross`. Testen Sie zunächst das „Prográmmchen". Um das Spiel zu starten, evaluieren Sie dann den Ausdruck `(repl3 check-guess3)`.

Beispiel 2.5.10. Entwickeln Sie die Prozedur `check-guess`. Sie liest vier Farben ein, die ersten beiden sind geratene Farben, die letzten beiden sind Ziele. Das Programm liefert folgende Antworten:

`'perfekt`, wenn beide geratene Farben mit beiden Zielen übereinstimmen.

`'eine-farbe-korrekt`, wenn eine geratene Farbe einschließlich ihrer Postiton mit dem Ziel übereinstimmt.

`'die-farbe-erscheint`, wenn eine der geratenen Farben bei den Zielen erscheint.

`'alles-falsch` in allen übrigen Fällen

Testen Sie das Programm gründlich und spielen das Farbenratespiel `(mastermind)`, indem Sie die library auf `mastermind.ss` setzen und dann den den Ausdruck `(repl check-guess)` evaluieren. (Die Bibliothek unterstützt das Programm `repl`). Mögliche Lösung:

```
(define (check-guess r1 r2 z1 z2)
  (cond ((and (equal? r1 z1) (equal? r2 z2)) 'perfekt)
        ((or (equal? r1 z1) (equal? r2 z2)) 'eine-farbe-korrekt)
```

```
((or (or (equal? r1 z1) (equal? r1 z2))
     (or (equal? r2 z1) (equal? r2 z2))) 'die-farbe-erscheint)
(else 'alles-falsch)))
```

Beispiel: `(check-guess 'red 'blue 'green 'brown)` `(repl check-guess)`.

3. Rekursion

Rekursion ist ein wichtiges Konzept, das auch im Gymnasium schon ab Klasse 9 (bei der HERON-Formel zur Berechnung von $\sqrt{2}$) eine Rolle spielt.

Fakultätsfunktion

Als Beispiel für primitive Rekursion (auch lineare Rekursion genannt) betrachten wir die Definition der Fakultätsfunktion.

$$
\begin{aligned}
0! &= 1, \\
(n+1)! &= (n+1) \cdot n!.
\end{aligned}
$$

Eine direkte Übertragung dieser Definition in die Sprache von Scheme liefert

```
(define (factorial n)
  (if (= 0 n)
      1
      (* n (factorial (- n 1)))))
```

Eine „iterative“ Form dieser Definition ist

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

In `product` wird also das Ergebnis angesammelt. Man nennt deshalb ein solches zusätzliches Argument einen *Akkumulator*.

Die zweite Form ist „endrekursiv“, d.h. die Prozedur macht nach der Rückkehr aus dem rekursiven Aufruf keine weiteren Berechnungen mehr. Dies impliziert, daß der Wert der Funktion in diesem Falle gleich dem Wert des rekursiven Aufrufs ist. Endrekursive Funktionen lassen sich in Scheme besonders effizient berechnen.

3.1 Primitive Rekursion und Iteration

Kleinstes gemeinsames Vielfaches und größter gemeinsamer Teiler

Die folgenden Beispiele zeigen deutlich, wie man bereits in der 5. Klasse das Ineinanderschachteln von Funktionen gewinnbringend einsetzen kann. Die Prozedur kgV3 ist hier bereits relativ komplex. (Man verwendet die Funktionen ggT und kgV.)

Beispiel 3.1.1. Klasse 5: größter gemeinsamer Teiler. Bestimmen Sie den ggT zweier Zahlen a, b mit Hilfe des euklidischen Algorithmus. a, b sind zwei natürliche Zahlen.

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

(ggT 45 40)
(ggT 40 50)
```

Bestimmen Sie den ggT dreier Zahlen mit Hilfe der obigen Funktion. a, b, c sind drei natürliche Zahlen.

```
(define (ggT3 a b c)
  (ggT (ggT a b) c))

(ggT3 12 18 24)

(ggT3 45987 8756 987659)
```

Beispiel 3.1.2. Klasse 5: kleinstes gemeinsames Vielfaches. Bestimme das kgV zweier Zahlen. a, b sind zwei natürliche Zahlen.

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

(kgV 6 9)
```

Bestimme das kgV dreier natürlicher Zahlen mit Hilfe der obigen Funktion. a, b, c sind drei natürliche Zahlen.

```
(define (kgV3 a b c)
  (kgV (kgV a b) c))

(kgV3 4 12 14)
```

Quadratwurzeln nach der Newton-Methode

Als besonders interessantes Beispiel für ein rekursives Verfahren bietet sich die Berechnung von $\sqrt{2}$ mit der Heron-Formel an. Es ist der erste Algorithmus im Pflichtprogramm, und für Klasse 9 vorgesehen. Wir wollen diesen Algorithmus hier in etwas allgemeinerer Form entwickeln, und behandeln die Newton-Methode zur Berechnung von Quadratwurzeln.

In der Analysis kann man schon vor der Konstruktion der reellen aus den rationalen Zahlen Quadratwurzeln approximieren. Es seien $a > 0$ und $x_0 > 0$ gegeben. Die Folge x_n sei rekursiv definiert durch

$$x_{n+1} := \frac{1}{2}\Big(x_n + \frac{a}{x_n}\Big).$$

Dann gilt

$$x_n > 0 \qquad \text{für alle } n \in \mathbb{N}, \tag{3.1}$$

$$a \le x_{n+1}^2 \qquad \text{für alle } n \in \mathbb{N}, \tag{3.2}$$

$$x_{n+2} \le x_{n+1} \qquad \text{für alle } n \in \mathbb{N}, \tag{3.3}$$

und mit $y_n := \frac{a}{x_n}$

$$\begin{aligned}
y_{n+1}^2 &\leq a && \text{für alle } n \in \mathbb{N}, && (3.4)\\
y_{n+1} &\leq y_{n+2} && \text{für alle } n \in \mathbb{N}, && (3.5)\\
y_{n+1} &\leq x_{m+1} && \text{für alle } n, m \in \mathbb{N}, && (3.6)\\
x_{n+1} - y_{n+1} &\leq \frac{1}{2^n}(x_1 - y_1) && \text{für alle } n, m \in \mathbb{N}. && (3.7)
\end{aligned}$$

Beweis. Wir führen den Beweis (wie in [6]) in mehreren Schritten.

(3.1) Durch Induktion über n zeigt man leicht $x_n > 0$ für alle $n \in \mathbb{N}$.

(3.2) Es gilt $x_{n+1}^2 \geq a$ für alle n, denn

$$\begin{aligned}
x_{n+1}^2 - a &= \frac{1}{4}\Big(x_n^2 + 2a + \frac{a^2}{x_n^2}\Big) - a\\
&= \frac{1}{4}\Big(x_n^2 - 2a + \frac{a^2}{x_n^2}\Big)\\
&= \frac{1}{4}\Big(x_n - \frac{a}{x_n}\Big)^2\\
&\geq 0.
\end{aligned}$$

(3.3) Es gilt $x_{n+2} \leq x_{n+1}$ für alle n, denn

$$\begin{aligned}
x_{n+1} - x_{n+2} &= x_{n+1} - \frac{1}{2}\Big(x_{n+1} + \frac{a}{x_{n+1}}\Big)\\
&= \frac{1}{2x_{n+1}}\Big(x_{n+1}^2 - a\Big)\\
&\geq 0.
\end{aligned}$$

(3.4) Mit $y_n := \frac{a}{x_n}$ gilt $y_{n+1}^2 \leq a$ für alle n, denn nach (3.2) ist $\frac{1}{x_{n+1}^2} \leq \frac{1}{a}$, also auch

$$y_{n+1}^2 = \frac{a^2}{x_{n+1}^2} \leq \frac{a^2}{a} = a.$$

(3.5) Aus (3.3) folgt $y_{n+1} \leq y_{n+2}$ für alle n.

(3.6) Es gilt $y_{n+1} \leq x_{m+1}$ für alle $n, m \in \mathbb{N}$. Denn – etwa für $n \geq m$ – hat man $y_{n+1} \leq x_{n+1}$ (dies folgt aus (3.2) durch Multiplikation mit $\frac{1}{x_{n+1}}$), und $x_{n+1} \leq x_{m+1}$ nach (3.3).

(3.7) Es gilt

$$x_{n+1} - y_{n+1} \leq \frac{1}{2^n}(x_1 - y_1).$$

Wir zeigen dies durch Induktion über n. Induktionsanfang: Für $n = 0$ sind beide Seiten gleich. Induktionsschritt:

$$\begin{aligned}
x_{n+2} - y_{n+2} &\leq x_{n+2} - y_{n+1}\\
&= \frac{1}{2}(x_{n+1} + y_{n+1}) - y_{n+1}\\
&= \frac{1}{2}(x_{n+1} - y_{n+1})\\
&\leq \frac{1}{2^{n+1}}(x_1 - y_1) && \text{nach Induktionsvoraussetzung.}
\end{aligned}$$

□

Um dieses Newton-Verfahren zu implementieren, definieren wir zunächst

```
(define (average x y)
  (/ (+ x y) 2))
```

Das im Satz formulierte Iterationsverfahren wird nun wie folgt implementiert.

```
(define (mysqrt x)
  (sqrt-iter 1 x))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) .001))

(define (improve guess x)
  (average guess (/ x guess)))
```

Einige Beispiele:

```
(mysqrt 2) ==> 577/408
(round (* (expt 10 16) (mysqrt 2))) ==> 14142156862745098

(mysqrt 9) ==> 65537/21845
(round (* (expt 10 16) (mysqrt 9))) ==> 30000915541313802
```

3.2 Baumrekursion

Eine weitere häufig auftretende Form der Rekursion ist die sogenannte ungeschachtelte Rekursion (auch Baumrekursion genannt). Hier dürfen mehrere ungeschachtelte Aufrufe der zu definierenden Funktion vorkommen. Ein typisches Beispiel dafür ist die Folge der FIBONACCI-Zahlen:

$$\mathsf{Fib}(n) := \begin{cases} 0 & \text{falls } n = 0, \\ 1 & \text{falls } n = 1, \\ \mathsf{Fib}(n-1) + \mathsf{Fib}(n-2) & \text{sonst.} \end{cases}$$

Hieraus erhalten wir unmittelbar die folgende Definition in Scheme:

```
(define (fib n)
  (cond ((= 0 n) 0)
        ((= 1 n) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Man beachte jedoch, daß eine hiernach durchgeführte Berechnung sehr ineffizient ist, da viele Mehrfachberechnungen durchgeführt werden. Deshalb ist die folgende iterative Version vorzuziehen.

```
(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= 0 count)
      b
      (fib-iter (+ a b) a (- count 1))))
```

Ein weiteres Beispiel für eine ungeschachtelte Rekursion liefern die Binomialkoeffizienten. Hier tritt als zusätzliche Besonderheit auf, daß die Parameterwerte verändert werden. Für natürliche Zahlen $n \geq 1$ und k setzen wir

$$\binom{n}{k} := \prod_{j=1}^{k} \frac{n-j+1}{j} = \frac{n(n-1) \cdot \ldots \cdot (n-k+1)}{1 \cdot 2 \cdot \ldots \cdot k}.$$

Die Zahlen $\binom{n}{k}$ heißen *Binomialkoeffizienten*.

Aus der Definition folgt unmittelbar

$$\binom{n}{k} = 0 \qquad \text{für } k > n, \tag{3.8}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \binom{n}{n-k} \qquad \text{für } 0 \leq k \leq n. \tag{3.9}$$

Wir zeigen, daß für $1 \leq k \leq n$ gilt

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}. \tag{3.10}$$

Beweis. Für $k = n$ ist dies offenbar richtig. Für $1 \leq k \leq n-1$ hat man

$$\begin{aligned}\binom{n-1}{k-1} + \binom{n-1}{k} &= \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-k-1)!} \\ &= \frac{k(n-1)! + (n-k)(n-1)!}{k!(n-k)!} \\ &= \frac{n!}{k!(n-k)!} \\ &= \binom{n}{k}.\end{aligned}$$

□

Zur Implementierung verwendet man entweder die Formeln (3.8) und (3.9) mit der Fakultätsfunktion

```
(define (binom n k)
  (if (< n k)
      0
      (quotient (factorial n) (* (factorial k) (factorial (- n k))))))
```

oder aber die Rekursionsformel:

```
(define (binom-rek n k)
  (cond ((zero? k) 1)
        ((> k n) 0)
        ((= 1 n) 1)
        (else (+ (binom-rek (- n 1) (- k 1))
                 (binom-rek (- n 1) k)))))
```

Ein Problem bei dieser rekursiven Implementierung sind wieder die durch den zweifachen Aufruf verursachten Mehrfachberechnungen.

3.3 Höherstufige Prozeduren

Betrachten wir die folgenden drei Prozeduren. Die erste berechnet die Summe aller ganzen Zahlen zwischen `a` und `b`.

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

Die zweite berechnet die Summe aller Quadrate der ganzen Zahlen zwischen `a` und `b`.

```
(define (sum-sqares a b)
  (if (> a b)
      0
      (+ (square a) (sum-squares (+ a 1) b))))
```

Die dritte berechnet gewisse Partialsummen der folgenden (sehr langsam) gegen $\frac{\pi}{8}$ konvergenten Reihe

$$\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \cdots,$$

und zwar bei Eingabe von $a = 4i - 3$ und $b = 4j - 3$ (mit $i \leq j$) die Partialsumme vom i-ten bis zum j-ten Glied (einschließlich).

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

Man erhält zum Beispiel

```
(* 8.0 (pi-sum 1 100)) ==> 3.1215946525910105
(* 8.0 (pi-sum 1 1500)) ==> 3.1402593208490512
```

Alle drei Definitionen folgen offenbar demselben Schema. Es liegt deshalb nahe, diese gemeinsame Form zu abstrahieren und einen allgemeinen Summationsoperator wie folgt zu definieren.

```
(define (sum summand-fct next-fct a b)
  (if (> a b)
      0
      (+ (summand-fct a)
         (sum summand-fct next-fct (next-fct a) b))))
```

Man beachte, daß hier Prozeduren als Argumente übergeben werden. In diesem Sinn ist also der definierte Summenoperator höherstufig.

Beispiel 3.3.1. Klasse 11: Integration durch fortgesetzte Summation. Funktionen übergeben.

3.4 Blockstrukturen

Mit „let“ vereinbart man lokale Variable. Scheme erhält so eine Blockstruktur, vergleichbar mit dem “begin – end” von Pascal. Es hat die folgende Form:

```
(let ((<Variable0> <Wert0>)
      (<Variable1> <Wert1>)
         ...
      (<VariableN> <WertN>))
   <Scheme Ausdruecke>)
```

Die genannten Variablen werden neu erzeugt und mit den gegebenen Werten initialisiert. In den folgenden Schemeausdrücken bis zur schließenden Klammer des „let“ können die neuen Variablen dann wie gewöhnlich verwendet werden.

Ist zum Beispiel `point` ein Vektor mit `x`- und `y`-Wert, so kann man schreiben:

```
(let ((x (vector-ref point 0))
      (y (vector-ref point 1)))
   (display "Polar: ")
   (display (sqrt (+ (* x x) (* y y))))
   (dispay ", ")
   (display (atan (/ x y)))
   (newline))
```

3.5 `do`-Schleifen

Oft ist es ein Algorithmus nicht in rekursiver Form gegeben, sondern in einer sogenannten „Schleifen"-Form. Dann bietet es sich an, ihn auch in dieser Weise zu implementieren. In Scheme ist das entsprechende Konstrukt das `do`, das sich in der Form an das `let`-Konstrukt anlehnt.

Einfache Schleifen schreibt man mit dem `do`-Befehl. Er hat die Form

```
(do ((<Laufvariable> <Anfangswert> <Nachfolger>))
    (<Test>)
  <Scheme Ausdruecke>)
```

Abgesehen von dem `Test` und den `Nachfolger` Ausdrücken erinnert das „do" sehr an das „let"; in der Tat sind die Unterschiede sehr gering. Zuerst werden die genannten Variablen erzeugt und mit den gegebenen Anfangswerten initialisiert (ähnlich wie das „let" kann das „do" natürlich auch mehrere Laufvariable verwalten).

Die Auswertung der Schleife beginnt mit dem Test. Ist dieser Test wahr, so wird die Schleife beendet. Andernfalls werden wie beim „let" die folgenden Schemeausdrücke ausgewertet. Neu ist, daß im Anschluß daran die `Nachfolger` Ausdrücke benutzt werden, um neue Werte für die Variablen zu berechnen, und danach die Schleife von vorne beginnt.

Das folgende Beispiel druckt die Quadratzahlen bis 100.

```
(do ((i 0 (+ i 1)))
    ((> i 10))
  (display (* i i))
  (newline) )
```

Das Schachbrett-Problem

Nach einer berühmten Legende soll der Erfinder des Schachspiels als Lohn für seine Erfindung auf dem ersten Feld des Schachbretts ein Reiskorn und auf jedem folgenden Feld doppelt so viele Körner wie auf dem vorhergehenden verlangt haben.

```
(do ((r 1 (* r 2))
     (f 1 (+ f 1))
     (sum 1 (+ sum (* r 2))))
    ((> f 30))
  (display f) (display #\tab)
  (display r) (display #\tab)
  (display sum) (newline))
```

```
1       1       1
2       2       3
3       4       7
4       8       15
5       16      31
6       32      63
7       64      127
8       128     255
9       256     511
10      512     1023
11      1024    2047
12      2048    4095
13      4096    8191
14      8192    16383
15      16384   32767
16      32768   65535
17      65536   131071
```

18	131072	262143
19	262144	524287
20	524288	1048575
21	1048576	2097151
22	2097152	4194303
23	4194304	8388607
24	8388608	16777215
25	16777216	33554431
26	33554432	67108863
27	67108864	134217727
28	134217728	268435455
29	268435456	536870911
30	536870912	1073741823

Faktorisierung durch Teilen

(nach KNUTH, The Art of Computer Programming (Band 2) [10, S. 380]). Der Algorithmus ist so wörtlich wie möglich übernommen. Allerdings ist die Folge der d_k primitiver als nötig, aber der Algorithmus ist sowieso nur für kleine N effizient.

```
(define (factor_by_division X)
  (define (dk+1 dk)
   (if (= dk 2) 3 (+ dk 2)))
  (do ((p ())                          A1. [Initialize.] Set t ← 0,
       (dk 2)                              k ← 0,
       (n X))                              n ← X.
      ((= n 1)                         A2. [n = 1?] If n = 1,
       p)                                  the algorithm terminates.
    (let ((q (quotient n dk))          A3. [Divide.] Set q ← ⌊n/d_k⌋,
          (r (modulo n dk)))               r ← n mod d_k.
      (if (zero? r)                    A4. [Zero remainder.] If r ≠ 0, go to A6.
          (begin                       A5. [Factor found.]
            (set! p (cons dk p))           Increase t by 1, and set p_t ← d_k,
            (set! n q))                    n ← q. Return to A2.
          (if (> q dk)                 A6. [Low quotient?] If q > d_k,
              (set! dk (dk+1 dk))          increase k by 1 and return to A3.
              (begin                   A7. [n is prime.]
                (set! p (cons n p))        Increase t by 1, set p_t ← d_k,
                (set! n 1)))))))           and terminate the algorithm
```

Man beachte:

- Die Scheme Implementierung verwendet eine Funktion `dk+1` zur Konstruktion einer Folge von möglichen Teilern d_k. Da aber zu jedem Zeitpunkt nur ein einzelner Wert von d_k benötigt wird, wird der Index k nur implizit im Wert von d_k gespeichert. Anstelle der Vergrößerung von k um 1 (im Schritt A6) wird die Funktion `dk+1` aufgerufen.
- Die Folge der Primfaktoren $p_1, \ldots, p_t$ ist als Liste `p` implementiert, die die p_i in umgekehrter Reihenfolge enthält. Wieder ist der Index t nicht explizit angegeben. Anstelle der Vergrößerung von t um 1 (im Schritt A7) fügen wir mit `cons` ein neues Element an die Liste an.
- Die Scheme Implementierung verwendet aus Gründen der Einfachheit und Klarheit nicht das Scheme-Äquivalent „named let" eines „go to", sondern verwendet die strukturierten `do` und `if` Konstrukte. Konsequenterweise gehen wir deshalb im Schritt A6 nicht zu Schritt A3 zurück (wie vom Algorithmus verlangt), sondern stattdessen zu Schritt A2. Die „Strafe" dafür ist ein zusätzlicher Test, ob $n = 1$. Die Terminierung des Algorithmus in Schritt A7 ist ebenfalls weniger elegant realisiert: wir setzen $n \leftarrow 1$ und gehen zu Schritt A2 zurück; damit geben wir dem strukturierten Ausgang aus der `do`-Schleife den Vorzug vor einem unstrukturierten „named let".

Zum Beispiel erhält man

```
(factor-by-division 47100194747174954)
==> (999863 281 53 47 23 19 11 7 2)
```

Die innere Schleife läuft bis $d_k = 1001$ durch alle ungeraden Zahlen.

```
(factor-by-division (* 999863 999983))
```

dauert „ewig". Die innere Schleife läuft bis $d_k = 999863$ durch alle ungeraden Zahlen also etwa 1000 mal länger als das vorige Problem. Zusätzliche kleinere Faktoren verändern die Laufzeit nicht (kaum). Die Laufzeit ist proportional dem Maximum des zweitgrößten Primfaktors und der Wurzel des größten Primfaktors.

Durch Änderung der Folge der d_k kann die Laufzeit um ca. 40% gesenkt werden (Elimination der Vielfachen von 3 und 5). Dann dauert es nur noch 600 mal so lange und ist immer noch unpraktikabel. Für kleinere Zahlen ist das Verfahren auch mit der einfachen Methode überraschend schnell.

Für den Fall, daß diese Änderungen unakzeptabel erscheinen, und auch zum Nachweis, daß sogar ein Assembler-ähnlicher Programmierstil in Scheme möglich ist, hier noch die folgende Implementierung desselben Algorithmus:

```
(define (factor_by_division X)
  (define (dk+1 dk)
   (if (= dk 2) 3 (+ dk 2)))
  (define p ())
  (define dk 2)
  (define n X)
  (let A2 ()
    (if (= n 1) p
  (let A3 ()
    (define q (quotient n dk))
    (define r (modulo n dk))
    (if (zero? r)
      (begin
        (set! p (cons dk p))
        (set! n q)
        (A2))
      (if (> q dk)
        (begin
          (set! dk (dk+1 dk))
          (A3))
        (begin
          (set! p (cons n p))
          p)))))))
```

A1. [Initialize.]
Set $t \leftarrow 0$,
$k \leftarrow 0$,
$n \leftarrow X$.
A2. [$n = 1$?]
If $n = 1$ the algorithm terminates.
A3. [Divide.]
Set $q \leftarrow \lfloor n/d_k \rfloor$,
$r \leftarrow n \bmod d_k$.
A4. [Zero remainder.] If $r \neq 0$, go to A6.
A5. [Factor found.]
Increase t by 1, and set $p_t \leftarrow d_k$,
$n \leftarrow q$.
Return to A2.
A6. [Low quotient?] If $q > d_k$,
increase k by 1
and return to A3.
A7. [n is prime.]
Increase t by 1, set $p_t \leftarrow d_k$,
and terminate the algorithm

4. Weitere Daten: Paare und Listen, Symbole, Zeichenreihen, Vektoren

4.1 Paare und Listen

Paare

Wenn wir etwa rationale Zahlen aus Zähler und Nenner zusammensetzen wollen, benötigen wir offenbar einen Weg, aus zwei Datenobjekten – hier Zähler und Nenner – ein neues zusammenzusetzen. Eine solche Paarbildung ist die Grundform zur Bildung zusammengesetzter Daten in LISP. Zur Bildung eines Paares aus zwei Argumenten verwenden wir die primitive Prozedur `cons`. Aus einem Paar kann man die erste und die zweite Komponente ablesen mittels der primitive Prozeduren `car` und `cdr`. Die Bezeichnungen `car` und `cdr` gehen zurück auf die ursprüngliche Implementierung von LISP auf einer IBM 704. `car` steht für "contents of address register" und `cdr` steht für "contents of decrement register".

```
(define x (cons 1 2)) ==> x
(car x) ==> 1
(cdr x) ==> 2
```

Man beachte, daß ein Paar ein gewöhnliches Datenobjekt ist, das wie jedes andere mit einem Namen versehen und manipuliert werden kann.

```
(define x (cons 1 2)) ==> x
(define y (cons 3 4)) ==> y
(define z (cons x y)) ==> z
(car (car z)) ==> 1
(car (cdr z)) ==> 3
```

In LISP verwendet man die Paarbildung als universellen Baustein zur Bildung komplexer Datenobjekte.

Beispiel 4.1.1. Klasse 6: rationale Zahlen. Nehmen wir zunächst an, wir hätten schon eine Implementierung der rationalen Zahlen durchgeführt, und zwar durch Angabe eines *Konstruktors* `make-rat`, der aus Zähler und Nenner eine rationale Zahl konstruiert, und zweier *Selektoren* `zaehler` und `nenner`, die aus einer rationalen Zahl den Zähler bzw. den Nenner ablesen. Ohne diese Implementierung genauer zu kennen, können wir Addition, Subtraktion, Multiplikation, Division und Gleichheit rationaler Zahlen definieren:

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

(define (-rat x y)
  (make-rat (- (* (zaehler x) (nenner y))
               (* (nenner x) (zaehler y)))
            (* (nenner x) (nenner y))))

(define (*rat x y)
  (make-rat (* (zaehler x) (zaehler y))
            (* (nenner x) (nenner y))))
```

```
(define (/rat x y)
  (make-rat (* (zaehler x) (nenner y))
            (* (nenner x) (zaehler y))))

(define (=rat x y)
  (= (* (zaehler x) (nenner y))
     (* (zaehler y) (nenner x))))
```

Wir wollen jetzt den Konstruktor `make-rat` und die beiden Selektoren `zaehler` und `nenner` implementieren. Dazu benötigen wir offenbar einen Weg, aus zwei Datenobjekten – hier Zähler und Nenner – ein neues zusammenzusetzen. Wir verwenden dazu die primitive Prozedur `cons`:

```
(define (make-rat n d) (cons n d))
(define (zaehler x) (car x))
(define (nenner x) (cdr x))
```

Zum Ausdruck der rationalen Zahlen verwenden wir die folgende Prozedur.

```
(define (print-rat x)
  (newline)
  (display (zaehler x))
  (display "/")
  (display (nenner x)))
```

Dann erhält man zum Beispiel

```
(print-rat (make-rat 2 6)) ==> 2/6
```

Man beachte, daß wir in unserer Implementierung der Operationen auf rationalen Zahlen vorausgesetzt haben, daß wir den Konstruktor `make-rat` und die Selektoren `zaehler` und `nenner` zur Vefügung haben. Es war nicht nötig, diese Implementierung genauer zu kennen. Das einzige, das wir über den Konstruktor `make-rat` und die Selektoren `zaehler` und `nenner` wissen mußten, war, daß die Anwendung eines Selektors auf ein durch den Konstruktor gebildetes Objekt die entsprechende Komponente reproduziert.

Listen

Paare werden in SCHEME hauptsächlich zur Bildung von *Listen* verwendet. Listen werden dargestellt mittels iterierter Paarbildung, wobei das `cdr`-Feld des letzten Paares leer bleibt. Genauer definiert man Listen rekursiv durch die folgenden Klauseln.

1. Die leere Liste ist eine Liste.
2. Ist a ein Datenobjekt und ℓ eine Liste, so ist das Paar, dessen `car`-Feld das Datenobjekt a und dessen `cdr`-Feld die Liste ℓ enthält, eine Liste.

Die Datenobjekte in den `car`-Feldern der zur Listenbildung verwendeten Paare sind die *Elemente* der Liste. Man beachte, daß als neues Datenobjekt die *leere Liste* `()` benötigt wird. Die primitive Prozedur `null?` fragt ab, ob ein Datenobjekt die leere Liste ist. Für eine genauere Diskussion und eine Beschreibung der zur Listenbearbeitung zur Verfügung stehenden Prozeduren sei auf Abschnitt 6.3 in der Sprachdefinition [9] verwiesen.

Durch Schachtelung der Listenbildung kann man auch endlich verzweigte *Bäume* darstellen.

Insbesondere sind die folgenden Prozeduren wichtig:

`list?`	Test
`list`	
`length`	Länge
`append`	zusammenhängen
`reverse`	umkehren
`list-ref`	
`member`	

Wichtig ist weiterhin die Prozedur `apply`, die eine Prozedur auf eine Liste von Argumenten anwendet. Beispiel:

```
(apply + (list 3 4)) ==> 7
```

Ferner wird oft die (höherstufige) Prozeduren `map` verwendet. `map` nimmt eine Prozedur und eine Liste und wendet die Prozedur der Reihe nach auf alle Elemente der Liste an. Zum Beispiel ist die Erstellung von Wertetabellen mit `map` besondes einfach (s. Beispiel 4.1.2).

Wir wollen hier eine Kurze Einführung des λ-Operators `lambda` einschieben, da seine Verwendung für die Erstellung von Wertetabellen besonders geeignet ist. Scheme ist insbesondere deshalb eine funktionale Programmiersprache, weil in ihr die Funktionen selbst Datenobjekte sind und den anderen Datenobjekten vollkommen gleichgestellt sind. Man kann Funktionen speichern oder vergleichen und man kann neue Funktionen zur Laufzeit erzeugen. Dazu dient das Schlüsselwort `lambda`, das in seiner Namensgebung auf die Ursprünge von Scheme im mathematischen λ-Kalkül verweist.

Eine Funktion mit den Parametern `x1 x2 ... xN` erzeugt man mit

`(lambda (x1 x2 ... xN)` *`Scheme-Ausdrücke`*`)`

Ganz genau so wie bei der Definition mit `define` werden zur Laufzeit beim Aufruf der Funktion zuerst die Parameter `x1 x2 ... xN` an die aktuellen Parameterwerte gebunden und dann werden die *Scheme-Ausdrücke* der Reihe nach ausgewertet. Der letzte Ausdruck liefert den Rückgabewert der Funktion.

Funktionen, die man mit `lambda` erzeugt, werden in der Regel als Argumente an andere Funktionen übergeben oder in einer Variablen gespeichert.

Beispiel 4.1.2. Klasse 8: Wertetabellen.

```
(define a (list 1 2 3 4))  ==>  a
a                          ==>  (1 2 3 4)
(map square a)             ==>  (1 4 9 16)
```

Die Verwendung von `lambda` ist hier besonders bequem. Will man zum Beispiel eine Wertetabelle der Funktion $x^2 + 4$ im Intervall von -3 bis 3 erzeugen, so kann man wie folgt vorgehen.

```
(map (lambda (x) (+ (* x x) 4)) '(-3 -2 -1 0 1 2 3))  ==>  (13 8 5 4 5 8 13)
```

oder etwas allgemeiner

```
(define (intervall a b) ; a,b ganze Zahlen mit a<b
  (do ((i a (+ i 1))
       (res '() (cons i res)))
      ((< b i) (reverse res))))

(map (lambda (x) (+ (* x x) 4)) (intervall -3 3))  ==>  (13 8 5 4 5 8 13)
```

Beispiel 4.1.3. Klasse 8: Pascalsches Dreieck. Aufgrund von (3.10) kann man die Binomialkoeffizienten mittels des PASCAL*schen Dreiecks* berechnen:

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}
$$

Unter Verwendung dieser Darstellung kann man leicht ein Programm zur Berechnung der Binomialkoeffizienten schreiben:

```
(define (binom-coeff n k)
  (list-ref (pasc-layer n) k))

(define (pasc-layer n)
  (if (zero? n)
      '(1)
      (let ((l (pasc-layer (- n 1))))
        (map + (cons 0 l) (append l '(0))))))
```

Dieses Programm ist sehr kurz, aber längst nicht so schnell wie ein mit der Fakultät geschriebenes.

Beispiel 4.1.4. Klasse 12: Kombinatorik: Ziehen ohne Zurücklegen. Lösungsmethode: Funktionen, die Listen von Varianten produzieren. Aus solchen Listen kann man dann mittels `map` neue Listen von Varianten produzieren, und sie mit `apply` und `append` zu einer einzigen Liste zusammenfassen.

`(auswahl liste k)`. Auswahl von k Elementen aus einer Liste ($0 \leq k \leq$ `(length liste)`).
Rückgabe: Liste mit Lösungen.
Wenn $k = 0$: eine Lösung: `()`
Wenn $k =$ `(length liste)`: eine Lösung: `liste`.
Sonst: erstes Element wird ausgewählt oder nicht. Lösungen im ersten Fall:

`(cons (car liste) x)` für alle Lösungen `x` von `(auswahl (cdr liste) (- k 1))`.

Lösungen im zweiten Fall: `(auswahl (cdr liste) k)`.

```
(define (auswahl liste k)
  (if (= k 0)
      (list ())
      (if (= k (length liste))
          (list liste)
          (append
            (map (lambda (x) (cons (car liste) x))
                 (auswahl (cdr liste) (- k 1)))
            (auswahl (cdr liste) k)))))
```

`(binom n k)`. Auswahl von k Elementen aus n Elementen.
Rückgabe: Anzahl der Lösungen.
Wenn $k = 0$: eine Lösung: 1.
Wenn $k = n$: eine Lösung: 1.
Sonst: erstes Element wird ausgewählt oder nicht.
Lösungen im ersten Fall: `(binom (- n 1) (- k 1))`.
Lösungen im zweiten Fall: `(binom (- n 1) k)`.
Änderungen:

$$\begin{aligned} \texttt{auswahl} &\mapsto \texttt{binom}, \\ \texttt{liste} &\mapsto \texttt{n}, \\ \texttt{(cdr liste)} &\mapsto \texttt{(- n 1)}, \\ \texttt{(length liste)} &\mapsto \texttt{n}, \\ \texttt{append} &\mapsto \texttt{+}. \end{aligned}$$

```
(define (binom n k)
  (if (= k 0)
      1
      (if (= k n)
          1
```

```
          (+ (binom (- n 1) (- k 1))
             (binom (- n 1) k)))))
```

Beispiel: `(auswahl '(1 2 3 4 5 6 7 8) 4)`.

```
((1 2 3 4)
 (1 2 3 5)
 (1 2 3 6)
 (1 2 3 7)
 (1 2 3 8)
 (1 2 4 5)
 (1 2 4 6)
 (1 2 4 7)

 . . .

 (3 5 6 7)
 (3 5 6 8)
 (3 5 7 8)
 (3 6 7 8)
 (4 5 6 7)
 (4 5 6 8)
 (4 5 7 8)
 (4 6 7 8)
 (5 6 7 8))
```

Ferner erhält man

```
(binom 8 4) ==> 70
(binom (length '(1 2 3 4 5 6 7 8)) 4) ==> 70
(length (auswahl '(1 2 3 4 5 6 7 8) 4)) ==> 70
```

Beispiel 4.1.5. Klasse 12: Kombinatorik: Ziehen mit Zurücklegen

```
(define (ziehung liste k)
  (if (= k 0)                                     ; wenn k=0,
      (list ())                                   ; Loesung leere Liste
      (apply append                               ; sonst
             (map (lambda (element)               ; fuer alle Elemente
                    (map (lambda (rest)           ; fuer alle k-1 -Ziehungen
                           (cons element rest))   ; eine neue Loesung
                         (ziehung liste (- k 1))))
                  liste))))
```

Als ein weiteres Beispiel für die Verwendung von Listen betrachten wir *Polynome*. Ein Polynom $a_nX^n + \cdots + a_1X + a_0$ mit $a_n \neq 0$ sei als Liste $(a_0\ \ldots\ a_n)$ dargestellt (man beachte die Reihenfolge!). Das Nullpolynom wird durch die leere Liste () repräsentiert.

Beispiel 4.1.6. Klasse 11: Summe von Polynomen. Zu schreiben ist eine Funktion `poly+` von zwei Argumenten, die die Summe zweier Polynome berechnen soll:

```
(define (poly+ p q)    ; p und q seien (Darstellungen von) Polynome(n)
  (cond ((null? p) q)  ; Wenn p das Nullpolynom ist, ist q das Ergebnis
        ((null? q) p)  ; analog umgekehrt
        (else (let ((k (+ (car p) (car q)))
                       ; k ist die Summe der konstanten Glieder
                    (s (poly+ (cdr p) (cdr q))))
                       ; s ist die Summe der "verkuerzten" Polynome
                       ; [p/X] und [q/X] (dabei sei p = [p/X]*X + p(0))
```

```
                (if (and (null? s) (zero? k))
                    '()  ; wenn s und k null sind, ist das Nullpolynom
                         ; das Ergebnis
                    (cons k s)))))) ; sonst k + s*X
```

Anmerkung. Man kann `poly-` analog definieren (Subtraktion von Polynomen), es ist nur das Zeichen + in der Zeile `(else ... )` durch − zu ersetzen.

Beispiel 4.1.7. Klasse 11: Skalarmultiplikation von Polynomen. Zu schreiben ist eine Funktion `poly*skal` von zwei Argumenten, einem Polynom p und einer Zahl a, die das Ergebnis der Multiplikation von p mit a liefern soll.

```
(define (poly*skal p a)  ; p Polynom, a Zahl
  (if (zero? a)
      '()     ; ist a=0, so muss das Nullpolynom () herauskommen
      (map (lambda (x) (* a x)) p)))
            ; sonst ist jeder Koeffizient in p mit a zu multiplizieren
```

Beispiel 4.1.8. Klasse 11: Produkt von Polynomen. Zu schreiben ist eine Funktion `poly*` von zwei Polynom-Argumenten p und q, die das Ergebnis der Multiplikation von p und q liefert.

```
(define (poly* p q)  ; p und q Polynome
  (if (null? p)
      '()      ; 0*q=0
      (poly+ (poly*skal q (car p)) (cons 0 (poly* (cdr p) q)))))
         ; sonst p = p0 + X*p1, und p*q = p0*q + X*(p1*q)
```

Beispiel 4.1.9. Klasse 11: Division von Polynomen. (Polynomdivision braucht man in Klasse 11, zur Lösung von Gleichungen 3. Grades: eine Lösung wird geraten). Zu schreiben ist eine Funktion `polydivide`, die zu ihren Argumenten p und q (Polynome) Quotient a und Rest r berechnet (d.h. $p = a \cdot q + r$ und $r = 0$ oder $\deg(r) < \deg(q)$). (Diese Aufgabe ist etwas schwieriger. Es empfiehlt sich, hier die Polynome "von hinten", d.h. beginnend mit dem Leitkoeffizienten, zu bearbeiten. Deswegen werden die Polynome erst einmal umgedreht. Außerdem ist es für die Rechnung praktischer, wenn man die Polynome so normalisiert, daß q normiert ist, d.h. Leitkoeffizient 1 hat. Der Quotient ändert sich dadurch nicht; der Rest muß am Ende wieder mit dem Leitkoeffizienten von q multipliziert werden.)

Wir setzen eine Hilfsfunktion `polydivide1` voraus, die zwei Argumente, p und q, bekommt, die "umgedrehte" Polynome sind, wobei q normiert ist, und als Ergebnis die Liste $(a'\ r')$ aus dem "umgedrehten" Quotienten und dem "umgedrehten" Rest liefert. Für die Hauptfunktion erhalten wir dann:

```
(define (polydivide p q)  ; p und q Polynome
  (if (null? q)
      (begin (newline)
             (display "Division durch Null!")
             '(() ()))
           ; Wenn q=0 ist, Warnung ausgeben, zwei Nullpolynome als Wert
      (let* ((pr (reverse p)) ; pr=p umgedreht
             (qr (reverse q)) ; qr=q umgedreht
             (lkfq (car qr))  ; lkfq = Leitkoeffizient von q
             (l (polydivide1 (poly*skal pr (/ lkfq))
                             (poly*skal qr (/ lkfq)))))
                ; l wird an die Liste (a' r') gebunden
        (list (reverse (car l)) (poly*skal (reverse (cadr l)) lkfq)))))
           ; jetzt wird das Ergebnis (a r) zusammengebaut: a entsteht aus
           ; a' durch Umdrehen, r aus r' durch Umdrehen und Multiplikation
           ; mit q0.
```

Nun die Hilfsfunktion:

```
(define (polydivide1 p q)  ; p und q umgedrehte Polynome, q normiert
  (cond ((null? p) (list '() '())) ; p=0, dann a'=r'=0
        ((zero? (car p)) (polydivide1 (cdr p) q))
           ; fuehrende Null beseitigen
        ((< (length p) (length q)) (list '() p))
                   ; deg(p) < deg(q), dann a'=0, r'=p
        (else (let ((l (polydivide1
                         (polydivide2 (car p) (cdr p) (cdr q)) q)))
          ; polydivide2 subtrahiert das (car p)-fache von q "von vorn"
          ; von p, l ist die Liste (a'' r') aus dem Quotienten dieser
          ; Differenz mit q und dem Rest (der bereits der Rest
          ; von p mod q ist)
               (list (cons (car p) (car l)) (cadr l))))))
                  ; Es ist a'=(Leitkoeff. von p)*X**n + a'', n passend

(define (polydivide2 a pl ql)
          ; a Zahl, pl, ql Listen, length(pl)>=length(ql)
  (if (null? ql)
      pl     ; ql ist aufgebraucht, pl bleibt uebrig
      (cons (- (car pl) (* a (car ql)))
            (polydivide2 a (cdr pl) (cdr ql)))))
             ; sonst vorne Subtrahieren
             ; und an verarbeiteten Rest anhaengen
```

Beispiel 4.1.10. Klasse 11: GgT von Polynomen. Zu schreiben ist eine Funktion `polygcd`, die den größten gemeinsamen Teiler ihrer zwei Argumente p und q (Polynome) berechnet. Man kann hierzu den üblichen Algorithmus verwenden.

```
(define (polygcd p q)  ; p und q Polynome
  (if (null? q)
      p                                      ; q=0, dann p
      (polygcd q (cadr (polydivide p q)))))  ; sonst ggT(q, p mod q)
```

Als Ergänzung sollte man noch eine Funktion zur Ausgabe von Polynomen auf den Bildschirm zur verfügung haben:

```
(define (polywrite p) ; p Polynom
  (if (null? p)
      (display 0)  ; Nullpolynom gibt 0
      (do ((n (- (length p) 1) (- n 1)) ; n ist der aktuelle Exponent
           (l (reverse p) (cdr l)) ; l ist die Liste der Koeffizienten
                                   ; (absteigend)
           (flag #t #f))           ; Flag fuer Anfang (wegen Vorzeichen)
          ((null? l))     ; Koeffizienten abgearbeitet
          (if (not (zero? (car l))) ; Koeff. muss /= 0 sein
              (begin (polywrite1 (car l) flag n) ; gibt ein Glied aus
                     (if (not (zero? n)) ; X^0 wird weggelassen
                         (begin (display "X")
                                (if (not (= n 1)) ; X^1 wird X
                                    (begin (display "^")
                                           (display n)))))))))
  (newline))   ; Zeile abschliessen

(define (polywrite1 k flag n) ; k Zahl, flag boolesch, n natuerliche Zahl
  (if (not flag) (display " ")) ; wenn nicht am Anfang, ein Leerzeichen
  (if (or (not flag) (negative? k))
```

```
        (display (if (negative? k) "- " "+ "))) ; evtl. Vorzeichen ausgeben
    (if (or (not (= 1 (abs k))) (zero? n))
        (display (abs k))))     ; Betrag des Koeffizienten ausgeben
```

4.2 Symbole und die Notwendigkeit der Quotierung

Man kann einem symbolischen Namen einen Wert zuordnen mittels

```
(define <Symbol> <Wert>)
```

Die Zuordnung eines symbolischen Namens zu einer Funktion bewerkstelligt man mit

```
(define (<Funktionsname> <Parameternamen>) <Werte>)
```

Beim Funktionsaufruf werden die Parameter wie mit einem `define` den aktuellen Werten im Funktionsaufruf zugeordnet. *Werte* steht für einen oder mehrere Schemeausdrücke, die die Parameternamen enthalten dürfen. Sie werden nun der Reihe nach ausgewertet und der letzte berechnete Wert wird dann der Rückgabewert der Funktion.

Neben den Buchstaben sind auch die meisten Sonderzeichen zur Bildung von Symbolen zugelassen. So ist etwa

```
(define (++ x) (+ x 1))
```

eine gültige Definition der Increment-Funktion `++`.

Bisher haben wir alle unsere Datenobjekte letzten Endes aus Zahlen konstruiert. Wir wollen uns jetzt die Möglichkeit verschaffen, auch Symbole als Datenobjekte zu verwenden. Beispiele:

```
(a b c d)
((a 1) (b 7) (c 3))
```

Listen, die auch Symbole enthalten, haben eine Form, wie sie auch bisher schon häufig vorgekommen ist.

```
(* (+ 1 2) (+ x 7))

(define (factorial n) (if (= 0 n) 1 (* n (factorial (- n 1)))))
```

Um mit Symbolen umgehen zu können, brauchen wir die Möglichkeit der *Quotierung*. Wenn wir zum Beispiel die Liste `(a b)` bilden wollen, können wir nicht einfach `(list a b)` auswerten, da dann der Interpreter nach Werten von `a` und `b` suchen und nicht die Symbole selbst nehmen würde. Dieses Phänomen ist aus natürlichen Sprachen gut bekannt. Wir verwenden deshalb den Quotierungoperator `quote` und schreiben `(quote a)`, wenn wir `a` quotieren wollen. Für `(quote a)` kann man kürzer auch `'a` schreiben.

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

Auch zusammengesetzte Objekte kann man quotieren.

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

4.3 Zeichen und Zeichenreihen

Einzelne Zeichen schreibt man als `#\`*`Zeichen`* oder `#\`*`Zeichenname`*. Zum Beispiel ist `#\a` ein kleines „a“ und `#\newline` das Zeilenende Zeichen.

Zeichenreihen (Strings) sind ein eigener Datentyp. Stringkonstanten schreibt man indem man die Zeichen in doppelte Anführungszeichen setzt, zum Beispiel „Hallo !“.

Strings kann man mit der Funktion string aus einzelnen Zeichen erzeugen, wie etwa (string #\H #\i #\space #\!), oder auch mit (make-string *k* *Zeichen*), wobei k die Länge des neuen Strings angibt der mit *Zeichen* aufgefüllt wird.

Auf einzelne Zeichen eines Strings greift man mit (string-ref *String* *k*) (das k-te Zeichen von *String*) zu.

Nützliche Funktionen sind auch (substring *String* *Anfang* *Ende*), string-append (zusammenhängen) und string-copy.

4.4 Vektoren

Vektoren erlauben einen Zugriff auf die Elemente durch Indizierung. Die Elemente sind dabei von 0 an aufsteigend indiziert. Der Bereich der gültigen Indizes eines Vektors sind also alle exakten nicht negativen ganzen Zahlen kleiner als die Länge des Vektors. Vektoren erzeugt man mit (make-vector <Länge> <Wert>) oder mit (vector <Wert0> ... <Wertn>).

Von den etwa aus Pascal bekannten Arrays unterscheiden sich die Vektoren in Scheme dadurch, daß sie dynamisch sind, d.h. daß ihre Länge nicht von vornherein festgelegt sein muß. Man kann also etwa leicht in Scheme die Matrizenmultiplikation für $n \times n$-Matrizen (mit variablem n) mittels Vektoren programmieren.

Mit vector-length bestimmt man die Länge eines Vektors; mit (vector-ref <Vektor> <Index>) greift man auf die einzelnen Elemente eines Vektors zu; mit (vector-set! <Vektor> <Index> <Wert>) weist man einem Element des Vektors einen neuen Wert zu.

Beim Arbeiten mit Vektoren verwendet man typischerweise do-Schleifen, die einen Index im Bereich $0 \leq i <$ (vector-length <Vektor>) verändert. Die Programmiertechnik unterscheidet sich so kaum von dem von Pascal bekannten Vorgehen.

Beispiel 4.4.1. Klasse 7: Sieb des Erathostenes. Wir geben hier eine Lösung mit Hilfe von Vektoren; dabei läßt sich die Verbindung zur anschaulichen Konstruktion des Siebs besonders gut erkennen. Auf eine Optimierung mit den geraden Zahlen wurde der Einfachheit halber verzichtet.

```
(define (vector-to n) (make-vector (+ n 1) #t))

(define (eratosthenes v) ; v Vektor
  (vector-set! v 0 #f) ; 0 ist keine Primzahl
  (vector-set! v 1 #f) ; 1 ist keine Primzahl
  (do ((i 2 (+ i 1)))
      ((>= i (/ (vector-length v) 2)))
    (if (vector-ref v i)
        (do ((j 2 (+ j 1)))
            ((>= (* i j) (vector-length v)))
          (vector-set! v (* i j) #f))))
  v)

(define (eratosthenes-to n)
  (eratosthenes (vector-to n)))

(define (primes-to n)
  (let ((v (eratosthenes-to n)))
    (do ((l ())
         (i n (- i 1)))
        ((= i 0) l)
      (if (vector-ref v i)
          (set! l (cons i l))))))
```

5. Anwendungen

5.1 Numerische Mathematik

Die Numerische Mathematik befaßt sich mit dem Problem, numerische Lösungen, und sehr oft auch numerische Approximationen für eine Lösung, für ein gegebenes Problem zu finden. Wenn man die Frage nach einer Zahl, die die Gleichung $x^2 = 2$ erfüllt, einfach mit $\sqrt{2}$ beantwortet, so wird ein Algebraiker mit dieser Antwort vollkommen zufrieden sein. Ein Numeriker ist mit solch einer symbolischen Lösung noch nicht fertig. Er möchte eine numerische Lösung oder wenigstens eine genügend genaue Lösung.

Ein Beispiel für numerisches Programmieren ist uns schon in Abschnitt 3.1 begegnet: die Berechnung von Quadratwurzeln nach der Newton Methode.

Die Genauigkeit der gefundenen Näherung wurde dabei durch die Funktion `good-enough?` eingestellt. Wenn wir uns für sehr genaue Näherungen interesieren können wir definieren:

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) (expt 10 -16)))
```

dies sollte uns eine Zahl liefern, deren Quadrat bis auf 16 Stellen nach dem Komma mit 2 übereinstimmt.

Auf die Frage `(mysqrt 2)` erhält man dann

```
886731088897/627013566048
```

Die Form des Ergebnisses zeigt an, dass ausgehend von der exakten Zahl 2, hier (mit entsprechendem Rechenaufwand) nur mit exakten Zahlen gerechnet wurde. Man kann hier vieleicht nicht der Versuchung wiederstehen einmal `(mysqrt 2.0)` zu versuchen. Dabei rechnet Scheme, ausgehend von der inexakten Zahl 2.0, nur mit inexakten Zahlen. Haben wir nun so etwas wie 1.414213562373095 als Ergebnis erwartet, so werden wir enttäuscht. Statt eine Ergebnis zu liefern, geht Scheme in eine Endlosschleife. Der Grund dafür ist einfach, daß im Bereich der inexakten Zahlen die üblichen Gesetze der Arithmetik wie Assoziativität und Distributivität nicht mehr gelten und der Algorithmus deswegen versagt. Möglicherweise gibt es auch überhaupt keine inexakte Zahl, die nahe genug an der tatsächlichen Wurzel aus 2 liegt.

(Hinweis: Der beschriebene Effekt hängt von der jeweiligen Scheme Implementierung inexakter Zahlen ab und kann unter Umständen schon bei geringeren, oder aber auch erst bei höheren Anforderungen an die Genauigkeit auftreten.)

Mit dem Übergang zu inexakten Zahlen tun sich also ganz neue Probleme auf. Mit solchen und anderen Problemen befaßt sich die Numerische Mathematik, ein weites Gebiet mathematischer Forschung, das wir hier nicht weiter ausbreiten können. Man sollte jedoch vorgewarnt sein. Bei numerischen Rechnungen mit sehr kleinen Zahlen oder Zahlen, die sehr nahe beieinander liegen, können ganz unerwartete Effekte auftreten. Die Berechnung numerischer Näherungen bietet sich aber in verschiedenen Jahrgangsstufen (Klasse 9: Kreiszahl π, Klasse 10: Nullstellen, Klasse 11: Differenzenquotient und Differenzialquotient, Klasse 12: Integral, Obersummen und Untersummen) als interessantes Experimentierfeld an.

Im folgenden stellen wir noch zwei Verfahren zur Berechnung von Nullstellen vor.

Das einfachste ist die Regula Falsi. Ausgehend von einem Intervall, an dessen Enden die Werte einer stetigen Funktion verschiedene Vorzeichen haben, wird durch fortgesetzte Intervallhalbierung eine Nullstelle der Funktion, die in diesem Intervall existieren muß, möglichst genau ermittelt. Zunächst muss man ein geeignetes Intervall finden. Hierzu sucht man mit Hilfe einer Wertetabelle ein Intervall der Länge 1, in welchem eine Nullstelle liegt.

```
; Funktion:
(define (f x)
```

```
  (+ (expt x 3) (* 3 x x) (- (* 3 x)) (- 6)))

; Wertetabelle:
; die x-Werte
(define x-werte (list -5 -4 -3 -2 -1 0 1 2 3 4 5))

; die y-Werte
(map f x-werte)
```

Die Funktion `besser` ermittelt nun einen Teilpunkt des gegebenen Intervalls.

```
(define (besser x y)
  (/ (+ x y) 2))
```

Die Variable `genau` gibt die Genauigkeit der gesuchten Lösung an. Man könnte `genau` auch als zusätzlichen Parameter bei der Funktion `loese` verwenden.

```
(define genau (expt 10 -30))
```

Durch Betrachtung der Vorzeichen der Werte von f an den Grenzen der beiden Teilintervalle findet man ein kleineres Intervall, das immer noch eine Nullstelle enthält.

```
(define (loese x y)
  (let* ((a (besser x y))
         (fa (f a)))
    (cond ((<= (abs (- y x)) genau) a )
          ((< (* fa (f x)) 0) (loese x a))
          ((< (* fa (f y)) 0) (loese a y))
          (else (error "loese" "Keine Nullstelle gefunden")))))
```

Die Anzeigefunktion `show` liefert dann eine kommentierte Ausgabe der Lösung.

```
(define (show a)
    (display "Loesung exakt: ") (newline)
    (display a) (newline)
    (display "Loesung inexakt: ") (newline)
    (display (exact->inexact a)) (newline)
    (display "Ziffernfolge der exakten Loesung:") (newline)
    (display (round (/ a genau))) (newline)
    (display "Funktionswert an der gesuchten Stelle: ") (newline)
    (display (exact->inexact (f a))) (newline))
```

Beispiel:

```
(show (loese -2 -1))
Loesung exakt:
-436794599502275153858418571144562332473267554147631/374144419156711147O...
Loesung inexakt:
-1.1674491911085352
Ziffernfolge der exakten Loesung:
-1167449191108535156274410599528432978508986O8527949
Funktionswert an der gesuchten Stelle:
-1.5937663482584213e-51
```

Die Konvergenzgeschwindigkeit bei der Regula Falsi ist vergleichsweise gering: bei jeder Iteration wird das Lösungsintervall halbiert und man gewinnt genau eine binäre Stelle.

Das folgende Verfahren, die sogenannte Sekantenmethode, verwendet zur Verbesserung der Approximation auf dem Intervall $[a, b]$ nicht den Intervallmittelpunkt $(a + b)/2$, sondern den Schnittpunkt der Geraden durch die Punke $(x_n, f(x_n))$ und $(x_{n-1}, f(x_{n-1}))$ (Sekante) mit der x-Achse: $x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$. Das Programm sieht dann so aus:

```
(define (f x)
  (+ (expt x 3) (* 3 x x) (- (* 3 x)) (- 6)))

(define (besser x fx y fy)
   (let ((h (* fx (/ (- x y) (- fx fy)))))
        (- x h)))

(define genau (expt 10 -50))

(define (loese x y)
    (loese-sekante x (f x) y (f y)))

(define (loese-sekante  x1 fx1 x0 fx0)
  (let ((x2 (besser x1 fx1 x0 fx0)))
    (if (<= (abs (- x2 x1)) genau)
        x2
        (loese-sekante x2 (f x2) x1 fx1))))
```

Beispiel:

```
(show (loese -2 -1))
Loesung exakt:
-19447948446007141657522123823398539536364028578779371289591988683068567...
Loesung inexakt:
-1.1674491911085352
Ziffernfolge der exakten Loesung:
-116744919110853515627441059952 8
Funktionswert an der gesuchten Stelle:
-1.50890799270184e-60
```

Bei der Sekantenmethode verdoppelt sich die Anzahl der richtigen Stellen in jedem Iterationsschritt, so daß die Genauigkeit schon nach nur 8 Iterationen (mehr als) 38 Dezimalstellen beträgt. Die Genauigkeit kann man leicht überprüfen, wenn man in der Funktion `besser` eine eintsprechende Ausgabe einbaut:

```
(define (besser x fx y fy)
  (let ((h (* fx (/ (- x y) (- fx fy)))))
    (display (exact->inexact h)) (newline)
    (- x h)))
```

man erhält:

```
 (show (loese -2 -1))
-0.8
-0.040336134453781515
0.0078079564113353525
-2.2616128700794703e-5
-1.4720346178514301e-8
2.8289367174486442e-14
-3.536105530332311e-23
-8.494410265982968e-38
2.5506051588919313e-61
```

Man entnimmt der Ergebnisausgabe aber auch, daß die Brüche, die bei der Berechnung mit der Sekantenmethode auftreten, ungleich komplexer sind als bei der Rechnung mit der Regula Falsi. Entsprechend ist die Laufzeit des Sekantenverfahrens nicht um so viel schneller, als man nur bei Betrachtung der Konvergenzgeschwindigkeit erwarten würde.

Zu den Aufgaben der numerischen Mathematik gehört nun eine genaue Analyse, an welchen Stellen exakte Arithmetik von Vorteil und wo sie nur Zeitverschwendung ist.

Ersetzt man die exakte Rechnung durch inexakte Rechnung, so beschleunigt das die Rechnung mit der Sekantenmethode ungemein (man teste etwa `(loese -2.0 -1.0)`). Jedoch hat man dann mit ganz anderen Problemen zu kämpfen. So zum Beispiel mit dem numerischen Unterschied bei der Berechnung von mathematisch äquivalenter Formeln.

Bei der Berechnung der verbesserten Näherung verwenden wir die Formel:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

Die mathematisch äquivalente Formel

$$x_{n+1} = \frac{x_{n-1}f(x_n) - x_n f(x_{n-1})}{f(x_n) - f(x_{n-1})}$$

die also bei exakter Arithmetik auch die gleichen Werte liefert, ergibt mit inexakten Zahlen ein bedeutend schlechteres Konvergenzverhalten. Der Grund liegt einfach darin, daß der feine Unterschied zwischen x_n und x_{n-1} der im Laufe der Iteration erreicht wird, durch die sehr viel gröberen Unterschiede zwischen $f(x_n)$ und $f(x_{n-1})$ zugedeckt werden können.

Viele weitere interessante Beobachtungen können beim Experimentieren mit numerischem Rechnen in Scheme gewonnen werden. So ist z.B. die Sekantenmethode angewandt auf die Endpunkte des Intervalls, statt auf die beiden letzten Näherungswerte, eher schlechter ist als die der Regula Falsi. Die Analyse der auftretenden Phänomene übersteigt aber oft den Rahmen der Schulmathematik. Der interessierte Leser sei auf das Buch [4] verwiesen.

5.2 Iteration über Listen

Ein wesentliches Merkmal der Programmiersprache Scheme ist nicht nur die besondere Behandlung von Funktionen als Datentypen, womit Scheme das funktionale Programmieren ermöglicht, sondern auch die bevorzugte Behandlung von Listen als eingebauter Datentyp. In vielen Programmiersprachen ist die besondere Bedeutung von Paaren, und damit Listen, als grundlegender Datentyp nicht erkannt und somit wird dieser Datentyp in der Sprache auch nicht besonders unterstützt. Wer Scheme als „zweite Programmiersprache" lernt, muß deswegen gerade hier neu dazu lernen. Um den Übergang zu erleichtern, wollen wir nun die Unterschiede bei der Listenprogrammierung anhand eines Beispiels verdeutlichen.

Betrachten wir dazu folgende Aufgabe: Gegeben ist eine Liste von Zahlen. Man drucke diese Liste zeilenweise aus, schließe die Ausgabe mit einem Strich ab und setze unter den Strich die Summe der Zahlen. Mit der Liste $4, 2, 5, 1$ ergibt sich damit die Ausgabe

```
4
2
5
1
========
12
```

Zuerst betrachten wir eine Lösung, wie man sie etwa in der Programmiersprache Pascal erstellen würde. Es werden zwei Schleifen verwendet, die erste zur Ausgabe, die zweite zur Summation. Es ist klar, daß man auch mit einer einzigen Schleife auskommen kann, aber wir möchten im weiteren zwei verschiedene Techniken zur Behandlung von Schleifen aufzeigen.

```
; iterative Loesung

(define (total zahlen)
   (do ((i 0 (+ i 1)))
       ((= i (length zahlen)))
       (display (list-ref zahlen i)) (newline))
   (display "=======") (newline)
```

```
    (do ((i 0 (+ i 1))
         (sum 0))
        ((= i (length zahlen)) (display sum) (newline))
        (set! sum (+ sum (list-ref zahlen i)))))
```

Diese Lösung ist gut, weil sie einen einheitlichen Programmierstil verwendet und konsequent umsetzt. Jeder Leser dieses Programms – soweit er mit dem entsprechenden Programmierstil vertraut ist –, ist in der Lage, die Funktion zu verstehen und bei Bedarf auch anzupassen.

Die nächste Lösung, beruht darauf, daß eine Liste in der Tat eine rekursive Datenstruktur ist, denn eine Liste, wenn es nicht die leere Liste ist, besteht aus dem ersten Element und einer Liste (dem Rest).

```
; rekursive Loesung

(define (total zahlen)
   (define (print zahlen)
        (if (pair? zahlen)
            (begin
                (display (car zahlen)) (newline)
                (print (cdr zahlen)))))
   (define (sum zahlen)
        (if (null? zahlen)
             0
            (+ (car zahlen) (sum (cdr zahlen)))))
   (print zahlen)
   (display "=======") (newline)
   (display (sum zahlen)) (newline))
```

Auch diese Lösung bedient sich eines konsequenten Stils. Die Verwendung der Rekursion erzwingt die Verwendung einer eigenen Funktion für jede Schleife. Hätten wir uns oben entschlossen, das Problem mit nur einer Schleife zu programmieren, so wären wir hier auch mit nur einer Funktion ausgekommen. Die Einführung von eigenen Funktionen für die Schleifen ist nicht unbedingt negativ zu sehen. Daraus entsteht ein durchaus heilsamer Zwang, über die Strukturierung der Funktion nachzudenken, und die gefundene Struktur im gewonnenen Programm deutlich sichtbar zu machen.

Die rekursive Implementierung hat den großen Vorteil, daß sie die sequentielle Natur des Zugriffs auf eine Liste nutzt anstatt sie mittels `list-ref` zu umgehen. Für grosse Listen wird sich das auch in der Laufzeit der beiden Lösungen niederschlagen. Da `list-ref` zur Bestimmung des i-ten Listenelements eine Laufzeit proportional zu i hat (sequentieller Zugriff), hat die iterative Lösung eine Laufzeit proportional zum Quadrat der Listenlänge. Hingegen ist die Laufzeit der rekursive Lösung nur linear in der Listenlänge.

Bei beiden Lösungen störend ist jedoch der große Verwaltungsaufwand, der sich einmal in der Behandlung der Laufvariable i und dann in der Verwendung von `car`, `cdr`, `pair?` und `null?` zeigt. Scheme bietet aber eingebaute Funktionen, die direkt die Iteration über Listen unterstützen. Die Funktionen `apply` und `map` bieten spezielle Kontrollstrukturen zur Verarbeitung von Listen.

`apply` nimmt eine Funktion und eine Liste und verwendet die Liste als Parameter für einen Aufruf der gegebenen Funktion. Dies ist besonders nützlich, da viele eingebaute Funtionen eine variable Anzahl von Argumenten erlauben, und es in Scheme auch für den Anwender einfach und effizient möglich ist, neue Funktionen mit variabler Parameterliste zu schreiben.

`map` nimmt eine Funktion und eine Liste und wendet die Funktion auf jedes einzelne Element der Liste an. `map` ist also das allgemeine Schleifenkonstrukt für Listen. Als Ergebnis liefert `map`, wie nicht anders zu erwarten, die Liste der Einzelergebnisse.

```
; Loesung mit map und apply

(define (total zahlen)
   (map (lambda (x)
          (display x) (newline))
        zahlen)
   (display "=======") (newline)
```

```
    (display (apply + zahlen)) (newline))
```

Diese Lösung ist nicht nur die kürzeste und schnellste, sie ist auch sehr übersichtlich und damit leicht zu lesen und zu modifizieren. Der Verwaltungsaufwand für die Schleifen fällt vollständig weg. Damit entfällt auch die Quelle vieler möglicher Fehler bei der Initialisierung, der Schleifenfortschaltung und dem Abbruchkriterium. Eine Endlosschleife ist von vorne herein ausgeschlossen.

5.3 Endliche Mengen

Mengen kann man auf viele verschiedene Arten repräsentieren. Hierbei sind in erster Linie Effizienzgesichtspunkte zu beachten.

Wir verwenden die Methode der Datenabstraktion. Das heißt, daß wir Mengen ausschließlich mit den Operationen

`adjoin-set`, `empty-set`, `element-of-set?`, `empty-set?`, `union-set` und `intersection-set`

bearbeiten.

Als erstes implementieren wir Mengen als *ungeordnete Listen*.

```
(define empty-set '())

(define (empty-set? set) (null? set))

(define (element-of-set? x set)
  (cond ((empty-set? set) #f)
        ((equal? x (car set)) #t)
        (else (element-of-set? x (cdr set)))))

(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))

(define (intersection-set set1 set2)
  (cond ((or (empty-set? set1) (empty-set? set2)) empty-set)
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

Beispiel 5.3.1. Klasse 12: Mengen als ungeordnete Listen. Man implementiere entsprechend `union-set` für die Darstellung von Mengen als ungeordnete Listen.

Unter Effizienzgesichtspunkten ist diese Implementierung nicht sehr befriedigend. Die Grundoperation `element-of-set?` benötigt n Schritte, um eine Menge bestehend aus n Elementen daraufhin zu prüfen, ob das gegebene Objekt Element der Menge ist. Die benötigte Zeit wächst also wie $O(n)$, wenn n die Größe der Menge ist. Die Operation `adjoin-set`, die `element-of-set?` verwendet, braucht also auch $O(n)$ viele Schritte. Die Operation `intersection-set` verwendet für jedes Element von `set1` einen Test `element-of-set?`, braucht also insgesamt $O(n^2)$ viele Schritte. Dasselbe gilt für `union-set`.

Als nächstes implementieren wir Mengen als *geordnete Listen*. Dafür ist es notwendig, daß wir eine lineare Ordnung der Elemente gegeben haben. Man könnte etwa die lexikographische Ordnung für Symbole verwenden. Zur Vereinfachung nehmen wir hier an, daß nur ganze Zahlen als Elemente in Frage kommen.

```
(define (element-of set? x set)
  (cond ((empty-set? set) #f)
```

```
        ((= x (car set)) #t)
        ((< x (car set)) #f)
        (else (element-of set? x (cdr set)))))
```

Im Mittel benötigt man nur halb soviele Schritte wie bei der Implementierung von Mengen durch ungeordnete Listen. Dies ist aber immer noch von der Größenordnung $O(n)$.

Eine wesentlich größere Beschleunigung erhält man bei der Durchschnittsbildung:

```
(define (intersection-set set1 set2)
  (if (or (empty-set? set1) (empty-set? set2))
      empty-set
      (let ((x1 (car set1))
            (x2 (car set2)))
        (cond ((= x1 x2)
               (cons x1 (intersection-set (cdr set1) (cdr set2))))
              ((< x1 x2)
               (intersection-set (cdr set1) set2))
              ((> x1 x2)
               (intersection-set set1 (cdr set2)))))))
```

Diese Implementierung braucht nur noch $O(n)$ viele Schritte.

Beispiel 5.3.2. Klasse 12: Mengen als geordnete Listen. Man gebe eine entsprechende $O(n)$-Implementierung von `union-set` für die Darstellung von Mengen als geordnete Listen.

Schließlich implementieren wir Mengen als *binäre Bäume*. Hierbei wird an jeden Knoten ein Element geschrieben, und es wird verlangt, daß der linke Teilbaum nur kleinere, der rechte Teilbaum nur größere Elemente enthält. Wenn dann der Baum „ausgeglichen“ (oder balanziert) ist, benötigt die Abfrage `element-of set?` nur noch $O(\log n)$ viele Schritte.

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right) (list entry left right))
(define empty-set '())
(define (empty-set? set) (null? set))

(define (element-of set? x set)
  (cond ((empty-set? set) #f)
        ((= x (entry set)) #t)
        ((< x (entry set))
         (element-of-set? x (left-branch set)))
        ((> x (entry set))
         (element-of-set? x (right-branch set)))))
```

Das Hinzufügen eines Elements zu einer Menge wird ähnlich vorgenommen und erfordert ebenfalls $O(\log(n))$ viele Schritte.

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

Die Durchschnittsbildung benötigt allerdings $O(n\log(n))$ viele Schritte.

Ein Problem der Darstellung von Mengen als binäre Bäume besteht darin, daß die angegebenen Schranken für die Schrittzahlen nur dann zutreffen, wenn die Bäume tatsächlich ausgeglichen sind. Dies können wir zwar erwarten, wenn man Elemente „zufällig" hinzufügt, aber man kann natürlich nicht sicher sein. Ein Ausweg besteht darin, daß man nach jeweils einigen Hinzufügungsschritten eine Operation einschaltet, die eventuell unausgeglichene Bäume in ausgeglichene umformt.

5.4 Kombinatorik

Wir haben in den Beispielen 4.1.4 und 4.1.5 schon zwei Grundaufgaben der Kombinatorik behandelt. Als Ergänzung möge noch das folgende Beispiel dienen.

Beispiel 5.4.1. Klasse 11: Münzen-Wechseln. Wir schreiben ein Programm, das berechnet, auf wie viele Arten man 1 DM in Münzen wechseln kann. Es sei $b =$ zu wechselnder Betrag, $mm =$ Liste der Münzarten.

```
 (define (w b mm)
  (cond ((zero? b) 1)
        ((or (< b 0) (null? mm)) 0)
        (else (+ (w b (cdr mm)) (w (- b (car mm)) mm)))))

(define mm '(100 50 10 5 2 1))

(w 10 mm) ==> 11

(w 100 mm) ==> 2499
```

5.5 Programme aus Beweisen

Wir wollen diskutieren, wie man Existenzbeweise als Programme verwenden (d.h. interpretieren) kann. Im nächsten Abschnitt 5.6 zeigen wir, daß und wie diese in Beweisform gegebenen und deshalb „extrem kommentierten" Programme bei Vorliegen von zusätzlichen Informationen über die Daten verändert werden können.

Ein Wort der Vorsicht ist jedoch am Platz: es handelt sich hier um eine „im Prinzip" anwendbare Methode, die zur Zeit Gegenstand der Forschung ist und deren praktische Relevanz noch nicht abgeschätzt werden kann.

Ein weiteres Wort der Vorsicht: Ein (klassischer) Beweis für $\forall x \exists y\, A$ liefert i.a. *kein* Programm zur Berechnung von y aus x. Als Beispiel betrachten wir die folgende Aussage.

Es gibt irrationale Zahlen a, b mit a^b rational.

Einen Beweis erhält man wie folgt durch Fallunterscheidung.

Fall $\sqrt{2}^{\sqrt{2}}$ ist rational. Man wähle $a = \sqrt{2}$ und $b = \sqrt{2}$. Dann sind a, b irrational, und nach Annahme ist a^b rational.

Fall $\sqrt{2}^{\sqrt{2}}$ ist irrational. Man wähle $a = \sqrt{2}^{\sqrt{2}}$ und $b = \sqrt{2}$. Dann sind nach Annahme a, b irrational, und

$$a^b = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \left(\sqrt{2}\right)^2 = 2$$

ist rational. □

Solange wir nicht entschieden haben, ob $\sqrt{2}^{\sqrt{2}}$ nun rational ist oder nicht, wissen wir nicht, welche Zahlen a, b wir nehmen müssen. Damit haben wir ein Beispiel eines Existenzbeweises, der es nicht erlaubt, das als existent nachgewiesene Objekt auch tatsächlich anzugeben.

Ein einfaches Gegenbeispiel, das tatsächlich die *Unmöglichkeit* der allgemeinen Ablesbarkeit von Programmen aus klassischen Existenzbeweisen nachweist, wurde von Kreisel angegeben. Dazu verwenden wir ein Ergebnis aus der Theorie der Berechenbarkeit (Rekursionstheorie): man kann eine entscheidbare Relation R angeben, für die $\{ x \mid \exists y R(x,y) \}$ unentscheidbar ist.

Offenbar gilt schon in der (Minimal-)Logik

$$\vdash \forall x \exists y \forall z. R(x,z) \to R(x,y).$$

(dies ergibt sich aus $\vdash \exists z\, R(x,z) \to \exists y\, R(x,y)$ mit den bekannten Regeln über das Herausziehen von Quantoren aus Implikationen). Gäbe es nun eine berechenbare Funktion f so daß

$$\forall x, z. R(x,z) \to R(x, f(x)),$$

in $\mathbb{N}$ gilt, so wäre $\{ x \mid \exists y R(x,y) \}$ entscheidbar, denn $\exists z\, R(x,z)$ wäre wahr genau dann, wenn $R(x, f(x))$ ist. – Wir müssen uns also auf Formeln der Gestalt

$$\forall x \exists y\, A \quad \text{mit } A \text{ quantorenfrei}$$

beschränken.

Ein Beispiel aus dem Bereich der diskreten Programmierung ist das in Abschnitt 5.6 behandelte Wall Street Problem. Hier ist das Prinzip der Programmextraktion recht leicht einzusehen.

Man kann aber auch Gegenstände der Analysis, etwa Existenzsätze über reelle Zahlen, für die Programmextraktion verwenden. Als Beispiel behandeln wir einen konstruktiven Beweis des Zwischenwertsatzes; zu dem verwendeten Konzept einer konstruktiven Analysis findet man ausführliche Angaben in [3]. Zunächst einige Definitionen.

Ein *kompaktes Intervall* hat die Form

$$[a,b] := \{ x \in \mathbb{R} \mid a \le x \le b \}$$

mit $a, b \in \mathbb{R}$, $a < b$. Ein *endliches Intervall* ist ein kompaktes Intervall oder hat die Form

$$\begin{aligned} (a,b) &:= \{x \in \mathbb{R} | a < x < b\} \\ [a,b) &:= \{x \in \mathbb{R} | a \le x < b\} \\ (a,b] &:= \{x \in \mathbb{R} | a < x \le b\} \end{aligned}$$

mit $a, b \in \mathbb{R}$, $a < b$. Ein *Intervall* ist ein endliches Intervall oder hat die Form

$$\begin{aligned} (-\infty, a) &:= \{x \in \mathbb{R} | x < a\} \\ (-\infty, a] &:= \{x \in \mathbb{R} | x \le a\} \\ (a, \infty) &:= \{x \in \mathbb{R} | a < x\} \\ [a, \infty) &:= \{x \in \mathbb{R} | a \le x\} \\ (-\infty, +\infty) &:= \mathbb{R} \end{aligned}$$

mit $a \in \mathbb{R}$.

Sei I ein Intervall. Eine Funktion $f\colon I \to \mathbb{R}$ heißt *stetig*, wenn f auf jedem kompakten Teilintervall $[a,b] \subseteq I$ gleichmäßig stetig ist, d.h.

$$\forall \varepsilon \exists^* \delta \forall x, y {\in} [a,b]. |x - y| \le \delta \to |fx - fy| \le \varepsilon.$$

$\omega\colon \mathbb{Q}^+ \to \mathbb{Q}^+$ heißt *Modul* der gleichmäßigen Stetigkeit von f auf $[a,b]$, wenn gilt

$$\forall \varepsilon \forall x, y \in [a,b] (|x - y| \le \omega\varepsilon \to |fx - fy| \le \varepsilon).$$

Beispiele:

1. Jede konstante Funktion sowie die Indentität sind offenbar stetige Funktionen auf $\mathbb{R}$.

2. Sei $f\colon \mathbb{R}^+ \to \mathbb{R}$ definiert durch $f(x) = \frac{1}{x}$. f ist stetig, denn gilt $0 <_\eta a < b$ (wobei $0 <_\eta a$ bedeutet $\eta < a - 0$ mit $0 < \eta \in \mathbb{Q}$), so hat man für $x, y \in [a, b]$

$$\begin{aligned} \Big|\frac{1}{x} - \frac{1}{y}\Big| &= \frac{|y-x|}{|xy|} \\ &\le \frac{1}{\eta^2}|x-y| \\ &\le \varepsilon \quad \text{für } |x-y| \le \frac{\varepsilon}{\eta^2}. \end{aligned}$$

3. Sei $f\colon \mathbb{R} \to \mathbb{R}$ definiert durch

$$f(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!} =: e^x.$$

f ist wohldefiniert, denn $\sum_{i=0}^{\infty} \frac{x^i}{i!}$ ist eine Cauchyfolge reeller Zahlen, die einen eindeutig bestimmten Grenzwert besitzt. f ist stetig, denn hat man $a < b$, so gilt mit $c := \max(|a|, |b|)$ für $x, y \in [a, b]$

$$\begin{aligned} \Big|\sum_{i=0}^{\infty} \frac{x^i}{i!} - \sum_{i=0}^{\infty} \frac{y^i}{i!}\Big| &= \Big|(x-y)\sum_{i=1}^{\infty} \frac{x^{i-1} + x^{i-2}y + \dots + y^{i-1}}{i!}\Big| \\ &\le |x-y| \sum_{i=1}^{\infty} \frac{ic^{i-1}}{i!} \\ &= |x-y| \sum_{j=0}^{\infty} \frac{c^j}{j!} \\ &\le \varepsilon \quad \text{für } |x-y| \le \tfrac{\varepsilon}{e^c}. \end{aligned}$$

Lemma 5.5.1. *Sei f eine stetige Funktion auf einem kompakten Intervall $[a, b]$. Dann findet man*

$$\sup\{\, f(x) \mid x \in [a, b] \,\}.$$

Beweis. Es genügt zu zeigen, daß $\{\, f(x) \mid x \in [a, b] \,\}$ „total beschränkt“ (s. [3]) ist. Sei also ε gegeben. Man wähle m mit $\frac{1}{m} \le \omega\varepsilon$, ω Modul der gleichmäßigen Stetigkeit von f auf $[a, b]$, und setze $a_i := a + i\frac{b-a}{m}$ für $i \le m$. Wir zeigen, daß $\{f(a_0), f(a_1), \dots, f(a_m)\}$ ein „ε-Netz“ (s. [3]) für $\{\, f(x) \mid x \in [a, b] \,\}$ ist. Sei also $f(x)$ mit $x \in [a, b]$ gegeben. Zu x findet man (durch Approximation von x und den a_i) ein $j \le m$ mit $|x - a_j| \le \frac{1}{m}$. Wegen $\frac{1}{m} \le \omega\varepsilon$ folgt $|f(x) - f(a_j)| \le \varepsilon$. □

Bemerkung. Hieraus folgt, daß jedes Polynom

$$f(x) := a_m x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0$$

auf $\mathbb{R}$ stetig ist. Ferner folgt, daß mit f auch $|f| = \max(f, -f)$ stetig ist.

Satz 5.5.2. *(Schwache Form des Zwischenwertsatzes). Sei $f\colon [a, b] \to \mathbb{R}$ stetig, $a < b$, und es gelte $f(a) <_\eta 0 <_\eta f(b)$. Dann folgt*

$$\forall\varepsilon \exists^* c{\in}[a, b].\, |f(c)| \le \varepsilon.$$

Beweis. Sei ε gegeben; wir können $\varepsilon \le y$ annehmen. Zu ε wähle man δ mit

$$|x - y| \le \delta \to |f(x) - f(y)| \le \varepsilon.$$

$[a, b]$ teile man ein in $a = a_0 < a_1 < \ldots < a_m = b$ mit $|a_{i+1} - a_i| \le \delta$. Wir vergleichen jetzt jedes $f(a_i)$ mit $-\frac{\varepsilon}{2} < \frac{\varepsilon}{2}$. Nach Annahme ist $0 <_\varepsilon f(a_m)$; daher können nicht alle $f(a_i) < \frac{\varepsilon}{2}$ sein. Es ist $f(a_0) < \frac{\varepsilon}{2}$. Sei also j minimal mit

$$f(a_j) < \frac{\varepsilon}{2}, \quad -\frac{\varepsilon}{2} < f(a_{j+1}).$$

Wir vergleichen jetzt $f(a_j)$ mit $-\varepsilon < -\frac{\varepsilon}{2}$ und $f(a_{j+1})$ mit $\frac{\varepsilon}{2} < \varepsilon$.

Fall 1. $-\varepsilon < f(a_j)$. Dann ist $|f(a_j)| < \varepsilon$.

Fall 2. $f(a_{j+1}) < \varepsilon$. Dann ist $|f(a_{j+1})| < \varepsilon$.

Fall 3. $f(a_j) < -\frac{\varepsilon}{2}$ und $\frac{\varepsilon}{2} < f(a_{j+1})$. Dann ist $|f(a_{j+1}) - f(a_j)| > \varepsilon$, was wegen $|a_{j+1} - a_j| \le \delta$ nicht sein kann. Fall 3 kann also nicht eintreten. □

Satz 5.5.3. *(Zwischenwertsatz für monotone Funktionen). Sei* $f : [a, b] \to \mathbb{R}$ *stetig,* $a < b$*, und es gelte* $f(a) < 0 < f(b)$*. Ferner sei* f *streng monoton, d.h.*

$$\forall x, y{\in}[a, b].x < y \to f(x) < f(y).$$

Dann findet man eine Nullstelle c *von* f *in* $[a, b]$*, d.h.*

$$\exists^* c{\in}[a, b].\, f(c) = 0.$$

Beweis. *OBdA* seien $a, b \in \mathbb{Q}$. Wir konstruieren $c, d : \mathbb{N} \to \mathbb{Q}$ so, daß für alle m gilt

1. $a = c_0 \leq c_1 \leq \ldots \leq c_m < d_m \leq \ldots \leq d_1 \leq d_0 = b$,
2. $f(c_m) < 0 < f(d_m)$,
3. $d_m - c_m = \left(\frac{2}{3}\right)^m (b - a)$.

Seien $c_0, \ldots, c_m, d_0, \ldots, d_m$ bereits konstruiert, so daß (1)–(3) gelten. Sei $x = c_m + \frac{1}{3}(d_m - c_m)$, $y = c_m + \frac{2}{3}(d_m - c_m)$. Nach Voraussetzung ist $f(x) < f(y)$. Fall 1: $0 < f(y)$. Setze $c_{m+1} := c_m, d_{m+1} := y$. Fall 2: $f(x) < 0$. Setze $c_{m+1} := x, d_{m+1} := d_m$. Offenbar gelten dann wieder (1)–(3). c ist eine Cauchyfolge, denn für $m \geq n$ gilt

$$\begin{aligned} |c_m - c_n| &= c_m - c_n \\ &\leq d_n - c_n \\ &\leq \left(\frac{2}{3}\right)^n (b - a). \end{aligned}$$

Ebenso sieht man, daß d eine Cauchyfolge ist. $c = d$ ergibt sich aus (3), und aus (2) folgt $f(c) \leq 0 \leq f(d)$, also $f(c) = 0$ □

Aus diesem Beweis des Zwischenwertsatzes ergibt sich im Falle der Exponentialfunktion $f(x) = e^x$ folgendes Programm zur Berechnung des natürlichen Logarithmus

```
(define (runde a b)  ; berechnet n mit |n - a/b| <= 1/2
  (let ((q (quotient a b))
        (r (remainder a b)))
    (if (> (+ r r) b) (+ q 1) q)))

(runde 77 4) ==> 19

(define (ln2-approx k)
  (let ((zehn^k (expt 10 k)))
    (do ((j 0 (+ j 1))
         (zwei^j 1 (* 2 zwei^j))
         (drei^j 1 (* 3 drei^j))
         (p_j 0 (let* ((u_j (+ (* 3 p_j) zwei^j))
                       (v_j (+ u_j zwei^j)))
                  (do ((i 0 (+ i 1))
                       (u_j^i 1 (* u_j u_j^i))
                       (v_j^i 1 (* v_j v_j^i))
                       (z_i 1 (+ (* (+ i 1) 3 drei^j z_i)
                                 (* u_j u_j^i)))
                       (w_i 1 (+ (* (+ i 1) 3 drei^j w_i)
                                 (* v_j v_j^i)))
                       (drei^j+1i 1 (* 3 drei^j drei^j+1i))
                       (rechts #f (< (* i z_i)
                                     (* drei^j+1i (- (* 2 i i!) 1))))
                       (links #f (> (* i w_i)
                                    (* drei^j+1i (+ (* 2 i i!) 1))))
```

```
                    (i! 1 (* (+ i 1) i!)))
                   ((or rechts links) (newline)
                    (if rechts (+ (* 3 p_j) zwei^j) (* 3 p_j)))
                 (display " i=") (display i)))))
      ((< (* 6 zwei^j zehn^k) drei^j) (newline)
       (display "ln(2)*10^") (display k)
       (display " ist bis auf 1 genau ")
       (runde (* p_j zehn^k) drei^j))
     (display "j=") (display j))))

(ln2-approx 20) ==> ln(2)*10^20 ist bis auf 1 genau 69314718055994530942
```

5.6 Programmentwicklung durch Beweistransformation

Wir betrachten das folgende einfache aber reale Programmierproblem, das sogenannte *Wall Street Problem* (oder Maximalsegmentproblem) aus [2]:

> Nehmen wir an, wir haben eine Million Dollar zur Verfügung. Wir wollen feststellen, in welchem Zeitintervall wir diese Summe am besten in Aktien investiert hätten. Das soll heißen, in welchem Zeitintervall hätten wir den höchsten Spekulationsgewinn gemacht. Zur Verfügung stehen uns die täglichen Angaben über den Dow–Jones Index.

Wir wollen hier anhand dieses einfachen Beispiels einen besonderen Aspekt der Gewinnung von Programmen aus Beweisen demonstrieren, nämlich die Möglichkeit, allgemeine Programme an spezielle Situationen anzupassen und sie dabei – auch extensional – zu verändern. Daß dies möglich ist, wurde zuerst von Goad in seiner Dissertation [8] bemerkt.

Zu einer gegebenen Folge $x_0, x_1, \dots, x_n$ ganzer Zahlen ist also ein maximales Segment zu finden, d.h. eine durch Indizes i und k mit $i \leq k$ bestimmte Teilfolge aufeinander folgender Glieder, so daß die Summe $x_i + \dots + x_k$ maximal ist. – Wir wollen diese Aufgabe zunächst durch eine logische Formel spezifizieren.

Sei x_i die Differenz der Dow–Jones Indizes der Tage $i+1$ und i. Dann haben wir folgendes Maximalsegmentproblem zu betrachten. Gegeben sei eine Folge $x_0, x_1, \dots, x_n, \dots$ ganzer Zahlen, also $x_i \in \mathbb{Z}$. Dann gibt es ein maximales Segment $x_i + \dots + x_k$ (mit $i \leq k \leq n$).

$\forall$	für alle natürlichen Zahlen
$\exists$	es gibt eine natürliche Zahl
$\leq$	ist kleiner oder gleich
$\mathsf{seg}(i,k)$	Segment $x_i + \dots + x_k$

$$\forall n \exists i, k (i \leq k \leq n \wedge \forall i', k'. i' \leq k' \leq n \to \mathsf{seg}(i', k') \leq \mathsf{seg}(i, k)].$$

Hier haben wir zur Abkürzung z.B. $\exists i, k$ für $\exists i \exists k$ und $i \leq k \leq n$ für $i \leq k \wedge k \leq n$ geschrieben.

Wir geben jetzt eine arithmetische Herleitung dieser Existenzaussage an. Hier spielen Induktionsschlüsse eine wesentliche Rolle.

Zunächst formulieren wir die Behauptung etwas allgemeiner und verwenden anstelle von $x_i + \dots + x_k$ ein beliebiges Maß $\mathsf{seg}(i,k)$ für $x_i, \dots, x_k$, also eine Abbildung $\mathsf{seg}\colon \mathbb{N} \times \mathbb{N} \to X$, wobei X eine beliebige (partiell) geordnete Menge ist. Gesucht sind dann i, k so daß $\mathsf{seg}(i,k)$ maximal in X ist. Ein Beispiel ist etwa $x_i \in \mathbb{Q}^+$ und $\mathsf{seg}(i,k) := \prod_{i \leq j \leq k} x_j$.

Natürlich kann man dieses Problem einfach durch Probieren aller Möglichkeiten lösen; dies sind $O(n^2)$ viele. Dieser trivialen Lösung entspricht auch der im folgenden angegebene Beweis für die allgemeine Behauptung. Genauer heißt dies: Aus dem konstruktiv geführten Beweis kann man einen Algorithmus ablesen, der dem simplen Durchprobieren entspricht.

Wir wollen dann anschließend zeigen, daß für unser konkretes Problem mit der Summe $x_i + \dots + x_k$ als Maß der Beweis sich vereinfachen läßt, indem man an geeigneten Stellen die Monotonie der Summe verwendet. Aus diesem vereinfachten Beweis läßt sich dann ein besserer Algorithmus ablesen, der nur noch $O(n)$ viele Schritte benötigt. Wir haben also hier durch Anpassen eines allgemeinen Beweises an eine spezielle Situation erreicht, daß der aus dem Beweis abzulesende Algorithmus wesentlich verbessert wurde.

Wir skizzieren zuerst den allgemeinen (und trivialen) Beweis. Zu zeigen ist

Lemma 5.6.1. *(Spezifikation).*

$$\forall n \exists i, k. i \le k \le n \wedge \forall i', k'. i' \le k' \le n \to \mathsf{seg}(i', k') \le \mathsf{seg}(i, k).$$

Beweis. Induktion über n. Der Induktionsanfang ist trivial, und im Schritt bestimmt man zunächst ein maximales Endsegment von $x_0, \dots, x_n, x_{n+1}$ und bildet dann das Maximum von diesem maximalen Endsegment und dem nach IH gegebenen maximalen Segment. □

Die erste Feststellung ist hier, daß man sowieso neben dem gesuchten maximalen Segment auch noch das maximale Endsegment berechnen muß. Es liegt deshalb nahe, die Spezifikation entsprechend zu erweitern.

Lemma 5.6.2. *(Erweiterte Spezifikation). Für alle n gilt*

$$\exists i, k. i \le k \le n \wedge \forall i', k'. i' \le k' \le n \to \mathsf{seg}(i', k') \le \mathsf{seg}(i, k) \tag{5.1}$$

$$\exists j{\le}n \forall j'{\le}n\, \mathsf{seg}(j', n) \le \mathsf{seg}(j, n) \tag{5.2}$$

Beweis. Induktion über n. *Fall* 0. Wähle $i = k = 0$ und $j = 0$. *Fall* $n+1$. Nach IH haben wir bereits i_n, k_n sowie j_n.

Wir wollen zunächst j_{n+1} konstruieren, haben also zu zeigen

$$\exists j{\le}n+1 \forall j'{\le}n+1\, \mathsf{seg}(j', n+1) \le \mathsf{seg}(j, n+1). \tag{5.3}$$

Da uns j_n hier offensichtlich nicht weiterhilft, verallgemeinern wir die Behauptung (5.3) so, daß wir sie durch eine Nebeninduktion beweisen können, nämlich zu

$$\forall m{\le}n+1 \exists \ell{\le}m \forall \ell'{\le}m\, \mathsf{seg}(\ell', n+1) \le \mathsf{seg}(\ell, n+1). \tag{5.4}$$

Den Beweis führen wir durch Nebeninduktion nach m, wobei n als Parameter festgehalten wird. *Fall* 0. Wähle $\ell = 0$. *Fall* $m+1$. Nach NIH haben wir bereits ein ℓ_m. Ist $\mathsf{seg}(\ell_m, n+1) < \mathsf{seg}(m+1, n+1)$, so sei $\ell_{m+1} := m+1$; andernfalls setzen wir $\ell_{m+1} := \ell_m$. – Damit ist (5.4) bewiesen, und mit $m := n+1$ können wir unser gesuchtes j_{n+1} definieren als $j_{n+1} := \ell_{n+1}$.

Jetzt können wir i_{n+1}, k_{n+1} konstruieren. Ist $\mathsf{seg}(j_{n+1}, n+1) < \mathsf{seg}(i_n, k_n)$, so nehmen wir das alte maximale Segment und setzen $i_{n+1} := i_n$, $k_{n+1} := k_n$; andernfalls ist das maximale Endsegment das maximale Segment und wir setzen $i_{n+1} := j_{n+1}$, $k_{n+1} := n+1$. □

Offenbar ist der diesem Beweis entsprechende Algorithmus quadratisch.

Wir wollen jetzt zeigen, wie man bei zusätzlichen Kenntnissen über die Eingangsdaten den Beweis vereinfachen kann. In unserer Ausgangssituation galt

$$(x_i + \dots + x_k \le x_j + \dots + x_k) \to (x_i + \dots + x_k + x_{k+1} \le x_j + \dots + x_k + x_{k+1}).$$

Wir betrachten deshalb den Spezialfall

$$\forall i, j, k. \mathsf{seg}(i, k) \le \mathsf{seg}(j, k) \to \mathsf{seg}(i, k+1) \le \mathsf{seg}(j, k+1). \tag{5.5}$$

Diese Zusatzannahme ermöglicht es uns, die Existenz (5.3) eines maximalen Endsegments direkter zu beweisen, und zwar

– ohne Verwendung der Verallgemeinerung (5.4) und der zu ihrem Beweis nötigen Nebeninduktion, aber
– mit Hilfe der IH – also j_n – und der Fallunterscheidung $\mathsf{seg}(j_n, n+1) < \mathsf{seg}(n+1, n+1)$ bzw. $\ge$.

Diesen durch die Zusatzannahme (5.5) ermöglichten verkürzten Beweis wollen wir jetzt genauer durchführen. Dazu modifizieren wir den obigen Beweis wie eben beschrieben.

Beweis. (Verkürzte Form). Zu zeigen ist wieder (5.1) und (5.2). Verwendet wird Induktion über n. *Fall* 0. Wähle $i = k = 0$ und $j = 0$. *Fall* $n+1$. Nach IH haben wir bereits i_n, k_n sowie j_n.

Wir wollen zunächst j_{n+1} konstruieren, haben also zu zeigen

$$\exists j{\leq}n+1 \forall j'{\leq}n+1\, \mathsf{seg}(j', n+1) \leq \mathsf{seg}(j, n+1). \tag{5.6}$$

Die IH liefert uns für beliebige $j' \leq n$

$$\mathsf{seg}(j', n) \leq \mathsf{seg}(j_n, n)$$

und deshalb aufgrund der Zusatzannahme (5.5) auch

$$\mathsf{seg}(j', n+1) \leq \mathsf{seg}(j_n, n+1). \tag{5.7}$$

Ist nun $\mathsf{seg}(j_n, n+1) < \mathsf{seg}(n+1, n+1)$, so sei $j_{n+1} := n+1$. Dann haben wir für beliebiges $j' \leq n+1$ wie gewünscht stets $\mathsf{seg}(j', n+1) \leq \mathsf{seg}(j_{n+1}, n+1)$: im Fall $j' \leq n$ folgt dies aus (5.7) und der Fallunterscheidungsannahme, und im Fall $j' = n+1$ ist es trivial.

Ist andererseits $\mathsf{seg}(n+1, n+1) \leq \mathsf{seg}(j_n, n+1)$, so setzen wir $j_{n+1} := j_n$. Dann haben wir für beliebiges $j' \leq n+1$ wieder $\mathsf{seg}(j', n+1) \leq \mathsf{seg}(j_{n+1}, n+1)$: im Fall $j' \leq n$ ist dies gerade (5.7), und im Fall $j' = n+1$ ist es die Fallunterscheidungsannahme.

Jetzt können wir wie vorher i_{n+1}, k_{n+1} konstruieren. □

Wir wollen uns jetzt die in diesen Beweisen enthaltenen Algorithmen genauer ansehen. Wir haben zunächst einen Existenzbeweis angegeben, der dem einfachen Durchprobieren entsprach; er enthielt zwei übereinanderliegende Induktionen. Der durch Interpretation dieses Beweises als Programm entstehende Algorithmus ordnet jedem n Zahlen i_n, k_n, j_n wie folgt zu.

$$\begin{aligned}
&n = 0: \quad i_0 = k_0 = j_0 = 0\\
&n+1: \quad \text{Gegeben: } i_n, k_n, j_n\\
&\qquad j_{n+1} := \ell_{n+1} \text{ mit}\\
&\qquad\quad \begin{cases} \ell_0 := 0\\ \ell_{m+1} := \begin{cases} m+1 & \text{falls } \mathsf{seg}(\ell_m, n+1) < \mathsf{seg}(m+1, n+1)\\ \ell_m & \text{sonst}\end{cases}\end{cases}\\
&\qquad (i_{n+1}, k_{n+1}) := \begin{cases} (i_n, k_n) & \text{falls } \mathsf{seg}(j_{n+1}, n+1) < \mathsf{seg}(i_n, k_n)\\ (j_{n+1}, n+1) & \text{sonst}\end{cases}
\end{aligned}$$

Wir haben dann zusätzliche Kenntnisse über die Daten angenommen, nämlich daß die Segmentfunktion monoton ist (wie dies bei der Summe der Fall ist). Unter dieser Zusatzannahme ließ sich der Beweis vereinfachen, und zwar so, daß nur noch eine Induktion auftrat. Dem vereinfachten Beweis entspricht folgender Algorithmus:

$$\begin{aligned}
&n = 0: \quad i_0 = k_0 = j_0 = 0\\
&n+1: \quad \text{Gegeben: } i_n, k_n, j_n\\
&\qquad j_{n+1} := \begin{cases} n+1 & \text{falls } \mathsf{seg}(j_n, n+1) < \mathsf{seg}(n+1, n+1)\\ j_n & \text{sonst}\end{cases}\\
&\qquad (i_{n+1}, k_{n+1}) \quad \text{wie eben}
\end{aligned}$$

Wir haben also einen linearen (statt quadratischen) Algorithmus gewonnen. Man beachte, daß das von dem verbesserten Algorithmus gelieferte maximale Segment *verschieden* sein kann von dem durch den ursprünglichen Algorithmus gefundenen. Dies ist ein Effekt, der durch reine Programmtransformation (etwa partielle Auswertung) prinzipiell nicht erreicht werden kann.

5.7 Ausblick

Viele weitere Gegenstände können im Gymnasialunterricht unter Zuhilfenahme von Scheme behandelt werden. Beispiele sind:

- Die Lösungen einer quadratischen Gleichung mit Fallunterscheidung und Ausgabe der beiden Lösungen.
- In der Geometrie können die Volumina zusammengesetzter Körper sehr gut mit den zugehörigen Formeln ermtittelt werden. Die Gesamtformel, die von Hand berechnet werden muß, kann zusätzlich überprüft werden. Weitere lohnende Gegenstände in der Geometrie sind etwa der Schnitt Ebene–Gerade, die Hesse-Normalform und die Vektorrechnung.
- In Jahrgangstufe 10 bieten dann die approximative Berechnung von π (etwa durch In- und Umkreise von 2^n-Ecken), sowie Flächen und Volumenberechnungen ein weites Anwendungsfeld. Zusätzlich sollten exponentielles Wachstum sowie geometrische Folgen und Reihen vorkommen.
- Die Oberstufe, in der man eigentlich die Früchte seiner Arbeit ernten sollte, gestaltet sich momentan immer mehr zur programmierfreien Zone. Der Grenzwertbegriff, die Sekanten- und Tangentensteigung, das Lösen von linearen Gleichungssystemen (Gauß-Algorithmus, wird auch für 3 und mehr Unbekannte gebraucht), das bestimmte Integral, die Simulation und Auswertung von Zufallsexperimenten (Glücksrad, Roulette, Ziegenproblem), das Rechnen mit Vektoren liefern eine Fülle von Anwendungsmöglichkeiten.
- Symbolisches Differenzieren, mit Ausdrucken des Lösungsweges.
- Datenstrukturen, Konstruktoren und Selektoren. Es sollte aufgezeigt werden, daß die Inhalte des Grundkurses Mathematik/Informatik erarbeitet werden können (Markovketten).
- Erstellung eigenständiger Scheme-Programme.
- Ausblicke auf die weiteren Möglichkeiten von Scheme, wie etwa das Schreiben eines eigenen Interpreters oder andere komplexe Projekte etwa bei Facharbeiten; siehe dazu etwa das Skript

  `http://www.mathematik.uni-muenchen.de/~schwicht/lectures/scheme/ws99/s.ps`

- Simulation mit Zufallszahlen, z.B. exponentieller Zerfall.

  ```
  (zerfall 0.5 '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ))
  ```

  Man geht etwa wie folgt vor.
  1. Drucken der Liste
  2. Erzeugen einer neuen Liste wobei jedes Element aus der alten Liste mit Wahrscheinlichkeit p in die neue Liste übernommen wird.
  3. Rekursiver Aufruf von Zerfall mit der neueu Liste (oder Schleife) bis die Liste leer ist.

  Dies ergibt

  ```
  (zerfall 0.7 '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ))
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0
  0 0 0 0 0 0
  0 0 0 0 0
  0 0 0
  0 0 0
  0 0
  0 0
  0
  0
  0

  (zerfall 0.5 '( S p a s s - m i t - S c h e m e - i n - d e r - S c h u l e))
  S p a s s - m i t - S c h e m e - i n - d e r - S c h u l e
  ```

```
p a s - i t - h e e i - e - c u l
p a - t h e - - u l
a - h - - l
a - h l
a l
a l
a
```

Literatur

1. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Struktur und Interpretation von Computerprogrammen*. Springer Verlag, Berlin, Heidelberg, New York, 1991.
2. Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.
3. Errett Bishop and Douglas Bridges. *Constructive Analysis*, volume 279 of *Grundlehren der mathematischen Wissenschaften*. Springer, Berlin, 1985.
4. Björck and Dahlquist. *Numerische Methoden*. Oldenbourg Verlag, München, Wien, 1972.
5. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs. An Introduction to Programming and Computing*. MIT Press, 2000. Also available under URL `http://www.cs.rice.edu/CS/PLT/Teaching/Lectures/Released/Book/`.
6. Otto Forster. *Analysis 1*. Vieweg, 1999. 5-te Auflage.
7. Daniel P. Friedman and Matthias Felleisen. *The Seasoned Schemer*. MIT Press, 1996.
8. Christopher Alan Goad. *Computational uses of the manipulation of formal proofs*. PhD thesis, Stanford University, August 1980. Stanford Department of Computer Science Report No. STAN–CS–80–819.
9. Richard Kelsey, William Clinger, and Jonathan Rees (Editors). *Revised^5 Report on the Algorithmic Language Scheme*, 1998. http://www.swiss.ai.mit.edu/projects/scheme/.
10. Donald E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison Wesley, 3 edition, 1998.

Index