Nichtnumerisches Programmieren (Scheme)

G. Kraus

Ferienkurs 4.-14.10.2005

Dieses Manuskript ist eine Arbeitsunterlage mit Entwurfscharakter. Die Programmbeispiele sind z.T. in älteren Scheme-Versionen gerechnet worden und entsprechen nicht durchwegs der aktuellen Sprachdefinition.
Das Manuskript enthält auch Material, das in der Vorlesung nur skizziert worden ist.

Inhaltsverzeichnis

1 Prozeduren

1.1 Elemente der Programmierung

Lisp (McCarthy 1960) ist nach Fortran die älteste noch allgemein benutzte Programmiersprache.
Die Spache Scheme ist eine modernere Lisp-Version, geschrieben 1975 am MIT und damit ebenfalls älter als viele ihrer Benutzer.
Ist das im Internet-Zeitalter nicht alles veraltet?
Man kann heute praktisch alles irgendwie bewerkstelligen, aber Übersicht und Einsicht zu gewinnen, wird immer schwieriger. Scheme bietet bei einer sehr einfachen und klaren Struktur durch besondere Flexibilität sehr weitreichende Anwendungsmöglichkeiten und bleibt dabei stets transparent.
Um mit den einführenden Worten des Scheme-Reports zu sprechen: Bei Programmiersprachen sollten nicht Merkmale und Eigenschaften aufeinandergehäuft werden, sondern es sollten die Schwächen und Einschränkungen beseitigt werden, welche die zusätzlichen Eigenschaften nötig machten. Scheme beweist, daß eine sehr kleine Anzahl von Regeln für die Bildung von Ausdrücken ausreicht, um eine sehr effiziente Sprache zu erzeugen, die zudem flexibel genug ist, um die meisten und wichtigsten heutigen Programmieraufgaben zu unterstützen.
Darüberhinaus ist die Sprache Scheme auch im hohen Maße geeignet, die Arbeitsweise von Algorithmen klar und eindeutig zu formulieren.
Letztendlich eignen sich die Lisp-Sprachen in besonderer Weise zur Behandlung abstrakter Probleme, besonders auch von Problemen der künstlichen Intelligenz und der Computeralgebra. Auch bei Verwendung von Anwenderpaketen (Mathematica, Maple, Axiom u.a.) greift man auf Lisp-Strukturen zurück.

Am LRZ stehen mehrere Lisp- und Scheme-Implementierungen zur Verfügung, insbesondere:

— PC-Scheme (eine Implementierung von TI-Scheme) an den PCs, Aufruf: pcs.

— MIT-Scheme und LMU-Scheme (O. Forster) an den Sun-Workstationen, Aufruf: scheme bzw. lmuscheme.

Die Sprachdefinition von Scheme kann unter
`http://www.swiss.ai.mit.edu/projects/scheme/`
aus dem Internet heruntergeladen werden.
Per ftp unter `ftp.mathematik.uni-muenchen.de` im Verzeichnis `pub/forster/lmuscheme` kann lmuscheme (für Unix) heruntergeladen werden, aus dem Internet erhält man unter `www.scheme.com` das Programm Petite Chez Scheme (für Unix oder Windows).
Scheme wird beendet mit `(exit)`.

1.1.1 Ausdrücke

Scheme verwendet die **Präfixschreibweise:**

```
(+ 3 4) ==> 7
(- 27 19) ==> 8
(* 13 6) ==> 78
(/ 27 9) ==> 3
(/ 10 6) ==> 1.66666666666667
(+ 2.7 10) ==> 12.7
(+ 3 4 2 10) ==> 19
(* 2 6 3) ==> 36
```

Die Präfixschreibweise kann geschachtelt verwendet und strukturiert aufgeschrieben werden, die Auswertung erfolgt dann von innen nach außen:

```
(* (+ 2
      (* 4 6))
   (+ 3 5 7)) ==> 390
```

Jede offene Klammer muß wieder geschlossen sein. Der Scheme-Interpreter arbeitet im Zyklus

lesen — auswerten — drucken

(**read-eval-print-loop**).

1.1.2 Namen und Belegungen

Es ist möglich, Namen Werte zuzuweisen und dann diese Namen als Abkürzungen für die Werte zu benutzen:

```
(define size 2) ==> size
size ==> 2
(* 5 size) ==> 10
(+ (* 5 size) (* size size)) ==> 14

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

Scheme benutzt zur Speicherung der Wertbindungen Tabellen, sog. **Rahmen (frames)**, die in einer **Umgebung (environment)** genannten Liste zusammengefüt sind.
In anderen Lisp-Sprachen hat man andere Speichermechanismen.
Die `define`-Funktion unterscheidet sich von den bisher betrachteten primitiven Funktionen, da ihre Argumente nicht ausgewertet werden. `define` ist eine sog. **spezielle Form**. Spezielle Formen besitzen eigene Regeln für die Auswertung ihrer Argumente.

Aufgabe 1.1.1 *Man schreibe folgende Ausdrücke in Präfix-Schreibweise und berechne sie in Scheme:*

1. $245 * (1212 + 25 * (345 + 3))$
2. $1 + 2 + 3 + 4 * 6 + 7 * 8 + 9$

1.1.3 Benutzerdefinierte Prozeduren

Außer den primitiven Prozeduren wie + und * kann auch der Benutzer eigene Prozeduren definieren:

```
(define (square x) (* x x))
```

Scheme (wie jede Lisp-Sprache) beruht auf dem **Lambda-Kalkül** (Alonzo Church 1941); die obige Prozedur ist gleichwertig mit

```
(define square (lambda (x) (* x x)))
```

Zur allgemeinen Form von lambda-Ausdrücken siehe die Sprachdefinition.
Beispiele:

```
(square 3) ==> 9
(square (+ 2 5)) ==> 49
(square (square 3)) ==> 81
(define (sum-of-squares x y) (+ (square x) (square y)))
(sum-of-squares 3 4) ==> 25
```

Benutzerdefinierte Funktionen werden, ebenso wie die Bindungen von Werten an Namen, in die Umgebungslisten eingetragen.

1.1.4 Auswertung benutzerdefinierter Prozeduren

Die Auswertung einer Kombination mit einer definierten Prozedur als Operator geschieht wie folgt:

1. Die Argumente werden ausgewertet und die Werte an die formalen Parameter gebunden.
2. Der Kern des lambda-Terms wird mit diesen neuen Wertbindungen ausgewertet.

Eine korrekte Behandlung der Wertbindungen bei Prozeduraufrufen ist ein wesentliche Knackpunkt bei Computersprachen.
Der Mechanismus der Wertbindungen bei älteren Lisp-Sprachen ist anders als in Scheme und führt bei den älteren Lisp-Sprachen zu bestimmten Problemen.

1.1.5 Bedingte Ausdrücke

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

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

In Scheme gibt es für diesen Zweck die spezielle Form `cond`:

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

`cond` ist eine spezielle Form und hat daher eine spezielle Auswertungsregel. In der allgemeinen Form

```
  (cond (test1 consequent1)
        (test2 consequent2) ...
        (testk consequentk))
```

werden die Ausdrücke `test1, test2, ...` nacheinander ausgewertet, bis ein von `#f` verschiedener Wert auftritt und dann der entsprechende danebenstehende Ausdruck.
In Scheme gilt jeder von `#f` verschiedener Wert als wahr (auch der Wert **NIL** = `()`, der im klassischen Lisp für falsch steht!)
Alternative Möglichkeiten sind:

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

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

Aufgabe 1.1.2 *Erkläre den Unterschied zwischen* `if` *und der folgenden Prozedur* `new-if`*:*

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

Aufgabe 1.1.3 *Man schreibe eine Prozedur, die zu vorgegebener Schranke M (natürliche Zahl) ein möglichst kleines n (natürliche Zahl) berechnet mit*

$$n! > M$$

Als Lösung bietet sich eine **rekursive Funktion** `b0` *an, die die Fakultät entsprechend der rekursiven Definition aufbaut und jeweils den Vergleich durchführt:*

```
(define (b0 M n facn)
    (if (< M facn)
        n
        (b0 M (+ 1 n) (* (+ 1 n) facn))))
```

```
> (b0 (expt 10 1) 0 1)
4
> (b0 (expt 10 10) 0 1)
14
> (b0 (expt 10 100) 0 1)
70
> (b0 (expt 10 1000) 0 1)
450
> (b0 (expt 10 10000) 0 1)
3249
> (b0 (expt 10 100000) 0 1)
25206
> (b0 (expt 10 1000000) 0 1)
205023
```

Beim Aufruf dieser Funktion muß neben der Schranke M die erste natürliche Zahl 0 und ihre Fakultät $0! = 1$ eingegeben werden. Diese Unbequemlichkeit kann man vermeiden durch Einführung einer Hauptfunktion `b1`*, die* `b0` *als rekursive Hilfsfunktion aufruft:*

```
> (define (b1 M) (b0 M 0 1))
> (b1 (expt 10 100))
70
```

Etwas besser ist es, die Hilfsfunktion der Hauptfunktion unterzuordnen:

```
> (define (b1 M)
    (define (b0 M n facn)
      (if (< M facn)
        n
        (b0 M (+ 1 n) (* (+ 1 n) facn))))
    (b0 M 0 1))
> (b1 100)
5
```

Die Verwendung rekursiver Funktionen läßt sich vermeiden durch Verwendung einer `do`*-Schleife:*

```
> (define (b2 M)
    (do
      ((n 0 (+ 1 n))
       (facn 1 (* (+ 1 n) facn)))
      ((< M facn) n)))
> (b2 100)
5
```

Im Gegensatz zu den rekursiven Prozeduren wird diese Prozedur **iterativ** *genannt. Es ist ein klassischer Grundsatz, daß iterative Funktionen bei der Ausführung weniger Speicherplatz beanspruchen als rekursive, da bei jedem Prozeduraufruf die lokalen Variablen der aufgerufenen Prozedur gespeichert werden müssen. Scheme berechnet jedoch auch rekursive Prozeduren mit konstantem (d.h.: minimalem) Speicherbedarf, wenn die rekursiven Prozeduren iterative Prozesse beschreiben (s.u).*

Aufgabe 1.1.4 *Bestimme zu vorgegebener Schranke M die kleinste natürliche Zahl n mit $10^n > M$*

Aufgabe 1.1.5 *Die n-te Partialsumme*

$$\sum_{m=0}^{n} \frac{1}{m!}$$

der e-Reihe kann in der Form $\frac{z(n)}{n!}$ geschrieben werden. Schreiben Sie eine Prozedur, die $z(n)$ zu gegebenem n berechnet!
Lösung: Man sieht leicht, daß folgendes gilt:

$$z(0) = 1 \quad \text{und} \quad z(n+1) = (n+1) \cdot z(n) + 1$$

Also:

```
(define (e-count n)
  (if (zero? n)
      1
```

```
      (+ 1 (* n (e-count (- n 1))))))

> (e-count 0)
1
> (e-count 1)
2
> (e-count 2)
5
> (e-count 3)
16
> (e-count 10)
9864101
> (e-count 100)
25368695556012729741527074821228022044514757856629814223277518598
74492539083864465189404854251520497932674077323280034936095134998
4969417670976449032316399200 1
> (e-count 1000)
10938019770709878381838642078033804290071956812454476122822812712
11990864719089676472387110064816694734957784976430979699239344362
31411589872016968717369980131324436579745225045317343249197327597
76111537073871046098290833302642895627153258650379995481897349179
53664902835492545539870529118213555817798834933835387299749069576
74957698889594002794048079499987268330768345470655943182899946890
81510993586783557788818886404725634093024075458245625314221952832
76832875970915612664124876203400808609765495899589738165433030632
18212471131851071428183657532426246266821185296738103112145883929
05182974884090925163492628322475415313646801346848703662635368027
46032502623317281062713756119777729769863899147715394983212872029
02628438020346552035516379624390438919259183580110845234330770128
83534635479246836631219021058731655331579152123138456703197006761
94528709215391836015603019641420546816411620200351230420003216595
88014419675572145237787455543233199207711456290076604902840560445
72935363787840250909389111541834011068106151830070468075608443368
50710586869171139117622655372182726319280572816642266064324832107
57286479850123945778418482239412838359960359301704448868979054722
23693391445598959666384818793054349302510928192328463715030002780
01090951959000653566470475673720400329045441542490526991982254256
73044194920138936817213229225091083761147882052968833395980210010
77304673277361681080992768639307344360645897621823858212022614609
28528192800964734934649260272806083273605653113836564998395257562
25473730773217265138751105949717577471645801831007927220542531909
16034180168110453049093184252541711417178253131597396872082363763
84444489908883103522824237448829186846605257597954616691352965149
56492874936713503672424616504871211398325409346751185571013012713
13804567190312221881301597660718805018377602703800959659013500614
```

```
0968513695317283071834931593285101343397577256734869402826986533 4
6752692758683147333485763417145449390982007517136125940396404598 2
8229351811126663217492061283169796854263489607456114534249612352 7
7976761950962875259567184905250316243755233923166475421159812323 8
6039932393223242820472461906487259093631357583733456749111679442 5
9845389952278098384282516239308383778364826945658788280187994862 9
2303294974774559966547433779589696297298275219444276182101129522 8
2392906303105695220722111658369369913412105341794049726314635892 7
1961943958907018917699222429493249080572615575457469393299786480
7382542519971179948164770899011040614836810296293265851895862946 4
9905240760430608199036210658505666121284331964891538189827998149 0
34317727937074241537692531060200 01
>
```

Aufgabe 1.1.6 *Man schreibe eine Scheme-Prozedur zur Berechnung der (verallgemeinerten) Binomialkoeffizienten*

$$\binom{x}{k} = \prod_{j=1}^{k} \frac{x-j+1}{j}$$

Aufgabe 1.1.7 *Für $n \in \mathbb{N}$ sei $\mathbb{Z}_n := \mathbb{Z}/n\mathbb{Z}$ die n-elementige zyklische Gruppe. Man schreibe Scheme-Prozeduren, welche die folgenden Abbildungen realisieren:*

1. *die Addition in $\mathbb{Z}_n$,*
2. *die Multiplikation in $\mathbb{Z}_n$,*
3. *die kanonische Abbildung $\mathbb{Z} \times \mathbb{Z}_n \to \mathbb{Z}_n, \quad (k, \bar{l}) \mapsto k \cdot \bar{l}$.*

Auf ähnliche Weise sind auch viele nichtnumerische Probleme einfach zu lösen. Wegen der notwendigen Diskussion der Datenstrukturen werden solche Beispiele aber auf später verschoben. Stattdessen nun noch ein aufwendigeres numerisches Beispiel:

1.1.6 Quadratwurzeln nach der Newton-Methode

Als weiteres Beispiel behandeln wir eine Implementierung der Newton-Methode zur Berechnung von Quadratwurzeln. In der Analysis kann man schon vor der Konstruktion der reellen aus den rationalen Zahlen den folgenden Satz beweisen.

Satz 1.1.8 *(Approximation von $\sqrt{a}$).*
Es seien $a > 0$ und $x_0 > 0$ gegeben. Die Folge x_n sei rekursiv definiert durch

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

Dann gilt:

1. $(x_n)_{n\varepsilon I\!N}$ *ist eine Cauchyfolge.*
2. *Wenn* $\lim_{n\to\infty} x_n = b$*, so ist* $b^2 = a$*.*

Beweis: Siehe z.B. Forster, Analysis I.

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 (sqrt x)
  (sqrt-iter 1 x))

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

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

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

Hier wurden alle diese Prozeduren `sqrt, sqrt-iter, good-enough?` und `improve` zu der globalen Umgebung hinzugefügt. Eine Alternative besteht darin, die Hilfsprozeduren `sqrt-iter, good-enough?` und `improve` nach außen unsichtbar zu machen und nur innerhalb der Definition von `sqrt` bereitzustellen. Das läßt sich wie folgt erreichen.

```
(define (sqrt x)
  (define (good-enough? guess x)
     (< (abs (- (square guess) x)) .001))
  (define (improve guess x)
     (average guess (/ x guess)))
  (define (sqrt-iter guess x)
     if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
   (sqrt-iter 1 x))
```

Hierbei läßt sich noch eine weitere Vereinfachung durchführen. Die Variable `x` wird in `sqrt` gebunden. Da die Definitionen von `good-enough?, improve`

und `sqrt-iter` sich im **Bindungsbereich (scope)** von `x` befinden, muß die Variable `x` nicht noch einmal explizit als Parameter übergeben werden. Man spricht hier von einem **"lexikalischen Bindungsbereich" (lexical scoping)**.

```
(define (sqrt x)
  (define (good-enough? guess)
     (< (abs (- (square guess) x)) .001))
  (define (improve guess)
     (average guess (/ x guess)))
  (define (sqrt-iter guess)
     (if (good-enough? guess)
         guess
         (sqrt-iter (improve guess))))
  (sqrt-iter 1))
```

```
> (sqrt 2)
577/408
> (sqrt 9)
65537/21845
> (round (* (expt 10 16) (sqrt 2)))
14142156862745098
```

Aufgabe 1.1.9 *Kann man in der oben definierten Funktion* `sqrt-iter` *die spezielle Form* `if` *ersetzen durch die Prozedur*

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

Aufgabe 1.1.10 *Man schreibe eine Scheme-Prozedur zur Bestimmung einer k-ten Wurzel $\sqrt[k]{a}$, $a > 0$. Hinweis: Man verwende für $x_0 > 0$ die wie folgt rekursiv definierte Folge (x_n):*

$$x_{n+1} := \frac{1}{k}((k-1)x_n + \frac{a}{x_n^{k-1}})$$

1.2 Prozeduren und die von ihnen erzeugten Prozesse

Prozeduren legen den jeweils nächsten Schritt fest, den der Rechner zu tun hat. Der globale Verlauf des Rechneprozesses kann sehr unterschidlich ausfallen.

1.2.1 Primitive Rekursion und Iteration

Betrachten wir zunächst zwei Prozeduren zur Berechnung der Fakultät:

1. Rekursive Prozedur:

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

2. Iterative Prozedur:

```
(define (fakiter n)
   (do ((0 n (- i 1)) (a 1 (* a i)))
        ((zero? i) a))))
```

Beide Prozeduren berechnen offenbar das gleiche Ergebnis, aber sie bewirken unterschiedliche Prozesse. Der Aufruf der rekursiven Prozedur ergibt einen **linear rekursiven Prozeß**. Der Aufruf von `(fakrek 6)` erzeugt folgende Funktionsaufrufe:

```
(fakrek 6)
(* 6 (fakrek 5))
(* 6 (* 5 (fakrek 4)))
(* 6 (* 5 (* 4 (fakrek 3))))
(* 6 (* 5 (* 4 (* 3 (fakrek 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (fakrek 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 (fakrek 0)))))))
(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 1)))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Bei der Abarbeitung der rekusiven Prozedur wird eine Kette von Funktionsanweisungen (hier: Multiplikationen) aufgebaut, die erst später, also verzögert, ausgeführt werden. Der Interpreter muß also Zwischeninformationen speichern; die Menge der zu speichernden Informationen nimmt linear mit der Rekursionstiefe zu.
Anders ist es bei der iterativen Prozedur. Die in den einzelnen Schritten der `do`-Schleife auszuführenden Rechnungen sind:

Stufe	i	Rechnung
1	0	`(* 1 1)`
2	1	`(* 1 1)`
3	2	`(* 1 2)`
4	3	`(* 2 3)`
5	4	`(* 6 4)`
6	5	`(* 24 5)`
7	6	`(* 120 6)`

Hier muß sich er Interpreter keine verzögerten Multiplikationen merken; der Prozeß ist durch die an jeder Stelle des Berechnung vorhandenen Informationen bereits vollständig bestimmt. Die Anzahl der Multiplikationen nimmt linear mit n zu. Ein solcher Prozeß heißt **linearer iterativer Prozeß**.
Es ist bemerkeneswert, daß auch rekursive Prozeduren iterative Prozesse entstehen lassen können. Beispiel:

```
(define (fak2 n)

  (define (fak-iter produkt zaehler max-zaehler)
     (if (> zaehler max-zaehler)
         produkt
         (fak-iter (* zaehler produkt)
                   (+ zaehler 1)
                   max-zaehler)))

  (fak-iter 1 1 n))
```

Der Aufruf von `(fak2 6)` bewirkt folgende Funktionsaufrufe:

```
(fak2 6)
(fak-iter 1 1 6)
(fak-iter 1 2 6)
(fak-iter 2 3 6)
(fak-iter 6 4 6)
(fak-iter 24 5 6)
(fak-iter 120 6 6)
(fak-iter 720 7 6)
720
```

1.2.2 Baumrekursion

Berechnung der Fibonacci-Zahlen durch doppelt-rekursive Funktion:

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

```
                (fib (- n 1))
                (fib (- n 2))))
   )
)
```

Diese Funktion ergibt einen **baumrekursiven Prozeß**:
Schnellere, iterative Berechnung der Fibonacci-Zahlen:

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

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

> (fib 20)
6765
> (fib 40)
102334155
> (fib 50)
12586269025
> (fib 60)
1548008755920
> (fib 80)
23416728348467685
> (fib 90)
2880067194370816120
> (fib 100)
354224848179261915075
>
```

Baumrekursive Prozesse sind adäquat bei entsprechenden Datenstrukuren, z.B. bei LISP-Daten.

2 Daten

2.1 Datenabstraktion

2.1.1 Paare und Listen, konkrete Datendarstellung, Arithmetische Operationen für rationale Zahlen

In Scheme sind Daten und Programme von der gleichen Struktur, sog. *symbolische Ausdrücke*. Insbesondere fallen darunter

— **Atome**; diese können sein

1. *Konstanten*, z.B.
   (a) *numerische Atome* wie z.B. -2 oder 3.14159, in MIT-Scheme auch 5/3 oder 1+3i
   (b) *Charaktere* wie z.B. `#\a, #\A,`
   (c) *Zeichenketten (Strings)*
   Strings können sich auch über mehrere Zeilen erstrecken.

literale Atome (Symbole, Namen, Identifier) wie z.B. x, ab2, radius, +, -, *, `/`, `#F`, `#T`. Symbole können *Variable* sein, an die andere Ausdrücke als Werte gebunden sind, oder *Schlüsselwörter (keywords)*, wenn sie Prozeduren darstellen.

— **Listen**, das sind Ausdrücke der Form $(l_1 l_2 \ldots l_n)$ mit $n \in \mathbb{N}_o$; dabei sind $l_1 l_2 \ldots l_n$ beliebige Atome, Listen oder auch Paare (s.u.). Einfache Beispiele: (1 2 3), (1 (1 2) 3), (* 1 2), (* a (+ 2 3)), ().

2.1.2 Paare, cons, car, cdr

Im praktischen Gebrauch stehen Atome und Listen im Vordergrund. Zu den elementarsten Grundbegriffen von Lisp gehört aber der **Paar**begriff *(dotted pair)*.

Algorithmus 2.1.1 (Paare, cons, car, cdr) *1. Ein* Paar *hat die Form* `(x . y)`*, wobei* `x` *und* `y` *beliebige Lisp-Ausdrücke sein dürfen.*

2. `(cons x y)` *liefert das Paar* `(x . y)`

3. Statt `(x . ())` *schreibt man* `(x)`*, allgemeiner wird stets ein Klammerpaar mit vorausgehendem Punkt weggelassen.*

4. `(car '(x y))` *hat den Wert* `x`*.*

5. `(cdr '(x y))` *hat den Wert* `y`*.*

6. Statt `(car (car exp))` *schreibt man* `(caar exp)`*, analog sind* `cddr`, `cadr`, `cdar`, `caaar` *usw. definiert.*

```
1 ]=> (define p (cons 4 5))

;Value: p

1 ]=> p

;Value: (4 . 5)

1 ]=> (pair? p)

;Value: #T

1 ]=> (car p)

;Value: 4

1 ]=> (define q (cons 3 p))

;Value: q
```

```
1 ]=> q

;Value: (3 4 . 5)

1 ]=> (cons 4 ())

;Value: (4)

1 ]=> (car '(1 2 3 4 5))

;Value: 1

1 ]=> (cdr '(1 2 3 4 5))

;Value: (2 3 4 5)

1 ]=> (cdar '(1 2 3 4 5))

not a pair 1

2 Error->

1 ]=> (cadr '(1 2 3 4 5))

;Value: 2

1 ]=> (cadddr '(1 2 3 4 5))

;Value: 4
```

Mit `cons` wird jeweils ein neues Paar gebildet, daher:

```
1 ]=> (eq? (cons 1 ()) (cons 1 ()))

;Value: ()
```

2.2 Prozeduren für Listen

Algorithmus 2.2.1 `(list exp ... )` *erzeugte eine Liste aus den Werten der als Parameter angegebenen Ausdrücken.*

```
1 ]=> (list (* 2 3) 'a (car '(5 6)))

;Value: (6 a 5)
```

`list` ist (wie z.B. auch `+, *`) eine Prozedur, die eine beliebige Anzahl von Argumenten zuläßt. Eine Definition von solchen Prozeduren mit `define` erfordert eine genauere Analyse von Funktionsdefinitionen und wird später geliefert.

Algorithmus 2.2.2 `(append lis1 lis2)` *hängt zwei Listen aneinander.*

```
1 ]=> (append '(1 2 3) '(3 5 7))

;Value: (1 2 3 3 5 7)

1 ]=> (define (append1 lis1 lis2)
        (cond
           ((null? lis1)  lis2)
           (else          (cons (car lis1) (append1 (cdr lis1) lis2))))))




;Value: append1

1 ]=> (append1 '(1 2 3) '(3 5 7))

;Value: (1 2 3 3 5 7)
```

Algorithmus 2.2.3 `(reverse lis)` *liefert eine Liste mit den Listenelementen in umgekehrter Reihenfolge.*

```
1 ]=> (reverse '(1 2 3 4))

;Value: (4 3 2 1)

1 ]=> (define (reverse1 lis)
        (cond
           ( (null? lis)  () )
           (  else        (append
                            (reverse1 (cdr lis))
                            (cons (car lis) () ))) )))


;Value: reverse1

1 ]=> (reverse1 '(1 2 3 4))
```

Algorithmus 2.2.4 `(map proc lis)` *hat als Wert die Liste, die sich aus Anwendung der Prozedur* `proc` *auf die einzelnen Listenelemente ergibt.*

```
1 ]=> (map 1+ '(1 2 3 4))
```

```
;Value: (2 3 4 5)

1 ]=> (define (map1 proc lis)
        (cond
           ( (null? lis) () )
           ( else        (cons
                            (proc (car lis))
                            (map1 proc (cdr lis)) )) )

)

;Value: map1

1 ]=> (map1 1+ '(1 2 3 4))

;Value: (2 3 4 5)
```

2.3 Umgebungen, Lambda-Ausdrücke

2.3.1 Lambda-Ausdrücke

Kernstück der Funktionsdefinitionen in Lisp ist eine Implementierung des λ-Kalküls der Logik (Alonzo Church 1941). Ein λ-Ausdruck kann als unbenamte Prozedur aufgefaßt werden.

Algorithmus 2.3.1 `(lambda formals exp-1 ... )`*, dabei sind* `formals`*: Ein Identifier oder eine Liste von Identifiern. Es sind folgende Varianten für* `formals` *möglich:*

1. *Eine Liste* `(var-1 ... var-n)`*. Diese Variablen sind dann die formalen Parameter des Lambda-Ausdrucks. Bei Aufruf des Lambda-Ausdrucks werden diese formalen Parameter an die Werte der aktuellen Parameter gebunden (die Anzahl der Parameter muß gleich sein) und die Ausdrücke* `exp-1 ...` *mit diesen Parametern ausgewertet.*

   ```
   1 ]=> ((lambda (x y) (+ x y)) 1 2)

   ;Value: 3
   ```

2. *Eine einzelne Variable* `var`*. Bei Auswertung wird an diese Variable eine Liste gebunden, deren Einträge die aktuellen Parameter sind, und die Ausdrücke* `exp-1 ...` *mit dieser Parameterliste ausgewertet.*

   ```
   1 ]=> ((lambda x x) 1 2 3 4)
   ```

```
;Value: (1 2 3 4)

1 ]=>  (define
    ++
    (lambda
        x
        (define (sumlist lis)
                    (cond
                       ( (null? lis) 0 )
                       ( else   (+ (car lis) (sumlist (cdr lis))) ) ))
        (sumlist x) ))


;Value: ++

1 ]=> (++ 1)

;Value: 1

1 ]=> (++ 1 2 3)

;Value: 6
```

3. *Ein Ausdruck der Form* `(var-1 ... . var-n)` *(uneigentliche Liste). Das ist eine Kombination der beiden vorherigen Fälle. Für die ersten (n-1) Variablen müssen passende aktuelle Parameter vorhanden sein, deren Werte an diese Variablen gebunden werden. Evtl. weitere Parameter werden zu einer Liste zusammengefaßt und diese an die Variable* `var-n` *gebunden.*

```
1 ]=> ((lambda (x y . z) z) 1 2 3 4 5 6 7)

;Value: (3 4 5 6 7)
```

Funktionsdefinitionen mit `define` *sind in der Tat syntaktische Abkürzungen für die Wertbindung eines Lambda-Ausdrucks an eine Variable.*

```
1 ]=> (define sq (lambda (x) (* x x)) )

;Value: sq

1 ]=> sq
```

```
;Value: #[compound-procedure 12 sq]

1 ]=> (define (sqq x) (* x x))

;Value: sqq

1 ]=> sqq

;Value: #[compound-procedure 13 sqq]

1 ]=> (eq? sq sqq)

;Value: ()

1 ]=> (equal? sq sqq)

;Value: ()

1 ]=> (sq 3)

;Value: 9

1 ]=> (sqq 3)

;Value: 9
```

In manchen Scheme-Versionen ist die folgende Form von `lambda` vorhanden:
Algorithmus 2.3.2 `named-lambda formals exp-1 ...`*: wie* `lambda`*, jedoch enthält die Liste* `formals` *als erstes Element ein litarales Atom, das als Namen des Lambfda-Ausdrucks dient. In TI-Scheme wird der Lambda-Ausdruck an diese Variable als Wert gebunden, um rekursive Aufrufe zu ermöglichen. In MIT-Scheme dient die Benennung nur zur Orientierung bei der Fehlersuche (Debugging).*

```
1 ]=> (define fakt
   (named-lambda (fakt n)
     (if (zero? n) 1 (* n (fakt (-1+ n))))))


;Value: fakt

1 ]=> (fakt 3)

;Value: 6
```

2.3.2 Lokale Umgebungen, let und letrec

In Scheme wird mit jedem Lambda-Ausdruck eine neue *lokale Umgebung* gebildet, die die formalen Parameter des Lambda-Ausdrucks enthält, zunächst ohne Wertbindung. Diese lokale Umgebung ist der bei der Definition des Lambda-Ausdrucks gültigen nächsthöheren bzw. globalen Umgebung untergeordnet, d.h., Variable im Rumpf der Prozedur, die nicht in der lokalen Umgebung vorkommen (sog. *freie Variable*, erhalten ihren Wert aus der nächsthöheren Umgebung zur Zeit der Funktionsdefinition *(statische Variablenbindung)*. Beim Aufruf der Prozedur werden die lokalen Parameter an die Werte der aktuellen Parameter gebunden.

```
 1 ]=> (define x 3)

;Value: x

1 ]=> (define (*x y) (* x y))

;Value: *x

1 ]=> (*x 2)

;Value: 6

1 ]=> (define (x-test x y)
               (define (new-*x y) (* x y))
               (write x)
               (newline)
               (write y)
               (newline)
               (write (*x y))
               (newline)
               (new-*x 2))

(x-test 8 2)
8
2
6
16
```

Ein besonders eindrucksvolles Beispiel ist folgendes:

```
1 ]=>  (define x 1)

;Value: x
```

```
1 ]=> (define (f x) (g 2))

;Value: f

1 ]=> (define (g y) (+ x y))

;Value: g

1 ]=> (f 5)

;Value: 3
```

Eine alternative Form zur Bildung neuer lokaler Umgebungen ist
Algorithmus 2.3.3 (let) `(let ((var form) ...) exp ...)`. *Bei Aufruf wird jeder Variablen* `var` *der Wert von* `form` *als Wert zugewiesen und dann die Ausdrücke* `exp ...` *ausgewertet.*
`(let ((var form) ...) exp ...)` ist äquivalent zu `((lambda (var ...) exp ...) form ...)`. Es wird daher bei Aufruf eine neue lokale Umgebung gebildet. Die neuen Variablen erhalten in dieser lokalen Umgebung die Werte der Ausdrücke `form`; bei der Bildung dieser Werte wird aber nicht die neue lokale Umgebung benutzt, sondern die nächsthöhere Umgebung.

```
1 ]=> (let ((x 2) (y 3))
           (* x y))

;Value: 6

1 ]=> (let ((x 2) (y 3))
           (let ((foo (lambda (z) (+ x y z)))
                 (x 7))
             (foo 4)))

;Value: 9
```

Algorithmus 2.3.4 (let*) `(let* ((var form) ...) exp ...)`. *Wie* `let`*, jedoch werden bei der Auswertung der Ausdrücke* `form` *die vorher gebundenen Werte* `var` *bereits benutzt.*
Algorithmus 2.3.5 *Die folgenden Ausdrücke sind äquivalent:*

1. `(let* ((var-1 form-1) (var-2 form-2) ... (var-n form-n)) exp ...)`
2. ```
   (let ((var-1 form-1)
      (let ((var-2 form-2))
      ...
          (let ((var-n form-n))
             exp ... ) ... ))
   ```
   ```

3.
```
((lambda (var-1)
    ((lambda (var-2)
    ...
      ((lambda (var-n)
      exp ...) form-n) ...) form-1)
```

```
1 ]=> (let ((x 2) (y 3))
            (let* ((foo (lambda (z) (+ x y z)))
                   (x 7))
              (foo 4)))


;Value: 9

1 ]=> (let ((x 2) (y 3))
            (let* ((x 7)
                   (foo (lambda (z) (+ x y z))))
              (foo 4)))

;Value: 14
```

Algorithmus 2.3.6 (letrec) `(letrec ((var form) ...) exp ...)`. *Wie* `let`*, jedoch mit folgender Behandlung der Variablen: Bei Aufruf von* `letrec` *wird eine neue lokale Umgebung gebildet und die Variablen* `var` *in dieser Umgebung definiert, ohne daß sie an Werte gebunden werden. Dann werden in dieser Umgebung die Ausdrücke* `form` *ausgewertet (in irgendeiner Reihenfolge) und an die Variablen gebunden, anschließend die Ausdrücke* `exp`*. Wert ist der Wert des zuletzt ausgewerteten Ausdrucks.*
Es gilt folgende Einschränkung: Die Auswertung der Ausdrücke `form` *muß möglich sein, ohne daß die Werte der Variablen* `var` *direkt benutzt oder verändert werden.*
Die letzte Bedingung ist in folgendem Beispiel nur scheinbar verletzt:

```
1 ]=> (let ((x 2) (y 3))
            (letrec ((x 7)
                   (foo (lambda (z) (+ x y z))))
              (foo 4)))

;Value: 14
```

Es wird zunächst eine zu `let` gehörige lokale Umgebung gebildet, in der `x` den Wert `2` hat, `y` den Wert `3`. Mit dem Aufruf von `letrec` entsteht eine neue lokale Umgebung, die der vorigen untergeordnet ist; in dieser Umgebung werden die Variablen `x` und `foo` zunächst ohne Wertbindung definiert. Die

Wertbindung erfolgt erst nach der Auswertung von `7` und `(lambda (z) (+ x y z))`, und zwar an die Werte dieser Ausdrücke, also an das numerische Atom `7` und an den durch das Lambda definierte Prozedur. In der zum Lambda-Ausdruck gehörenden nochmals untergeordneten lokalen Umgebung kommt `x` nicht vor; daher gilt für `x` der Wert aus der `letrec`-Umgebung, und dort ist er bei Auswertung des Lambda-Ausdrucks `7`. Es kommt also auch nicht auf die Reihenfolge der Variablendefinitionen an:

```
1 ]=> (let ((x 2) (y 3))
            (letrec ((foo (lambda (z) (+ x y z)))
                     (x 7))
              (foo 4)))

;Value: 14
```

`letrec` wird meistens so verwendet, daß den Variablen Lambda-Ausdrücke zugewiesen werden. Dadurch wird eine wechselseitige Rekursion ermöglicht:

```
1 ]=> (letrec ((even?
                    (lambda (n)
                      (if (zero? n)
                          #t
                          (odd? (- n 1)))))
                   (odd?
                    (lambda (n)
                      (if (zero? n)
                          #f
                          (even? (- n 1))))))
            (even? 88))

;Value: #T
```

2.4 Seiteneffekte, Zuweisungen (Assignments) und Sequenzen

In der ursprünglichen Konzeption bestand für Lisp die Idee einer rein *applikative* Sprache, was bedeuten würde, daß es bei jeder Prozedur ausschließlich auf ihren Funktiosnwert ankommt. Dieses angedachte Konzept, in dem ein Programm nur im Aufeinanderanwenden von Prozeduren bestehen kann, ist lediglich theoretisch verfolgt worden, tatsächlich widerspricht bereits die Verwendung von Variablen diesem Ansatz.
Die Funktion `cons` bildet ein neues Paar, also ein Paar von Zeigern auf die beiden Ausdrücke, aus denen das Paar besteht. Diese Ausdrücke werden aber nicht geändert; sie tauchen lediglich neu als Ziele von Zeigern auf.
Der Ausdruck `(define a 3)` liefert den Wert `a`: als Seiteneffekt wird die Liste `(a 3)` in die jeweilige Umgebung eingefügt. Die Umgebung wird also

geändert.

Algorithmus 2.4.1 (set!) `(set! var exp)` *ändert den Wert von* `var` *in den Wert von* `exp`*. Die Variable* `var` *muß gebunden sein (in einer Umgebung vorkommen); es muß ihr aber kein Wert zugewiesen sein.*

Auf gebundene Variablen der globalen Umgebung ist die Wirkung von `set!` die gleiche wie die von `define`. Mit `set!` können jedoch auch *lokale Variablen* in einer Weise geändert werden, die mit `define` nicht möglich ist.

Die folgende Funktion `accumulator` addiert die Argumente bei jedem Aufruf:

```
1 ]=> (define accumulator
         (let ((foo 0))
           (lambda (n)
             (set! foo (+ foo n))
             foo)))
;Value: accumulator

1 ]=> (accumulator 2)

;Value: 2

1 ]=> (accumulator 3)

;Value: 5
```

Mit `accumulator` ist also ein Prozedur mit einem *lokalen Zustand* definiert, von dem der Funktionswert abhängt. In Abelson-Sussman ist ausführlich beschrieben, wie eine solche Problematik bei der Verwaltung eines Bankkontos vorkommt.

Ersetzt man in dieser Prozedur `set!` durch `define`, erhält man folgende Fehlermeldung:

```
1 ]=> (define accumulator
         (let ((foo 0))
           (lambda (n)
             (define foo (+ foo n))
             foo)))

;Value: accumulator

1 ]=> (accumulator 2)
Unassigned variable foo
```

Um den Fehler einzugrenzen, wird die Definition abgeändert:

```
1 ]=> (define accumulator
         (let ((foo 0))
           (lambda (n)
             (define foo (+ 1 n))
```

```
            foo)))

;Value: accumulator

1 ]=> (accumulator 2)

;Value: 3


1 ]=> (accumulator 3)

;Value: 4
```

Der Grund ist, daß mit dem `define` innerhalb des Lambda-Ausdrucks eine neue lokale Variable `foo` ohne Wertzuweisung gebunden wird, also nicht mehr auf die entsprechende Variable in der übergeordneten Umgebung zugegriffen wird.
Algorithmus 2.4.2 (Wichtige Zuweisungsprozeduren) `delete!, delq!, reverse!, set-car!, set-cdr!, string-fill!, string-set!`
Eine andere Prozeduren mit Seiteneffekt ist z.B. `write`.

2.5 Konvergente Folgen rationaler Zahlen

Beispiele Folgen rationaler Zahlen ...
Definition konvergente Folge.
Eine gegen $a \in \mathbb{Q}$ konvergente Folge in $\mathbb{Q}$ besteht danach aus folgenden Daten:

1. der Folge (a_n), das ist eine Abbildung $\mathbb{N} \to \mathbb{Q}$,
2. der Zahl a,
3. einer Abbildung (**sog. Konvergenzmodul**) $N : \mathbb{Q}_+^* \to \mathbb{N}$ wie folgt: $\forall \varepsilon > 0 \; \forall n \geq N(\varepsilon) \; |a_n - a| \leq \varepsilon$.

Dieses Konzept kann wie folgt implementiert werden:

```
> (define (make-conv s l m) (cons s (cons l m)))
> (define (seq x) (car x))
> (define (lim x) (car (cdr x)))
> (define (mod x) (cdr (cdr x)))

(define ex1
  (make-conv (lambda (n) 27)
    27
    (lambda (eps) 0)))
```

```
(define ex1
  (make-conv (lambda (n) 27)
    27
    (lambda (eps) 0)))
>
ex1
(#<procedure> 27 . #<procedure>)
> (car ex1)
#<procedure>
> (cadr ex1)
27

> ((seq ex1) 3)
27
> (define ex2
(make-conv (lambda (n) (/ 1 n))
0
(lambda (eps) (ceiling (/ 1 eps)))))

> (define ex3
(make-conv (lambda (n) (/ n (+ n 1)))
1
(lambda (eps) (ceiling (/ 1 eps)))))

> (define ex4
(make-conv (lambda (n) (/ n (expt 2 n)))
0
(lambda (eps) (ceiling (max 4 (/ 1 eps))))))

> ((seq ex4) 7)
7/128
> (define ex4
(make-conv (lambda (n) (/ n (expt 2. n)))
0
(lambda (eps) (ceiling (max 4. (/ 1. eps))))))
> ((seq ex4) 7)
0.0546875
> ((seq ex4) 70)
5.929230630780102e-20
>
```

2.6 Ströme, verzögerte Obkekte

2.6.1 Ströme

Ströme sind wie Listen aufgebaut, jedoch erfolgt die Auswertung der Einträge erst auf spezielle Anforderung. Zur Konstruktion wird `cons-stream` verwendet, statt `car` verwendet man `head`, statt `cdr` `tail`. Man kann auf diese Weise z.B. die „Liste“ der ganzen Zahlen als ein Datenobjekt angeben:
Algorithmus 2.6.1 *Beispiele*

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (1+ n))))

(define naturals (integers-starting-from 0))

(define (nth n s)
 (cond
  ( (empty-stream? s) '() )
  ( (= n 1) (head s) )
  ( (nth (-1+ n) (tail s)) )) )

(define (head-s n s)
 (cond
  ( (empty-stream? s) the-empty-stream )
  ( (< n 1) the-empty-stream )
  ( (cons-stream (head s) (head-s (-1+ n) (tail s))) )) )

(define (tail-s n s)
 (cond
  ( (empty-stream? s) the-empty-stream )
  ( (< n 1) s )
  ( (tail-s (-1+ n) (tail s)) )) )
```

2.6.2 Implementierung von Strömen

Die o.g. Prozeduren gehören nicht zur Sprachdefinition von Scheme.

2.6.3 `delay` und `force`

`delay` ist eine zur Sprachdefinition von Scheme gehörige spezielle Form, die wie folgt eingeführt werden könnte:

```
(define-syntax delay
    (syntax-rules ()
      ((_ e) (lambda () e))))
```

`force` ist eine Scheme-System-Prozedur, die wie folgt definiert werden könnte:

```
> (define (force1 e) (e))
```

2.6.4 `cons-stream`

```
(define-syntax cons-stream
    (syntax-rules ()
      ((_ h t) ((lambda () (cons h (delay t)))))))


(define the-empty-stream '())
(define (empty-stream? s) (null? s))
(define (head s) (car s))
(define (tail s)  (force (cdr s)))



Alternativ-Ansätze:

 > (define-syntax cons-stream6
    (syntax-rules ()
      ((_ h t) (eval '(lambda () (cons h (delay t)))))))

(define-syntax cons-stream1
    (syntax-rules ()
      ((_ h t) ( (cons h (delay t))))))

(define-syntax cons-stream2
    (syntax-rules ()
      ((_ h t) (lambda (h t) '(cons h (delay t))))))

(define-syntax cons-stream
    (syntax-rules ()
      ((_ h t) ((eval (lambda () (cons h (delay t)))))))

(define-syntax cons-stream4
    (syntax-rules ()
      ((_ h t) ( (cons h (delay t)))))))
```

2.6.5 Veranschaulichung von Strömen

Zur Veranschaulichung von Strömen verwenden wir folgende Ausgabefunktion:

Algorithmus 2.6.2

```
 (define (for-each-s proc s)
 (cond
  ( (empty-stream? s) (newline) 'done )
  ( (proc (head s)) (for-each-s proc (tail s)) )) )

(define (write-s s)
  (define (write1 exp)
    (display "   ")
    (write exp))
  (for-each-s write1 s))
```

Beispielaufrufe:

```
1 ]=> naturals

;Value: (0 . #[promise 5])

1 ]=> (head naturals)

;Value: 0

1 ]=> (tail naturals)

;Value: (1 . #[promise 13])

1 ]=> naturals

;Value: (0 . #[promise 5])

1 ]=> (head naturals)

;Value: 0

1 ]=> (tail naturals)

;Value: (1 . #[promise 13])

1 ]=> (define hs (head-s 100 naturals))

;Value: hs

1 ]=> (head hs)
```

```
;Value: 0

1 ]=> (tail hs)

;Value: (1 . #[promise 14])

1 ]=> (nth 20 hs)

;Value: 19

1 ]=> (write-s hs)
   0   1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16   17

;Value: done
```

Eine andere Möglichkeit ist die Verwendung der Systemfunktion `stream->list`, am besten kombiniert mit der System-Ausgabefunktion `pp`.

Algorithmus 2.6.3

```
(define (memo-proc proc)
  (let ((already-run? '#f f)
        (result '() )
  (lambda ()
     (cond ( (not already-run?)
                 (set! result (proc))
                 (set! already-run? '#t)
                 result )
           (  else  result ))) ) )

; In TI-Scheme kann man dann delay definieren als

(syntax (delay1 exp) (memo-proc (lambda () exp)))
```

2.6.6 Abstrakte Prozeduren für Ströme

Algorithmus 2.6.4

```
(define (head-pred-s pred s)
 (cond
  ( (empty-stream? s) the-empty-stream )
  ( (pred (head s)) the-empty-stream )
  ( (cons-stream (head s) (head-pred-s pred (tail s))) )) )

; (liefert den Teil des Stroms, bis pred gilt)

(define (tail-pred-s pred s)
 (cond
  ( (empty-stream? s) the-empty-stream )
```

```
  ( (pred (head s)) s )
  ( (tail-pred-s pred (tail s)) )) )

; (liefert den Teil des Stroms ab Gueltigkeit von pred)

(define (memq-s el s)
 (tail-pred-s (lambda (x) (eq? x el)) s))
```

Algorithmus 2.6.5 *Filterprozeduren*

```
(define (filter pred s )
 (cond
  ( (empty-stream? s ) the-empty-stream )
  ( (pred (head s)) (cons-stream (head s) (filter pred (tail s))) )
  ( (filter pred (tail s)) )) )



1 ]=> (write-s (filter odd? hs))
   1   3   5   7   9   11   13   15   17   19   21   23   25   27   29   31   33

;Value: done
```

Algorithmus 2.6.6 *Map-Stream*

```
(define (map-s proc s)
 (cond
  ( (empty-stream? s) the-empty-stream )
  ( (cons-stream (proc (head s)) (map-s proc (tail s))) )) )

(define (multiply-s factor s) (map-s (lambda (x) (* factor x)) s))

(define (-multiply-s factor s) (map-s (lambda (x) (* (- factor) x)) s))

(define (divide-s divisor s) (map-s (lambda (x) (/ x divisor)) s))

(define (g-multiply-s factor s) (map-s (lambda (x) (g-* factor x)) s))

(define (g-g-multiply-s factor s)
 (map-s (lambda (x) (g-* (g-minus factor) x)) s))

(define (g-divide-s divisor s) (map-s (lambda (x) (g-/ x divisor)) s))

(define (g-quot-divide-s divisor s) (map-s (lambda (x) (g-// x divisor)) s))
```

```
1 ]=> (write-s (multiply-s 3 hs))
   0   3   6   9   12   15   18   21   24   27   30   33   36   39   42   45   4

;Value: done
```

Algorithmus 2.6.7 *Akkumulation von Strömen und Listen*

```
(define (accumulate combiner initial-value s)
 (cond
  ( (empty-stream? s) initial-value )
  ( (combiner
     (head s)
     (accumulate combiner initial-value (tail s))) )) )

(define (sum-s s) (accumulate + 0 s))
(define (product-s s) (accumulate * 1 s))
(define (g-sum-s s) (accumulate g-+ 0 s))
(define (g-product-s s) (accumulate g-* 1 s))
```

Algorithmus 2.6.8 *S-Akkumulation von Strömen und Listen*

```
(define (s-accumulate combiner s1 s2)
 (cond
  ( (empty-stream? s1) s2 )
  ( (empty-stream? s2) s1 )
  ( (cons-stream
     (combiner (head s1) (head s2))
     (s-accumulate combiner (tail s1) (tail s2))) )) )

(define (add-s s1 s2) (s-accumulate + s1 s2))
(define (mult-s s1 s2) (s-accumulate * s1 s2))
(define (g-add-s s1 s2) (s-accumulate g-+ s1 s2))
(define (g-mult-s s1 s2) (s-accumulate g-* s1 s2))

(define (constant-pair-s s1 s2) (s-accumulate cons s1 s2))

(define (convolute s1 s2) (sum-s (mult-s s1 s2)))
(define (g-convolute s1 s2) (g-sum-s (g-mult-s s1 s2)))

(define (equ?-s s1 s2)
 (define (local-and x y) (and x y))
 (accumulate local-and #t (s-accumulate g-= s1 s2)))
```

Algorithmus 2.6.9 *S-Akkumulation von geordneten Strömen*

```
(define (ordered-el-s-accumulate << combiner el s)
 (cond
  ( (empty-stream? s) (cons-stream el the-empty-stream) )
  ( (atom? s) (ordered-el-s-accumulate << combiner el (list s)) )
  ( (<< (head s) el) (cons-stream el s) )
  ( (<< el (head s))
    (cons-stream (head s) (ordered-el-s-accumulate << combiner el (tail s))) )
  ( (cons-stream (apply combiner (list el (head s))) (tail s)) )))

(define (ordered-s-accumulate << combiner s1 s2)
 (cond
  ( (empty-stream? s1) s2 )
  ( (empty-stream? s2) s1 )
  ( (let
     (( h1 (head s1) ) ( h2 (head s2) ))
     (cond
      ( (<< h1 h2)
        (cons-stream h2 (ordered-s-accumulate << combiner s1 (tail s2))) )
      ( (<< h2 h1)
        (cons-stream h1 (ordered-s-accumulate << combiner (tail s1) s2)) )
     ( (cons-stream
        (apply combiner (list h1 h2))
        (ordered-s-accumulate << combiner (tail s1) (tail s2))) ))) )) )

(define (join-ordered-s << s1 s2)
 (ordered-s-accumulate << (lambda (x y) x) s1 s2))
```

Algorithmus 2.6.10 *Verbindung von Strömen*
Aneinanderfügen von zwei endlichen Strömen

```
(define (append-s s1 s2)
 (cond
  ( (empty-stream? s1) s2 )
  ( (cons-stream (head s1) (append-s (tail s1) s2)) ) )
```

Aneinanderfügen eines endlichen Stroms endlicher Ströme

```
(define (finite-flatten s)
 (accumulate append-s the-empty-stream s))
```

Dieser Algorithmus kann *nicht* verwendet werden fuer unendliche Ströme endlicher Ströme. Denn in der Definition von `accumulate` wird die Prozedur `combiner` aufgerufen, die ihr Argument auswertet und deshalb `accumulate`

rekursiv aufruft. Das führt bei unendlichen Strömen zu einer unendlichen Schleife. Man verwendet folgende Variation von `accumulate`:

Algorithmus 2.6.11
```
(define (accumulate-delayed combiner initial-value s)
 (cond
  ( (empty-stream? s) initial-value )
  ( (combiner
     (head s)
     (delay
      (accumulate-delayed combiner initial-value (tail s)))) )) )
```

Man muß dann eine entsprechende Variante von `append-s` *verwenden:*

```
(define (append-s-delayed s1 delayed-s2)
 (cond
  ( (empty-stream? s1) (force delayed-s2) )
  ( (cons-stream
     (head s1)
     (append-s-delayed (tail s1) delayed-s2)) )) )

(define (flatten s)
 (accumulate-delayed append-s-delayed the-empty-stream s))

(define (flatmap f s) (flatten (map-s f s)))
```

Anwendung: Strom aller Paare aus je zwei Elementen von zwei Strömen:

Algorithmus 2.6.12
```
(define (all-pairs-s s1 s2)
 (define (rec-pairs tail-s1 tail-s2 counter1 counter2)
  (let
   (( l-p
      (cond
       ( (empty-stream? tail-s1) the-empty-stream )
       ( (map-s
          (lambda (y) (cons (head tail-s1) y))
          (head-s counter2 s2)) )) )
    ( r-p
      (cond
       ( (empty-stream? tail-s2) the-empty-stream )
       ( (map-s
          (lambda (x) (cons x (head tail-s2)))
          (head-s counter1 s1)) )) )
    ( rec-p
      (cond
       ( (empty-stream? (tail tail-s1))
         (cond
```

```
          ( (empty-stream? (tail tail-s2)) the-empty-stream )
          ( (rec-pairs
              the-empty-stream
              (tail tail-s2)
              (1+ counter1) (1+ counter2)) )  )
       ( (empty-stream? (tail tail-s2))
         (rec-pairs
            (tail tail-s1)
            the-empty-stream
            (1+ counter1)
            counter2) )
       ( (rec-pairs
          (tail tail-s1)
          (tail tail-s2)
          (1+ counter1)
          (1+ counter2)) )  ))
  (append-s (append-s l-p r-p) rec-p)) )
 (rec-pairs s1 s2 0 1))
```

Alternativ:

Algorithmus 2.6.13 `(define (all-pairs-s s1 s2)`

```
 (define (pairs-up-to n s1 s2)
  (append-s
   (map-s
    (lambda (y) (cons (nth n s1) y))
    (head-s n s2))
   (map-s
    (lambda (x) (cons x (nth n s2)))
    (head-s (-1+ n) s1))))
 (flatmap
  (lambda (n) (pairs-up-to n s1 s2))
  naturals))
```

2.6.7 Primzahlen

Zur Berechnung der Primzahlen bzw. der primen Gauß-Zahlen kann man das *Sieb des Eratosthenes* verwenden. Die allgemeine Prozedur ist:

Algorithmus 2.6.14

```
(define (divides? z w) (zero? (modulo w z)))

(define (sieve s)
  (cons-stream
     (head s)
     (sieve
        (filter
           (lambda (x) (not (divides? (head s) x)))
           (tail s)))))
```

Eine endrekursive Prozedur zur Berechnung der primen Gauß-Zahlen ist:
Algorithmus 2.6.15 `(define (s-prime? z)`

```
 (define (iter ps)
   (cond
   ( (divides? (head ps) z) '() )
   ( else      (iter (tail ps)) )))
 (iter s-primes))

(define s-primes
 (cons-stream
  2
  (cons-stream
   3
   (filter s-prime? (tail-s 4 naturals)) )) )
```

Liste der ersten 1000 Primzahlen:

```
1 ]=> (define (write-s s)
  (define counter 1)
  (define (write1 exp)
    (write counter)
    (display "   ")
    (write exp)
    (cond ((zero? (modulo counter 3)) (newline))
          (else (display "        ")))
    (set! counter (1+ counter)))
  (for-each-s write1 s))


;Value: write-s

1 ]=> (write-s (head-s 1000  s-primes))
```

3 Eval und Apply — Scheme in Scheme

3.1 Verwaltung von Tabellen

Die Umgebungen in Scheme sind Tabellen und können als sog. *Assoziationslisten* oder kurz *A-Listen*, das sind eigentliche Listen der Form

```
( (a-1  b-1) (a-2 b-2) . . . (a-n  b-n) )
```

— manchmal versteht man unter A-Listen auch Listen von Paaren, also Listen der Form

```
( (a-1 . b-1) (a-2 . b-2) . . . (a-n . b-n) )
```

— realisiert werden. Für solche A-Listen kennt Scheme die Prozeduren `assoc, assq, assv`. Beim Aufruf von `(assoc obj alist)` erhält man als Wert den ersten in der A-Liste `alist` vorkommenden Eintrag mit `obj` als `car`. Dabei testet `assoc` mittels `equal?`, `assq` mittels `eq?` und `assv` mittels `eqv?`. Diese Prozeduren könnten folgendermaßen definiert werden:
Algorithmus 3.1.1 `(define (atom? x) (not (pair? x)))`

```
(define (assq key alist)
  (cond
    ( (null? alist) () )
    ( (atom? (car alist)) (assq key (cdr alist)) )
    ( (eq? key (caar alist)) (car alist) )
    ( else (assq key (cdr alist)) )) )

1 ]=> (assq 'assq (environment-bindings (the-environment)))

;Value: (assq #[compound-procedure 3 assq])
```

Für manche Zwecke (z.B. die generische Arithmetik), betrachtet man auch eindimensionale Tabellen als "A-Listen mit Kopf", also von der Form

```
( *table* (a-1  b-1) (a-2 b-2) . . . (a-n b-n) )
```

Die folgende Prozedur holt Informationen aus einer eindimensionalen Tabelle:
Algorithmus 3.1.2 `(define (lookup key table)`

```
  (let  ((record (assq key table)))
    (cond
      ( (null? record) () )
      ( else (cadr record) ))) )
```

Wir betrachten nun das umgekehrte Problem, nämlich neue Informationen in eine Tabelle einzufügen. Das Ziel ist hier nicht, eine neue, erweiterte Tabel-

le zusätzlich zu der schon vorhandenen anzulegen, sondern die vorhandene Tabelle abzuändern.

Algorithmus 3.1.3
```
(define (insert! key value table)
  (let  ((record (assq key table)))
    (cond
      ( (null? record)
          (set-cdr! table (cons (cons key value) (cdr table))) )
      ( else
          (set-cdr! record (list value)) )))
   "ok")
```

Zur Erzeugung einer eindimensionalen Tabelle verwendet man folgende Prozedur:
```
(define (make-table) (list '*table*))
```

Zweidimensionale Tabellen werden entsprechend gehandhabt. Der erste Schlüsselbegriff bestimmt eine eindimensionale Untertabelle, in welcher dann nach dem zweiten Schlüssel gesucht wird.

Algorithmus 3.1.4 *Zweidimensionale Tabellen*
```
(define (lookup key-1 key-2 table)
  (let ((subtable (assq key-1 table)))
    (cond
      ( (null? subtable) () )
      ( else
         (let ((record (assq key-2  subtable)))
           (cond
             ( (null? record) () )
             ( else (cadr record) ))) ))) )

(define (insert! key-1 key-2 value table)
  (let ((subtable (assq key-1   table)))
    (cond
      ( (null? subtable)
          (set-cdr! table
                    (cons
                      (list key-1 (cons key-2 value))
                      (cdr table))) )
      ( else
         (let  ((record (assq key-2 subtable)))
           (cond
             ( (null? record)
                 (set-cdr! subtable
                           (cons (cons key-2 value) (cdr subtable))) )
             ( else
                 (set-cdr! record value) ))) )))
```

```
   "ok")

(define (make-table) (list '*table*))
```

Sind Listen von Variablen und ihren zugehörigen Werten gegeben, kann eine Tabelle durch die folgende Prozedur durch die entsprechende A-Liste ergänzt werden:

Algorithmus 3.1.5 `(define (pairlis varlist valuelist alist)`

```
  (cond
    ( (null? varlist) alist )
    ( else   (cons
                (list (car varlist) (car valuelist))
                (pairlis (cdr varlist) (cdr valuelist) alist)) )) )
```

3.2 `eval und apply` — Scheme in Scheme

Algorithmus 3.2.1 `apply proc (arg-list))` *wendet die Prozedur* `proc` *auf die in* `(arg-list)` *enthaltenen Ausdrücke als Argument an; dies ist gleichwertig mit der Auswertung von* `(proc arg ...)`.

Algorithmus 3.2.2 `(eval exp env)` *wertet* `exp` *in der durch* `env` *gegebenen Umgebung aus.*

```
1 ]=> (define x 99)

;Value: x

1 ]=> (define y 100)

;Value: y

1 ]=> (define env
        (let ((x 4) (y 5))
           (the-environment)))

;Value: env

1 ]=>
;Value: ((x 4) (y 5))

1 ]=> (environment-bindings (the-environment))

;Value: ((assq #[compound-procedure 6 assq]) (x 99) (y 100) (env #[environment 4

1 ]=> (eval '(+ x y) env)

;Value: 9
```

```
1 ]=> (eval '(+ x y) (the-environment))

;Value: 199
```

Im folgenden sollen Prozeduren `eval1` und `apply1` definiert werden. Die Prozedur `eval1` soll Scheme-Ausdrücke auswerten können; dabei wird die Auswertbarkeit diverser primitiver Funktionsausdrücke vorausgesetzt. Zur Unterscheidung werden die benutzten Funktionsausdrücke mit einer nachgestellten `1` gekennzeichent, also z.B. `car1`. Die Grundform ist:

Algorithmus 3.2.3

```
(define
  (eval1 form envlist)
  (cond
    ( (atom? form)
        (cond
           ( (eq? form '#T)  '#T )          ; Auswertung von Konstanten
           ( (eq? form '#F)  '#F )
           ( (number? form) form ) ; ...
           ( else (lookup form envlist) )   ; Auswertung von Variablen
        )
    ) ; Klausel
    ( (atom? (car form))                    ; Auswertung von Prozeduren,
        (cond                               ; die mit define1 definiert wurden
           ( (eq? (car form) 'quote1) (cadr form) )
           ( (eq? (car form) 'cond1) (eval-cond1 (cdr form) envlist) )
           ( (procedure? (lookup (car form) envlist))
               (apply1
                 (lookup (car form) envlist)
                 (ev-list (cdr form) envlist)
                 envlist)
           )
           ( else                           ; Auswertung  anderer
               (apply1                      ; primitiver Prozeduren
                  (car form)
                  (ev-list (cdr form) envlist)
                  envlist)
           )
         )
    )  ; Klausel
    ( else                                  ; Auswertung von Ausdruecken, die
        (apply 1                            ; als Wert eine Prozedur annehmen
           (car form)                       ; (insb. Lambda-Ausdruecke)
           (ev-list (cdr form) envlist)
           envlist)
    ) ; Klausel
  ) ;cond
```

```
)   ;define
```

Diese Prozedur enthält diverse Hilsprozeduren.

Algorithmus 3.2.4 *1.*
```
(define (apply1 fn lis envlist)
      (cond
        ( (atom? fn)
            (cond
               ( (eq? fn 'car1)   (caar lis) )
               ( (eq? fn 'cdr1)   (cdar lis) )
               ( (eq? fn 'cons1)  (cons (car lis) (cadr lis)) )
               ( (eq? fn 'atom1?) (atom? (car lis)) )
               ( (eq? fn 'eq1?)   (eq? (car lis) (cadr lis)) )
               ( else
                   (apply1 (eval1 fn envlist) lis envlist) )) )
               ; (fn ist ein Atom, dessen Wert eine der o.g. primitiven
               ; Prozeduren oder ein Lambda-Ausdruck ist)
        ( (eq? (car fn) 'lambda)
            (eval1
               (caddr fn)                          ; Prozedurrumpf
               (pairlis (cadr fn) lis envlist)) ) ; erweiterte Umgebung
      )
    )
```

2.
```
(define (eval-cond1 clist envlist)
    (cond
      ( (null? clist) '() )
      ( (eval1 (caar clist) envlist)
          (cond
             ( (null? (cdar clist)) (eval1 (caar clist) envlist) )
             ( else                 (eval1 (cadar clist) envlist) )) )
      ( else
          (eval-cond1 (cdr clist) envlist) )
  ))
```

3.
```
(define (ev-list lis envlist)
     (cond
       ( (null? lis) '() )
       ( else  (cons (eval1 (car lis) envlist) (ev-list (cdr lis) envlist)) ))
1 ]=> (define env (environment-bindings (the-environment)))

;Value: env

1 ]=> (eval1 1 env)

;Value: 1

1 ]=> (eval1 '(cons1 1 2) env)
```

```
;Value: (1 . 2)

1 ]=> (eval1 '(car1 (cons1 1 2)) env)

;Value: 1

1 ]=> (eval1 '(quote1 (1 2 3)) env)

;Value: (1 2 3)
```

4 Datengesteuerte Prozeduren — generische Arithmetik

4.1 Arithmetische Datentypen

Algorithmus 4.1.1 *Typzuweisungen.*

```
 (define (attach-type type contents)
  (cons type contents))
(define (type datum)
  (cond  ((pair? datum) (car datum))
         ((integer? datum) 'integer)
         ((real? datum) 'real)
         ((rational? datum) 'rational)
         ((complex? datum) 'complex)
         ((error "Bad typed datum -- TYPE" datum))))

(define (contents datum)
  (cond  ((pair? datum) (cdr datum))
         ((number? datum) datum)
         ((error "Bad typed datum -- CONTENTS" datum))))
```

Es folgen einfache Beispiele von Mengendefinitionen durch derartige Typ-Deklarationen.

Algorithmus 4.1.2 *Komplexe Zahlen (Darstellung durch Polarkoordinaten)*

```
(define (polar? z)
  (eq? (type z) 'polar))

(define (make-polar r a)
  (attach-type 'polar (cons r a)))
```

Algorithmus 4.1.3 *Gaußsche Zahlen*

```
(define (gaussian? z)
  (eq? (type z) 'gaussian))

(define (make-gaussian x y)
  (attach-type 'gaussian (cons x y)))
```

Algorithmus 4.1.4 *Restklassen*

```
(define (residue? x)
  (eq? (type x) 'residue))

(define (make-residue m a)
  (attach-type 'residue (cons m a)))
```

4.2 Relationen und algebraische Operationen

Zum Begriff einer Menge gehört die Festlegung einer Äquivalenzrelation *Gleichheit* auf dieser Menge. Wenn dies auch zunächst als selbstverständliche Forderung erscheint, verbirgt sich hier dennoch eines der Kernprobleme der Computer-Algebra. Wann sind zwei algebraische Terme gleich: wenn sie genau gleich aufgeschrieben sind, wenn sie sich durch gewisse festzulegende Operationen ineinander überführen lassen, oder wenn sie bei gewissen Ersetzungen jeweils den gleichen numerischen Wert ergeben? Hier sind genaue Definitionen zu treffendie entsprechenden Prädikate können ganz einfach oder auch äußerst schwierig bzw. unlösbar sein.
Die *Ungleichheit* ist, streng genommen, ein von der Gleichheit unabhängiger Begriff. In den meisten Fällen kann man jedoch die Ungleichheit als die Negation der Gleichheit einführen.
Neben der Gleichheit sind andere Relationen wie z.B. $<$ zu definieren sowie die algebraischen Operationen Addition, Subtraktion, Multiplikation, Division bzw. Division mit Rest sowie weitere Prozeduren wie z.B. komplexe Konjugation, Zuordnung von Real- und Imaginärteil bzw. von Zähler und Nenner usw.
Alle diese Definitionen werden in Abhängigkeit vom Typ der Variablen definiert. Man hat also Prozeduren der folgenden Art:
Algorithmus 4.2.1 *Nullstellige Operationen*

```
(define (zero-integer x) '0)

(define (identity-integer x) '1)

(define (zero-real x) '0.0)

(define (identity-real x) '1.0)

(define (zero-rational x) '0.0)

(define (identity-rational x) '1.0)

(define (zero-complex x) '0.0)

(define (identity-complex x) '1.0)

(define (zero-polar x) (make-polar '0 '0))

(define (identity-polar x) (make-polar '1 '0))

(define (zero-gaussian x) (make-gaussian '0 '0))

(define (identity-gaussian x) (make-gaussian '1 '0))
```

```
(define (zero-residue m) (make-residue m '0))

(define (identity-residue m) (make-residue m '1))
```

Algorithmus 4.2.2 *Einstellige Operationen*

```
Typ integer:
============

(define (zero-integer? n) (zero? n) )

(define (unit-integer? n) (or (= n 1) ( = n (- 1))) )

(define (positive-integer? n) (positive? n) )

(define (negative-integer? n) (negative? n) )

(define (minus-integer n) (- n) )

(define (inverse-integer n)
  (cond
    ((unit-integer? n) n)
    ((error "Inverse does not exist -- INVERSE-INTEGER" n))) )

(define (deg-euclid-integer n) (abs n))

(define (normform-most-types x)
  x)

Typ real:
========

(define (zero-real? x) (zero? x) )

(define (unit-real? x) (not (zero? x)) )

(define (positive-real? x) (positive? x))

(define (negative-real? x) (negative? x))

(define (minus-real x) (- x) )

(define (inverse-real x)
  (cond
    ((unit-real? x) (/ 1 x))
    ((error "Inverse does not exist -- INVERSE-REAL" x))) )
```

```
Typ rational:
=============

(define (numerator-rational x) (numerator x) )

(define (denominator-rational x) (denominator x) )

(define (zero-rational? x) (zero? x) )

(define (unit-rational? x) (not (zero? x)) )

(define (positive-rational? x) (positive? x))

(define (negative-rational? x) (negative? x))

(define (minus-rational x) (- x) )

(define (inverse-rational x)
  (cond
    ((unit-rational? x) (/ 1 x))
    ((error "Inverse does not exist -- INVERSE-REAL" x))) )


Typ complex:
============

(define (real-part-complex x) (real-part x) )

(define (imag-part-complex x) (imag-part x) )

(define (zero-complex? x) (zero? x) )

(define (unit-complex? x) (not (zero? x)) )

(define (magnitude-complex x) (magnitude x) )

(define (angle-complex x) (angle x) )

(define (minus-complex x) (- x) )

(define (inverse-complex x)
  (cond
```

```
    ((unit-complex? x) (/ 1 x))
    ((error "Inverse does not exist -- INVERSE-REAL" x))) )

(define (conjugate-complex x) (conjugate x) )

(define (polar-complex x)
  (make-polar (magnitude x) (angle x)) )


Typ polar:
==========

(define (magnitude-polar z) (car (contents z)))

(define (angle-polar z) (cdr (contents z)))

(define (zero-polar? z) (zero? (magnitude-polar z)) )

(define (unit-polar? z) (not (zero-polar? z)) )

(define (real-part-polar z)
  (* (car (contents z)) (cos (cdr (contents z)))))

(define (imag-part-polar z)
  (* (car (contents z)) (sin (cdr (contents z)))))

(define (minus-polar z)
  (make-polar (magnitude-polar z) (+ pi (angle-polar z))) )

(define (inverse-polar z)
  (cond
    ( (unit-polar? z)
        (make-polar (/ 1 (magnitude-polar z)) (- (angle-polar z))) )
    ( (error "Inverse does not exist -- INVERSE-POLAR" z) )) )

(define (normform-polar z)
  (cond
    ( (zero-polar? z) (make-polar 0 0) )
    ( (make-polar
        (magnitude-polar z)
        (* 2 pi (mantisse (/ angle-polar 2 pi)))) )) )

(define (mantisse x)
  (- x (floor x)))
```

```
(define (conjugate-polar z)
  (make-polar (magnitude-polar z) (- (angle-polar z))) )

(define (complex-polar z)
  (make-complex (* (magnitude-polar z) (cos (angle-polar z)))
                  (* (magnitude-polar z) (sin (angle-polar z)))) )

Typ gaussian:
============

(define (zero-gaussian? z)
  (and (zero? (real-part-gaussian z))
       (zero? (imag-part-gaussian z))))

(define (magnitude-gaussian z)
   (let ((x (real-part-gaussian z)) (y (imag-part-gaussian z)))
   (magnitude-complex (+ x (* 0+i z)))))

(define (angle-gaussian z)
   (let ((x (real-part-gaussian z)) (y (imag-part-gaussian z)))
   (angle-complex (+ x (* 0+i z)))))

(define (polar-gaussian z)
   (let ((x (real-part-gaussian z)) (y (imag-part-gaussian z)))
   (polar-complex (+ x (* 0+i z)))))

(define (unit-gaussian? z)
  (= 1 (magnitude-complex z)))

(define (real-part-gaussian z) (car (contents z)))

(define (imag-part-gaussian z) (cdr (contents z)))

(define (minus-gaussian z)
  (make-gaussian (- (real-part-gaussian z))
                 (- (imag-part-gaussian z))) )

(define (inverse-gaussian z)
  (cond
    ( (unit-gaussian? z)
        ( (lambda (d)
                  (make-gaussian (/    (real-part-gaussian z)  d)
                                 (/ (- (imag-part-gaussian z)) d)
                  )
          )
```

```
          (+ (square (real-part-gaussian z))
             (square (imag-part-gaussianz))
          )
        )
     )
     ( (error "Inverse does not exist -- INVERSE-GAUSSIAN" z) )) )

(define (conjugate-gaussian z)
  (make-gaussian (real-part-gaussian z)
                 (- (imag-part-gaussian z))) )

(define (deg-euclid-gaussian z)
  (+ (square (real-part-gaussian z)) (square (imag-part-gaussian z))) )


Typ residue:
===========

(define (modulus-residue x) (car (contents x)) )

(define (representative-residue x) (cdr (contents x)) )

(define (normform-residue x)
  (make-residue (modulus-residue x)
                (modulo (representative-residue x) (modulus-residue x))) )

(define (zero-residue? x)
  (zero? (modulo (representative-residue x) (modulus-residue x))) )

(define (unit-residue? x)
  (= 1 (gcd (modulus-residue x) (representative-residue x))) )

(define (minus-residue x) (make-residue (modulus-residue x)
  (- (representative-residue x))) )

(define (inverse-residue x) (error "Noch nicht definiert -- INVERSE-RESIDUE" x))
```

Die jetzt an sich w—nschenswerte Prozedur (inverse-residue x) verwendet
den euklidischen Algorithmus und wird deshalb später eingef—hrt.

Algorithmus 4.2.3 *Gleichheits-Prädikate*

```
(define (equ-integer? m n) (= m n))

(define (equ-real? x y) (= x y))

(define (equ-rational? x y) (= x y))
```

```
(define (equ-complex? x y) (= x y))

(define (equ-gaussian? x y)
  (and (= (real-part-gaussian x) (real-part-gaussian y))
       (= (imag-part-gaussian x) (imag-part-gaussian y))))

(define (equ-polar? z w)
 (define (integer-real? x) (= x (floor x)))
 (cond
    ( (zero-polar? z) (zero-polar? w) )
    ( else            (and (= (car (contents z)) (car (contents w)))
                           (integer-real?
                              (/ (- (cdr (contents z)) (cdr (contents w)))
                                 2
                                 pi
                              )
                           )
                      )
    )
  )
)

(define (equ-residue? x y)
  (zero?
    (remainder
       (- (cdr (contents x)) (cdr (contents y)))
       (car (contents x)))))
```

Bei dieser Prozedur wird vorausgesetzt, da_ beide Restklassen nach dem gleichen Modul gebildet sind - das ist beim Aufruf der Prozedur zu beachten.

Algorithmus 4.2.4 *Ordnungs-Prädikate: Man kann Definitionen der folgenden Art treffen:*

```
(define (<-integer? m n)  (< m n))

(define (<=-integer? m n)  (<= m n))

(define (>-integer? m n)  (> m n))

(define (>=-integer? m n)  (>= m n))

(define (<-real? x y) (< x y))

(define (<=-real? x y)  (<= x y))
```

```
(define (>-real? x y)  (> x y))

(define (>=-real? x y) (>= x y))

(define (<-rational? x y) (< x y) )

(define (<=-rational? x y) (<= x y) )

(define (>-rational? x y) (> x y) )

(define (>=-rational? x y) (>= x y) )
```

Algorithmus 4.2.5 *Algebraische Operatoren*
Für die Typen integer, real, rational *und* complex *werden, soweit möglich, die gewöhnlichen Scheme-Prozeduren verwendet.*

```
Type integer:
=============

(define (quotient-integer a b) (quotient a b))

(define (modulo-integer a b)
  (modulo a b))

type polar:
===========

(define (+polar z w)
  (polar-complex
    (+complex (complex-polar z) (complex-polar w))) )

(define (-polar z w)
  (polar-complex
    (-complex (complex-polar z) (complex-polar w))) )

(define (*polar z w)
  (make-polar
    (* (magnitude-polar z) (magnitude-polar w))
    (+ (angle-polar z) (angle-polar w))) )

(define (/polar z w)
  (*polar z (inverse-polar w)) )

type gaussian:
================
```

```
(define (+gaussian z w)
  (make-gaussian (+ (real-part-gaussian z) (real-part-gaussian w))
                 (+ (imag-part-gaussian z) (imag-part-gaussian w))) )

(define (-gaussian z w)
  (make-gaussian (- (real-part-gaussian z) (real-part-gaussian w))
                 (- (imag-part-gaussian z) (imag-part-gaussian w))) )

(define (*gaussian z w)
  (make-gaussian
    (- (* (real-part-gaussian z) (real-part-gaussian w))
       (* (imag-part-gaussian z) (imag-part-gaussian w)))
    (+ (* (real-part-gaussian z) (imag-part-gaussian w))
       (* (imag-part-gaussian z) (real-part-gaussian w)))) )

(define (/gaussian z w)
  (*gaussian z (inverse-gaussian w)) )

(define (quotient-gaussian z w)
  (define (next-gaussian-complex z)
    (make-gaussian
      (round (real-part-complex z))
      (round (imag-part-complex z))) )
  (next-gaussian-complex
    (/complex
      (gaussian->complex z)
      (gaussian->complex w))) )


Hier werden Uebergangsfunktionen benutzt, die spaeter definiert werden.

type residue:
============

(define (+residue x y)
  (make-residue
    (modulus-residue x)
    (+ (representative-residue x) (representative-residue y))) )

(define (-residue x y)
  (make-residue
    (modulus-residue x)
    (- (representative-residue x) (representative-residue y))) )

(define (*residue x y)
```

```
  (make-residue
    (modulus-residue x)
    (* (representative-residue x) (representative-residue y))) )

(define (/residue x y)
  (*residue x (inverse-residue y)) )
```

4.3 Generische Operatoren

Algorithmus 4.3.1 *Die Prozeduren* **put** *und* **get**

```
(define operation-table (make-table))

(define (put type op proc)
  (insert! type op proc operation-table))

(define (get type op)
  (lookup type op operation-table))
```

Algorithmus 4.3.2 *Tabelleneinträge*

```
 type integer:
 ============

(put 'integer 'zero zero-integer)

(put 'integer 'identity identity-integer)

(put 'integer 'zero? zero-integer?)

(put 'integer 'unit? unit-integer?)

(put 'integer 'positive? positive-integer?)

(put 'integer 'negative? negative-integer?)

(put 'integer 'minus minus-integer)

(put 'integer 'inverse inverse-integer)

(put 'integer 'normform normform-most-types)

(put 'integer 'equ? equ-integer?)

(put 'integer '<? <-integer?)
```

```
(put 'integer '<=? <=-integer?)

(put 'integer '>? >-integer?)

(put 'integer '>=? >=-integer?)

(put 'integer '+ +)

(put 'integer '- -)

(put 'integer '* *)

(put 'integer '/ /integer)

 type real:
 =========

(put 'real 'zero zero-real)

(put 'real 'identity identity-real)

(put 'real 'zero? zero-real?)

(put 'real 'unit? unit-real?)

(put 'real 'positive? positive-real?)

(put 'real 'negative? negative-real?)

(put 'real 'minus minus-real)

(put 'real 'inverse inverse-real)

(put 'real 'normform normform-most-types)

(put 'real 'equ? equ-real?)

(put 'real '<? <-real?)

(put 'real '<=? <=-real?)

(put 'real '>? >-real?)

(put 'real '>=? >=-real?)
```

```
(put 'real '+ +)

(put 'real '- -)

(put 'real '* *)

(put 'real '/ /)

 type rational:
 =========

(put 'rational 'zero zero-rational)

(put 'rational 'identity identity-rational)

(put 'rational 'numer numerator-rational)

(put 'rational 'denom denominator-rational)

(put 'rational 'normform normform-most-types)

(put 'rational 'zero? zero-rational?)

(put 'rational 'unit? unit-rational?)

(put 'rational 'positive? positive-rational?)

(put 'rational 'negative? negative-rational?)

(put 'rational 'minus minus-rational)

(put 'rational 'inverse inverse-rational)

(put 'rational 'equ? equ-rational?)

(put 'rational '<? <-rational?)

(put 'rational '<=? <=-rational?)

(put 'rational '>? >-rational?)

(put 'rational '>=? >=-rational?)

(put 'rational '+ + )
```

```
(put 'rational '- - )

(put 'rational '* * )

(put 'rational '/ /  )

 type complex:
 ===============

(put 'complex 'zero zero-complex)

(put 'complex 'identity identity-complex)

(put 'complex 'real-part real-part-complex)

(put 'complex 'imag-part imag-part-complex)

(put 'complex 'zero? zero-complex?)

(put 'complex 'unit? unit-complex?)

(put 'complex 'magnitude magnitude-complex)

(put 'complex 'angle angle-complex)

(put 'complex 'minus minus-complex)

(put 'complex 'inverse inverse-complex)

(put 'complex 'normform normform-most-types)

(put 'complex 'conjugate conjugate-complex)

(put 'complex 'polar polar-complex)

(put 'complex 'equ? equ-complex?)

(put 'complex '+ + )

(put 'complex '- - )

(put 'complex '* * )

(put 'complex '/ / )
```

```
 type polar:
 ==========

(put 'polar 'zero zero-polar)

(put 'polar 'identity identity-polar)

(put 'polar 'real-part real-part-polar)

(put 'polar 'imag-part imag-part-polar)

(put 'polar 'zero? zero-polar?)

(put 'polar 'unit? unit-polar?)

(put 'polar 'magnitude magnitude-polar)

(put 'polar 'angle angle-polar)

(put 'polar 'minus minus-polar)

(put 'polar 'inverse inverse-polar)

(put 'polar 'normform normform-polar)

(put 'polar 'conjugate conjugate-polar)

(put 'polar 'complex complex-polar)

(put 'polar 'equ? equ-polar?)

(put 'polar '+ +polar)

(put 'polar '- -polar)

(put 'polar '* *polar)

(put 'polar '/ /polar)

 type gaussian:
 ===============

(put 'gaussian 'zero zero-gaussian)

(put 'gaussian 'identity identity-gaussian)
```

```
(put 'gaussian 'real-part real-part-gaussian)

(put 'gaussian 'imag-part imag-part-gaussian)

(put 'gaussian 'zero? zero-gaussian?)

(put 'gaussian 'unit? unit-gaussian?)

(put 'gaussian 'magnitude magnitude-gaussian)

(put 'gaussian 'angle angle-gaussian)

(put 'gaussian 'minus minus-gaussian)

(put 'gaussian 'inverse inverse-gaussian)

(put 'gaussian 'normform normform-most-types)

(put 'gaussian 'conjugate conjugate-gaussian)

(put 'gaussian 'polar polar-gaussian)

(put 'gaussian 'equ? equ-gaussian?)

(put 'gaussian '+ +gaussian)

(put 'gaussian '- -gaussian)

(put 'gaussian '* *gaussian)

(put 'gaussian '/ /gaussian)

 type residue:
 ============

(put 'residue 'zero zero-residue)

(put 'residue 'identity identity-residue)

(put 'residue 'modulus modulus-residue)

(put 'residue 'representative representative-residue)

(put 'residue 'normform normform-residue)
```

```
(put 'residue 'zero? zero-residue?)

(put 'residue 'unit? unit-residue?)

(put 'residue 'minus minus-residue)

(put 'residue 'inverse inverse-residue)

(put 'residue 'equ? equ-residue?)

(put 'residue '+ +residue)

(put 'residue '- -residue)

(put 'residue '* *residue)

(put 'residue '/ /residue)
```

Mit Hilfe der eingeführten Tabelle können *generische Operatoren* **add, sub,** ... eingeführt werden, die je nach Typ der Argumente passend definiert sind. Bei nullstelligen Operatoren muß der Typ explizit angegeben sein.
Algorithmus 4.3.3 *Zur Definition generischer Operatoren*

```
 Nullstellige Operatoren:
 ========================

(define (operate-0 op type)
  (let ((proc (get type op)))
    (cond
      ( (not (null? proc)) (proc type) )
      ( else (error "Operator undefined for this type -- OPERATE-0"
                    (list op type)) )) ))

 F|r den Typ residue ist eine Sonderbehandlung notwendig.

 Einstellige Operatoren:
 =======================

(define (operate-1 op arg)
  (let ((proc (get (type arg) op)))
    (cond
      ( (not (null? proc)) (proc arg) )
      ( else (error "Operator undefined for this type -- OPERATE-1"
                    (list op arg)) )) ))
```

```
Zweistellige Operatoren:
========================

(define (operate-2 op arg1 arg2)
  (let ((t1 (type arg1)))
    (cond
      ( (eq? t1 (type arg2))
          (let ((proc (get t1 op)))
            (cond
              ( (not (null? proc)) (proc arg1 arg2) )
              ( else (error "Operator undefined for this type -- OPERATE-2"
                            (list op arg1 arg2)) )) ) )
       ( else (error "Operands not of same type -- OPERATE-2"
                     (list op arg1 arg2)) ))) )


Die folgende Definition geschieht im Vorgriff auf die Einfuehrung von
Uebergangsprozeduren. Es wird au_erdem eine Sonderbehandlung fuer den
Uebergang zwischen Polynomen und den Elementen eines Quotientenringes
vorgesehen.

(define (operate-2 op arg1 arg2)
  (let ((t1 (type arg1)) (t2 (type arg2)))
    (cond
      ( (eq? t1 t2)
          (let ((proc (get t1 op)))
            (cond
              ( (not (null? proc)) (proc arg1 arg2) )
              ( else (error "Operator undefined for this type -- OPERATE-2"
                            (list op arg1 arg2)) )) ) )
      ( (cond
          ( (and (eq? t1 'poly) (eq? t2 'quot))
            (let (( arg1 (g-normform arg1) )
                  ( arg2 (g-normform arg2) ))
            (cond
              ( (or (poly? (g-numer arg2)) (poly? (g-denom arg2)))
                (let (( t1->t2 (get-coercion t1 t2) ))
                (cond
                  ( (not (null? t1->t2)) (operate-2 op (t1->t2 arg1) arg2) )
                  ( else (error "Operands not of same type -- OPERATE-2"
                                (list op arg1 arg2)) ))) )
              ( else
                (let (( t2->t1 (get-coercion t2 t1) ))
                (cond
```

```
                ( (not (null? t2->t1)) (operate-2 op arg1 (t2->t1 arg2)) )
                ( else (error "Operands not of same type -- OPERATE-2"
                              (list op arg1 arg2)) ))) ))) )
          ( (and (residue? arg1) (integer? arg2))
            (operate-2 op arg1 (make-residue (modulus-residue arg1) arg2)) )
          ( (and (residue? arg2) (integer? arg1))
            (operate-2 op (make-residue (modulus-residue arg2) arg1) arg2) )
          ( (let (( t1->t2 (get-coercion t1 t2) )
                  ( t2->t1 (get-coercion t2 t1) ))
            (cond
              ( (not (null? t1->t2)) (operate-2 op (t1->t2 arg1) arg2) )
              ( (not (null? t2->t1)) (operate-2 op arg1 (t2->t1 arg2)) )
              ( else (error "Operands not of same type -- OPERATE-2"
                            (list op arg1 arg2)) ))) ))) )
```

Mit Hilfe dieser Prozeduren können null-, ein- und zweistellige generische Operatoren eingeführt werden:

Algorithmus 4.3.4 `Nullstellige Operatoren:`

```
 ======================

(define (g-zero type) (operate-0 'zero type))

(define (g-identity type) (operate-0 'identity type))


 Einstellige Operatoren:
 ======================

(define (g-zero? x) (operate-1 'zero? x))

(define (g-unit? x) (operate-1 'unit? x))

(define (g-positive? x) (operate-1 'positive? x))

(define (g-negative? x) (operate-1 'negative? x))

(define (g-minus x) (operate-1 'minus x))

(define (g-inverse x) (operate-1 'inverse x))

(define (g-numer x) (operate-1 'numer x))

(define (g-denom x) (operate-1 'denom x))

(define (g-normform x) (operate-1 'normform x))
```

```
(define (g-real-part z) (operate-1 'real-part z))

(define (g-imag-part z) (operate-1 'imag-part z))

(define (g-magnitude z) (operate-1 'magnitude z))

(define (g-angle z) (operate-1 'angle z))

(define (g-conjugate z) (operate-1 'conjugate z))

(define (g-polar z) (operate-1 'polar z))

(define (g-complex z) (operate-1 'complex z))

(define (g-modulus x) (operate-1 'modulus x))

(define (g-representative x) (operate-1 'representative x))

 Zweistellige Operatoren:
 ========================

(define (g-= x y) (operate-2 '= x y))

(define (g-< x y) (operate-2 '< x y))

(define (g-<= x y) (operate-2 '<= x y))

(define (g-> x y) (operate-2 '> x y))


(define (g->= x y) (operate-2 '>= x y))

(define (g-+ x y) (operate-2 '+ x y))

(define (g-- x y) (operate-2 '- x y))

(define (g-* x y) (operate-2 '* x y))

(define (g-/ x y) (operate-2 '/ x y))
```

4.4 Operatoren für gemischte Typen

Das Problem, Operatoren für unterschiedliche Typen zu definieren (z.B. Addition einer reellen und einer komplexen Zahl), kann durch Einführung von geeigneten Übergangstabellen gelöst werden, für die, in Analogie zu den Prozeduren **put** und **get**, Prozeduren **put-coercion** und **get-coercion** definiert werden mit folgendes Syntax:

```
(put-coercion type1 type2 proc)

(get-coercion type1 type2)
```

Durch diese Prozeduren werden Übergangsprozeduren der folgenden Art in eine entsprechende Tabelle eingetragen bzw. aus dieser Tabelle ausgelesen:
Algorithmus 4.4.1 *Übergangsprozeduren*

```
(define (integer->rational n) n)

(define (integer->real n) n)

(define (integer->gaussian n) (make-gaussian n 0))

(define (integer->complex n) n)

(define (integer->polar n) (make-polar n 0))

(define (rational->real n) n)

(define (rational->complex n) n)

(define (rational->polar n)
  (make-polar (/ (numerator-rational n) (denominator-rational n)) 0))

(define (real->complex x) x)

(define (real->polar x) (make-polar x 0))

(define (gaussian->complex x)
 (+ (real-part-complex x) (* 0+i (imag-part-complex x))) )
```

Für die Übergangsfunktionen wird eine gesonderte Tabelle angelegt:
Algorithmus 4.4.2 `(define coercion-table (make-table))`

```
(define (put-coercion type op proc)
  (insert! type op proc coercion-table))

(define (get-coercion type op)
  (lookup type op coercion-table))
```

Algorithmus 4.4.3 *Einfügen Übergangsprozeduren in eine Tabelle*

```
(put-coercion 'integer 'rat integer->rational)

(put-coercion 'integer 'real integer->real)

(put-coercion 'integer 'gaussian integer->gaussian)

(put-coercion 'integer 'complex integer->complex)

(put-coercion 'integer 'polar integer->polar)

(put-coercion 'rational 'real rational->real)

(put-coercion 'rational 'complex rational->complex)

(put-coercion 'rational 'polar rational->polar)

(put-coercion 'real 'complex real->complex)

(put-coercion 'real 'polar real->polar)

(put-coercion 'complex 'polar polar-complex)

(put-coercion 'polar 'complex complex-polar)

(put-coercion 'gaussian 'complex gaussian->complex)
```

5 Quotientenringe und -körper

5.1 Einführung von Quotientenringen

Bei der Einführung neuer algebraischer Strukturen und Algorithmen werden nach Möglichkeit die generischen Operatoren verwendet. Dadurch erhält man allgemeingültige Resultate.
Wir betrachten hier nur die sog. *totalen Quotientenringe*, das sind die Brüche $\frac{n}{d}$, wobei d ein beliebiger Nichtnullteiler sein darf. Für Integritätsringe ist der totale Quotientenring ein Körper, der *Quotientenkörper* des Rings.
Für die betrachteten Ringe müssen Prädikate angegeben werden, mit denen man Nullteiler von Nichtnullteilern unterscheiden kann. Es wird benutzt, daß Einheiten keine Nullteiler sind. Daher kann für Körper auf die Prädikate für Einheiten zurückgegriffen werden. Das gleiche gilt für den Restklassenring Z_m, in dem ebenfalls jeder Nichtnullteiler Einheit ist.
Algorithmus 5.1.1 *Nullteiler*

```
(define (zero-divisor-integer? n)
  (g-zero? n))

(define (zero-divisor-gaussian? z)
  (g-zero? z))

(define (zero-divisor-poly? f)
  (g-zero? f))

(put 'integer 'zero-divisor? zero-divisor-integer?)

(put 'gaussian 'zero-divisor? zero-divisor-gaussian?)

(put 'poly 'zero-divisor? zero-divisor-poly?)

(define (g-zero-divisor? x) (operate-1 'zero-divisor? x))
```

Algorithmus 5.1.2

```
(define (quot? x)
  (eq? (type x) 'quot))

(define (make-quot n d)
  (attach-type 'quot (cons n d)))

Nullstellige Operationen:
=========================

(define (zero-quot type)
  (make-quot (g-zero type) (g-identity type)))

(define (identity-quot type)
  (make-quot (g-identity type) (g-identity type)))
```

```
(put 'quot 'zero zero-quot)

(put 'quot 'identity identity-quot)

Einstellige Operationen:
========================

(define (numer-quot x)
  (car (contents x)))

(define (denom-quot x)
  (cdr (contents x)))

(define (zero-quot? x)
  (g-zero? (numer-quot x)))

(define (unit-quot? x)
  (cond
    ( (g-unit? (numer-quot x)) )
    ( (g-zero? (numer-quot x)) nil )
    ( (not (g-zero-divisor? (numer-quot x))) )
    ( else nil )))

(define (minus-quot x)
  (make-quot (g-minus (numer-quot x)) (denom-quot x)))

(define (inverse-quot x)
  (cond
    ( (unit-quot? x) (make-quot (denom-quot x) (numer-quot x)) )
    ( (error "Inverse does not exist -- INVERSE-QUOT" x) )) )

Im folgenden Algorithmus werden gemeinsame Faktoren von Zaehler und
Nenner gekuerzt, wenn ein Algorithmus gcd definiert ist:

(define (normform-quot x)
  (let ( (t-mod (get (type (numer-quot x)) 'modulo)) )
    (let
      ( (xx
          (cond
            ( (null? t-mod) x )
            ( (let ( (d (g-gcd (numer-quot x) (denom-quot x))) )
                (make-quot
                  (g-/ (numer-quot x) d)
                  (g-/ (denom-quot x) d))) ))) )
```

```
       (cond
         ( (g-unit? (denom-quot xx))
             (g-normform (g-/ (numer-quot xx) (denom-quot xx))) )
         ( (make-quot
             (g-normform (numer-quot xx))
             (g-normform (denom-quot xx))) )))  )

Zweistellige Operationen:
=========================

(define (equ-quot? x y)
  (g-equ
    (g-* (numer-quot x) (denom-quot y))
    (g-* (denom-quot x) (numer-quot y))))

(define (+quot x y)
  (make-quot
    (g-+
      (g-* (numer-quot x) (denom-quot y))
      (g-* (denom-quot x) (numer-quot y)))
    (g-*
      (denom-quot x) (denom-quot y))))

(define (-quot x y)
  (+quot x (minus-quot x)))

(define (*quot x y)
  (make-quot
    (g-* (numer-quot x) (numer-quot y))
    (g-* (denom-quot x) (denom-quot y))) )

(define (/quot x y)
  (cond
    ( (unit-quot? y)
        (make-quot
          (g-* (numer-quot x) (denom-quot y))
          (g-* (denom-quot x) (numer-quot y))) )
    ( (error "Division not possible - /QUOT" (list x y)) )) )


Eintrag der Prozeduren in die Operatorentabelle:
================================================

(put 'quot 'zero zero-quot)
```

```
(put 'quot 'identity identity-quot)

(put 'quot 'numer numer-quot)

(put 'quot 'denom denom-quot)

(put 'quot 'zero? zero-quot?)

(put 'quot 'unit? unit-quot?)

(put 'quot 'minus minus-quot)

(put 'quot 'inverse inverse-quot)

(put 'quot 'normform normform-quot)

(put 'quot 'equ? equ-quot?)

(put 'quot '+ +quot)

(put 'quot '- -quot)

(put 'quot '* *quot)

(put 'quot '/ /quot)
```

Algorithmus 5.1.3 *Übergangsprozeduren*

```
(define (most-types->quot x)
  (make-quot x (g-identity (type x))))

(define (rational->quot x)
  (make-quot (numer-rat x) (denom-rat x)))

(define (poly->quot f)
  (make-quot f (g-identity (type-of-coefficients f))) )

(define (quot->poly x)
  (make-poly (list (make-monom x nil)) nil))

(put-coercion 'integer 'quot most-types->quot)

(put-coercion 'rational 'quot rational->quot)

(put-coercion 'real 'quot most-types->quot)

(put-coercion 'complex 'quot most-types->quot)
```

```
(put-coercion 'polar 'quot most-types->quot)

(put-coercion 'gaussian 'quot most-types->quot)

(put-coercion 'residue 'quot most-types->quot)

(put-coercion 'poly 'quot poly->quot)

(put-coercion 'quot 'poly quot->poly)
```

Man kann jetzt die Divisionsalgorithmen für Ringe neu definieren: Ist der Divisor keine Einheit, aber Nichtnullteiler, soll der Quotient im Quotientenring gebildet werden:

Algorithmus 5.1.4 *Erweitere Definition von Divisionsalgorithmen*

```
(define (//all-types x y)
  (cond
    ( (g-unit? y) (g-/ x y) )
    ( (g-zero? y)
        (error "Division by zero - //ALL-TYPES" (list x y)) )
    ( (not (g-zero-divisor? y))
        (g-/
          ((get-coercion (type x) 'quot) x)
          ((get-coercion (type y) 'quot) y)) )
    ( (error "Division by zero-divisor - //ALL-TYPES" (list x y)) )) )

(define (quot-inverse-most-types x)
  (//all-types 1 x))

(define (quot-inverse-poly f)
  (cond
    ( (zero-poly? f)
        (error "Division by zero -- QUOT-INVERSE-POLY" f) )
    ( (//all-types 1 f) )) )

(put 'integer '// //all-types)

(put 'rational '// //all-types)

(put 'real '// //all-types)

(put 'complex '// //all-types)

(put 'polar '// //all-types)

(put 'gaussian '// //all-types)
```

```
(put 'residue '// //all-types)

(put 'poly '// //all-types)

(put 'quot '// //all-types)

(put 'integer 'quot-inverse quot-inverse-most-types )

(put 'rational 'quot-inverse quot-inverse-most-types )

(put 'real 'quot-inverse quot-inverse-most-types )

(put 'complex 'quot-inverse quot-inverse-most-types )

(put 'polar 'quot-inverse quot-inverse-most-types )

(put 'gaussian 'quot-inverse quot-inverse-most-types )

(put 'residue 'quot-inverse quot-inverse-most-types )

(put 'poly 'quot-inverse quot-inverse-poly)

(put 'quot 'quot-inverse quot-inverse-most-types )

(define (g-// x y) (operate-2 '// x y))

(define (g-quot-inverse x) (operate-1 'quot-inverse x))
```

6 Euklidische Ringe

6.1 Division mit Rest

Ein Ring R heißt *euklidisch*, wenn es eine Abbildung $d : R \to N_0$ gibt, so daß gilt:
Für alle $a, b \in R,\ b \neq 0$ gibt es $q, r \in R$ mit $d(r) < d(b)$ und $a = bq + r$.
Algorithmus 6.1.1 *Division mit Rest für ganze Zahlen und ganze Gauss-Zahlen*

```
Type integer:
============
\bv
(define (deg-euclid-integer n)
  (abs n))

(define (quotient-integer a b)
  (quotient a b))

(define (modulo-integer a b)
  (modulo a b))

Type gaussian:
=============


(define (deg-euclid-gaussian z)
  (+ (square (g-real-part z)) (square (g-imag-part z))) )

(define (modulo-gaussian z w)
  (g-- z (g-* (g-quotient z w) w)))

Tabelleneintraege:
=================

(put 'integer 'deg-euclid deg-euclid-integer)

(put 'integer 'quotient quotient-integer)

(put 'integer 'modulo modulo-integer)

(put 'gaussian 'deg-euclid deg-euclid-gaussian)

(put 'gaussian 'quotient quotient-gaussian)

(put 'gaussian 'modulo modulo-gaussian)
```

```
(define (g-deg-euclid x) (operate-1 'deg-euclid x))

(define (g-quotient x y) (operate-2 'quotient x y))

(define (g-modulo x y) (operate-2 'modulo x y))
```

Zusammenfassung:

```
(define (g-div x y)
  (let ( (q (g-quotient x y)) )
    (cons q (g-- x (g-* q y)))) )
```

6.2 Der euklidische Algorithmus

Wir definieren eine Prozedur mit der Syntax

`(g-euclid a b)`
welche als Argumente zwei beliebige Elemente a, b eines euklidischen Rings hat und als Wert eine Liste (dst) liefert mit $d = \gcd(a, b$ und $d = sa + tb$.

Algorithmus 6.2.1 *Euklidischer Algorithmus*

```
(define (g-euclid a b)
  (cond
    ( (g-zero? b) (list a 1 0) )
    ( (let ( (div (g-div a b)) )
        (let ( (eucl (g-euclid b (cdr div))) )
          (list
            (car eucl)
            (caddr eucl)
            (g-- (cadr eucl) (g-* (car div) (caddr eucl)))))) )) )

(define (g-norm-euclid a b)
  (let ( (eucl (g-euclid a b)) )
    (cond
      ( (g-unit? (car eucl))
          (let ( (inv (g-inverse (car eucl))) )
            (list
              1
              (g-normform (g-* inv (cadr eucl)))
              (g-normform (g-* inv (caddr eucl))))) )
      ( else
         (list
           (car eucl)
           (g-normform (cadr eucl))
           (g-normform (caddr eucl))) ))) )
```

```
(define (g-gcd a b)
  (cond
    ( (g-zero? b) a )
    ( (g-gcd b (g-modulo a b)) )) )

(define (g-norm-gcd a b)
  (g-norm-ideal (g-gcd a b)))

(define (g-gcd a b)
  (car (g-euclid a b)))
```

Als Anwendung können wir nun die Definition der Prozedur `inverse/residue` nachtragen:

Algorithmus 6.2.2 *Bildung des Inversen mod m*

```
(define (inverse-residue x)
  (cond
    ( (g-unit? x)
        (make-residue
          (g-modulus x)
          (cadr (g-euclid (g-representative x) (g-modulus x)))) )
    ( (error "Not existing inverse - INVERSE-RESIDUE" x) )) )
```

Algorithmus 6.2.3 *Beispiele*

```
(define z1 (make-gaussian 5 6))

(define z2 (make-gaussian 45 50))

(g-quotient z1 z2)

;Value: (gaussian 0 . 0)

1 ]=> (g-gcd z1 z2)

;Value: (gaussian 0 . 1)

1 ]=> (g-gcd (g-* 3 z1) (g-* 9 z2))

;Value: (gaussian -3 . 0)
```

7 Restklassenringe euklidischer Ringe

7.1 Ideale

Die Ideale werden durch eine (endliche) Basis explizit (als Liste) eingeführt. In einem euklidischen Ring ist jedes Ideal Hauptideal.

Algorithmus 7.1.1 *Ideale in euklidischen Ringen*

```
(define (principal-ideal-euclid ideal)
  (cond
    ( (null? ideal) nil )
    ( (null? (cdr ideal)) (normalize-principal-ideal ideal) )
    ( (principal-ideal-euclid
        (cons
          (g-gcd (car ideal) (cadr ideal))
          (cddr ideal))) )) )

(define (normalize-principal-ideal ideal)
  (list (g-norm-ideal (g-normform (car ideal)))) )

(define (norm-ideal-integer n)
  (abs n))

(define (norm-ideal-gaussian z)
  (make-imag-positive-gaussian (make-real-positive-gaussian z)))

(define (make-real-positive-gaussian z)
  (g-* (sign (g-real-part z)) z))

(define (make-imag-positive-gaussian z)
  (cond
    ( (negative? (g-imag-part z))
        (g-* (make-gaussian 0 1) z) )
    ( else z )) )

(define (norm-ideal-poly f)
  (cond
    ( (g-zero? f) f )
    ( (zero? (degree-poly f)) 1 )
    ( (g-* (g-quot-inverse (coefficient (h-term f))) f) )) )

(define (norm-ideal-other-types x)
  (cond
    ( (g-zero? x) 0 )
    ( (g-unit? x) 1 )
    ( (g-normform x) )) )

(put 'integer 'norm-ideal norm-ideal-integer)

(put 'gaussian 'norm-ideal norm-ideal-gaussian)

(put 'poly 'norm-ideal norm-ideal-poly)
```

```
(put 'rat 'norm-ideal norm-ideal-other-types)

(put 'real 'norm-ideal norm-ideal-other-types)

(put 'rectangular 'norm-ideal norm-ideal-other-types)

(put 'polar 'norm-ideal norm-ideal-other-types)

(put 'residue 'norm-ideal norm-ideal-other-types)

(put 'quot 'norm-ideal norm-ideal-other-types)

(define (g-norm-ideal x) (operate-1 'norm-ideal x))
```

7.2 Normalformen

Zunächst wird, in Analogie zur Einführung der Restklassenringe Z/mZ, eine generische Definition von Restklassenringen modulo einem Ideal gegeben. Die Ideale werden durch eine (endliche) Basis explizit (als Liste) eingeführt.
Die Algorithmen für Restklassen werden meist so ausgeführt, daß sie für beliebige Basen eines Ideals angewendet werden können, also nicht nur für Basen aus einem Element.

Algorithmus 7.2.1 *Restklassenringe*

```
(define (res? x)
  (eq? (type x) 'res))

(define (make-res ideal representative)
  (attach-type 'res (cons ideal representative)))

Einstellige Operationen:
========================

(define (ideal-res x)
  (car (contents x)) )

(define (representative-res x)
  (cdr (contents x)) )

(define (minus-res x)
  (make-res (ideal-res x) (g-minus (representative-res x))) )

Uebergang zu den Restklassen nach dem zugehoerigen Hauptideal
=============================================================
bei euklidischen Ringen:
```

```
======================

(define (principalize-euclid x)
  (make-res
    (principal-ideal-euclid (ideal-res x))
    (representative-res x)) )

Inversenbildung in Restklassenringen euklidischer Ringe:
========================================================

Hier werden Restklassen nach Hauptidealen vorausgesetzt.

(define (unit-res? x)
  (unit? (g-gcd (car (ideal-res x)) (representative-res x))) )

(define (inverse-res x)
  (let ( (eucl (g-norm-euclid (representative-res x) (car (ideal-res x)))) )
    (cond
      ( (= 1 (car eucl))
          (make-res (ideal-res x) (cadr eucl)) )
      ( (error "Non-existing inverse -- INVERSE-RES" x) ))) )


Zweistellige Operationen:
=========================

(define (+res x y)
  (make-res
    (ideal-res x)
    (g-+
      (representative-res x)
      (representative-res y))))

(define (-res x y)
  (+res x (minus-res y)))

(define (*res x y)
  (make-res
    (ideal-res x)
    (g-*
      (representative-res x)
      (representative-res y))))

Tabelleneintraege:
==================
```

```
(put 'res 'ideal ideal-res)

(put 'res 'representative representative-res)

(put 'res 'minus minus-res)

(put 'res 'unit? unit-res?)

(put 'res 'inverse inverse-res)

(put 'res '+ +res)

(put 'res '- -res)

(put 'res '* *res)
```

Definition 7.2.2 (Kanonische Normalformen) *Eine Abbildung* $x \mapsto normform(x)$ *heißt* kanonischer Normalform-Algorithmus*, wenn gilt:* $x \cong y \Longleftrightarrow normform(x) = normform(y)$.

Algorithmus 7.2.3 *Kanonische Normalform in Restklassenringen euklidischer Ringe*

```
(define (normform-euclid-res x)
  (make-res
    (ideal-res x)
    (g-normform
      (normalize-representative (ideal-res x) (representative-res x)))) )

(define (normalize-representative ideal rep)
  (cond
    ( (null? ideal) rep )
    ( (g-normform
        (normalize-representative
          (cdr ideal)
          (g-modulo rep (car ideal)))) )) )

(put 'res 'normform normform-euclid-res)

(define (g-normform-euclid x) (operate-1 'normform-euclid x))
```

7.3 Der Chinesische Restsatz

Sei R ein kommutativer Ring mit 1.

Definition 7.3.1 *Zwei Ideale* $A, B \subset R$ *heißen* relativ prim*, wenn gilt:* $A + B = R$.

Bemerkung 7.3.2 *a) Zwei Hauptideale* $A = aR, B = bR$ *in einem euklidi-*

schen Ring sind genau dann relativ prim, wenn gilt: $\gcd(a,b) = 1$. *(Dies ist eine Folgerung aus dem euklidischen Algorithmus.)*
b) Sei $\pi_A : R \to R/A$ *die kanonische Restklassenabbildung.* A, B *sind genau dann relativ prim, wenn gilt:* $\pi_A(B) = R/A$. *(Dies folgt aus* $\pi_A(B) = A + B/A$.*)*
Dies ist eine Folgerung aus dem euklidischen Algorithmus.
Lemma 7.3.3 *Seien* $A_1, \ldots, A_k$ *relativ prim. Dann gilt:*
a) $A_1 \ldots A_{k-1}$ *und* A_k *sind relativ prim. (Für diese Behauptung muß eigentlich nur vorausgesetzt werden:* A_i, A_k *relativ prim für* $i = 1, \ldots, k-1$*).*
b) $A_1 \ldots A_k = A_1 \cap \ldots \cap A_k$
Beweis:
a) Ist $\pi : R \to R/A_k$ die Restklassenabbildung, so gilt nach der Bemerkung oben: $\pi(A_i) = R/A$ für $i = 1, \ldots, k-1$. Es folgt: $\pi(A_1 \ldots A_{k-1}) = R/A \ldots R/A = R/A$.
b) Induktion nach k.
Für $k = 1$ ist nichts zu beweisen.
Sei $k > 1$. Induktionsvoraussetzung: $A_1 \ldots A_{k-1} = A_1 \cap \ldots \cap A_{k-1} =: A$ Zu zeigen: $AA_k = A \cap A_k$. Offensichtlich gilt „$\subset$“. Da $A = A_1 \ldots A_{k-1}$ und A_k relativ prim sind, gibt es Elemente $a \in A,\ a_k \in A_k$ mit $a + a_k = 1$. Für $x \in A \cap A_k$ folgt $x = ax + a_k x \in AA_k$daraus folgt „$\supset$“.
Satz 7.3.4 (Chinesischer Restsatz) *Seien* $A_1, \ldots A_k \subset R$ *paarweise teilerfremde Ideale, und sei* $A := A_1 A_2 \ldots A_k$. *Die Abbildung* $\chi : R \to (R/A_1) \times \ldots \times (R/A_k), \quad x \mapsto (x + A_1, \ldots, x + A_k)$ *ist surjektiv, und es gilt* $\ker(\chi) = A$.
Im Fall von Hauptidealen $A_i = m_i R$ macht der chinesische Restsatz eine Aussage über die Lösbarkeit simultaner Kongruenzen der folgenden Art: Sind $m_1, \ldots, m_k$ teilerfremde Elemente, so gibt es zu beliebig vorgegebenen Ringelementen $r_1, \ldots r_k$ eine (modulo $m := m_1 m_2 \ldots m_k$ eindeutig bestimmte) Lösung r für die Kongruenzen

$$x \equiv r_1 mod m_1$$

$$x \equiv r_2 mod m_2$$

$$\vdots$$

$$x \equiv r_k mod m_k$$

Beweis: Sei $i \in \{1, \ldots, k\}$. Nach Voraussetzung gibt es für jedes $j \in \{1, \ldots, k\}$ Elemente $x_{ij} \in A_i$ und $y_{ij} \in A_j$ mit $x_{ij} + yij = 1$. Sei $\delta_i := y_{i1} \ldots y_{i,i-1} y_{i,i+1} \ldots y_{ik}$. Dann gilt: $\delta_i \equiv 1 mod A_i$, $\delta_i \equiv 0 mod A_j$ für $j \neq i$. Dann ist

$$r := \delta_1 r_1 \ldots \delta_k r_k$$

die gewünschte Lösung. Für weitere Details des Beweises vgl. z.B. Schneider: Algebra und Elemente der Zahlentheorie.

Wir bringen hier zunächst den Satz in einer weiteren äquivalenten Formulierung und dann eine rekursive Konstruktion der Lösung:

Satz 7.3.5 (Chinesischer Restsatz - alternative Formulierung) *Seien $x_1 = r_1 + A_1 \in R/A_1, \ldots x_k = r_k + A_k \in R/A_k$ Restklassen mit paarweise teilerfremden Idealen $A_1, \ldots, A_k \subset R$, und sei $A := A_1 A_2 \ldots A_k$. Dann gibt es genau eine Restklasse $x = r + A \in R/A$, so daß für $i = 1, \ldots k$ gilt: $r + A_i = x_i$.*

Rekursiver Beweis: Im Fall $k = 1$ wähle $x := x_1$. Sei nun $k > 1$, und sei $x' = r' + A'$ die Lösung zu den Restklassen $x_1, \ldots, x_{k-1}$. Dann sind A_k und A' teilerfremd, es gibt also Elemente $a_k \in A$ und $a' \in A'$ mit $a_k + a' = 1$. Mit $A := A_k A'$, $r := r' + (r_k - r')a'$ folgt die Behauptung.

Algorithmus 7.3.6 *Chinesischer Rest-Algorithmus für euklidische Ringe. Argument: Eine Liste von Restklassen nach teilerfremden Hauptidealen. Wert: Die nach dem Chinesischen Restsatz ermittelte Restklasse.*

```
(define (g-cra xl)
  (cond
    ( (null? xl) nil )
    ( (null? (cdr xl)) (car xl) )
    ( (let ( (xx (g-cra (cdr xl))) )
        (let ( (eucl (g-norm-euclid
                       (car (ideal-res (car xl)))
                       (car (ideal-res xx)))) )
          (cond
            ( (= 1 (car eucl))
                (g-normform
                  (make-res
                    (list (g-* (car (ideal-res (car xl))) (car (ideal-res xx))))
                    (g-+
                      (representative-res xx)
                      (g-*
                        (g--
                          (representative-res (car xl))
                          (representative-res xx))
                      (g-*
                        (caddr eucl)
                        (car (ideal-res xx)))))))) )
             ( (error "Ideals not relatively prime -- G-CRA" xl) )))) )) )
(define (g-norm-cra xl)
  (g-normform (g-cra xl)))
```

Algorithmus 7.3.7 *Beispiele*

```
(define
  xl1
  (list
    (make-res (list 3) 1)
    (make-res (list 5) 3)
```

```
    (make-res (list 7) 0)
    (make-res (list 11) 10)) )

(define
  xl2
  (list
    (make-res (list g1) 1)
    (make-res (list g2) 3)))

(define
  xl3
  (list
    (make-res (list g1) g2)
    (make-res (list g2) g1)))
```

8 Polynomarithmetik

8.1 Darstellung von Polynomen

Ziel ist die Behandlung des Polynomrings $S = R[X_1, \ldots, X_n]$ über einem beliebigen Ring R. Die Polynome $f \in S$ können auf verschiedene Weise dargestellt werden:
Summe von Monomen mit Koeffizienten in R,
Summe von homogenen Polynomen,
Entwicklung nach X_n (mit entsprechender Darstellung der Koeffizienten $\in R[X_1, \ldots, X_{n-1}]$), also Betrachtung des Polynomrings $R[X_1][X_2], \ldots, [X_n]$.
Man kann ferner diverse Vereinfachungen der Bezeichnungsweise einführen, z.B. indem man die Variablennamen nicht explizit angibt, sondern Listen betrachtet, in denen nur die Koeffizienten und die Exponenten vorkommen. Oder man gibt nur Listen von Koeffizienten an, wobei von einer festen Ordnung der Monome ausgegangen wirdin diesem Fall müssen auch Koeffizienten $= 0$ aufgeführt werden.
Bei Polynomen in einer Unbestimmten bieten sich noch folgende Darstellungsmöglichkeiten an:
als Produkt von Linearfaktoren,
als Liste der Nullstellen des Polynoms,
als Liste der Werte des Polynoms an vorgegebenen Punkten.
Je nach konkreter Aufgabenstellung kann die eine oder die andere Vorgehensweise vorzuziehen sein.
Unser Verfahren verlangt nicht, daß wir uns von vornherein auf eine bestimmte Darstellungsart festlegen. So ähnlich, wie wir komplexe Zahlen durch Real- und Imaginärteil oder in Polarkoordinatendarstellung behandlen konnten, können auch hier verschiedene Repräsentationen neben- und miteinander verwendet werden.
In dieser Vorlesung soll allerdings nur eine Darstellungsform eingeführt werden: Polynome werden explizit als Summe (d.h.: als Liste) von Monomen eingeführt, wobei die Variablennamen in jedem Monom angegeben werden. Monome mit Koeffizient 0 werden nicht angegeben.
Die Monome sind nicht in willkürlicher Ordnung in dieser Liste eingetragen. Die Reihenfolge hängt vielmehr ab von der Angabe einer in der Definition des Polynoms zusätzlich angegeben Ordnungsfunktion für die Monome (z.B. lexikographische oder grad-lexikographische Ordnung der Monome)
Auch für die Variablen kann die Ordnung frei definiert werden. Geeignete Festlegungen der Ordnungen für die Monome und für die Variablen sind wichtig bei gewissen Algorithmen für Polynome.
Da der Koeffizientenring beliebig ist, ist die Darstellungsform von Polynomen als Elemente von $R[X_1][X_2], \ldots, [X_n]$ in diesem Verfahren als Spezialfall mit inbegriffen.

Algorithmus 8.1.1 *Einführung von Polynomen*

```
(define (poly? f)
  (eq? (type f) 'poly))
```

```
(define (make-poly mlist olist)
  (attach-type 'poly (cons mlist olist)))
(define (mlist f)
  (cond
    ( (poly? f) (car (contents f)) )
    ( else nil )))
(define (olist f)
  (cond
    ( (poly? f) (cdr (contents f)) )
    ( else nil )))
(define (monom-ordering-poly f)
  (monom-ordering-olist (olist f)) )
(define (monom-ordering-olist ol)
  (cond
    ( (null? ol) (give-global-monom-ordering) )
    ( else (car ol) )) )
(define (give-global-monom-ordering)
  (cond
    ( (unbound? global-monom-ordering) 'lexicographic )
    ( else global-monom-ordering )) )
(define (var-ordering-poly f)
  (var-ordering-olist (olist f)) )
(define (var-ordering-olist ol)
  (cond
    ( (null? ol) (give-global-var-ordering) )
    ( (null? (cdr ol)) (give-global-var-ordering) )
    ( else (cadr ol) )) )
(define (give-global-var-ordering)
  (cond
    ( (unbound? global-var-ordering) 'alphabetic )
    ( else global-var-ordering )) )

Fuer spaetere Anwendungen wird noch eingefuehrt:
(define (h-term f)
  (cond
    ( (not (poly? f)) (list f) )
    ( (null? (mlist f)) nil )
    ( else (car (mlist f)) )) )
(define (rest-list f)
  (cond
    ( (not (poly? f)) nil )
    ( (null? (mlist f)) nil )
    ( else (cdr (mlist f)) )) )
(define (rest-poly f)
  (cond
```

```
    ( (not (poly? f)) 0 )
    ( (null? (mlist f)) (make-poly nil (olist f)))
    ( else (make-poly (cdr (mlist f)) (olist f)) )) )
```

Die bei der Einführung der Polynome angegebene **olist** ist dabei eine Liste, die die Prozeduren angibt, nach denen die Monome und die Variablen geordnet werden sollen. Sind keine entsprechenden Prozeduren in dieser Liste eingetragen, sollen die Algorithmen die Ordnung nach Standard-Vorgaben vornehmen (lexikographische Ordnung der Monome, alphabetische Ordnung der Variablen bzw. andere global vorgegeben Ordnungen).
Die arithmetischen Operationen und sonstigen Operatoren für Polynome werden auf entsprechende Operationen für Monomlisten zurückgeführt:
Algorithmus 8.1.2 *Operatoren für Polynome*

```
Nullstellige Operationen:
=========================
(define the-empty-olist nil)
(define the-empty-mlist nil)
(define the-empty-plist nil)
(define (zero-poly type-of-coefficients)
  (make-poly the-empty-mlist the-empty-olist) )
(define (identity-poly type-of-coefficients)
  (make-poly
    (list (make-monom 1 the-empty-plist))
    the-empty-olist) )

Einstellige Operationen:
========================
(define (zero-poly? f)
  (zero-monom? (h-term f)) )
(define (unit-poly? f)
  (unit-monom? (h-term f)) )
(define (degree-poly f)
  (degree-mlist (mlist f)) )
(define (var-degree-poly var f)
  (var-degree-mlist var (mlist f)) )
(define (minus-poly f)
  (make-poly (minus-mlist (mlist f)) (olist f)) )
(define (inverse-poly f)
  (cond
    ( (unit-poly? f) (g-inverse (coefficient (h-term f))) )
    ( (error "Non existing inverse -- INVERSE-POLY" f) )) )
(define (type-of-coefficients f)
  (type (coefficient (h-term f))) )

Zweistellige Operationen:
```

```
=======================
(define (+poly f1 f2)
  (let ( (ol (check-olists (olist f1) (olist f2))) )
       (make-poly (+mlist ol (mlist f1) (mlist f2)) ol)) )
(define (-poly f1 f2)
  (+poly f1 (minus-poly f2)))
(define (*poly f1 f2)
  (let ( (ol (check-olists (olist f1) (olist f2))) )
       (make-poly (*mlist ol (mlist f1) (mlist f2)) ol)) )
(define (/poly f1 f2)
  (*poly f1 (quot-inverse-poly f2)))
(define (check-olists ol1 ol2)
  (cond
    ( (null? ol1) ol2 )
    ( (cond
        ( (null? ol2) ol1 )
        ( (equal? ol1 ol2) ol1 )
        ( (error "Inconsistent orderings -- CHECK-OLISTS"
                 (list ol1 ol2)) )) )) )
(define (normform-poly f)
  (cond
    ( (zero-poly? f) 0 )
    ( (zero? (degree-poly f)) (g-normform (coefficient (h-term f))) )
    ( (make-poly (normform-mlist (mlist f) (olist f)) (olist f)) )) )
```

8.2 Monome und Monomlisten

Wir führen jetzt die im letzten Abschnitt bereits erwähnten Prozeduren für die Monomlisten von Polynomen ein.
Monomlisten sind Listen von Monomen in absteigender Reihenfolge der gegebenen Ordnung von Monomen.
Monome sind „A-Listen mit Kopf“, also eindimensionale Tabellen im Sinn von Abschnitt 2.1.4. Der **car** dieser Listen enthält den Koeffizienten des Monoms, dann kommt eine Liste von Paaren, bei denen jeweils der **car** den Variablennamen, der **cdr** den zugehörigen Exponenten enthält. Die Variablen sind in der angegebenen Ordnung von Variablen aufgeführt.
Es werden spezielle Operationen für Variable und für Monome und Monomlisten eingeführt. Diese Operationen werden aber nicht in die Operatoren-Tabelle eingetragen, weil wir für Variable und Monome nicht die Typbezeichnungen und generischen Operatoren einführen wollen, sondern in eine spezielle Tabelle mit diesen Ordnungen.
Algorithmus 8.2.1 *Ordnungen für Variable und Monome*

```
Ordnungen von Variablen:
========================
```

```
(define v-order-table (make-table))
(define (put-v-order v-order op proc)
  (insert! v-order op proc v-order-table))
(define (var-<-alphabetic? var1 var2)
  (cond
    ( (char? var1)
        (cond
          ( (char? var2) (char-ci<? var1 var2) )
          ( (string? var2) (string-ci<? (make-string 1 var1) var2) )
          ( (error "Variables of wrong type -- VAR-<-ALPHABETIC?"
                   (list var1 var2)) )) )
    ( (string? var1)
        (cond
          ( (string? var2) (string-ci<? var1 var2) )
          ( (char? var2) (string-ci<? var1 (make-string 1 var2)) )
          ( (error "Variables of wrong type -- VAR-<-ALPHABETIC?"
                   (list var1 var2)) )) )
    ( (error "Variables of wrong type -- VAR-<-ALPHABETIC?" (list var1 var2)) ))

(put-v-order 'alphabetic '< var-<-alphabetic?)
(define (get-v-order v-order op)
  (lookup v-order op v-order-table))
(define (<-var v1 v2 v-order)
  (let ((proc (get-v-order v-order '<)))
    (cond
      ( (not (null? proc)) (proc v1 v2) )
      ( (error "Order of variables undefined -- <-VAR" (list v1 v2)) ))) )
(define (>-var v1 v2 v-order)
  (<-var v2 v1 v-order))

Ordnungen von Monomen:
======================
(define m-order-table (make-table))
(define (put-m-order m-order op proc)
  (insert! m-order op proc m-order-table))

(put-m-order 'lexicographic '< <-lexicographic)

(put-m-order 'degree-lexicographic '< <-degree-lexicographic)

Die Definition dieser Ordnungen wird weiter unten angegeben.
(define (get-m-order m-order op)
  (lookup m-order op m-order-table))
(define (<-monom m1 m2 ol)
  (let ((m-order (monom-ordering-olist ol)) (v-order (var-ordering-olist ol)))
```

```
    (let ((proc (get-m-order m-order '<)))
      (cond
        ( (not (null? proc)) (proc m1 m2 v-order) )
        ( (error "Order of monomials undefined -- <-MONOM" (list m1 m2)) )))) )
(define (>-monom m1 m2 ol)
  (<-monom m2 m1 ol))
```

Man betrachtet zunächst Monome:

Algorithmus 8.2.2 *Operationen für Monome*

```
Allgemeine Prozeduren:
======================
(define (make-monom coefficient plist)
  (cond
    ( (null? coefficient) nil )
    ( (g-zero? coefficient) nil )
    ( (cons coefficient plist) )) )
(define (coefficient monom)
  (cond
    ( (null? monom) 0)
    ( (atom? monom) monom )
    ( (car monom) )) )
(define (powers-list monom)
  (cond
    ( (null? monom) nil )
    ( (atom? monom) nil )
    ( (cdr monom) )) )
(define (variables-list monom)
  (map car (powers-list monom)))
(define (exponents-list monom)
  (map cdr (powers-list monom)))
(define (minus-monom m)
  (make-monom (g-minus (coefficient m)) (powers-list m)))
(define (zero-monom? m)
  (cond
    ( (null? m) )
    ( (g-zero? (coefficient m)) )) )
(define (unit-monom? m)
  (and (g-unit? (coefficient m))
       (null? (powers-list m))) )

Grad von Monomen:
=================
(define (degree-monom monom)
  (sum-list (exponents-list monom)))
(define (sum-list li)
```

```
  (eval (cons '+ li)))
(define (var-degree-monom var monom)
  (let ((p (assq var (powers-list monom))))
    (cond
      ( (null? p) 0 )
      ( else (cdr p) ))) )
```

Die folgende Prozedur erstellt zu einer vorgegebenen Liste von
Variablen die Liste der in einem Monom vorkommenden Exponenten
dieser Variablen:

```
(define (exp-list monom varlist)
  (define (vardeg var)    die Prozedur vardeg wird als lokal-
                          definierte Prozedur eingefuehrt, da
                          innerhalb des Prozedur-Koerpers von
                          dem durch das define implizit
                          aufgerufenen lambda.
    (var-degree-monom var monom))
  (map vardeg varlist))
```

Hilfsprozeduren fuer geordnete Listen:
==========

Die Listen sind in absteigender Folge geordnet.
Bei der folgenden Prozedur kommen Elemente, die sich der
Ordnung nach nicht unterscheiden, nur einmal vor:

```
(define (join-ordered-lists list1 list2 <-proc)
  (cond
    ( (null? list1) list2 )
    ( (join-ordered-lists
        (cdr list1)
        (join-element (car list1) list2 <-proc)
        <-proc) )) )
(define (join-element element li <-proc)
  (cond
    ( (null? li) (list element) )
    ( (<-proc (car li) element) (cons element li) )
    ( (<-proc element (car li))
      (cons
        (car li)
        (join-element element (cdr li) <-proc)) )
    ( li )) )
```

Bei der nun folgenden Prozedur kommen Elemente, die sich der
Ordnung nach nicht unterscheiden, unter Umstaenden mehrmals vor:

```
(define (join2-ordered-lists list1 list2 <-proc)
```

```
  (cond
    ( (null? list1) list2 )
    ( (join2-ordered-lists
        (cdr list1)
        (join2-element (car list1) list2 <-proc)
        <-proc) )) )
(define (join2-element element li <-proc)
  (cond
    ( (null? li) (list element) )
    ( (<-proc (car li) element) (cons element li) )
    ( (<-proc element (car li))
      (cons
        (car li)
        (join-element element (cdr li) <-proc)) )
    ( (cons element li) )) )

Lexikographische Ordnung von Listen ganzer Zahlen:
==================================================
(define (<-lexi-integer? l1 l2)
  (cond
    ( (null? l1)
      (cond
        ( (not (null? l2))
            (error "Different lengths of lists -- LEXI-INTEGER" l2) )
        ( else nil )) )
    ( (null? l2) (error "Different lengths of lists -- LEXI-INTEGER" l1) )
    ( (= (car l1) (car l2)) (<-lexi-integer? (cdr l1) (cdr l2)) )
    ( (< (car l1) (car l2)) )
    ( else nil )) )

Beispiel: (0 1) < (0 2) < (0 3) < (1 1)

Lexikographische Ordnung von Monomen:
=====================================

Es wird zunaechst die Liste der in beiden Monomen vorkommenden
Variablen gebildet. Dann erstellt man die Listen der Exponenten
dieser Variablen in jedem der Monome. Man wendet dann die Prozedur
<-lexi-integer auf diese Listen an.
(define (<-lexicographic? m1 m2 v-order)
  (let ((j-v-list (joined-var-list m1 m2 v-order)))
    (<-lexi-integer? (exp-list m1 j-v-list) (exp-list m2 j-v-list))))
(define (joined-var-list m1 m2 v-order)
  (define (<-proc v1 v2)
    (<-var v1 v2 v-order))
```

```
  (join-ordered-lists
    (variables-list m1)
    (variables-list m2)
    <-proc) )

Grad-lexikographische Ordnung von Monomen:
==========================================
(define (<-degree-lexicographic? m1 m2 v-order)
  (cond
    ( (< (degree-monom m1) (degree-monom m2)) )
    ( (> (degree-monom m1) (degree-monom m2)) nil )
    ( (<lexicographic? m1 m2 v-order) )))

(put-m-order 'lexicographic '< <-lexicographic?)

(put-m-order 'degree-lexicographic '< <-degree-lexicographic?)

Multiplikation von Monomen:
===========================
(define (*monom m1 m2 v-order)
  (make-monom (g-* (coefficient m1) (coefficient m2))
              (*plist (powers-list m1) (powers-list m2) v-order)))
(define (*plist pp1 pp2 v-order)
  (cond
    ( (null? pp1) pp2 )
    ( (*plist (cdr pp1) (*vp-plist (car pp1) pp2 v-order) v-order) )) )
(define (*vp-plist vp plist v-order)
  (cond
    ( (null? plist)
      (cond
        ( (zero? (cdr vp)) nil )
        ( (list vp) )) )
    ( (>-var (car vp) (caar plist) v-order)
        (cons vp plist) )
    ( (eq? (car vp) (caar plist))
        (cons (cons (car vp) (+ (cdr vp) (cdar plist))) (cdr plist)) )
    ( (cons (car plist) (*vp-plist vp (cdr plist) v-order)) )) )

Normalform von Monomen:
=======================
(define (normform-monom m ol)
  (cond
    ( (null? m) nil )
    ( (let ( (v-order (var-ordering-olist ol)) )
        (make-monom
```

```
          (g-normform (coefficient m))
          (normform-plist (powers-list m) v-order))) )) )
(define (normform-plist pl v-order)
  (cond
    ( (null? pl) nil )
    ( (adjoin-p-plist
        v-order
        (car pl)
        (normform-plist (cdr pl) v-order)) )) )
(define (adjoin-p-plist v-order p pl)
  (cond
    ( (null? p) pl )
    ( (null? pl)
      (cond
        ( (zero? (cdr p)) nil )
        ( (list p) )) )
    ( (>-var (car p) (caar pl) v-order)
      (cond
        ( (zero? (cdr p)) pl )
        ( else (cons p pl) )) )
    ( (<-var (car p) (caar pl) v-order)
        (cons (car pl) (adjoin-p-plist v-order p (cdr pl))) )
    ( (let ( (expon (+ (cdr p) (cdar pl))) )
        (cond
          ( (zero? expon) (cdr pl) )
          ( (cons
              (cons (car p) expon)
              (cdr pl)) ))) )) )
```

Manchmal ist es zweckmaessig, auch negative Exponenten zuzulassen. Man kann dann definieren:

```
(define (inverse-monom m v-order)
  (cond
    ( (null? m)
        (error "Inverse does not exist -- INVERSE-MONOM" m) )
    ( (make-monom
        (g-quot-inverse (coefficient m))
        (inverse-plist (powers-list m) v-order)) )) )
(define (inverse-plist pl v-order)
  (cond
    ( (null? pl) nil )
    ( (*plist
        (list (cons (caar pl) (- (cdar pl))))
        (inverse-plist (cdr pl) v-order)
        v-order) )) )
```

Algorithmus 8.2.3 *Operationen für Monomlisten*

```
Einstellige Operationen:
========================
(define (degree-mlist mlist)
  (cond
    ( (null? mlist) (- 1) )
    ( (eval (cons 'max (map degree-monom mlist))) )) )
(define (var-degree-mlist var mlist)
  (define (var-degree monom)
    (var-degree-monom var monom))
  (eval (cons 'max (map var-degree mlist))) )

(define (minus-mlist mlist)
  (map minus-monom mlist))


Addition von Monomlisten:
=========================
(define (+mlist ol ml1 ml2)
  (cond
    ( (null? ml1) ml2)
    ( (+mlist
         ol
         (cdr ml1)
         (+monom-monomlist ol (car ml1) ml2)) ) )
(define (+monom-monomlist ol m ml)
  (cond
    ( (null? m) ml )
    ( (null? ml) (list m) )
    ( (>-monom  m (car ml) ol)
        (cons m ml) )
    ( (equal? (powers-list m) (powers-list (car ml)))
        (let ((coeff (g-+ (coefficient m) (coefficient (car ml)))))
          (cond
            ( (g-zero? coeff) (cdr ml) )
            ( (cons
                (make-monom coeff (powers-list m))
                (cdr ml)) ))) )
     ( (cons (car ml) (+monom-monomlist ol m (cdr ml))) ) )

Multiplikation von Monomlisten:
===============================
(define (*mlist ol ml1 ml2)
  (cond
```

```
    ( (null? ml1) the-empty-mlist )
    ( (+mlist
        ol
        (*monom-monomlist ol (car ml1) ml2)
        (*mlist ol (cdr ml1) ml2)) )) )
(define (*monom-monomlist ol m ml)
  (define (*m-monom monom)
    (*monom m monom (var-ordering-olist ol)))
  (map *m-monom ml) )

Normalform von Monomlisten:
===========================
(define (normform-mlist ml ol)
  (cond
    ( (null? ml) nil )
    ( (adjoin-m-mlist
        ol
        (normform-monom (car ml) ol)
        (normform-mlist (cdr ml) ol)) )) )
(define (adjoin-m-mlist ol m ml)
  (cond
    ( (null? m) ml )
    ( (null? ml) (list m) )
    ( (>-monom m (normform-monom (car ml) ol) ol) (cons m ml) )
    ( (<-monom m (normform-monom (car ml) ol) ol)
        (cons
          (normform-monom (car ml) ol)
          (adjoin-m-mlist ol m (cdr ml))) )
    ( (let ((coeff (g-+ (coefficient m) (coefficient (car ml)))))
        (cond
          ( (g-zero? coeff) (cdr ml) )
          ( (cons
              (make-monom coeff (powers-list m))
              (cdr ml)) ))) )) )
```

Algorithmus 8.2.4 *Einfügen der Prozeduren in die Operatorentabelle*

```
(put 'poly 'zero zero-poly)

(put 'poly 'identity identity-poly)

(put 'poly 'zero? zero-poly?)

(put 'poly 'unit? unit-poly?)

(put 'poly 'degree degree-poly)
```

```
(put 'poly 'var-degree var-degree-poly)

(put 'poly 'normform normform-poly)

(put 'poly 'minus minus-poly)

(put 'poly 'inverse inverse-poly)

(put 'poly '+ +poly)

(put 'poly '- -poly)

(put 'poly '* *poly)

(put 'poly '/ /poly)
(define (g-degree f) (operate-1 'degree f))
(define (g-var-degree var f) (operate-2 'var-degree var f))
(define (g-id a)
  (cond
    ( (poly? a)
        (cond
          ( (g-zero? a) 1 )
          ( (g-identity (type (coefficient (h-term a)))) )) )
    ( (g-identity (type a)) )) )
(define (g-ze a)
  (cond
    ( (poly? a)
        (cond
          ( (g-zero? a) 0 )
          ( (g-zero (type (coefficient (h-term a)))) )) )
    ( (g-zero (type a)) )) )
```

8.3 Übergangsprozeduren

Algorithmus 8.3.1 *Uebergangsprozeduren*

```
(define (integer->poly n)
  (make-poly
    (list (make-monom n the-empty-plist))
    the-empty-olist) )
(define (rat->poly x)
  (make-poly
    (list (make-monom x the-empty-plist))
    the-empty-olist) )
(define (real->poly x)
```

```
  (make-poly
    (list (make-monom x the-empty-plist))
    the-empty-olist) )
(define (residue->poly x)
  (make-poly
    (list (make-monom x the-empty-plist))
    the-empty-olist) )
(define (rectangular->poly z)
  (make-poly
    (list (make-monom z the-empty-plist))
    the-empty-olist) )
(define (polar->poly z)
  (make-poly
    (list (make-monom z the-empty-plist))
    the-empty-olist) )

(put-coercion 'integer 'poly integer->poly)

(put-coercion 'rat 'poly rat->poly)

(put-coercion 'real 'poly real->poly)

(put-coercion 'residue 'poly residue->poly)

(put-coercion 'rectangular 'poly rectangular->poly)

(put-coercion 'polar 'poly polar->poly)
```

8.4 Beispiele

Algorithmus 8.4.1 *Beispiele*

```
(define f (make-poly '( (1 (#\z . 2) (#\y . 3)) (5 (#\y . 1) (#\x . 2)))
                     nil ))
(define g (minus-poly f))
(define a (make-poly (append (mlist f) (mlist g)) nil))
(define h
  (make-poly
    (list
      (make-monom z '((#\z . 2) (#\y . 3)))
      (make-monom w '((#\y . 1) (#\x . 2))))
    nil ))

(define k (minus-poly h))

(define l
```

```
  (make-poly
    (list
      (make-monom x '((#\z . 2) (#\y . 3)))
      (make-monom y '((#\y . 1) (#\x . 2))))
    nil ))

(define m (minus-poly l))
(define s1 (g-+ f g))

(define s2 (g-+ f h))

(define s3 (g-+ f l))

(define s4 (g-+ h l))
(define p1 (g-* f g))

(define p2 (g-+ f h))

(define p3 (g-* f l))

(define p4 (g-+ h l))

(define p5 (g-* s3 s4))

(define p6 (g-+ p3 p4))

(define p8
  (make-poly
    (list
      (make-monom p3 '((#\z . 2) (#\y . 3)))
      (make-monom  l '((#\y . 1) (#\x . 2)))
      (make-monom (g-+ p3 l) nil))
    nil ))
(define (g-square x)
  (g-* x x))
```

9 Restklassenringe von Polynomringen

9.1 Polynomreduktion

Die Polynomreduktion fungiert als Ersatz für die in euklidischen Ringen verwendete Division mit Rest bei der Bildung von Normalformalgorithmen. Die hier vorgestellten Algorithmen sind für Polynomringe in endlich vielen Unbestimmten über einem Körperentsprechende Algorithmen können für Polynomringe über euklidischen Ringen entwickelt werden.

Algorithmus 9.1.1 *Polynomreduktion*

```
(define (max-pol-order-monom m)
  (minus (eval (cons 'min (exponents-list m)))) )
(define (divides-monom? m1 m2 v-order)
  (not
    (positive?
      (max-pol-order-monom
        (*monom (inverse-monom m1 v-order) m2 v-order)))) )
(define (quotient-monom m1 m2 v-order)
  (let ( (q (*monom m1 (inverse-monom m2 v-order) v-order)) )
    (cond
      ( (positive? (max-pol-order-monom q)) nil )
      ( else q ))) )
(define (reduce-poly f h)
  (cond
    ( (g-zero? h)
        (error
          "Reduction with zero-polynomial not allowed - REDUCE-POLY"
          (list f h)) )
    ( else
       (let ( (ol (check-olists (olist f) (olist h))) )
         (let ( (pair (pair-poly h)) )
           (g-normform
             (g-norm-ideal
               (car
                 (reduce-poly-pair
                   f
                   (car pair)
                   (cdr pair)
                   ol)))))) )) )
(define (pair-poly h)
  (cons (h-term h) (g-minus (rest-poly h))))
(define (reduce-poly-pair f a b ol)
  (cond
    ( (null? f) '(0) )
    ( (g-zero? f) '(0) )
    ( else
```

```
      (let ( (q (quotient-monom (h-term f) a (var-ordering-olist ol))) )
        (cond
          ( (null? q)
              (let ( (rpp (reduce-poly-pair (rest-poly f) a b ol)) )
                (cons
                  (g-+
                    (make-poly (list (h-term f)) nil)
                    (car rpp))
                  (cdr rpp))) )
            ( else
               (cons
                 (g-+
                   (*poly
                     (make-poly (list q) nil)
                     b)
                   (rest-poly f))
                  q) ))) )) )
```

9.2 Reduktions-Normalform-Algorithmen

Gegeben sei eine Restklasse x mit Repräsentant f und Ideal A. Ein Reduktions-Normalform-Algorithmus wendet auf den Repräsentanten f die oben beschriebene Reduktion nach den Basiselementen $h \in A$ an solange, bis das Verfahren abbricht.

Definition 9.2.1 *Eine Monom-Ordnung $< -monom$ heißt „zulässig“, wenn gilt:*
1. Für alle Monome m gilt $(< -monom 1 m)$.
2. Die Monom-Ordnung ist verträglich mit der Monommultiplikation.

Man kann zeigen:

Satz 9.2.2 *Bei einer zulässigen Monom-Ordnung bricht ein Reduktions- Normalform-Algorithmus stets nach endlich vielen Schritten ab.*

Man erhält unterschiedliche Reduktions-Normalform-Algorithmen je nach der Strategie, nach welcher die einzelnen Polynome h der Idealbasis auf die einzelnen Monome von f angewendet werden.

Algorithmus 9.2.3 *Ein Reduktions-Normalform-Algorithmus*

```
(define (total-reduction-poly f h)
  (cond
    ( (g-zero? h)
        (error
          "Reduction with zero-polynomial not allowed - REDUCE-POLY"
          (list f h)) )
    ( else
       (let ( (ol (check-olists (olist f) (olist h))) )
         (let ( (pair (pair-poly h)) )
```

```
           (g-normform
             (g-norm-ideal
               (car
                 (total-reduction-poly-pair
                   f
                   (car pair)
                   (cdr pair)
                   ol)))))) )) )
(define (total-reduction-poly-pair f a b ol)
  (cond
    ( (null? f) '(0) )
    ( (g-zero? f) '(0) )
    ( else
       (let ( (rpp (reduce-poly-pair f a b ol)) )
         (cond
           ( (null? (cdr rpp)) rpp )
           ( else
              (total-reduction-poly-pair (car rpp) a b ol) ))) )) )
(define (total-reduction-ideal f aa)
  (cond
    ( (null? aa) f )
    ( else
       (total-reduction-ideal
         (total-reduction-poly f (car aa))
         (cdr aa)) )) )
(define (normform-reduction-res x)
  (make-res
    (ideal-res x)
    (total-reduction-ideal
      (representative-res x)
      (ideal-res x))) )
```

Algorithmus 9.2.4 *Beispiele*

```
(define
  f1
  (make-poly
    (list
      '( 3 (#\y . 1) (#\x . 2) )
      '( 2 (#\y . 1) (#\x . 1) )
      '( 1 (#\y . 1) )
      '( 9 (#\x . 2) )
      '( 5 (#\x . 1) )
      '(-3 ) )
    nil ))

(define
```

```
  f2
  (make-poly
    (list
      '( 2 (#\y . 1) (#\x . 3) )
      '(-1 (#\y . 1) (#\x . 1) )
      '(-1 (#\y . 1) )
      '( 6 (#\x . 3) )
      '(-5 (#\x . 2) )
      '(-3 (#\x . 1) )
      '( 3 ) )
    nil ))

(define
  f3
  (make-poly
    (list
      '( 1 (#\y . 1) (#\x . 3) )
      '( 1 (#\y . 1) (#\x . 2) )
      '( 3 (#\x . 3) )
      '( 2 (#\x . 2) ))
    nil ) )
(define aa (list f1 f2 f3))
(define
  f1-res
  (make-poly
    (list
      '( (residue 7 . 3) (#\y . 1) (#\x . 2) )
      '( (residue 7 . 2) (#\y . 1) (#\x . 1) )
      '( (residue 7 . 1) (#\y . 1) )
      '( (residue 7 . 9) (#\x . 2) )
      '( (residue 7 . 5) (#\x . 1) )
      '( (residue 7 . -3) ) )
    nil ))

(define
  f2-res
  (make-poly
    (list
      '( (residue 7 . 2) (#\y . 1) (#\x . 3) )
      '( (residue 7 . -1) (#\y . 1) (#\x . 1) )
      '( (residue 7 . -1) (#\y . 1) )
      '( (residue 7 . 6) (#\x . 3) )
      '( (residue 7 . -5) (#\x . 2) )
      '( (residue 7 . -3) (#\x . 1) )
      '( (residue 7 . 3) ) )
```

```
    nil ))

(define
  f3-res
  (make-poly
    (list
      '( (residue 7 . 1) (#\y . 1) (#\x . 3) )
      '( (residue 7 . 1) (#\y . 1) (#\x . 2) )
      '( (residue 7 . 3) (#\x . 3) )
      '( (residue 7 . 2) (#\x . 2) ))
    nil ) )
(define aa-res (list f1-res f2-res f3-res))
(define
  g
  (make-poly
    (list
      '( 5 (#\y . 2) )
      '( 2 (#\y . 1) (#\x . 2) )
      '( (rat 5 . 2) (#\y . 1) (#\x . 1) )
      '( (rat 5 . 2) (#\y . 1) (#\x . 1) )
      '( (rat 3 . 2) (#\y . 1) )
      '( 8 (#\x . 2) )
      '( (rat 3 . 2) (#\x . 1) )
      '( (rat -9 . 2) ))
    nil))
(define xx (make-res aa g))
(define xx1 (make-res aa f1))

(define xx2 (make-res aa f2))

(define xx3 (make-res aa f3))
```

9.3 Gröbner-Basen

Der Reduktions-Normalform-Algorithmus ist i.a. kein kanonischer Normalform-Algorithmus. Z.B. reduzieren sich die oben angegebenen Restklassen $xx2$ und $xx3$ bei Anwendung des Algorithmus nicht zur Nullklasse.

Definition 9.3.1 *Eine Idealbasis heißt Gröbner-Basis, wenn jeder Reduktions-Normalform-Algorithmus nach dieser Basis kanonisch ist.*

Wir werden den *Vervollständigungsalgorithmus* angeben, mit dessen Hilfe man von einer beliebigen Basis zu einer Gröbner-Basis übergehen kann. Der Vervollständigungsalgorithmus basiert auf der nun im folgenden beschriebenen Charakterisierung von Gröbner-Basen durch sog. *S-Polynome*.

Definition 9.3.2 *Seien h_1, h_2 Polynome mit den führenden Termen a_1, a_2. Sei weiter $b_i := a_i - h_1$, $i = 1, 2$. Sei s das kleinste gemeinsame Vielfache von*

a_1 und a_2, und zwar gelte $s = \sigma_i a_i$ füri $= 1, 2$. Das Polynom $SP(h_1, h_2) := \sigma_1 b_1 - \sigma_2 b_2$ heißt das S-Polynom *von h_1 und h_2.*

Algorithmus 9.3.3 *S-Polynome*

```
(define (s-poly h1 h2)
  (s-poly-ol h1 h2 (check-olists (olist h1) (olist h2))))
(define (s-poly-ol h1 h2 ol)
  (let ( (pair1 (pair-poly h1))
         (pair2 (pair-poly h2))
         (v-order (var-ordering-olist ol)) )
    (let ( (multiples (lcm-factors-monom (car pair1) (car pair2) v-order)) )
      (g--
        (g-*
          (g-normform (make-poly (list (cadr multiples)) ol))
          (g-normform (cdr pair1)))
        (g-*
          (g-normform (make-poly (list (caddr multiples)) ol))
          (g-normform (cdr pair2)))))))
(define (lcm-factors-monom m1 m2 v-order)
  (let ( (j-v-list (joined-var-list m1 m2 v-order)) )
    (define (power-lcm var)
      (cons var (max (var-degree-monom var m1) (var-degree-monom var m2))))
    (let ( (lcm-pl
             (normform-plist
               (map power-lcm j-v-list)
               v-order)) )
      (let ( (lcm (make-monom 1 lcm-pl)) )
        (list
          lcm
          (quotient-monom lcm m1 v-order)
          (quotient-monom lcm m2 v-order))))))
```

Satz 9.3.4 (Charakterisierung von Gröbner-Basen) *Eine Idealbasis E ist genau dann eine Gröbner-Basis, wenn gilt: Ist RNA ein Reduktions-Normalform-Algorithmus in Bezug auf diese Idealbasis, so gilt für je zwei Basiselemente $h_1, h_2 \in E$: $RNA(SP(h_1, h_2) = 0$.*

Der Beweis dieses Satzes ist nicht trivial, vgl. das Manuskript „Theorie und Praxis der Polynomideale“ und die dort angegebene Literatur.

Definition 9.3.5 *Eine Gröbner-Basis E heißt* reduziert*, wenn jedes ihrer Polynome den höchsten Koeffizienten 1 hat und in Bezug auf die übrigen Polynome nicht reduzierbar ist.*

Satz 9.3.6 *Jedes Ideal besitzt eine (bis auf Reihenfolge) eindeutig bestimmte reduzierte Gröbner-Basis.*

Zum Beweis wird wieder auf das o.a. Manuskript verwiesen. Der folgende Algorithmus hat als Argument eine beliebige Idealbasis E und als Wert die zugehörige reduzierte Gröbner-Basis F. Die Basis wird bei der Ausführung des Algorithmus stets so umgeordnet, daß die führenden Terme der Basisele-

mente in schwach monoton fallender Folge kommenbei der als Ergebnis erhaltenen reduzierten Gröbner-Basis stehen sie dann sogar in strikt monoton fallender Folge.

Algorithmus 9.3.7 *Vervollständigungsalgorithmus*

```
Hilfsprozeduren zur Anordnung von Basen:
========================================
(define (<-poly f g)
  (let (( ol (check-olists (olist f) (olist g)) ))
    (<-monom (h-term f) (h-term g) ol)))
(define (>-poly f g)
  (<-poly g f))

(put 'poly '< <-poly)

(put 'poly '> >-poly)
(define (order-base e)
  (join2-ordered-lists e nil <-poly))

Der Vervollstaendigungsalgorithmus:
===================================
(define (groebner e)
  (cond
    ( (null? e) e )
    ( (let (( e (order-base e) ))
      (let (( i (indices e) )
            ( ol (olist (car e)) ))
      (let (( reduced-pair (reduce-base e e i ol) ))
      (let (( e (groebner-kernel (car reduced-pair) (cdr reduced-pair) ol) ))
      (order-base (car (reduce-base e e nil ol))) )))) )) )
(define (indices e)
  (cond
    ( (null? e) nil )
    ( (null? (cdr e)) nil )
    ( (append (firstlist e) (indices (cdr e))) )) )
(define (firstlist e)
  (define (make-pair element)
    (cons (car e) element))
  (map make-pair (cdr e)))
(define (groebner-kernel e i ol)
  (cond
    ( (null? i)

      (newline)
      (newline)
```

```
      (display "Groebner-Basis:")
      (newline)
      (newline)

      e )
    ( (let (( h (total-reduction-ideal (s-poly (caar i) (cdar i)) e) ))

      (newline)
      (display "Bildung des S-Polynoms der folgenden Polynome:")
      (newline)
      (pp (caar i))
      (newline)
      (pp (cdar i))
      (newline)
      (display "Reduziertes S-Polynom:")
      (newline)
      (pp h)
      (newline)

      (cond
        ( (g-zero? h)
          (groebner-kernel e (cdr i) ol) )
        ( (let (( e (cons h e) )
                ( i (append (cdr i) (firstlist (cons h e))) ))
          (let (( reduced-pair (reduce-base e e i ol) ))
          (groebner-kernel (car reduced-pair) (cdr reduced-pair) ol))) ))) ) )

Der Hilfsalgorithmus zur Reduktion der Basis:
=============================================
(define (reduce-base old-e new-e i ol)
  (cond
    ( (null? old-e) (cons new-e i) )
    ( (let (( ff (g-norm-ideal (total-reduction-ideal (car old-e) (cdr new-e)))
            ( old-e (cdr old-e) )
            ( new-e (cdr new-e) )
            ( i (remove-indices (car old-e) i) ))
      (cond
        ( (g-zero? ff) (reduce-base old-e new-e i ol) )
        ( (reduce-base
            old-e
            (append new-e (list ff))
            (append i (firstlist (cons ff new-e)))
            ol) ))) )) )

(define (remove-indices element i)
```

```
  (cond
    ( (null? i) nil )
    ( (eq? (car i) element) (remove-indices element (cdr i)) )
    ( (cons
        (car i)
        (remove-indices element (cdr i))) )  )
```

Algorithmus 9.3.8 *Beispiele*

```
(define gb-aa (groebner aa))

(pp gb-aa)

Man erhaelt die Antwort

;((POLY ((1 (#\y . 1))
         (-3)))
(POLY ((1 (#\x . 1)))))
(define gb-aa-res (groebner aa-res))

(pp gb-aa-res)
(define ee
  (list
    (make-poly
      '( ( 1 (#\y . 1) (#\x . 3))
         (-2 (#\x . 2)))
      nil)
    (make-poly
      '( ( 1 (#\y . 3) (#\x . 1))
         (-3 (#\y . 1) (#\x . 1)))
      nil)))
(define gb-ee (groebner ee))

(pp gb-ee)
```

10 Unifikation und Resolution

Es geht hier um eine elementare KI-Anwendung (sog. künstliche Intelligenz), und zwar um das automatisierte Beweisen.
In der Aussagenlogik (wo es um Aussagen ohne Variablen und Quantoren geht) hat man den Resolutionsalgorithmus für Grundklauseln (Robinson 1965), anwendbar auf Aussagen, die als Hornformel gegeben sind; sie sind also insbesondere in konjunktiver Normalform (KNF). Der Resolutionsalgorithmus wird angewendet auf die Formel *Voraussetzung und (logisches und) Negation der Behauptung* und ergibt, wenn die Behauptung aus der Voraussetzung logisch hergeleitet werden kann, in endlich vielen Schritten eine leere Formel, d.h. ein logisches Falsch.
Der Resolutionsalgorithmus ist auch in der Prädikatenlogik 1. Stufe (Variablen und Quantoren, aber keine Funktionsausdrücke als Variable) anwendbar. Hier muß aber eine geeignete Substitution für die Variablen gefunden werden, um den Widerspruch herzuleiten. Das ist mittels der Unifikation möglich.

Unifikationsalgorithums aus der Dokumentation von Petite Chez Scheme:

```
(define unify #f)
(let ()
  ;; occurs? returns true if and only if u occurs in v
  (define occurs?
    (lambda (u v)
      (and (pair? v)
           (let f ((l (cdr v)))
             (and (pair? l)
                  (or (eq? u (car l))
                      (occurs? u (car l))
                      (f (cdr l)))))))))

  ;; sigma returns a new substitution procedure extending s by
  ;; the substitution of u with v
  (define sigma
    (lambda (u v s)
      (lambda (x)
        (let f ((x (s x)))
          (if (symbol? x)
              (if (eq? x u) v x)
              (cons (car x) (map f (cdr x)))))))))

  ;; try-subst tries to substitute u for v but may require a
  ;; full unification if (s u) is not a variable, and it may
  ;; fail if it sees that u occurs in v.
  (define try-subst
    (lambda (u v s ks kf)
      (let ((u (s u)))
```

```
        (if (not (symbol? u))
            (uni u v s ks kf)
            (let ((v (s v)))
              (cond
                ((eq? u v) (ks s))
                ((occurs? u v) (kf "cycle"))
                (else (ks (sigma u v s)))))))))

  ;; uni attempts to unify u and v with a continuation-passing
  ;; style that returns a substitution to the success argument
  ;; ks or an error message to the failure argument kf.  The
  ;; substitution itself is represented by a procedure from
  ;; variables to terms.
  (define uni
    (lambda (u v s ks kf)
      (cond
        ((symbol? u) (try-subst u v s ks kf))
        ((symbol? v) (try-subst v u s ks kf))
        ((and (eq? (car u) (car v))
              (= (length u) (length v)))
         (let f ((u (cdr u)) (v (cdr v)) (s s))
           (if (null? u)
               (ks s)
               (uni (car u)
                    (car v)
                    s
                    (lambda (s) (f (cdr u) (cdr v) s))
                    kf))))
        (else (kf "clash")))))

  ;; unify shows one possible interface to uni, where the initial
  ;; substitution is the identity procedure, the initial success
  ;; continuation returns the unified term, and the initial failure
  ;; continuation returns the error message.
  (set! unify
    (lambda (u v)
      (uni u
           v
           (lambda (x) x)
           (lambda (s) (s u))
           (lambda (msg) msg)))))
```

```
(unify 'x 'y)  y
```

```
(unify '(f x y) '(g x y))  "clash"
(unify '(f x (h)) '(f (h) y))  (f (h) (h))
(unify '(f (g x) y) '(f y x))  "cycle"
(unify '(f (g x) y) '(f y (g x)))  (f (g x) (g x))
(unify '(f (g x) y) '(f y z))  (f (g x) (g x))
```

In algebraischen Zusammenhängen (gleichungsdefinierte algebraische Strukturen, universelle Algebra) kann man statt des Resolutionsalgorithmus auch sog. Termersetzungssysteme verwenden. Hier zeigt es sich, daß man die gegebenen Gleichungen (z.B. die Axiome für eine Gruppe) i.a. durch Hinzunahme weitere, aus den gegebenen Gleichungen ableitbare Gleichungen vervollständigen muß (Algorithmus von Knuth und Bendix). Das Termersetzungssystem liefert dann einen kanonishcen Normalformalgorithmus.
Ein analoges Vorgehen bzgl. des Rechnens modulo einem Polynomideal (von mehreren Variablen) liefert die sog. Gröbner-Basen der Idealtheorie.