Eine kurze Einführung in MAGMA

Ulrich Thiel[1]

INHALTSVERZEICHNIS

§1. Einleitung 1
§2. Grundlagen 1
§2A. Erste Schritte in MAGMA 1, §2B. Logik und Struktur von MAGMA 3, §2C. Implementierte Funktionen (Intrinsics) 5, §2D. Bezeichner 8, §2E. Aussagenlogik (Booleans) 8, §2F. Konditionalausdrücke 9, §2G. Mengen 11, §2H. Sequenzen 13, §2I. Tupel 14, §2J. Abbildungen 15, §2K. Intrinsics, Funktionen und Prozeduren 15, §2L. Kommentare 18, §2M. Schleifen 19, §2N. Strings 20, §2O. Ausgabe 20
§3. Gruppen 21
§3A. Freie Gruppen 22, §3B. Endlich präsentierte Gruppen 24, §3C. Gruppen mit Termersetzungssystem 28, §3D. Permutationsgruppen (konkret) 29, §3E. Permutationsgruppen (abstrakt) 32, §3F. Matrixgruppen 34, §3G. PC-Gruppen 35, §3H. Datenbank endlicher Gruppen 35
§4. Ringe 36
§4A. Moduln und Algebren 37, §4B. Freie Algebren 37, §4C. Endlich präsentierbare Algebren 39, §4D. Polynomringe 41, §4E. Algebraische Zahlkörper 44
§5. Moduln und lineare Algebra 45
§5A. Matrizen, Vektoren und lineare Algebra 45

Dies ist eine vorläufige Version einer kurzen Einführung in das Computeralgebrasystem MAGMA. Kommentare, Fehlermeldungen und Anregungen sind jederzeit erwünscht. Da ich dieses Skript noch erweitern und verbessern möchte, muss ich hiermit die elektronische Bereitstellung dieses Dokuments und seiner Teile ohne meine Zustimmung untersagen.

[1] thiel at mathematik.uni-stuttgart.de, www.mathematik.uni-stuttgart.de/~thiel

§1. Einleitung

MAGMA ist ein Computeralgebra-System, das von der Computational Algebra Group der University of Sydney entwickelt wird (siehe [1] und [Magma]). Es entstand aus dem Computeralgebra-System CAYLEY, das von Greg Butler und John Cannon zwischen 1982 und 1993 entwickelt wurde (siehe [2]). Die erste Version von MAGMA wurde 1993 veröffentlicht; die aktuelle Version (Stand: April 2013) ist 2.19-4. Zu den Hauptentwicklern zählen – neben vielen anderen – John Cannon und Allan Steel.

MAGMA , dessen Name angelehnt ist an die algebraischen Struktur namens Magma, stellt eine mathematisch rigorose Umgebung zum Arbeiten mit algebraischen Strukturen – wie zum Beispiel Gruppen, Ringe, Körper, Algebren und Moduln – zur Verfügung und unterscheidet sich damit von vielen anderen Computeralgebrasystemen. Daneben ist es aber auch die unglaubliche Flexibilität und Bandbreite an bereits implementierten Strukturen und Funktionen, die MAGMA so besonders macht.

Die mathematische Rigorosität von MAGMA hat allerdings zur Folge, dass ein Anfänger sich zunächst in der der Bedienung schwer tun kann und gezwungen ist, sich Klarheit über einige algebraische Konzepte zu verschaffen. In dieser kurzen Einführung wollen wir uns sowohl mit der Bedienung von MAGMA vertraut machen als auch grundlegende algebraische Begriffe erlernen bzw. wiederholen.

Einen Nachteil und Kritikpunkt von MAGMA möchte ich jedoch gleich zu Beginn festhalten: Es ist nicht kostenlos und der Quellcode ist nicht frei verfügbar. Daher möchte ich an dieser Stelle auch kostenlose Alternativen auflisten, die zumindest gewisse Teile der Funktionalität von MAGMA abdecken oder gar besser umsetzen, aber *meiner* Meinung nach in ihrer inneren Struktur, Funktionalität, Flexibilität und Geschwindigkeit MAGMA nicht gewachsen sind: GAP, SAGE, SINGULAR und noch viele mehr.

Bemerkung. Ich werde teilweise englische Begriffe in einem deutschen Text verwenden. Das ist zwar schlechter Stil, aber das künstliche Eindeutschen gewisser Fachbegriffe erscheint mir eine noch schlechtere Alternative zu sein.

§2. Grundlagen

Wir werden hier die grundlegenden Strukturen und Konzepte von MAGMA besprechen. Besonderen Wert legen wir dabei auf das Verständnis von Objekttypen, deren zugehörigen Strukturen, und Beziehungen zwischen diesen Strukturen. Die Diskussion der Implementierung eigener Funktionen zögern wir ein wenig hinaus, bis wir genug Beispiele grundlegender Datentypen gesammelt haben.

§2A. Erste Schritte in MAGMA

2.1. MAGMA kann auf dem Studpool über den Befehl `magma` gestartet werden. Von außen kann man sich per SSH auf dem Studpool (z.B. auf `stud10`) einloggen und MAGMA starten:

```
$ ssh LOGIN@stud10.mathematik.uni-stuttgart.de
LOGIN@stud10.mathematik.uni-stuttgart.de's password:
NAME@stud10:~$ magma
Magma V2.19-2 (STUDENT)   Wed Apr  3 2013 13:40:57
Type ? for help.  Type <Ctrl>-D to quit.
>
```

Auffällig ist sicherlich, dass MAGMA keine graphische Oberfläche hat wie zum Beispiel MAPLE. Das ist richtig, und die benötigen wir auch nicht, denn unser Anliegen ist nicht, Funktionen oder ähnliches zu visualisieren, sondern uns mit algebraischen Strukturen beschäftigen.

2.2. Auf der Internetseite

http://magma.maths.usyd.edu.au

von MAGMA kann man viele hilfreiche Informationen finden. Die wichtigste ist die Online-Dokumentation, die unter

http://magma.maths.usyd.edu.au/magma/handbook/

verfügbar ist. Daneben gibt es auf der Internetseite

http://magma.maths.usyd.edu.au/calc/

noch einen Online Calculator, mit dem man einfache Rechnungen durchführen kann.

2.3. Wir können nun erstmal ein paar einfache Rechnungen durchführen, um uns an MAGMA zu gewöhnen:

```
> 2+3;
 5
> 2*(-3);
 -6
> (-3)^117;
-665559370338678226078955492410964829530176158347352261 63
> 3/7+2/17;
65/119
> (1/13)/(-5/19);
-19/65
```

Jedes Kommando in MAGMA wird mit dem Semikolon ; abgeschlossen und mit der Eingabetaste ausgeführt. Wir sehen, dass wir in MAGMA sofort im Ring $\mathbb{Z}$ der ganzen Zahlen und im Körper $\mathbb{Q}$ der rationalen Zahlen rechnen können, ohne dies MAGMA speziell mitteilen zu müssen.

2.4. Mit dem Befehl `quit` kann MAGMA beendet werden:

```
> quit;

Total time: 0.500 seconds, Total memory usage: 19.34MB
```

2.5. Eine längere Rechnung in MAGMA kann mit dem Befehl `Ctrl` + `C` abgebrochen werden, bzw. genauer: Damit wird ein Unterbrechungssignal an den Prozess gesandt. Es kann dann noch einige Zeit dauern, bis MAGMA die Rechnung wirklich stoppt. Aber Achtung: `Ctrl` + `C` zweimal schnell hintereinander beendet MAGMA!

2.6. Aufgabe. Geben Sie den Ausdruck

$$1+\cfrac{1^2}{3+\cfrac{2^2}{5+\cfrac{3^2}{7+\frac{4^2}{9}}}}$$

in der Form $\frac{a}{b}$ an. Es handelt sich hierbei um den Anfang von Lamberts Kettenbruch für $\frac{4}{\pi}$. Eine wichtige Regel für die Computerprogrammierung im Allgemeinen können wir an dieser Aufgabe schon einüben:

> Die Anzahl öffnender Klammern muss gleich der Anzahl schließender Klammern sein.

In jedem Fall ist MAGMA also ein sehr guter Taschenrechner, kann aber natürlich noch viel mehr. Bevor wir uns aber um die weitere Funktionalität kümmern, müssen wir zunächst die Logik und Struktur von MAGMA besprechen.

§2B. Logik und Struktur von MAGMA

2.7. Jedes Objekt in MAGMA hat einen wohldefinierten *Typ* und gehört einer darüber liegenden wohldefinierten *Struktur*, dem *Parent* ("'Elternstruktur"'), an. Diese Informationen verwendet MAGMA zur korrekten Behandlung der Objekte und sie können wie folgt ermittelt werden:

```
> Type(17);
RngIntElt
> Parent(17);
Integer Ring
> Type(3/7);
FldRatElt
> Parent(3/7);
Rational Field
```

Wir sehen also, dass 17 vom Typ RngIntElt ist und als Parent Integer Ring hat. Das heißt nichts anderes, als dass 17 von MAGMA als eine ganze Zahl (der Typ) und damit als Element des Rings $\mathbb{Z}$ der ganzen Zahlen (der Parent) angesehen wird. Wenn wir dann zum Beispiel eine Addition 17+12 ausführen, weiß MAGMA, dass er zwei Elemente des Rings ganzer Zahlen addieren soll und benutzt dafür die Additionsfunktion der Struktur Integer Ring. All das geschieht automatisch in MAGMA und analog verhält es sich mit rationalen Zahlen wie $\frac{3}{7}$ in dem Beispiel.

2.8. Die Strukturen Integer Ring und Rational Field können in MAGMA mit folgenden Befehlen erzeugt werden:

```
> Integers();
Integer Ring
> Integers();
> Rationals();
Rational Field
```

Die Rückgabewerte sind genau die gleichen wie von Parent(17) bzw. Parent(3/7). Wir werden diese Befehle in Kürze zur Typenumwandlung einsetzen.

2.9. Verschiedene Strukturen A und B können miteinander in Beziehung stehen. So kann es zum Beispiel eine natürliche *Einbettung* $A \hookrightarrow B$ von A nach B geben, die es auf natürliche Weise erlaubt, ein Objekt vom Typ A zu einem Objekt vom Typ B zu machen, oder es kann eine natürliche *Projektion* $B \twoheadrightarrow A$ geben, die es auf natürliche Weise erlaubt, ein Objekt vom Typ B zu einem Objekt vom Typ A zu machen. Dieses Denken in Typen, Strukturen und deren Beziehungen ist ein fundamentales Konzept in MAGMA. Das Umwandeln von Objekten mittels solcher Strukturbeziehungen heißt

Coercion in MAGMA und wird mit dem Operator ! ausgeführt. Haben wir zum Beispiel eine (wie auch immer definierte) Beziehung $A \to B$ und ist $a \in A$ (d.h. der Parent von a ist A), so wird die Coercion in MAGMA mittels `B!a` ausgeführt und ergibt ein zu B gehöriges Objekt, d.h. $B!a \in B$.

2.10. Betrachten wir ein konkretes Beispiel dieses abstrakten Konzepts. Natürlich ist jede ganze Zahl auf kanonische Weise auch eine rationale Zahl, d.h. nicht anderes, als dass wir eine kanonische Einbettung $\mathbb{Z} \hookrightarrow \mathbb{Q}$ haben. In der Sprache von MAGMA heißt das, dass wir eine Einbettung `Integer Ring` $\hookrightarrow$ `Rational Field` haben sollten, und dieser Einbettung ist sich MAGMA natürlich bewusst:

```
> Rationals()!17;
17
> Type(Rationals()!17);
FldRatElt
> Parent((Rationals()!17));
Rational Field
```

In dem Beispiel haben wir die ganze Zahl 17 mittels Coercion ! zu einer rationalen Zahl gemacht. Diese Art offensichtliche Typenumwandlung wird von MAGMA aber automatisch ausgeführt – zum Beispiel geschieht dies, wenn wir eine ganze Zahl und eine rationale Zahl addieren:

```
> 17+3/7;
122/7
```

Die Sinnhaftigkeit dieses Ausdrucks basiert einfach auf der Kommutativität des Diagramms

$$\begin{array}{ccccc} \mathbb{Q} & \times & \mathbb{Q} & \xrightarrow{+} & \mathbb{Q} \\ \uparrow & & \uparrow & & \uparrow \\ \mathbb{Z} & \times & \mathbb{Z} & \xrightarrow{+} & \mathbb{Z} \end{array}$$

Es gibt natürlich auch Fälle, in denen eine Strukturbeziehung nicht vorhanden ist und eine Coercion keinen Sinn ergibt:

```
> Integers()!(3/7);

>> Integers()!(3/7);
              ^
Runtime error in '!': Rational argument is not a whole integer
LHS: RngInt
RHS: FldRatElt
```

Die *echt*-rationale Zahl $\frac{3}{7}$ können wir natürlich nicht als ganze Zahl interpretieren und genauso wenig kann MAGMA das – deshalb gibt es eine Fehlermeldung.

2.11. Bemerkung. Folgendes Beispiel zeigt, dass das abstrakte Konzept von Strukturen und Coercion in MAGMA manchmal Probleme bereiten kann, wenn man sich nicht vollkommen bewusst ist, was man tut:

```
> 3/7+4/7;
1
> 3/7+4/7;
```

```
1
> Type(3/7+4/7);
FldRatElt
```

Wir haben also mit einer Operation von echt-rationalen Zahlen eine echt-ganze Zahl produziert. Diese ist aber trotzdem vom Typ FldRatElt und wird von MAGMA nicht automatisch mittels Coercion zum Typ RngIntElt umgewandelt. Das ist zwar in Ordnung, kann aber zu Problemen führen, wenn man ganzzahlige Ergebnisse von Rechnungen in $\mathbb{Q}$ als Typ RngIntElt benötigt.

> Rechnet man mit rationalen Zahlen und produziert dabei eine ganze Zahl, so ist diese immer noch vom Type FldRatElt! Um dies zu einer ganzen Zahl zu machen, muss man eine Coercion ausführen!

2.12. Sofern wir nur in $\mathbb{Z}$ und $\mathbb{Q}$ rechnen wollten, wären solche abstrakten Konzepte viel zu übertrieben. Da MAGMA aber auch wesentlich kompliziertere algebraische Strukturen mit viel subtileren Beziehungen zueinander beherrscht, muss man sich früher oder später um genau diese Konzepte Gedanken machen – umso früher umso besser.

§2C. Implementierte Funktionen (Intrinsics)

2.13. In MAGMA ist bereits eine Vielzahl von Funktionen, sogenannte *Intrinsics*, für den Umgang mit Objekten implementiert. Diese sind gewöhnlich mit – mehr oder weniger – einleuchtenden Namen versehen und funktionieren nach dem Schema:

IntrinsicName(EingabeObjekt$_1$, ..., EingabeObjekt$_n$); $\rightsquigarrow$ AusgabeObjekt

Eine Intrinsic erwartet eine bestimmte Anzahl von Eingabeobjekten eines bestimmen Typs. Mit diesen Eingabeobjekten tut die Intrinsic dann irgendetwas und gibt am Ende ein Ausgabeobjekt eines bestimmten Typs zurück. Es ist auch möglich, dass kein Objekt zurückgegeben wird oder, dass kein Eingabeobjekt erwartet wird. Schauen wir uns ein paar Beispiele an:

```
> Divisors(416);
[ 1, 2, 4, 8, 13, 16, 26, 32, 52, 104, 208, 416 ]
> PrimeDivisors(416);
[ 2, 13 ]
> Factorization(416);
[ <2, 5>, <13, 1> ]
> GreatestCommonDivisor(24,180);
12
```

Die erste Intrinsic Divisors gibt die Teiler der ganzen Zahl 416 zurück, die zweite gibt die Primteiler zurück, die dritte gibt die Primfaktorisierung $416 = 2^5 \cdot 13^1$ zurück, und die letzte gibt den größten gemeinsamen Teiler von 24 und 180 zurück – all dies kann MAGMA bereits und die Namen der Intrinsics sind auch relativ einleuchtend.

2.14. Wenn man sich nicht sicher ist, was eine Intrinsic – wie z.B. Divisors – genau tut, so kann man sich durch Eingabe des Intrinsicnamens ohne Argument eine Hilfe ausgeben lassen:

```
> Divisors;
Intrinsic 'Divisors'
```

```
Signatures:

    (<RngIntElt> n) -> SeqEnum

        A sequence containing all divisors of the positive integer n.

    (<RngIntEltFact> Q) -> SeqEnum

        Given an integer factorization sequence Q of n, return a
        sequence containing all the divisors of n, including 1 and n.

    (<RngOrdElt> x) -> SeqEnum

        All divisors (up to units) of x, which must lie in a maximal
        order.

    (<RngOrdIdl> x) -> SeqEnum

        All divisors of x, which must be an integral ideal of a
        maximal order.
```

Wir sehen hier in der ersten Zeile, dass Divisors ein *Intrinsic* ist, d.h. eine in MAGMA bereits fest implementierte – intrinsische – Funktion. Danach werden die sogenannten *Signaturen* gelistet, d.h. mögliche Eingabetypen und Ausgabetypen für diese Intrinsic. Die erste Signatur ist genau die für uns relevante. Sie besagt, dass wir der Intrinsic Divisors ein Objekt vom Typ RngIntElt, also eine ganze Zahl, übergeben können, und als Rückgabe ein Objekt vom Typ SeqEnum erhalten. Bei dem Typ SeqEnum handelt es sich einfach um Listen in MAGMA , die wir weiter unten noch genauer besprechen. Meistens ist zu einer Signatur auch noch ein Kommentar angegeben, der auch als Hilfe dient.

In der Ausgabe oben sehen wir, dass die Intrinsic Divisors noch für einige andere Typen definiert ist, die wir jetzt aber noch nicht verstehen müssen. Diese Intrinsic ist also *überladen*, d.h. der gleiche Intrinsicname ist für verschiedene Eingabetypen definiert – ein weiteres essentielles Konzept von MAGMA.

2.15. Aufgabe. Schauen Sie sich die Signaturen der Funktion Factorization an und finden Sie die im obigen Beispiel verwendete Signatur.

2.16. Aufgabe. Wir haben in §2B bereits die Intrinsics Type und Parent benutzt. Schauen Sie sich die Signaturen dieser Intrinsics an.

2.17. Die Addition + und Multiplikation * von ganzen bzw. rationalen Zahlen sind ebenfalls Intrinsics. Da man natürlich nicht Additionen in der Form +(2,3), sondern in der Form 2+3 eingeben möchte, sind dies nochmals spezielle Formen von Intrinsics, verhalten sich aber genauso. Wir können uns sogar die Signaturen dieser Intrinsics ansehen (hier nur der relevante Ausschnitt):

```
> '+';
Intrinsic '+'

Signatures:
```

```
(<RngIntElt> x, <RngIntElt> y) -> RngIntElt

    Sum of x and y.

(<FldRatElt> x, <FldRatElt> y) -> FldRatElt

    Sum of x and y.
```

Spätestens für Addition und Multiplikation wird deutlich, dass das Konzept der Überladung von Intrinsicnamen sehr wichtig ist – schließlich wollen wir nicht für jeden Typen, für den eine Addition definiert ist, einen neuen Intrinsicnamen vergeben, denn das wäre irgendwann schwer zu merken!

2.18. Da, wie bereits erwähnt, eine unglaubliche Vielzahl von Intrinsics schon implementiert ist, bekommt man früher oder später das Problem, dass man sich nicht mehr erinnern kann, wie eine bestimmte Intrinsic hieß. Hierbei ist die Tabulatortaste (TAB) ein sehr wichtiges Hilfsmittel. Gibt man nur einen Teil eines Intrinsicnamens ein und drückt zweimal schnell die Tabulatortaste, so bekommt man eine Auflistung aller Intrinsics (und ebenfalls aller benutzerdefinierten Funktionen und Prozeduren), die mit diesem Teil anfangen.

Wollen wir zum Beispiel die Primteiler einer Zahl berechnen und können uns nicht mehr an den korrekten Namen der Intrinsic erinnern, so können wir vielleicht zumindest erahnen, dass die Intrinsic mit Prime anfängt. Geben wir dies ein und drücken zweimal schnell die Tabulatortaste, so erhalten wir

```
> Prime
Prime                           PrimePolynomials
PrimeBasis                      PrimePowerKernelMatrix
PrimeBound                      PrimePowerNullspaceMatrix
PrimeComponents                 PrimePowerOrderElement
PrimeDivisors                   PrimePowerRepresentation
PrimeField                      PrimeRing
PrimeForm                       Primes
PrimeIdeal                      PrimesInInterval
PrimeOrderElement               PrimesUpTo
> Prime
```

Beim Durchsuchen der Liste finden wir die korrekte Funktion PrimeDivisors. Die Tabulatortaste hat darüber hinaus die Funktion der automatischen Vervollständigung der Eingabe.

2.19. Aufgabe. Berechnen Sie den Ausdruck

$$117! = \prod_{i=1}^{117} i \ .$$

Hinweis: Es handelt sich hierbei um die Fakultät von 117. Der englische Ausdruck dafür ist *factorial*.

2.20. Aufgabe. Berechnen Sie die Anzahl aller 6-elementigen Teilmengen einer 49-elementigen Menge. Hinweis: Es handelt sich hierbei um den Binomialkoeffizienten $\binom{49}{6}$.

§2D. Bezeichner

2.21. In MAGMA kann man für jedes Objekt auch eine Kurzbezeichnung einführen. Dies geschieht durch `Bezeichner := Ausdruck` und ist ungemein wichtig bei komplexeren Arbeiten. Wir schauen uns ein Beispiel an:

```
> x := 3;
> y := 2;
> x+y;
5
> x := y;
> x;
2
> y;
2
> y := 17;
> x;
2
> y;
17
> z := 7;
> x := y - z;
> x;
10
```

2.22. Als Bezeichner kann jede *Konkatenation* (d.h. Aneinanderreihung) von Buchstaben a bis z und A bis Z, von Zahlen 0 bis 9, und dem Symbol _ verwendet werden, wobei ein Bezeichner immer mit einem Buchstaben beginnen muss:

```
> 0z := 10;

>> 0z := 10;
     ^
User error: bad syntax
```

2.23. Aufgabe. Bestimmen Sie ohne MAGMA auszuführen den Wert von `x` in folgendem Beispiel:

```
> x := 2;
> y := 3;
> z := y;
> x := x - z;
```

2.24. Wir können sogar Bezeichner für Intrinsics anlegen:

```
> Addiere := '+';
> Addiere(17,3);
20
```

§2E. Aussagenlogik (Booleans)

2.25. MAGMA beherrscht die Aussagenlogik. Für Wahrheitswerte (*Booleans*) stellt MAGMA einen eigenen Datentyp namens `BoolElt` zur Verfügung. Es handelt sich

dabei einfach um die Ausdrücke `true` (wahr) und `false` (false). Mit diesen kann man dann die gewöhnlichen logischen Operationen wie `and`, `or` und `not` durchführen:

```
> x := true;
> y := false;
> x or y;
true
> x and y;
false
> not x;
false
> x eq y;
false
```

2.26. MAGMA wertet logische Ausdrücke immer von links nach rechts aus.

2.27. Aufgabe. Erstellen Sie mit MAGMA eine Wahrheitstabelle des logischen Ausdrucks

$$((p \to q) \wedge (r \to s) \wedge (p \vee \neg s)) \to (q \vee \neg r)$$

Was ist Ihre Schlussfolgerung? Hinweis: $p \to q$ ist logisch äquivalent zu $\neg p \vee q$.

2.28. Viele Intrinsics geben Booleans zurück. Es handelt sich dabei meist um Funktionen, die ein Objekt oder mehrere Objekte auf irgendwelche Eigenschaften überprüfen und genau dann `true` zurückgeben, wenn diese Eigenschaften erfüllt sind:

```
> IsPrime(17);
true
> IsPrime(16);
false
```

Weitere solche Intrinsics sind Vergleichstests, wie z.B. `eq` für $=$, `lt` für $<$, `le` für $\leq$, `gt` für $>$ und `ge` für $\geq$:

```
> 5 eq 7;
false
> 5 le 7;
true
> 5 ge 7;
false
```

§2F. Konditionalausdrücke

2.29. Mit Konditionalausdrücken kann man den Programmfluss von MAGMA beeinflussen. Diese Konditionalausdrücke haben folgende Struktur:

```
if BoolAusdruck1 then
   Anweisungen1
elif BoolAusdruck2 then
   Anweisungen2
else
   Anweisungen3
end if;
```

Hierbei wird zunächst getestet ob der logische Ausdruck `BoolAusdruck1` wahr ist. In diesem Fall wird `Anweisungen1` ausgeführt und sonst nichts weiter. Ist `BoolAusdruck1` falsch, so wird getestet, ob `BoolAusdruck2` wahr ist. In diesem Fall wird `Anweisungen2` ausgeführt und sonst nichts weiter. Ist `BoolAusdruck2` falsch, so wird `Anweisungen3` ausgeführt.

Die Zweige `elif` und `else` können auch weggelassen werden, es können aber auch beliebig viele `elif` Zweige auftreten. Die `if` Konstruktionen lassen sich beliebig verschachteln.

```
> x := 11;
> y := 0;
> if x gt 11 then
if> y := 1;
if> elif x eq 11 then
if> y := 0;
if> else
if> y := -1;
if> end if;
y;
0
```

2.30. Ich selbst finde die obige Eingabe von Konditionalausdrücken in MAGMA etwas lästig und bevorzuge eine zwar schwerer lesbare, aber wesentliche kompaktere Schreibweise:

```
if BoolAusdruck1 then; Anweisungen1; elif BoolAusdruck2 then;
Anweisungen2; else; Anweisungen3; end if;
```

Schauen wir uns ein Beispiel an:

```
> x := 11; y := 0;
> if x gt 11 then; y:=1; elif x eq 11 then; y:=0; else; y:=-1; end if;
> y;
0
> x := 10;
> if x gt 11 then; y:=1; elif x eq 11 then; y:=0; else; y:=-1; end if;
> y;
-1
> x := 12;
> if x gt 11 then; y:=1; elif x eq 11 then; y:=0; else; y:=-1; end if;
> y;
1
```

2.31. Aufgabe. Sei $f : \mathbb{N}_{>0} \to \mathbb{N}_{>0}$ mit

$$f(n) := \begin{cases} \frac{n}{2} & \text{falls } n \text{ gerade} \\ 3n+1 & \text{sonst.} \end{cases}$$

Schreiben Sie einen Konditionalausdruck in MAGMA, der einen zuvor definierten Startwert n in MAGMA zu $f(n)$ modifiziert Berechnen Sie damit zum Beispiel $f^{30}(17)$. Was fällt Ihnen an der Sequenz auf? Achtung: Sie werden wahrscheinlich bei Ihrem ersten Versuch – selbst, wenn Sie scheinbar alles richtig gemacht haben – auf eine Fehlermeldung stoßen! Sie müssen diese Fehlermeldung verstehen und selbst beheben!

§2G. Mengen

2.32. Ein wichtiges Konzept in MAGMA ist, dass man Objekte zu Mengen zusammenfassen kann. Mengen bilden einen eigenständige Datentyp in MAGMA namens `SetEnum` und das Zusammenfassen von Objekten zu Mengen ist sehr intuitiv:

```
> M := {2,3,5,7,11};
> M;
{ 2, 3, 5, 7, 11 }
> Type(M);
SetEnum
```

2.33. Es ist wichtig, zu beachten, dass die Mengen in MAGMA sich wirklich wie mathematische Mengen verhalten. Zum Beispiel werden Wiederholungen und Umordnungen bei der Mengendefinition ignoriert:

```
> {1,2,2,2,2,2};
{ 1, 2 }
> { 3,1,9,2};
{ 1, 2, 3, 9 }
```

2.34. Sehr häufig arbeitet man mit Mengen, die ein gewisses Intervall ganzer Zahlen enthalten. In MAGMA existiert zur Bildung solcher Mengen die Abkürzung `{a..b}`, wobei a und b ganze Zahlen sind:

```
> {0..11};
{ 0 .. 11 }
```

2.35. Da Mengen einen eigenen Datentyp bilden, können wir sogar wieder Mengen von Mengen bilden! Ein Beispiel:

```
> M := {2,3,5,7,11};
> N := {4,6,8,9,10};
> P := {{}, M,N};
> P;
{
    { 4, 6, 8, 9, 10 },
    {},
    { 2, 3, 5, 7, 11 }
}
```

Im obigen Beispiel sehen wir, dass auch die leere Menge $\emptyset = \{\}$ in MAGMA existiert.

2.36. Jede Menge in MAGMA hat ein sogenanntes *Universum*. Es handelt sich dabei um eine Struktur, in der alle Elemente der Menge leben (genauer: in die alle Elemente coerced werden können). Dieses Universum ermittelt man mit dem Befehl `Universe`. Bei einer Menge ganzer Zahlen wie im obigen Beispiel, ist die Struktur `Integer Ring` das Universum.

Wichtig für die Mengenbildung ist, dass die Elemente der zu bildenden Menge wirklich ein gemeinsames Universum haben. Wir schauen uns ein Beispiel an, wo dies offensichtlich existiert und eins, wo dies offensichtlich nicht existiert:

```
> M := {2,3};
> N := {3/7,11/3};
> Universe(M);
```

```
Integer Ring
> Universe(N);
Rational Field
> M := {2,3};
> N := { {2,3} };
> Universe(M);
Integer Ring
> Universe(N);
Set of subsets of Integer Ring
> P := { M,N };

>> P := { M,N };
         ^
Runtime error in { ... }: Cannot coerce argument 2 into the universe
```

2.37. Mit Mengen, die in ein gemeinsames Universum eingebettet werden können, kann man in MAGMA alle gewöhnlichen Mengenoperationen wie Vereinigung join und Schnitt meet durchführen, und die Anzahl der Elemente mittels # ermitteln:

```
> M := {0..10};
> N := {-2,0,2,4,6,8,10,12};
> M meet N;
{ 0, 2, 4, 6, 8, 10 }
> M join N;
{ -2, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12 }
> M diff N;
{ 1, 3, 5, 7, 9 }
> #(M diff N);
5
```

2.38. Mittels in kann man testen, ob Objekte in einer gegeben Menge liegen oder nicht:

```
> M := {2,3,6,7};
> 3 in M;
true
> 9 in M;
false
> 3 notin M;
false
```

Analog kann man mit subset testen, ob eine Menge in einer anderen Menge enthalten ist (es handelt sich dabei um $\subseteq$, nicht um $\subsetneq$):

```
> {2,3,4} subset {1..10};
true
```

2.39. Aufgabe. Sei $M := \{1, \ldots, 6\}$. Bestimmen Sie mit MAGMA die Menge aller Teilmengen und die Menge aller drei-elementigen Teilmengen von M. Hinweis: Es gibt schon entsprechende Intrinsics, Sie müssen diese nur finden.

2.40. Ist M eine Menge und P eine "Eigenschaft" auf M, d.h. eine Funktion $P : M \to \{\text{true}, \text{false}\}$, so betrachtet man oft die Teilmenge $\{x \in M \mid P(x) = \text{true}\}$. Solche

Teilmengenbildungen mittels Konditionalausdrücken unterstützt MAGMA ebenfalls mittels $\{x : xinM|P(x)\}$. Ein Beispiel:

```
> M := {1..10};
> {x : x in M | IsEven(x) };
{ 2, 4, 6, 8, 10 }
> {x : x in M | IsOdd(x) and x notin {1,3,7,9} or x gt 6 };
{ 5, 7, 8, 9, 10 }
```

2.41. Aufgabe. Bestimmen Sie die Menge M aller Primzahlen p, die sich als Summe dreier Primzahlen a, b, c mit $3800 \leq a, b, c \leq 4100$ schreiben lassen. Gibt es Primzahlen zwischen $\min(M)$ und $\max(M)$, die nicht in M liegen?

§2H. Sequenzen

2.42. Wollen wir in MAGMA eine Reihe von Objekten aus einem gemeinsamen Universum unter Beachtung der Reihenfolge und mit der Möglichkeit von Wiederholungen abspeichern, so sind die Mengen nicht der richtige Datentyp. Aber auch für dieses Problem stellt MAGMA einen eigenen Datentyp, die *Sequenzen* (Typ `SeqEnum`), zur Verfügung. Die Erzeugung von Sequenzen funktioniert genauso wie bei Mengen:

```
> S := [ 5,9,1,1,1,2 ];
> S;
[ 5, 9, 1, 1, 1, 2 ]
> Type(S);
SeqEnum
> 5 in S;
true
> [ x : x in S | x gt 1 ];
[ 5, 9, 2 ]
```

2.43. Natürlich können wir auch wieder Sequenzen von Sequenzen (oder Mengen von Sequenzen oder Sequenzen von Mengen!) bilden:

```
> [ [1,1,1], [1,1,1], [9], [] ];
[
    [ 1, 1, 1 ],
    [ 1, 1, 1 ],
    [ 9 ],
    []
]
```

2.44. Da eine Sequenz die Reihenfolge der Objekte beachtet, können wir auch gezielt auf das n-*te Folgenglied* einer Sequenz zugreifen. Dies geschiet mittels $S[n]$. Man kann auch eine zusammenhängende Teilfolge mittels `S[m..n]` für $1 \leq m \leq n \leq \#S$ ausgeben lassen

```
> S := [9,1,11];
[ 9, 1, 11 ]
> S[2];
1
> S[3] := 85;
> S;
```

```
[ 9, 1, 85 ]
> S := [1,2,3,4,5,6,7,8];
> S[3..7];
[ 3, 4, 5, 6, 7 ]
```

Achtung. Das erste Folgenglied einer Sequenz S ist S[1]. In anderen Programmiersprachen (z.B. C) ist das erste Folgenglied das 0-te.

2.45. Wie bei Mengen erhält man die Anzahl der Folgenglieder einer Sequenz S mittels #S.

2.46. Das Analogon zu join für Mengen ist cat für Sequenzen. Mit diesem Befehl werden zwei Sequenzen ordnungstreu aneinandergehängt:

```
> S := [1,4,9];
> T := [1,9,4];
> S cat T;
[ 1, 4, 9, 1, 9, 4 ]
```

2.47. In MAGMA kann man leicht Sequenzen auf die offensichtliche Art zu Mengen "degenerieren":

```
> S := [1,1,1,9,2,2];
> SequenceToSet(S);
{ 1, 2, 9 }
```

Andererseits kann man auch Mengen zu Sequenzen machen, wobei hier darauf zu achten ist, dass MAGMA die Sequenz dann in irgendeiner Reihenfolge anlegt (diese ist zwar meistens reproduzierbar, jedoch sollte man sich darauf nicht immer verlassen):

```
> S := {1,2,3};
> SetToSequence(S);
[ 1, 2, 3 ]
```

§2I. Tupel

2.48. Bisher war es uns nicht möglich, Objekte aus verschiedenen Universen zusammenzufassen – sowohl Mengen als auch Sequenzen benötigen für ihre Erzeugung immer ein gemeinsames Universum der Objekte. Die *Tupel* in MAGMA (vom Typ Tup) geben uns eine Möglichkeit eine feste Anzahl Objekten aus beliebigen Universen (unter Beachtung der Reihenfolge und mit möglichen Wiederholungen) zusammenzufassen – bei Tupeln handelt es sich also um Elemente aus einem kartesischen Produkt von beliebigen Strukturen. Das Erzeugen von Tupeln erfolgt über < Objekt$_1$, Objekt$_2$, ..., Objekt$_n$>, wobei MAGMA hier selbst in jeder Komponente das korrekte Universum wählen wird, wenn man das nicht explizit angibt.

```
> T := <5, {1,3}, [1,2,3]>;
> T;
<5, { 1, 3 }, [ 1, 2, 3 ]>
> Type(T);
Tup
> Parent(T);
Cartesian Product<Integer Ring, Set of subsets of Integer Ring, Set of
```

```
sequences over Integer Ring>
> T[3];
[ 1, 2, 3 ]
> T[1] := 17;
> T;
<17, { 1, 3 }, [ 1, 2, 3 ]>
> T[1] := {1,2};

>> T[1] := {1,2};
     ^
Runtime error in :=: New component not in cartesian product
```

§2J. Abbildungen

2.49. MAGMA beherrscht auch das Konzept von *Abbildungen* (Typ Map) zwischen Strukturen. Sind A und B Strukturen und $f : A \to B$ eine Abbildung, so kann man f in MAGMA mittels map<A -> B | Γ_f> definieren, wobei $\Gamma_f := \{(a, f(a)) \mid a \in A\} \subseteq A \times B$ der *Graph* von f ist und als Menge (oder Sequenz) von Tupeln <a,f(a)> übergeben wird. Ein Beispiel

```
> f := map<{1,2,3} -> {1,2,3} | { <1,3>, <2,2>, <3,1> }>;
> f;
Mapping from: { 1, 2, 3 } to { 1, 2, 3 }
    <1, 3>
    <2, 2>
    <3, 1>
> Type(f);
Map
> f(1);
3
> f(2);
2
> f(3);
1
```

§2K. Intrinsics, Funktionen und Prozeduren

2.50. Wir haben nun schon eine Reihe von bereits implementierten Intrinsics benutzt und wollen nun diskutieren, wie man eigene Intrinsics implementiert. Zunächst besprechen wir aber zwei (besonders für den Anfänger) einfachere Varianten, wie man eigene Programme in MAGMA implementieren kann: die Funktionen und die Prozeduren.

2.51. Bei einer *Funktion* in MAGMA (Typ UserProgram) handelt es sich einfach um eine feste Vorschrift, die aus einer Reihe von Eingabeobjekten eine Reihe von Ausgabeobjekten produziert und dieses zurückgibt:

```
function MeineFunktion(EingabeObjekt_1, ..., EingabeObjekt_n)
    Anweisungen;
    return AusagbeObjekt_1, AusgabeObjekt_2, ..., AusgabeObjekt_m;
end function;
```

Bei der Funktionsdefinition muss man sich (im Gegensatz zu den Intrinsics wie wir sehen werden) keine besonderen Gedanken um Datentypen machen. Ein Beispiel:

```
> f := function(n)
function> return n^2;
function> end function;
> Type(f);
UserProgram
> f(5);
25
```

2.52. Ähnlich wie bei Konditionalausdrücken gibt es auch eine Kurzschreibweise für die Funktionsdefinition, sofern das Ausgabeobjekt durch nur einen Ausdruck beschrieben werden kann:

```
func<EingabeObjekt_1, ..., EingabeObjekt_n | Ausdruck >;
```

Obiges Beispiel in Kurzschreibweise ist also:

```
> f := func<n|n^2>;
> f(4);
16
```

2.53. Komplexere Programme will man natürlich nicht in der MAGMA-Eingabeaufforderung erstellen. Man kann all dies in eine Datei auslagern (die man mit einem beliebigen Editor editieren kann) und diese Datei kann man in MAGMA mittels `load "dateiname"` einbinden. Ändert man etwas in der Datei, so muss man sie jedes Mal auf diese Weise wieder neu in MAGMA laden.

Ein simpler Texteditor für das Terminal ist zum Beispiel *nano*. Will man die Datei *meinefunktionen.m* editieren (oder neu erstellen), so erfolgt das im Terminal mittels `nano meinefunktionen.m`. Es erscheint ein neues Fenster, in dem man seine Programme schreiben kann. Man kann beliebig viele Funktionen in eine Datei schreiben und diese auch untereinander verwenden. Speichern kann man die Datei mittels Ctrl + O, beenden kann man nano mittels Ctrl + X. In MAGMA lädt man die Datei dann mittels `load "meinedatei.m"` und es stehen alle Funktionen der Datei zur Verfügung – sofern man keinen Fehler bei der Programmierung gemacht hat.

Sie können sich auf dem studpool von einem Linux Computer aus auch mittels `ssh -X -C` einloggen. Dann stehen Ihnen auch graphische Editoren wie *kwrite* zur Verfügung.

2.54. Ein wichtiges Konzept bei der Programmierung ist, dass sich eine Funktion selbst wieder aufrufen kann – man nennt dies *Rekursion*. In der folgenden Aufgabe kann das Prinzip der Rekursion eingeübt werden.

2.55. Aufgabe. Die *Fibonacci-Folge* $(F_n)_{n\in\mathbb{N}}$ ist rekursiv definiert durch:

- $f_0 := 0$ und $f_1 := 1$.
- $f_n := f_{n-1} + f_{n-2}$ für $n \geq 2$.

Implementieren Sie eine Funktion `MyFibonacci(n)`, die F_n mittels Rekursion zurückgibt. Berechnen Sie `MyFibonacci(35)` und vergleichen Sie die Laufzeit mit der in MAGMA bereits implementierten Intrinsic `Fibonacci(35)`. Was ist Ihre Erklärung für den Laufzeitunterschied?

2.56. Aufgabe. Implementieren Sie eine Funktion `MyFibonacci`, die schneller ist, als die naive rekursive Version.

2.57. Wenn wir ein Objekt einer Funktion übergeben und dieses übergebe Objekt ändern, so wirkt sich diese Änderung *nicht* auf das Originalobjekt aus – in der Funktion wird eine *Kopie* dieses Objekts angelegt.

```
 > f := function(n)
function> n := 0;
function> return n;
function> end function;
> n := 10;
> f(10);
0
> n;
10
```

Für den direkten Zugriff auf ein Objekt stellt MAGMA einen weiteren Typ von Benutzerprogrammen zur Verfügung, die *Prozeduren* (ebenfalls vom Typ `UserProgram`). Im Gegensatz zu Funktionen geben Prozeduren kein Objekt zurück, erlauben es aber ein übergebenes Objekt direkt zu ändern. Prozeduren haben zwei verschiedenen Typen von Eingabeobjekten: gewöhnliche und dann in der Prozedur unveränderliche Eingabeobjekte, und *Referenzen* auf Objekte. Eine Referenz eines Objekts `X` erhält man mittels `~X`. Übergibt man an eine Prozedur also Referenzen auf Objekte, so wirken sich Veränderungen auf diese Objekte auch direkt auf das referenzierte Objekt, d.h. auf das Originalobjekt, aus.

```
procedure MeineProzedur(EingabeObjekt_1, ..., EingabeObjekt_n,
~Referenz_1, ..., ~Referenz_m)
```

Betrachten wir ein Beispiel:

```
> p := procedure(m,~n)
procedure> n := m+1;
procedure> end procedure;
> Type(p);
UserProgram
> n:=5; m:=11;
> p(m,~n);
> m;
11
> n;
12
> n := p(n);

>> n := p(n);
          ^
Runtime error: Attempt to call user procedure as a function
```

in dem Beispiel wird das Objekt m ganz gewöhnlich übergegen und eine Referenz auf das Objekt n übergeben. Änderungen an n innerhalb der Prozedur wirken sich dann direkt auf das Originalobjekt n aus. Das Objekt m können wir innerhalb der Prozedur jedoch nicht verändern.

2.58. Aufgabe. Implementieren Sie eine Prozedur `EvenPart(~M)`, die von einer Menge M ganzer Zahlen alle ungeraden Zahlen entfernt.

2.59. Aufgabe. Implementieren Sie eine Prozedur $RemoveDuplicates(\tilde{}S)$, die von einer Sequenz S alle Duplikate (unter Beibehaltung der Ordnung!) entfernt.

2.60. Nun kommen wir schließlich zu den Intrinsics selbst. Es ist für den Benutzer auch möglich, eigene Intrinsics zu programmieren, die MAGMA so behandelt, als wären es feste Bestandteile von MAGMA . Da man Intrinsics wie besprochen überladen kann – d.h. es gibt mehrere Intrinsics des gleichen Namens, die aber unterschiedlich auf verschiede Eingabetypen reagieren können – muss man für die Implementierung von Intrinsics auch die Typen der Eingabeobjekte angeben. Intrinsics kann man nicht in der MAGMA-Eingabeaufforderung implementieren – sie müssen in einer Datei programmiert werden, die dann mittels `Atach("datei.m")` in MAGMA eingebunden wird. Eine Intrinsic wird wie folgt definiert

```
intrinsic MeineIntrinsic(EingabeObjekt_1::Typ_1, ...,
EingabeObjekt_n::Typ_n) -> AusgabeTyp_1, ..., AusgabeTyp_m
{Signatur-Beschreibung. Diese ist verpflichtend, kann aber leer sein.}
    Anweisungen;
    return AusgabeObjekt_1, ..., AusgabeObjekt_m;
end intrinsic;
```

Eine Intrinsic akzeptiert ebenfalls auch Referenzen als Eingaben – dann darf jedoch kein Objekt zurückgegeben werden. Ein Beispiel:

```
intrinsic Quadrieren(n::RngIntElt) -> RngIntElt
{Gibt n^2 zurueck.}
    return n^2;
end intrinsic;

intrinsic Quadrieren(S::SetEnum) -> SetEnum
{Gibt alle Paare von Elementen aus S zurueck.}
    return { <x,y> : x,y in S };
end intrinsic;
```

Schreiben wir das in eine Datei quadieren.m und rufen `Attach("quadrieren.m")` auf, so werden die beiden Intrinsics in MAGMA eingefügt. Die Signaturen mit Kommentaren erhält man dann wie üblich über `Quadrieren` ohne Argument.

§2L. Kommentare

2.61. Es ist hilfreich (und bei größeren Projekten eine Pflicht!) Quellcode zu dokumentieren. MAGMA stellt dafür zwei Wege zur Verfügung. Sowohl sämtlicher Text *in der selben Zeile* wie `//` als auch sämtlicher Text *zwischen* `/*` und `*/` wird von MAGMA als Kommentar gewertet.

```
Anweisungen1;
// Dies ist ein Kommentar
Anweisungen2; // Anweisungen2 ist kein Kommentar mehr
/*
    Das ist ein
    laengerer Komentar
    ueber mehrere Zeilen.
*/
Anweisungen3;
```

§2M. Schleifen

2.62. Ein in fast allen Programmiersprachen vorkommendes Konzept ist das der Schleifen. Dabei wird zwischen zwei Schleifentypen unterschieden: Den for-Schleifen und den while-Schleifen.

2.63. Bei einer *for-Schleife* holt man Objekte (oder sogar Tupel von Objekten) aus einer gegebenen Struktur (man *iteriert* über die Objekte der Struktur) und führt für jedes der Objekte einen bestimmten Block von Anweisungen durch. Die Form einer for-Schleife in MAGMA ist:

```
for Objekt_1, ..., Objekt_n in Struktur do
    Anweisungen;
end for;
```

Dabei ist wichtig, dass die Struktur, über die iteriert wird, einen *Iterator* besitzt, d.h. MAGMA muss eine Möglichkeit haben, alle Objekte der Struktur aufzuzählen. Darum muss man sich aber im allgemeinen nicht kümmern. Ein Beispiel

```
> n := 0;
> for i in {1..5} do
for> n +:= i; // n +:= i ist eine Kurzform fuer n := n + i
for> end for;
> n;
15
```

2.64. Wie üblich kann man Schleifen auch wieder verschachteln. Dass man gleich über Tupel von Objekten aus der Struktur iterieren kann, erlaubt es, einige Schleifen wesentlich kompakter und ohne Verschachtelung zu schreiben. So ist zum Beispiel

```
> n:=0;
> for i in {1..3} do
for> for j in {1..3} do
for|for> n +:= i*j;
for|for> end for;
for> end for;
> n;
36
```

äquivalent zu

```
> n := 0;
> for i,j in {1..3} do
for> n +:= i*j;
for> end for;
> n;
36
```

2.65. Wie bei den Konditionalausdrücken gibt es auch eine kompakte Schreibweise für for-Schleifen:

```
for Objekt_1, ..., Objekt_n in Struktur do; Anweisungen; end for;
```

2.66. Bei einer *while-Schleife* wird solange ein Block von Anweisungen durchgeführt, wie ein bestimmter boolscher Ausdruck wahr ist. Die Form einer while-Schleife in MAGMA ist:

```
while BoolAusdruck do
    Anweisungen;
end while;
```

Ein Beispiel:

```
> n := 10;
> while n gt 0 do
while> n -:= 1; // Eine Kurzschreibweise fuer n := n - 1
while> end while;
> n;
0
```

2.67. Wieder gibt es auch eine Kurzschreibweise für die while-Schleife:

```
while BoolAusdruck do; Anweisungen; end while;
```

2.68. Sowohl die for- als auch die while-Schleifen kann man durch die Befehle continue und break beeinflussen. Erreicht der Anweisungsblock einer Schleife den continue Befehl, so wird direkt zu nächsten Iteration der Schleife gesprungen; wird ein break Befehl erreich, so wird die Schleife ganz abgebrochen.

§2N. Strings

2.69. MAGMA hat einen eigenen Datentyp zur Verarbeitung von Zeichenketten (*Strings*), nämlich MonStgElt. Als String wird in MAGMA alles intepretiert, was zwischen zwei Anführungszeichen " steht.

```
> str := "Hallo";
> str;
Hallo
> Type(str);
MonStgElt
```

2.70. Für Strings stellt MAGMA verschiedene Operationen wie die *Konkatenation* mittels cat oder $*$ zur Verfügung.

```
> str1 := "Hallo";
> str2 := " ";
> str3 := "Welt!";
> str := str1 * str2 * str3;
> str;
Hallo Welt!
```

§2O. Ausgabe

2.71. Mit dem Befehl

```
print Ausdruck1, ..., Ausdruck_n;
```

kann man in MAGMA den Wert von Ausdrücken ausgeben. Das ist besonders hilfreich für Ausgaben innerhalb von Funktionen, Prozeduren und Intrinsics. Ein Beispiel:

```
> n := 5;
> str := "Hallo";
> print n, str;
5 Hallo
```

2.72. Besonders nützlich ist, dass man den Wert von Ausdrücken mittels Sprint in Strings umwandeln kann und diese Strings dann zu komplexeren Ausdrücken zusammensetzen kann:

```
> n := 5;
> M := {1,2,3};
> Sprint(n)*" liegt nicht in der Menge "*Sprint(M)*".";
5 liegt nicht in der Menge { 1, 2, 3 }.
```

§3. Gruppen

Zur Beschreibung von Gruppen im Computer muss es endliche Regeln geben, die diese Gruppen (oder zumindest Teilinformationen) beschreiben. Dies schließt von vornherein bereits einige Gruppen aus (z.B. nicht endlich präsentierte Gruppen wie das Kranzprodukt $\mathbb{Z} \wr \mathbb{Z}$). Es gibt verschiedene Wege, Gruppen im Computer zu beschreiben und je nach Art der Beschreibung gibt es unterschiedlich mächtige bzw. sinnvolle Methoden um in diesen zu arbeiten.

MAGMA kann mit verschiedensten Arten (d.h. Beschreibungen) von Gruppen umgehen und stellt auch für jede Art einen eigenen Typ und eigene Intrinsics zur Verfügung. Meistens gibt es auch Intrinsics, die von einer Beschreibung in eine andere Beschreibung umwandeln können. Man kann sowohl mit endlichen als auch mit unendlichen Gruppen in MAGMA arbeiten – natürlich nur sofern es endliche Regeln gibt, die diese Gruppen beschreiben. Wir wollen und können nur einen ganz kleinen Teil davon besprechen.

3.1. Unter den endlichen Gruppen sind folgende Arten in MAGMA vorhanden:
(a) Permutationsgruppen (Typ GrpPerm).
(b) Endliche Matrixgruppen (Typ GrpMat).
(c) Endliche auflösbare Gruppen in power-conjugate Darstellung (Typ GrpPC).
(d) Endliche abelsche Gruppen (Typ GrpAb).
(e) Endlichen polyzyklische Gruppen (Typ GrpGPC).

3.2. Unter den (un)endlichen Gruppen sind folgende Arten in MAGMA vorhanden:
(a) Endlich präsentierte Gruppen (Typ GrpFp).
(b) Gruppen mit Termersetzungssystem (Typ GrpRWS).

3.3. Ganz allgemein basiert nahezu jede Beschreibung einer Gruppe im Computer auf der Angabe einer endlichen Menge von *Erzeugern* und einer endlichen Menge von *Regeln*, die beschreiben, wie mit den Erzeugern zu rechnen ist. Die Art der Erzeuger, die Art der Regeln und die zu Verfügung stehenden Methoden in der Gruppe zu rechnen variieren je nach Art der Beschreibung.

3.4. Je nach Kontext muss der Anwender selbst entscheiden, wie er eine gegebene Gruppe oder ein gegebenes gruppentheoretisches Problem in MAGMA beschreibt.

Prinzipiell kann man in MAGMA aber verschiedene Typen von Gruppen ineinander umwandeln (zumindest kann man es versuchen, nicht immer ist dies möglich).

Ich werde hier lediglich einige verschiedenen Typen und Möglichkeiten aufzeigen – den Rest muss der Anwender selbst entscheiden.

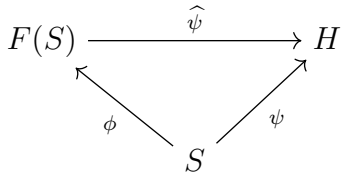
3.5. Das Definieren von Morphismen, von Untergruppen und von Quotienten werde ich nur einmal bei freien Gruppen und bei endlich präsentierbaren Gruppen besprechen. Bei allen anderen Gruppentypen funktioniert das dann analog. Gleiches gilt für die Berechnung des Zentrums – und eigentlich aller Intrinsics für Gruppen. Wie immer gilt: Je nach Gruppentyp können gemeinsame und unterschiedliche Intrinsics zur Verfügung stehen.

§3A. Freie Gruppen

3.6. Wir beginnen mit endlich präsentierten Gruppen, weil deren Darstellung und Funktionsweise wahrscheinlich am einleuchtendsten ist, und die meisten Intrinsics auch analog für andere Typen von Gruppen funktionieren. Weiterhin ist die in 3.3 zugrundeliegende Idee für jegliche Beschreibung einer Gruppe im Computer – nämlich mittels Erzeugern und je nach Typ verschieden gearteter Regeln – hier am deutlichsten.

Zur Definition endlich präsentierter Gruppen benötigen wir jedoch zunächst den Begriff der freien Gruppen.

3.7. Definition. Eine *freie Gruppe* über einer Menge S ist eine Gruppe $F(S)$ zusammen mit einem Morphismus $\phi : S \to F(S)$ von Mengen, sodass es für jeden anderen Mengen-Morphismus $\psi : S \to H$ in eine Gruppe H genau einen Gruppenmorphismus $\widehat{\psi} : F(S) \to H$ gibt, sodass das Diagramm



kommutiert.

3.8. Bemerkung. Falls eine freie Gruppe $F(S)$ über S existiert, so folgt aus der charakterisierenden *universellen Eigenschaft*, dass $G(S)$ eindeutig bis auf Isomorphie ist.

3.9. Bemerkung. Die universelle Eigenschaft heißt nichts weiter, als dass ein Gruppenmorphismus aus einer freien Gruppe $F(S)$ in eine Gruppe bereits eindeutig festgelegt ist durch die Bilder der Elemente $s \in S$.

3.10. Theorem. Eine freie Gruppe existiert über jeder Menge S. ■

Beweisidee. Sei $F(S)$ die Menge aller Ausdrücke der Form $s_1^{e_1} s_2^{e_2} \cdots s_n^{e_n}$ mit $s_i \in S$ und $e_i \in \mathbb{Z}$. Diese Mengen kann man jetzt mit den offsichtlichen Regeln $s^e s^f = s^{e+f}$ zu einer Gruppe mit dem leeren Ausdruck als Identität machen. Dies ist dann in der Tat bereits eine freie Gruppe über S. ■

3.11. Definition. Eine Gruppe G heißt *frei*, falls es eine Teilmenge $S \subseteq G$ gibt, sodass $G \cong F(S)$. Man nennt S dann ein *freies Erzeugendensystem* von G.

3.12. Theorem. Hat G sowohl S als auch S' als freies Erzeugendensystem, so haben S und S' die gleiche Mächtigkeit. Diese nennt man dann den *freien Rang* von G. ■

3.13. Beispiel. Die freie Gruppe vom Rang 1 ist einfach $\mathbb{Z}$.

3.14. Aufgabe. Wieso ist die freie Gruppe vom Rang $n \geq 2$ nicht $\mathbb{Z}^n$?

3.15. In MAGMA kann man nun die freie Gruppe vom Rang n mittels `FreeGroup(n)` erzeugen und in ihr rechnen. Zugriff auf den i-ten freien Erzeuger erhält man mittels `F.i`, wenn F eine freie Gruppe ist:

```
> F := FreeGroup(3);
> F;
Finitely presented group F on 3 generators (free)
> Type(F);
GrpFP
> Generators(F);
{ F.1, F.2, F.3 }
> F.1*F.3^2;
F.1 * F.3^2
> F.2^-1*F.2;
Id(F)
> F.1*F.2 eq F.2*F.1;
false
```

Der Typ einer freien Gruppe ist `GrpFP`, was allgemeiner der Typ für endlich präsentierbare Gruppen ist (siehe nächsten Abschnitt).

3.16. Es ist hilfreich, beim Anlegen der freien Gruppe gleich Bezeichner für die freien Erzeuger anzulegen:

```
> F<s,t,u> := FreeGroup(3);
> Generators(F);
{ s, t, u }
> s^2*u*t;
s^2 * u * t
```

3.17. Mittels `Eltseq` (das steht für *element to sequence*) kann man von einem Wort $W = g_{i_1}^{\pm 1} \cdots g_{i_m}^{\pm 1}$ einer Gruppe mit Erzeugern $(g_i)_{i=1}^n$ die Sequenz $(\pm i_j)_{j=1}^m$ ausgeben, die man danach wieder mittels Coercion zu einem Wort der Gruppe machen kann. Dies ist hilfreich beim Konstruieren von Gruppenelementen oder beim späteren Umwandeln von Gruppenelementen zwischen verschiedenen Beschreibungen der gleichen Gruppe.

```
> F<s,t,u> := FreeGroup(3);
> Eltseq(s*t^2*u^-1);
[ 1, 2, 2, -3 ]
> F![1,2,2,-3];
s * t^2 * u^-1
```

3.18. Die charakterisierende Eigenschaft freier Gruppen erlaubt es uns, ganz einfach Gruppen-Homomorphismen zwischen freien Gruppen zu definieren: Wir müssen einfach nur Bilder der freien Erzeuger vorgeben. In MAGMA läuft dies mit dem Konstruktor `hom` ganz analog zum Konstruktor `map` für Abbildungen zwischen endlichen Mengen:

```
> F<s,t,u> := FreeGroup(3);
> f := hom<F->F | { <s,t>, <t,u>, <u,s>} >;
> Type(f);
HomGrp
> f(s);
t
> f(t);
u
> f(u);
s
```

§3B. Endlich präsentierte Gruppen

Freie Gruppen alleine sind natürlich etwas langweilig, aber es wird wesentlich interessanter, wenn wir Quotienten freier Gruppen betrachten.

3.19. Definition. Ist S eine Menge und R eine Menge von *Wörtern* in S, d.h. im Prinzip $R \subseteq F(S)$, so bezeichnen wir mit $\langle S \mid R\rangle$ den Quotienten $F(S)/\langle R\rangle$, wobei $\langle R\rangle$ den durch R erzeugten Normalteiler in $F(S)$ bezeichnet. Man nennt $\langle S \mid R\rangle$ die *Gruppe mit Erzeugern S und Relationen R*, und nennt das Paar (S, R) die *Präsentation* dieser Gruppe.

3.20. Definition. Ist G eine Gruppe und $G \cong \langle S \mid R\rangle$, so nennen wir $(S \mid R)$ eine *Präsentation* von G.

3.21. Aufgabe. Zeigen Sie, dass jede Gruppe G unendliche viele Präsentationen besitzt. In jedem Fall existiert aber eine Präsentation und daher können wir auf diese Weise *jede* Gruppe beschreiben.

3.22. Definition. Eine Gruppe G heißt *endlich präsentierbar*, falls G eine Präsentation (S, R) besitzt, sodass sowohl S als auch R endlich sind.

3.23. Beispiel.

(a) $\langle x \mid x^n\rangle$ is eine endliche Präsentation von $\mathbb{Z}/n\mathbb{Z}$.
(b) $\langle x, y \mid xy = yx\rangle$ is eine endliche Präsentation von $\mathbb{Z} \times \mathbb{Z}$.
(c) $\langle x, y \mid xy = yx, x^m, y^n\rangle$ is eine endliche Präsentation von $\mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$.

Obwohl es in MAGMA auch eine Abkürzung zur Erzeugung endlich präsentierter Gruppen gibt, wollen wir zunuächst den Weg über den Quotienten einer freien Gruppe beschreiten.

3.24. Die durch eine Teilmenge R erzeugte Untergruppe einer freien Gruppe F kann man in MAGMA mittels `sub<F|R>` erzeugen:

```
> F<x,y> := FreeGroup(2);
> R := sub<F|{x^2,y^2}>;
> R;
```

```
Finitely presented group R on 2 generators
Generators as words in group F
    R.1 = x^2
    R.2 = y^2
> Type(R);
GrpFP
```

3.25. Den durch eine Teilmenge (oder Untergruppe) R einer freien Gruppe F definierten Quotienten $F/\langle R\rangle$ kann man in MAGMA mittels `quo<F|R>` erzeugen. Dabei wird neben dem Quotienten $F/\langle R\rangle$ auch der Quotientenmorphism $F \twoheadrightarrow F/\langle R\rangle$ zurückgegeben:

```
> F<x,y> := FreeGroup(2);
> R := sub<F|{x^2,y^2}>;
> Q,q := quo<F|R>;
> Q;
Finitely presented group Q on 2 generators
Relations
    Q.1^2 = Id(Q)
    Q.2^2 = Id(Q)
> q;
Mapping from: GrpFP: F to GrpFP: Q
Composition of Mapping from: GrpFP: F to GrpFP: F and
Mapping from: GrpFP: F to GrpFP: Q
> Type(Q);
GrpFP
```

Auf genau diese Art können wir in MAGMA endlich präsentierbare Gruppen erzeugen. Es gibt aber noch eine kürzere Möglichkeit.

3.26. Die endlich präsentierbare Gruppe $\langle S \mid R\rangle$ kann man in MAGMA auch direkt mittels `Group< S | R>` erzeugen. Die Relationen kann man auch in der Form `a = b`, was äquivalent zu `ab^-1` ist, übergeben. Ein Beispiel

```
> G<s,t> := Group< s,t | s^2, t^2, s*t*s = t*s*t >;
> G;
Finitely presented group G on 2 generators
Relations
    s^2 = Id(G)
    t^2 = Id(G)
    s * t * s = t * s * t
> Type(G);
GrpFP
```

3.27. Obwohl es leicht ist, endlich präsentierbare Gruppen in MAGMA zu konstruieren, ist das Rechnen in Ihnen unglaublich schwierig. Die zentralen Probleme in diesem Kontext wurden von Dehn schon 1911 in den folgenden drei Fragen thematisiert:

(a) (*Wortproblem*) Ist $G = \langle S \mid R\rangle$ und w eine Wort über S, d.h., $w \in F(S)$. Stellt w die Identität von G dar, d.h. ist $w \equiv 1 \bmod \langle R\rangle$?

(b) (*Konjugationsproblem*) Ist $G = \langle S \mid R\rangle$ und seien w, v Wörter über S. Sind w und v konjugiert in G?

(c) (*Isomorphieproblem*) Ist $G = \langle S \mid R \rangle$ und $G' = \langle S' \mid R' \rangle$. Sind G und G' isomorph?

3.28. Beispiel. Betrachten wir zum Beispiel die endlich präsentierte Gruppe $D_3 := \langle x, y \mid x^2, y^2, (xy)^3 \rangle$. Was ist mit dem Element $(yx)^3$? Es taucht zwar nicht in den angegebenen Relationen auf, ist es aber trotzdem trivial? Die Antwort ist: ja! Wir können das wie folgt beweisen:

$$x(yx)^3 = x(yxyxyx) = (xyxyxy)x = x$$

und daher ist $(yx)^3 = 1$. Aber woher soll der Computer das wissen?

3.29. Es stellte sich heraus, dass keine der Dehnschen Fragen eine algorithmische Lösung besitzt. Von Novikov wurde eine endlich präsentierbare Gruppe konstruiert, in der das Wortproblem und damit auch das Konjugationsproblem unentscheidbar ist. Für das Isomorphieproblem wurde ebenfalls gezeigt, dass es unentscheidbar ist.

3.30. Wegen des Wortproblems rechnet MAGMA auch in einer endlich präsentierten Gruppe wie in der darüber liegenden freien Gruppe. Das betrifft insbesondere den Gleichheitstest von zwei Wörtern. Der Kommentar zur Signatur `eq` für Elemente aus endlich präsentierten Gruppen weist auch darauf hin:

```
u eq v : GrpFPElt, GrpFPElt -> BoolElt

Return true if the free reductions of the words u and v are identical.
```

Ein Beispiel:

```
> G<s,t> := Group< s,t | s^2, t^2, s*t*s = t*s*t >;
> s*t*s eq t*s*t;
false
> s^2 eq Identity(G);
false
```

3.31. Trotzdem gibt es algorithmische Methoden, die in einigen Fällen Antworten zu den Fragen liefern können und die es ermöglichen, eine endlich präsentierbare Gruppe in andere Typen – z.B. in Gruppen mit Termerersetzungssystem, in Permutationsgruppen, in Matrixgruppen, in PC-Gruppen – umzuwandeln, in denen die Dehnschen Fragen beantwortet werden können. Diese Gruppen betrachten wir in den folgenden Abschnitten, nachdem wir noch einige Methoden besprochen, die schon direkt für endlich präsentierbare Gruppen funktionieren.

3.32. In MAGMA kann man mittels des Konstruktors `sub` Untergruppen einer endlich präsentierbaren Gruppe durch die Angabe von Erzeugern konstruieren:

```
> G<s,t> := Group< s,t | s^2, t^2, s*t*s = t*s*t >;
> H := sub<G|s>;
> H;
Finitely presented group H on 1 generator
Generators as words in group G
    H.1 = s
> Type(H);
GrpFP
```

```
> Relations(H);
[]
```

Man erkennt in dem Beispiel ein Problem, nämlich dass MAGMA für die Untergruppe H keine Relationen angibt. Das stimmt natürlich nicht, denn H ist eine Untergruppe von G und in der gelten Relationen. Da MAGMA weiß, von welcher Gruppe H eine Untergruppe ist, bezieht MAGMA diese Informationen daher. Das kann allerdings verwirrend sein und insbesondere sehen wir so keine Präsentation für die Untergruppe H. Mit der Intrinsic `Rewrite` kann man versuchen, die in G geltenden Relationen in den Erzeugern von H umzuschreiben und so eine Präsentation von H zu erhalten:

```
> G<s,t> := Group< s,t | s^2, t^2, s*t*s = t*s*t >;
> H := sub<G|s>;
> H;
Finitely presented group H on 1 generator
Generators as words in group G
    H.1 = s
> K := Rewrite(G,H);
> K;
Finitely presented group K on 1 generator
Generators as words in group G
    K.1 = s
Relations
    K.1^2 = Id(K)
```

Da dieser Prozess des Umschreibens von H nicht-trivial ist – es wird ein Algorithmus namens *Reidemeister–Schreier Rewriting* verwendet – muss dies nicht immer zum Erfolg führen!

3.33. Ein weiterer wichtiger Algorithmus ist der *Todd–Coxeter Algorithmus*, der versucht, zu einer Untergruppe H einer endlich präsentierbaren Gruppe G ein Repräsentantensystem von G/H anzugeben. Mittels `ToddCoxeter(G,H)` startet man diesen Algorithmus und bekommt, falls erfolgreich, den Index $[G : H]$, die Nebenklassenabbildung und die Nebenklassentabelle zurück:

```
> G<s,t> := Group< s,t | s^2, t^2, s*t*s = t*s*t >;
> H := sub<G|s>;
> ToddCoxeter(G,H);
3 Mapping from: Cartesian Product<{ 1 .. 3 }, GrpFP: G> to { 1 .. 3 }
   $1 $2
1.  1  2
2.  3  1
3.  2  3

5 5
```

In obigem Beispiel ist der Index von H in G also insbesondere gleich 3.

3.34. Der Todd–Coxeter Algorithmus angewandt auf die triviale Untergruppe gibt insbesondere eine Methode, die Ordnung einer endlich präsentierbaren Gruppe zu bestimmen – falls diese endlich ist und die Methode funktioniert. Die Intrinsic `Order` versucht genau das – mit vielen weiteren Tricks – und gibt genau eins der folgenden Ergebnisse zurück:

(a) 0, falls die Methode nicht erfolgreich ist.
(b) n, falls die Methode funktioniert und die Gruppe endliche Ordnung n hat.
(c) `Infinity`, falls die Methode funktioniert und beweisen kann, dass die Gruppe keine endliche Ordnung hat.

3.35. Mittels `hom` kann man genau wie für freie Gruppe auch Homomorphismen zwischen endlich präsentierbaren Gruppen konstruieren. Zwar ist ein Morphismus $\langle S \mid \rangle \to \langle S' \mid R' \rangle$ aus einer freien Gruppe in eine endlich präsentierbare immer bereits durch die Bilder der Elemente $s \in S$ eindeutig festgelegt und wohldefiniert, jedoch muss so ein Morphismus natürlich nicht einen Morphismus $\langle S \mid R \rangle \to \langle S', R' \rangle$ induzieren – dies funktioniert nur genau dann, wenn dieser Morphismus $\langle R \rangle$ im Kern hat.

3.36. MAGMA wird wegen des oben diskutierten Wortproblems nicht überprüfen, ob ein mittels `hom` konstruierter Morphismus tatsächlich ein Morphismus ist! Zwar gibt es einige Intrinsics, die dabei helfen können, dies zu prüfen, jedoch sollte sich der Benutzer immer selber sicher sein, dass er wirklich einen Morphismus definiert.

§3C. Gruppen mit Termersetzungssystem

3.37. Definition. Ein *Termersetzungssystem* (*rewrite system*) besteht aus einem Alphabet Σ und einer Menge von Regeln, nach der Wörter über Σ umgewandelt werden können. Man symbolisiert das Anwenden einer Regeln meist durch $x \to y$, d.h. man produziert mit einer Regel aus einem Wort x ein Wort y.

Ein Wort heißt dann *irreduzibel* oder *in Normalform*, falls keine Regeln mehr auf das Wort angewandet werden können, um es umzuwandeln.

Ein Termersetzungsstem heißt *terminierend*, falls es keine unendliche Kette $x_0 \to x_1 \to x_2 \to \cdots$ gibt. In einem terminierenden Termersetzungssystem besitzt jedes Wort mindestes eine Normalform.

Ein Termersetzungssystem *konvergiert*, falls es termierend ist und jedes Wort eine eindeutige Normalform besitzt, d.h. egal auf welche Weise man ein Wort nach den Regeln umformt, man produziert nach endlich Schritten immer dieselbe Normalform.

3.38. In einem konvergierenden Termersetzungssystem ist auch das *Wortproblem* entscheidbar, d.h. man kan prüfen ob zwei Wörter über dem Alphabet äquivalent modulo der Umschreiberegeln sind.

3.39. Sei $\langle S \mid R \rangle$ eine endlich präsentierbare Gruppe. Daraus kann man sich ein Termersetzungssystem produzieren, indem man S als Alphabet wählt und eine Regel $x \to y$ einführt, sofern $x = y \in \langle S \mid R \rangle$ gilt, die Worte $x, y \in F(S)$ also modulo $\langle R \rangle$ äquivalent sind. Natürlich nehmen wir mit $x \to y$ aber auch die Regel $y \to x$ auf, sodass dieses Termersetzungssystem nicht terminierend sein wird. Noch immer ist aber nicht mehr klar, dass das Termersetzungssystem *konfluent* ist, d.h. jedes Element eine eindeutige Normalform besitzt. Der *Knuth–Bendix Algorithmus* ist ein Algorithmus, der *versucht*, für $\langle S \mid R \rangle$ ein konvergierendes Termersetzungssystem zu finden, indem er diese Regeln *vervollständigt*. Wenn dies erfolgreich ist, kann man das Wortproblem in $\langle S \mid R \rangle$ lösen.

3.40. In MAGMA kann man mittels RWSGroup(G) versuchen, aus einer endlich präsentierbaren Gruppe G ein konvergierendes Termersetzungssystem nach Knuth–Bendix zu erzeugen. Das Resultat dieses Befehls ist ein *Gruppe mit Termersetzungssystem* (Typ GrpRWS). In diesem neuen Objekt kann man genauso rechnen, wie in der Gruppe G, kann jetzt aber (falls der Algorithmus erfolgreich war) das Wortproblem lösen, indem man Normalformen für die Elemente vergleicht. Betrachten wir dazu das Beispiel von oben:

```
> G<s,t> := Group< s,t | s^2, t^2, s*t*s = t*s*t >;
> H := RWSGroup(G);
> H;
A confluent rewrite group.
Generator Ordering = [ $.1, $.1^-1, $.2, $.2^-1 ]
Ordering = ShortLex.
The reduction machine has 4 states.
    $.1^2 = Id($)
    $.2^2 = Id($)
    $.2 * $.1 * $.2 = $.1 * $.2 * $.1
    $.1^-1 = $.1
    $.2^-1 = $.2

> Type(H);
GrpRWS
> H.1^2;
Id(H)
> H.1*H.2*H.1 eq H.2*H.1*H.2;
true
```

Ebefalls das Beispiel D_3 von oben kann damit behandelt werden:

```
> G<x,y> := Group<x,y | x^2, y^2, (x*y)^3 >;
> H := RWSGroup(G);
> H;
A confluent rewrite group.
Generator Ordering = [ $.1, $.1^-1, $.2, $.2^-1 ]
Ordering = ShortLex.
The reduction machine has 4 states.
    $.1^2 = Id($)
    $.2^2 = Id($)
    $.1^-1 = $.1
    $.2^-1 = $.2
    $.2 * $.1 * $.2 = $.1 * $.2 * $.1

> (H.2*H.1)^3;
Id(H)
```

§3D. Permutationsgruppen (konkret)

3.41. Nun kommen wir zu einem bestimmten Typ endlicher Gruppen für den unglaublich viele algorithmische Methoden zur Verfügung stehen – die Permutationsgruppen.

3.42. Bevor wir beginnen, halten wir einen wichtigen Punkt fest: MAGMA verknüpft Abbildungen $f : A \to B$ und $g : B \to C$ von Mengen auf vielleicht ungewohnte Art von *links nach rechts*, d.h. `f*g` steht in MAGMA für die Abbildung $g \circ f$!

3.43. Definition. Für eine Menge X sei $\mathrm{S}(X)$ die *symmetrische Gruppe* über X, d.h. die Menge aller (mengentheoretischen) Bijektionen $X \to X$ mit der Hintereinanderausführung von *links nach rechts* als Verknüpfung. Die Definition der Hintereinanderausführung ist einfach eine Definitionsfrage und MAGMA (wie viele andere Computeralgebrasysteme) benutzt diese Variante. Im größeren Kontext hängt das damit zusammen, dass man alle Operationen als Operationen von *rechts* betrachtet.

3.44. Zum Beispiel ist $\mathrm{S}_n := \mathrm{S}(\{1, \ldots, n\})$, obwohl man für die Defition von S_n natürlich genausogut jede endliche Menge mit n Elementen nehmen kann.

3.45. Diese Gruppenoperation der symmetrischen Gruppen kennen wir (und auch der Computer!) sehr gut und sie sind leicht im Computer zu implementieren. Die symmetrische Gruppe S_n erzeugt man in MAGMA mittels `SymmetricGroup(n)`. Diese haben den Typ `GrpPerm`.

```
> S := SymmetricGroup(3);
> S;
Symmetric group S acting on a set of cardinality 3
Order = 6 = 2 * 3
    (1, 2, 3)
    (1, 2)
> Type(S);
GrpPerm
> Generators(S);
{
    (1, 2, 3),
    (1, 2)
}
> Set(S);
{
    (1, 3, 2),
    (2, 3),
    (1, 3),
    (1, 2, 3),
    (1, 2),
    Id(S)
}
```

Wir sehen, dass MAGMA für Elemente der symmetrischen Gruppe die Zykelschreibweise verwendet. Produkte von Zykeln kann man in MAGMA einfach in der gewohnten Form

$$(x_{i_1}, \ldots, x_{i_{r_1}})(x_{i_2}, \ldots, x_{i_{r_2}}) \cdots (x_{i_m}, \ldots, x_{i_{r_m}})$$

eingeben. Da das Produkt $(1, 2)(3, 4)$ von Zykeln ohne weiten Kontext zu jeder symmetrischen Gruppe S_n mit $n \geq 4$ gehören könnte, muss man Produkte von Zykeln in MAGMA mittels Coercion in eine symmetrische Gruppe ziehen:

```
> S := SymmetricGroup(5);
> (1,2)(3,4);

>> (1,2)(3,4);
   ^
Runtime error in elt< ... >: No permutation group context in
which to create cycle

> S!(1,2)(3,4);
(1, 2)(3, 4)
> Type(S!(1,2)(3,4));
GrpPermElt
```

3.46. Im Gegensatz zu endlich präsentierten Gruppen ist das Wortproblem in symmetrischen Gruppen entscheidbar – im schlimmsten Fall stellt man die Elemente einfach als Permutationen dar und vergleicht!

```
> S := SymmetricGroup(3);
> s := S!(1,2);
> t := S!(2,3);
> s*t*s eq t*s*t;
true
```

3.47. Wie bereits oben erwähnt liest MAGMA Komposition und daher auch Produkte von Permutationen immer von *links nach rechts*!

```
> S := SymmetricGroup(3);
> S!(1,2)*S!(2,3);
(1, 3, 2)
> S!(2,3)*S!(1,2);
(1, 2, 3)
```

Schauen wir uns dieses Beispiel genauer an:

$$(1,2)(2,3) = \begin{pmatrix}1&2&3\\2&1&3\end{pmatrix}\begin{pmatrix}1&2&3\\1&3&2\end{pmatrix} = \begin{pmatrix}1&2&3\\3&1&2\end{pmatrix} = (1,3,2)$$

$$(2,3)(1,2) = \begin{pmatrix}1&2&3\\1&3&2\end{pmatrix}\begin{pmatrix}1&2&3\\2&1&3\end{pmatrix} = \begin{pmatrix}1&2&3\\2&3&1\end{pmatrix} = (1,2,3)\ .$$

3.48. Definition. Eine (konkrete) *Permutationsgruppe* ist eine Untergruppe einer symmetrischen Gruppe S_n.

3.49. In MAGMA kann man mit dem Konstruktor `sub` leicht Untergruppen einer symmetrischen Gruppe, also Permutationsgruppen, erzeugen. Wir werden gleich sehen, dass diese Art Gruppen eigentlich alle endlichen Gruppen enthalten. Das ist besonders hilfreich, da für es für Permutationsgruppen sehr viele Algorithmen gibt, die es zum Beispiel erlauben, das Zentrum (`Center`), die Kommutatoruntergruppe (`CommutatorSubgroup`), die Konjugiertenklassen (`Classes`) und vieles mehr zu berechnen – also im Prinzip alles, was bei endlichen endlich präsentierbaren Gruppen sehr schwierig ist.

```
> S := SymmetricGroup(5);
```

```
> H := sub<S | [S!(1,4,2), S!(5,1)]>;
> H;
Permutation group H acting on a set of cardinality 5
    (1, 4, 2)
    (1, 5)
> Type(H);
GrpPerm
> Generators(H);
{
    (1, 5),
    (1, 4, 2)
}
> Order(H);
24
> Center(H);
Permutation group acting on a set of cardinality 5
Order = 1
> CommutatorSubgroup(H);
Permutation group acting on a set of cardinality 5
Order = 12 = 2^2 * 3
    (1, 5, 4)
    (2, 4, 5)
> Classes(H);
Conjugacy Classes of group H
----------------------------
[1]     Order 1         Length 1
        Rep Id(H)

[2]     Order 2         Length 3
        Rep (1, 2)(4, 5)

[3]     Order 2         Length 6
        Rep (1, 5)

[4]     Order 3         Length 8
        Rep (1, 4, 2)

[5]     Order 4         Length 6
        Rep (1, 4, 2, 5)
```

§3E. Permutationsgruppen (abstrakt)

3.50. Wir werden jetzt sehen, dass man jede endliche Gruppe als Permutationsgruppe auffasen kann. Da es, wie oben erwähnt, algorithmische Lösungen zu nahezu allen gruppentheoretischen Problemen in Permutationsgruppen gibt, können wir also im Prinzip auch all diese Probleme in beliebigen endlichen Gruppen algorithmisch lösen – sofern wir eine Möglichkeit finden, eine gegebene endliche Gruppe als Permutationsgruppe aufzufassen.

3.51. Definition. Sei G eine Gruppe. Eine *G-Menge* ist eine Menge X zusammen mit einer Abbildung $G \times X \to X$, $x \mapsto xg$, für die gilt

(a) $x1 = x$ für alle $x \in X$.
(b) $x(gh) = (xg)h$ für alle $x \in X$ und $g, h \in G$.
Man sagt in diesem Fall auch, dass G auf X (von rechts) *operiert*.

3.52. Ist X eine G-Menge, so erhalten wir einen Gruppenmorphismus $\varphi_G^X : G \to \mathrm{S}(X)$, $g \mapsto (x \mapsto xg)$. Ist andererseits $\varphi : G \to \mathrm{S}(X)$ ein Gruppenmorphismus, so erhalten wir mittels $xg := \varphi(g)(x)$ eine Operation von G auf X. Also sind Operationen von G auf X nichts anderes als Gruppenmorphismen $G \to \mathrm{S}(X)$.

3.53. Bemerkung. Unsere Operation von G auf X ist von rechts notiert, und da wir die Hintereinanderausführung in $\mathrm{S}(X)$ von links nach rechts definiert haben, ist φ_G^X in der Tat ein Gruppenmorphismus – ansonsten wäre es ein Antimorphismus, d.h. $\varphi_G^X(gh) = \varphi_G^X(h)\varphi_G^X(g)$!

3.54. Definition. Eine Operation von G auf einer Menge X heißt *treu*, falls $\varphi_G^X : G \to \mathrm{S}(X)$ injektiv ist. Das ist äquivalent zu der Bedingung, dass $xg = x$ für alle $x \in X$ bereits $g = 1$ impliziert.

3.55. Definition. Eine (abstrakte) *Permutationsgruppe* ist eine endliche Gruppe G zusammen mit einer treuen Operation auf einer endlichen Menge X, oder äquivalent, zusammen mit einem injektiven Morphismus $\varphi : G \to \mathrm{S}_n$ (auch *treue Permutationsdarstellung* genannt).

3.56. Ist $\varphi : G \to \mathrm{S}_n$ eine Permutationsgruppe, so können wir G mittels φ mit einer Untergruppe von S_n identifizieren, sodass wir ohne Beschränkung der Allgemeinheit sagen können, dass Permutationsgruppen genau die Untergruppen von symmetrischen Gruppen S_n sind.

3.57. Theorem (Cayley)**.** Jede endliche Gruppe besitzt eine treue Permutationsdarstellung und kann daher als Permutationsgruppe aufgefasst werden.

Beweis. Sei G eine endliche Gruppe. Für $g \in G$ sei $\varphi_g : G \to G$ die Abbildung $h \mapsto hg$. Dann ist $G \to \mathrm{S}(G)$, $g \mapsto \varphi_g$, ein injektiver Gruppenmorphismus und daher eine treue Permutationsdarstellung von G. ∎

3.58. Hat man eine treue Permutationsdarstellung $\varphi : G \to \mathrm{S}_n$, so kann man G mit seinem Bild in S_n, also mit einer (konkreten) Permutationsgruppe, identifizieren und alle Fragen über G dahin übertragen und algorithmisch lösen.

Es bleibt noch das Problem, algorithmisch eine treue Permutationsdarstellung von G zu finden. Ist zum Beispiel $G = \langle S \mid R \rangle$ eine endliche endlich präsentierbare Gruppe in MAGMA , so kann man zum Beispiel die treue Permutationsdarstellung von G nach Cayley mit dem Todd-Coxeter Algorithmus für die Nebenklassenabbildung und Nebenklassentabelle von der trivialemn Unertgruppe in G berechnen – dies ist also im Prinzip algortihmisch lösbar und MAGMA ermöglicht genau dies auch mittels `PermutationGroup`.

Obwohl die Permutationsdarstellung von G nach Cayley vielleicht noch einfach zu konstruieren ist, hat sie den Nachteil, dass wir in der symmetrischen Gruppe $\mathrm{S}_{|G|}$ der Ordnung $|G|! \gg |G|$ rechnen müssen! Die Intrinsic `PermutationGroup` versucht noch mit einigen Tricks den Grad dieser Darstellung zu reduzieren.

```
> D := Group<x,y | x^2,y^2,(x*y)^3>;
```

```
> Order(D);
6
> S,phi := PermutationGroup(D);
> S;
Permutation group S acting on a set of cardinality 3
Order = 6 = 2 * 3
    (2, 3)
    (1, 2)
> phi;
Isomorphism of GrpFP: D into GrpPerm: S, Degree 3, Order 2 * 3 induced by
    D.1 |--> (2, 3)
    D.2 |--> (1, 2)
```

Mit dem ebenfalls zurückgegebenen Morphismus φ kann man jetzt gruppentheoretische Probleme von G in die symmetrische Gruppe S_n übertragen und Ergbenisse dort mit der Umkehrabbildung `Inverse(phi)` wieder zurückziehen.

§3F. Matrixgruppen

3.59. Neben den Permutationgruppen gibt es einen weiteren Typ endlicher Gruppen, in denen MAGMA sehr gut rechnen kann und für den es viele algorithmische Lösungen gibt – die endlichen Matrixgruppen.

3.60. Definition. Eine *Matrixgruppe* vom *Grad* $n \in \mathbb{N}_{>0}$ über einem Körper K ist eine Untergruppe $G \leq \mathrm{GL}_n(K)$.

3.61. Wie üblich kann man eine Matrixgruppe $G \leq \mathrm{GL}_n(K)$ durch Angaben von Erzeugern beschreiben. Ist K ein unendlicher Körper (z.B. $\mathbb{Q}$), so hat man hier jedoch wieder das Problem, dass man entscheiden muss, ob die erzeugte Gruppe endlich ist oder nicht. Dafür gibt es aber wieder algorithmische Methoden, die versuchen, dies zu beantworten.

3.62. Um zu erläutern, wie man Matrixgruppen in MAGMA definiert, müssen wir zunächst klären, wie man Matrizen in MAGMA anlegt. Dies geschiet über den `Matrix` Konstruktor. Diese Intrinsic hat viele verschiedene Signaturen (d.h. es gibt viele Möglichkeiten, die eine Matrix beschreibenden Daten einzugeben), aber die einfachste und verständlichste ist wohl folgende:

```
(<Rng> R, <RngIntElt> m, <RngIntElt> n, <SeqEnum[Tup]> Q) -> Mtrx

    The m by n matrix over R, whose entries are given by the
    tuples in Q of the form <i, j, x> (specifying that x (coerced
    into R) is at entry (i, j)).
```

Die Matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \in \mathrm{Mat}_{n\times n}(\mathbb{Q})$$

kann man zum Beispiel wie folgt konstruieren:

```
> M := Matrix(Rationals(),3,3,[1,2,3,4,5,6,7,8,9]);
> M;
[1 2 3]
```

```
[4 5 6]
[7 8 9]
> Type(M);
AlgMatElt
```

3.63. Aufgabe. Bestimmen Sie Determinante und Rang der Matrix M aus 3.62.

3.64. Eine Matrixgruppe konstruiert man nun in MAGMA mittels des MatrixGroup Konstruktors. Hierfür muss man eine Sequenz von invertierbaren Matrizen gleicher Größe über dem gleichen Körper (oder Ring) übergeben:

```
MatrixGroup< n, R | L > : RngIntElt, Rng, List -> GrpMat
```

konstruiert die Matrixgruppe in $\mathrm{GL}_n(R)$ mit Erzeugern L. Ein Beispiel:

```
> M := Matrix(Rationals(),2,2,[ 0, 1, 1, 0 ]);
> N := Matrix(Rationals(),2,2,[ 0, -1, -1, 0 ]);
> G := MatrixGroup<2, Rationals() | [M,N]>;
> G;
MatrixGroup(2, Rational Field)
Generators:
    [0  1]
    [1  0]

    [ 0 -1]
    [-1  0]
> Type(G);
GrpMat
```

3.65. Aufgabe. Konstruieren Sie in MAGMA die von den Matrizen

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

erzeugte Matrixgruppe über $\mathbb{Q}$ und bestimmen Sie ihre Ordnung.

§3G. PC-Gruppen

3.66. Eine weitere Klasse endlicher Gruppen, für die viele algorithmische Lösungen verfügbar sind, sind die sogenannten *PC-Gruppen* (*power-conjugation*) vom Typ GrpPC. Es handelt sich dabei um endliche auflösbare Gruppen. Für diese existiert eine bestimmte Art von Präsentation, die es erlaubt Elemente eindeutig in Normalformen zu schreiben. Wir wollen jetzt jedoch hier nicht weiter darauf eingehen, erwähnen dies nur, weil viele Gruppen in den Datenbanken (siehe nächsten Abschnitt) als GrpPC abgespeichert sind.

§3H. Datenbank endlicher Gruppen

3.67. Zu jeder natürlichen Zahl n kann man fragen, wie viele Gruppen von der Ordnung n bis auf Isomorphie gibt und wie ein Repräsentantensystem dieser Gruppen aussieht. Zum Beispiel gibt es, wie man sich leicht überlegen kann, nur eine Gruppe der Ordnung 2, nämlich $\mathbb{Z}/2\mathbb{Z}$. Ebenfalls leicht sieht man, dass es genau zwei Gruppen der Ordnung 4 gibt, nämlich $\mathbb{Z}/4\mathbb{Z}$ und $\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$. Wie sieht es allerdings mit Gruppen der

Ordnung 12 aus? Natürlich gibt es auch darauf eine Antwort. Mittels Computern wurden solche Klassifikationen für kleine Ordnungen durchgeführt. Solche Datenbanken sind besonders hilfreich, wenn man nach Gegenbeispielen für Vermutungen sucht. MAGMA besitzt eine Datenbank kleiner endlicher Gruppen, die es ermöglicht, für eine gegebene Ordnung alle Gruppen dieser Ordnung zu durchlaufen. In dieser Datenbank ist folgendes enthalten:

(a) All groups of order up to 2000, excluding the groups of order 1024.
(b) The groups whose order is a product of at most 3 primes.
(c) The groups of order dividing p^6 for p a prime.
(d) The groups of order $q^n p$, where q^n is a prime-power dividing 28, 36, 55 or 74 and p is a prime different to q.

3.68. Auf die Datenbank kleiner Gruppen kann auf verschiedene Weisen zugegriffen werden. Mit `NumberOfSmallGroups(n)` bekommt man beispielsweise die Anzahl der Gruppen von Ordnung n und mit `SmallGroup(n, k)` bekommt man die k-te Gruppe der Ordnung n in der Datenbank.

```
> NumberOfSmallGroups(4);
2
> SmallGroup(4,1);
GrpPC of order 4 = 2^2
PC-Relations:
    $.1^2 = $.2
> SmallGroup(4,2);
GrpPC of order 4 = 2^2
PC-Relations:
    $.1^2 = Id($),
    $.2^2 = Id($)
```

Endliche auflösbare Gruppen werden in der Datenbank als PC-Gruppen aufgeführt. Nicht auflösbare Gruppen werden als Permutationsgruppen zurückgegeben.

3.69. Aufgabe. Zwar sollte man das Ergebnis dazu im Kopf haben, bestimmen Sie jedoch trotzdem mit MAGMA die kleinste nicht-auflösbare Gruppe.

§4. Ringe

Die Mathematik besteht natürlich aus noch viel mehr (algebraischen) Strukturen als aus Gruppen und glücklicherweise beherrscht MAGMA auch noch einige davon. Die nächste typische algebraische Struktur ist die eines Rings. Natürlich – wie bei Gruppen – müssen sich Ringe auch wieder über endliche Regeln beschreiben lassen, damit man sie im Computer modellieren kann. Zwei grundlegende Ringe haben wir bereits schon gesehen: Die ganzen Zahlen $\mathbb{Z}$ (`Integers`) und die rationalen Zahlen $\mathbb{Q}$ (`Rationals`). Weitere von MAGMA unterstütze Ringtypen sind:

(a) Quotienten von in MAGMA konstruierbaren Ringen, wie zum Beispiel $\mathbb{Z}/n\mathbb{Z}$.
(b) Polynomringe in mehreren Variablen über in MAGMA konstruierbaren Ringen.
(c) Quotientenkörper von in MAGMA konstruierbaren Integritätsbereichen, wie zum Beispiel rationale Funktionenkörper.
(d) Algebraische Zahlkörper, d.h. endliche Erweiterungen von $\mathbb{Q}$ (wie $\mathbb{Q}(\sqrt{2})$) und deren Zahlringe (wie $\mathbb{Z}[\sqrt{2}]$).

(e) Den Ring $\mathbb{Z}_p$ der p-adischen Zahlen.
(f) Freie (nicht-kommutative) Ringe und Tensor-Algebren.
(g) Endlich präsentierbare Algebren.
(h) Matrix-Algebren.
(i) Gruppen-Algebren.

§4A. Moduln und Algebren

Vielleicht ist nicht jeder mit dem Begriff eines Moduls und einer Algebra vertraut, deshalb wiederholen wir das zunächst kurz.

4.1. Definition. Sei A ein Ring. Ein (rechter) A-*Modul* ist eine abelsche Gruppe V zusammen mit einer Abbildung $V \times A \to V$, $(v, a) \mapsto va$, sodass gilt:

(a) $v1 = v$ für alle $v \in V$.
(b) $(v + w)a = va + vw$ für alle $v, w \in V$ und $a \in A$.
(c) $v(a + b) = va + vb$ für alle $v \in V$ und $a, b \in A$.
(d) $v(ab) = (va)b$ für alle $v \in V$ und $a, b \in A$.

4.2. Definition. Sei A ein Ring. Ein *Morphismus* von A-Moduln V, W ist eine Abbildung $f : V \to W$ abelscher Gruppen, sodass $f(va) = f(v)a$, d.h. f ist A-*linear*.

4.3. Bemerkung. Ist K ein Körper, so ist ein K-Modul nichts anderes als ein K-Vektorraum und Morphismen zwischen K-Moduln sind nichts anderes als lineare Abbildungen! Der Begriff eines Moduls ist also die natürliche Verallgemeinerung eines Vektorraums, wobei man als Skalare nicht mehr nur Elemente eines Körpers, sondern eines beliebigen (festen) Rings zulässt.

4.4. Definition. Sei R ein kommutativer Ring. Eine R-*Algebra* ist ein Ring A, der zugleich ein R-Modul ist, sodass gilt:

(a) $(ab)r = (ar)b = a(rb)$ für alle $r \in R$ und $a, b \in A$.

4.5. Definition. Sei R ein kommutativer Ring. Ein *Morphismus* von R-Algebren A, A' ist ein Ringmorphismus $f : A \to A'$, der zugleich ein Morphismus von R-Moduln, d.h. R-linear ist.

4.6. Beispiel. Jeder Ring A ist kanonisch eine $\mathbb{Z}$-Algebra.

4.7. Beispiel. Ist R ein kommutativer Ring, so ist der Polynomring $A := R[X_1, \ldots, X_n]$ kanonisch eine R-Algebra.

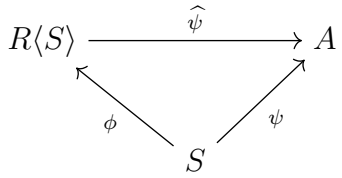
4.8. Beispiel. Jeder kommutative Ring R is kanonisch eine R-Algebra.

§4B. Freie Algebren

4.9. Ähnlich wie freie Gruppen gibt es auch freie Algebren. Dies sind genau die Ringe, in denen keine anderen Regeln als die unbedingt notwendigen gelten – sie sind daher insbesondere, wie freie Gruppen, nicht kommutativ.

4.10. Definition. Sei R ein kommutativer Ring und sei S eine Menge. Eine *freie* R-*Algebra* ist eine R-Algebra $R\langle S\rangle$ zusammen mit einem Morphismus $\phi : S \to R\langle S\rangle$, sodass es für jeden anderen Mengen-Morphismus $\psi : S \to A$ in eine R-Algebra A

genau einen R-Algebrenmorphismus $\widehat{\psi} : R\langle S\rangle \to A$, sodass das Diagramm



kommutiert.

4.11. Wie bei Gruppen ist eine freie R-Algebra, falls sie existiert, eindeutig bis auf Isomorphie von R-Algebren. Und ähnlich wie bei Gruppen ist es nicht so schwer die Existenz einer freien R-Algbra zu zeigen: $R\langle S\rangle$ besteht einfach aus allen R-Linearkombinationen von Ausdrücken der Form $\prod_{i=1}^{n} s_i^{m_i}$ mit $s_i \in S$ und $m_i \in \mathbb{N}$. Das sollte Sie sofort an den Polynomring $R[S]$ erinnern, außer, dass wir *keine* Kommutativität haben, also $st \neq ts$ in $R\langle S\rangle$ für $s, t \in S$ und $s \neq t$ gilt. In der Tat kann man aber die Definition einer freien R-Algebra zur Definition einer *freien kommutativen R-Algebra* abwandeln indem man überall die Bedingung *kommutativ* hinzufügt, und in diesem Fall ist der Polynomring $R[S]$ eben genau die freie *kommutative* R-Algebra. Mit Polynomringen werden wir uns später noch ausführlicher beschäftigen.

4.12. Aufgabe. Denken Sie über den Begriff eines *freien R-Moduls* und über dessen Zusammenhang mit freien R-Algebren nach.

4.13. Wie mit freien Gruppen kann man auch mit freien R-Algebren (über einer endlichen Menge natürlich nur) in MAGMA rechnen. Der Konstruktor dafür ist `FreeAlgebra(R,n)`, wobei n die Anzahl der Erzeuger (also der Elemente der Grundmenge) ist:

```
> A<x,y,z> := FreeAlgebra(Rationals(),3);
> A;
Free associative algebra of rank 3 over Rational Field
Order: Non-commutative Graded Lexicographical
Variables: x, y, z
> Type(A);
AlgFr
> 1/2*x*y + y^2 + z*x;
1/2*x*y + y^2 + z*x
> x*y eq y*x;
false
> Type(x);
AlgFrElt
```

4.14. Wie bei freien Gruppen ist ein Morphismus freier Algebren eindeutig durch das Bild der Erzeuger bestimmt und auf diese Weise kann man auch in MAGMA Morphismen freier Algebren definieren:

```
> A<x,y,z> := FreeAlgebra(Rationals(),3);
> f := hom<A->A | [z,x,y] >;
> f;
Mapping from: AlgFr: A to AlgFr: A
> f(x);
z
```

```
> f(y);
x
> f(z);
y
```

§4C. Endlich präsentierbare Algebren

4.15. Wie freie Gruppen sind auch freie Algebren etwas langweilig und es wird viel interessaner, wenn man Quotienten betrachtet. Die korrekten Analoga von Normalteilern im Konext von Algebren sind die Ideale.

4.16. Definition. Ein (zwei-seitiges) *Ideal* in einem Ring A ist eine additive Untergruppe I von A, die darüber hinaus abgeschlossen ist unter Multiplikation von links und rechts mit Elementen aus A, d.h. $xa \in I$ und $ax \in I$ für alle $x \in I$ und $a \in A$. In diesem Fall kann man einen wohldefinierten Quotientenring A/I (dessen zugrundeliegende additive Gruppe genau der Quotient A/I additiver Gruppen ist), bilden. Für das durch eine Teilmenge $I \subseteq A$ erzeugte Ideal (d.h. das kleinste I enthaltende Ideal) schreibt man auch $\langle I \rangle$.

4.17. Bemerkung. Ist R ein kommutativer Ring, A eine R-Algebra und I ein Ideal in A, so ist I auch insbesondere abgeschlossen unter Multiplikation mit Elementen von R und ist daher ein R-Untermodul des R-Moduls A. Der Quotient A/I ist daher wieder eine R-Algebra.

4.18. Beispiel. Das durch ein Element $m \in \mathbb{Z}$ erzeugte Ideal ist genau $\langle m \rangle = \mathbb{Z}m\mathbb{Z} = m\mathbb{Z} = \{mk \mid k \in \mathbb{Z}\}$. Der Quotient von $\mathbb{Z}$ nach diesen Ideal ist der altbekannte Ring $\mathbb{Z}/m\mathbb{Z}$.

4.19. Definition. Sei R ein kommutativer Ring. Man nennt einen Isomorphismus $A \cong R\langle S \rangle / I$ von R-Algebren, wobei S eine Menge und I ein Ideal in der freien R-Algebra $R\langle S \rangle$ ist, eine *Präsentation* von A.

4.20. Ähnlich wie bei Gruppen besitzt jede R-Algebra eine Präsentation, sodass man auf diese Weise im Prinzip jede Algebra im Computer definieren kann, sofern sie endlich präsentierbar ist.

4.21. Definition. Man nennt A *endlich präsentierbar*, falls A eine Präsentation $A \cong R\langle S \rangle / I$ besitzt, sodass S endlich und I endlich erzeugt ist.

4.22. Sofern der kommutative Grundring R in MAGMA konstruierbar ist, können wir leicht endlich präsentierbare Algebren in MAGMA konstruieren. Ganz allgemein erzeugt man ein durch I erzeugte Ideal eines Ring A mittels `ideal<A|I>` und den Quotienten $A/\langle I \rangle$ mittels `quo<A|I>`:

```
> A<x,y> := FreeAlgebra(Rationals(),2);
> I := ideal<A|x^2,y^2,x*y-y*x>;
> I;
Two-sided ideal of Free associative algebra of rank 2 over Rational
Field
Order: Non-commutative Graded Lexicographical
Variables: x, y
Homogeneous
```

```
Basis:
[
    x^2,
    y^2,
    x*y - y*x
]
> Type(I);
AlgFr
> Q,q := quo<A|I>;
> Q;
Finitely Presented Algebra of rank 2 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y
Quotient relations:
[
    x^2,
    y^2,
    x*y - y*x
]
> q;
Mapping from: AlgFr: A to AlgFP: Q
> Type(Q);
AlgFP
> Q.1*Q.2 eq Q.2*Q.1;
true
> Q.1^2;
0
> q(x);
x
> q(x^2);
0
```

4.23. Wie bei endlich präsentierbaren Gruppen haben wir bei endlich präsentierbaren Algebren auch mit einem Wortproblem zu kämpfen, denn wiederum ist nicht klar, wie man in dem Quotienten A/I rechnet. In dem Beispiel oben sehen wir aber, dass MAGMA hier – im Gegensatz zu endlich präsentierbaren Gruppen (siehe 3.30) – zumindest die offensichtlichen Relationen berücksichtigt. Das liegt daran, dass MAGMA für den Benutzer nicht sichtbar eine sogenannte *Gröbner-Basis* des Ideals I berechnet (das ist ein spezielles Erzeugendensystem von I) und dass man in diesem Fall in dem Quotienten A/I in der Tat rechnen kann. Aber so eine Gröbner-Basis muss nicht endlich sein und daher kann es passieren, dass man wirklich nicht ohne weiteres in dem Quotienten A/I rechnen kann – ähnlich wie bei endlich präsentierbaren Gruppen. In diesem Fall wird MAGMA zum Beispiel bei einer Gleicheitsabfrage von Elementen unendliche lange rechnen.

4.24. Es gibt aber auch Ringe in deren Quotienten man sehr gut rechnen kann. Dazu gehören zum Beispiel Quotienten von Polynomringen über Körpern (denn hier existiert in der Tat immer eine Gröbner-Basis und Algorithmen zu ihrer Berechnung).

§4D. Polynomringe

4.25. Den Polynomring $R[X_1, \ldots, X_n]$ über einem kommutativen Ring R in n-Variablen erzeugt man in MAGMA mittels PolynomialRing(R,n).

```
> P<X,Y,Z> := PolynomialRing(Rationals(),3);
> P;
Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: X, Y, Z
> Type(P);
RngMPol
> X*Y eq Y*X;
true
> Type(X);
RngMPolElt
> f := 1/2*X*Y*Z - Z^5*X^3 + 1;
> f^2;
X^6*Z^10 - X^4*Y*Z^6 - 2*X^3*Z^5 + 1/4*X^2*Y^2*Z^2 + X*Y*Z + 1
```

4.26. Ein Polynomring in einer Variablen (ein sogenannter *univariater* Polynomring) kann zwar über PolynomialRing(R,1) konstruiert werden, jedoch sollte man dies über den Konstruktor PolynomialRing(R) tun, denn univariate Polynomringe haben einen eigenen Typ RngUPol in MAGMA und es gibt spezielle Intrinsics nur für univariate Polynomringe.

```
> P := PolynomialRing(Rationals(),1);
> Type(P);
RngMPol
> P := PolynomialRing(Rationals());
> Type(P);
RngUPol
```

4.27. Sei $A := R[\boldsymbol{X}]$ ein Polynomring in den Variablen $\boldsymbol{X} := (X_i)_{i=1}^n$. Jedes Element $f \in A$ lässt sich dann schreiben als $f = \sum_{\alpha \in \mathbb{N}^n} c_\alpha \boldsymbol{X}^\alpha$, wobei $\boldsymbol{X}^\alpha := \prod_{i=1}^n X_i^{\alpha_i}$ mit $c_\alpha \in R$ und fast alle $c_\alpha = 0$. Man nennt die c_α mit $c_\alpha \neq 0$ die *Koeffizienten* von f, man nennt die $\boldsymbol{X}^\alpha$ mit $c_\alpha \neq 0$ die *Monome* von f, und man nennt die $c_\alpha \boldsymbol{X}^\alpha$ mit $c_\alpha \neq 0$ die *Terme* von f. Der *Grad* von f ist $\max\{|\alpha| \mid \alpha \in \mathbb{N}^n, c_\alpha \neq 0\}$, wobei $|\alpha| := \sum_{i=1}^n \alpha_i$. Auf alle diese Objekte hat man leichten Zugriff in MAGMA :

```
> P<X,Y,Z> := PolynomialRing(Rationals(),3);
> f := 1/2*X*Y*Z - Z^5*X^3 + 1;
> f;
-X^3*Z^5 + 1/2*X*Y*Z + 1
> Degree(f);
8
> Coefficients(f);
[ -1, 1/2, 1 ]
> Monomials(f);
[
    X^3*Z^5,
    X*Y*Z,
    1
```

```
]
> Terms(f);
[
    -X^3*Z^5,
    1/2*X*Y*Z,
    1
]
>
```

Die Sequenzen von Coefficients, Monomials und Terms sind dabei alle kompatibel, sodass sich ein Element daraus sofort wieder rekonstruieren lässt.

4.28. Folgendes Beispiel zeigt, wie flexibel MAGMA bei der Konstruktion von Polynomringen ist:

```
> P<X> := PolynomialRing(Rationals());
> Q<Y> := PolynomialRing(P);
> Q;
Univariate Polynomial Ring in Y over Univariate Polynomial Ring in X
over Rational Field
> X*Y;
X*Y
```

4.29. Mehrere Instanzen multivariater Polynomringe (gleichen Rangs über dem gleichen Ring) werden in MAGMA immer als verschieden betrachtet:

```
> P<X,Y,Z> := PolynomialRing(Rationals(),3);
> Q<U,V,W> := PolynomialRing(Rationals(),3);
> U in P;
false
```

Univariate Polynomringe (gleichen Rangs über dem gleichen Ring) hingegen werden immer als gleich angesehen:

```
> P<X> := PolynomialRing(Rationals());
> Q<U> := PolynomialRing(Rationals());
> U in P;
true
> P eq Q;
true
> X;
U
```

Will man verschiedene Variablen für univariate Polynomringe verwenden, so kann man dies über die Optional Global:=false im Konstruktor erreichen.

```
> P<X> := PolynomialRing(Rationals() : Global:=false);
> Q<U> := PolynomialRing(Rationals() : Global:=false);
> U in P;
true
> X;
X
> U;
U
```

4.30. Vielleicht ist Ihnen bereits aufgefallen, dass MAGMA für einen multivariaten Polynomring immer etwas wie Order: Lexicographical angibt. Es handelt sich dabei um die Information, welche Monomordnung auf dem Polynomring gewählt wurde und standardmäßig handelt es sich dabei um die lexikographische. Eine *Monomordnung* auf einem Polynomring $A := R[\boldsymbol{X}]$ in den Variablen $\boldsymbol{X} := (X_i)_{i=1}^n$ ist dabei eine Wohlordnung $\prec$ auf der Menge aller Monome in A, sodass gilt: Ist $f \prec g$ für Monome f und g, so gilt auch $fh \prec gh$ für alle Monome h. Neben der lexikographischen Ordnung gibt es auch noch andere Monomordnungen. Diese Ordnungen werden zur Berechnung der *Gröbner-Basen* von Idealen im Polynomring benötigt. Es handelt sich dabei um spezielle Erzeugendensysteme von Idealen, die, falls R ein Körper ist, nach Wahl einer Monomordnung immer existieren, berechnet werden können, und es erlauben in Quotienten zu rechnen. Betrachten wir ein Beispiel:

```
> P<X,Y,Z> := PolynomialRing(Rationals(),3);
> I := ideal<P|X*Y*Z - Z^3, X^2 - Z^2>;
> I;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: X, Y, Z
Homogeneous
Basis:
[
    X*Y*Z - Z^3,
    X^2 - Z^2
]
> GroebnerBasis(I);
[
    X^2 - Z^2,
    X*Y*Z - Z^3,
    X*Z^3 - Y*Z^3,
    Y^2*Z^3 - Z^5
]
> I;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: X, Y, Z
Homogeneous, Dimension >0
Groebner basis:
[
    X^2 - Z^2,
    X*Y*Z - Z^3,
    X*Z^3 - Y*Z^3,
    Y^2*Z^3 - Z^5
]
> Q,q := quo<P|I>;
> Q;
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: X, Y, Z
Quotient relations:
```

```
[
    X^2 - Z^2,
    X*Y*Z - Z^3,
    X*Z^3 - Y*Z^3,
    Y^2*Z^3 - Z^5
]
> Q.1*Q.2*Q.3 eq Q.3^3;
true
```

§4E. Algebraische Zahlkörper

4.31. Vielleicht ist Ihnen schon aufgefallen, dass man algebraisch mit Ausdrücken wie $\sqrt{2}$ in MAGMA nicht so direkt wie zum Beispiel in MAPLE rechnen kann:

```
> Sqrt(2);
1.41421356237309504880168872421
```

Der Befehl `Sqrt` konstruiert nicht das algebraische Objekt $\sqrt{2}$, sondern gibt nur eine rationale Zahl an, die $\sqrt{2}$ approximiert. Um zu verstehen, wie man abstrakt mit $\sqrt{2}$ in MAGMA rechnet, muss man erst einmal verstehen, was $\sqrt{2}$ eigentlich ist: $\sqrt{2}$ ist eine (der beiden!) Nullstellen des Polynoms $X^2 - 2 \in \mathbb{Q}[X]$! Man kann sich also $\sqrt{2}$ als irgendein über $\mathbb{Q}$ lebendes Objekt vorstellen, das aber mittels $\mathbb{Q}$ und einem Polynom beschrieben werden kann. Aber wo lebt jetzt $\sqrt{2}$? Man hätte gerne einen möglichst kleinen $\mathbb{Q}$ enthaltenden Körper K, in dem ein Element existiert, das eine Nullstelle von $X^2 - 2$ ist. Wie konstruiert man so einen Körper? Ganz einfach: Es ist $K = \mathbb{Q}[X]/\langle X^2 - 2\rangle$. Sicherlich ist $\mathbb{Q}$ in dem Ring K enthalten und man kann auch zeigen, dass K ein Körper ist (das liegt daran, dass $X^2 - 2$ irreduzibel in $\mathbb{Q}[X]$ ist, sodass $\langle X^2 - 2\rangle$ ein *maximales Ideal* in $\mathbb{Q}[X]$ ist). Und was ist jetzt $\sqrt{2}$? Es ist die Restklasse $\overline{X}$ von X in diesem Quotienten! Es gilt nämlich nach Definition $X^2 \equiv 2 \bmod \langle X^2 - 2\rangle$, d.h. $\overline{X}^2 = 2$.

Auf genau diese Weise kann man zu jedem irreduziblen Polynom $p \in \mathbb{Q}[X]$ einen $\mathbb{Q}$ enthaltenden Körper K konstruieren, in dem p eine Nullstelle hat. Dies kann man auch in MAGMA mit dem Befehl `NumberField(p)` tun.

```
> P<X> := PolynomialRing(Rationals());
> K<a> := NumberField(X^2-2);
> K;
Number Field with defining polynomial X^2 - 2 over the Rational Field
> Type(K);
FldNum
> a^2;
2
```

In diesem Beispiel spielt a die Rolle von $\sqrt{2}$. Normalerweise betachtet man Ausdrücke wie $\sqrt{2}$ in $\mathbb{R}$ und bezeichnet dann mit $\sqrt{2}$ die *positive* Nullstelle des Polynoms $X^2 - 2$. Vom algebraischen Standpunkt aus macht es allerdings keinen Unterschied, welche der beiden Nullstellen von $X^2 - 2$ wir mit $\sqrt{2}$ bezeichnen – in jedem Fall ist $-\sqrt{2}$ die andere Nullstelle. Dies ist ohnehin nicht mehr möglich, wenn diese Nullstellen nicht mehr in $\mathbb{R}$ sondern echt in $\mathbb{C}$ liegen, weil wir hier keine Ordnungsrelation und daher keine Begriffe wie *positiv* mehr haben. So ist zum Beispiel die imaginäre Einheit $\mathrm{i} \in \mathbb{C}$ eine Nullstelle des Polynoms $X^2 + 1$, mehr gibt es dazu nicht zu sagen.

§5. Moduln und lineare Algebra

§5A. Matrizen, Vektoren und lineare Algebra

5.1. Wir haben bereits bei den Matrixgruppen besprochen, wie man Matrizen über beliebigen Ringen in MAGMA definiert (siehe 3.62). Damit wir ein wenig lineare Algebra betreiben können, müssen wir noch wissen, wie man Vektoren konstruiert. Vektoren sind nichts anderes als Elemente eines Vektorraums, also benötigen wir zunächst Vektorräume. Den Standardvektorraum K^n über einem Körper K konstruiert man mittels VectorSpace(K,n).

```
> V := VectorSpace(Rationals(),3);
> V;
Full Vector space of degree 3 over Rational Field
> Type(V);
ModTupFld
> Basis(V);
[
    (1 0 0),
    (0 1 0),
    (0 0 1)
]
> V.1;
(1 0 0)
> V.2;
(0 1 0)
> V.3;
(0 0 1)
> Type(V.1);
ModTupFldElt
> V.1+1/2*V.2;
(  1 1/2   0)
> V![1,-17,12];
(  1 -17  12)
```

Bei der Ausgabe von Basis(V) sehen wir, dass MAGMA Vektoren als Zeilenvektoren schreibt. Zugriff auf die Basiselemente haben wir mittels V.i für $1 \leq i \leq n$. Wie in dem Beispiel oben zu sehen, kann man Vektoren mittels Coercion aus Sequenzen erzeugen.

5.2. Mit Vektorräumen kan man alle grundlegenden Operationen durchführen, wie Unterräume, Quotienten, Schnitte und Summen berechnen.

```
> V := VectorSpace(Rationals(),3);
> U := sub<V|1/2*V.1+V.2>;
> U;
Vector space of degree 3, dimension 1 over Rational Field
Generators:
(1/2   1   0)
Echelonized basis:
(1 2 0)
```

```
> W := sub<V|V.3>;
> W;
Vector space of degree 3, dimension 1 over Rational Field
Generators:
(0 0 1)
Echelonized basis:
(0 0 1)
> U + W;
Vector space of degree 3, dimension 2 over Rational Field
Echelonized basis:
(1 2 0)
(0 0 1)
> U meet W;
Vector space of degree 3, dimension 0 over Rational Field
> Q,q := quo<V|U>;
> Q;
Full Vector space of degree 2 over Rational Field
> q(V.1);
(-2  0)
> Basis(Q);
[
    (1 0),
    (0 1)
]
```

5.3. Da MAGMA Vektoren immer als Zeilenvektoren auffast, ist die Matrix-Vektor-Multiplikation immer von der Form $v \cdot A$ für einen Zeilenvektor v und eine Matrix A (gleicher Dimension über dem gleichen Ring). Genauer, operiert bei MAGMA der Matrixring $\mathrm{Mat}_{n\times n}(K)$ von *rechts* auf dem Vektorraum K^n. Der Grund hierfür ist einfach, dass sich auf dem Bildschirm Zeilenvektoren einfacher ausgeben lassen, als Spaltenvektoren. Das hat jedoch einige vielleicht ungewohnte Konsequenzen, wenn man nur die Operation von links, also von der Form $A \cdot v$ für Spaltenvektoren v, gewöhnt ist. Denn berechnet man zum Beispiel den Kern einer Matrix A in MAGMA, so handelt es sich dabei um alle Zeilenvektoren v, sodass $v \cdot A = 0$ gilt. Das kann sich von der Menge aller Spaltenvektoren v, sodass $A \cdot v = 0$ gilt, unterscheiden! Natürlich ist aber

$$\{\text{Zeilenvektoren } v \mid v \cdot A = 0\} = \{\text{Spaltenvektoren } v \mid A^{\mathrm{T}} \cdot v = 0\}$$

und daher kann man mittels Transponieren auch immer die entsprechenden Ergebnisse von links erhalten – daran muss man jedoch immer denken!

5.4. Folgendes Beispiel zeigt den Unterschied zwischen $\mathrm{Ker}(A)$ und $\mathrm{Ker}(A^{\mathrm{T}})$.

```
> A := Matrix(Rationals(),3,3,[3,6,9,1,2,3,6,12,18]);
> A;
[ 3  6  9]
[ 1  2  3]
[ 6 12 18]
> Kernel(Transpose(A));
```

```
Vector space of degree 3, dimension 2 over Rational Field
Echelonized basis:
(   1    0 -1/3)
(   0    1 -2/3)
> Kernel(A);
Vector space of degree 3, dimension 2 over Rational Field
Echelonized basis:
(   1    0 -1/2)
(   0    1 -1/6)
```

5.5. MAGMA bezieht sich bei Berechnungen mit Matrizen immer auf den Grundring der Matrizen. Berechnet man zum Beispiel Eigenwerte, so werden nur Eigenwerte über dem Grundring angegeben!

```
> A := Matrix(Rationals(),3,3,[1,2,3,4,5,6,7,8,9]);
> A;
[1 2 3]
[4 5 6]
[7 8 9]
> Eigenvalues(A);
{ <0, 1> }
> Eigenspace(A,0);
Vector space of degree 3, dimension 1 over Rational Field
Echelonized basis:
( 1 -2  1)
> p := CharacteristicPolynomial(A);
> p;
$.1^3 - 15*$.1^2 - 18*$.1
> Factorization(p);
[
    <$.1, 1>,
    <$.1^2 - 15*$.1 - 18, 1>
]
```

Die Matrix $A \in \mathrm{Mat}_{3\times 3}(\mathbb{Q})$ in dem Beispiel hat nur einen Eigenwert über $\mathbb{Q}$. Um die weiteren Eigenwerte zu bekommen, müssen wir die Matrix A auf den Zerfällungskörper des charakteristischen Polynoms hochheben:

```
> P<X> := PolynomialRing(Rationals());
> K<a> := NumberField(X^2 - 15*X - 18);
> B := ChangeRing(A,K);
> B;
[1 2 3]
[4 5 6]
[7 8 9]
> Parent(B);
Full Matrix Algebra of degree 3 over K
> Eigenvalues(B);
{
    <0, 1>,
    <a, 1>,
    <-a + 15, 1>
```

```
}
> Eigenspace(B,a);
Vector space of degree 3, dimension 1 over K
Echelonized basis:
(           1 1/18*(a + 6)  1/9*(a - 3))
> Eigenspace(B,-a+15);
Vector space of degree 3, dimension 1 over K
Echelonized basis:
(             1 1/18*(-a + 21)  1/9*(-a + 12))
> JordanForm(B);
[      0       0       0]
[      0       a       0]
[      0       0 -a + 15]
```

5.6. Lineare Gleichungssysteme der Form $v \cdot A = b$ kann man in MAGMA mittels IsConsistent(A,b) auf Konsistenz prüfen und deren Lösung bestimmen. Dabei wird ein Boolean zurückgegeben, der Auskunft über die Konsistenz gibt, und falls das System konsistenz ist, wird eine spezielle Lösung und der Kern von A zurückgegeben.

```
> A := Matrix(Rationals(),3,3,[1,2,3,4,5,6,7,8,9]);
> V := VectorSpace(Rationals(),3);
> b := V![1/2,0,1];
> IsConsistent(A,b);
false
> b := V![1,1,1];
> IsConsistent(A,b);
true (   0 -1/3  1/3)

Vector space of degree 3, dimension 1 over Rational Field
Echelonized basis:
( 1 -2  1)
```

Literatur

[1] Wieb Bosma, John Cannon und Catherine Playoust. *The Magma algebra system. I. The user language*. In: *Journal of Symbolic Computation* 24.3-4 (1997), S. 235–265. DOI: 10.1006/jsco.1996.0125. URL: http://dx.doi.org/10.1006/jsco.1996.0125.

[2] Greg Butler und John Cannon. *Cayley, version 4: The user language*. In: Bd. 358. Lecture Notes in Computer Science. Springer, 1989. URL: http://link.springer.com/chapter/10.1007%2F3-540-51084-2_43.

[Magma] *Magma Computational Algebra System by W. Bosma, J. Cannon and C. Playoust*. URL: http://magma.maths.usyd.edu.au/magma/.

Index

A-Modul, 37
A-linear, 37
G-Menge, 32
R-Algebra, 37
überladen, 6

Abbildung, 15

Boolean, 8

Coercion, 4

Einbettung, 3
element to sequence, 23
endlich präsentierbar, 24, 39
Erzeuger, 24

Fibonacci-Folge, 16
Folgenglied, 13
for-Schleife, 19
frei, 23
freie R-Algebra, 37
freie Gruppe, 22
freie kommutative R-Algebra, 38
freier Rang, 23
freies Erzeugendensystem, 23
Funktion, 15

Gröbner-Basis, 40, 43
Grad, 34, 41
Graph, 15
Gruppe mit Termersetzungssystem, 29

Ideal, 39
Intrinsic, 5, 6
irreduzibel, 28
Isomorphieproblem, 26
Iterator, 19
iterieten, 19

Knuth–Bendix Algorithmus, 28
Koeffizienten, 41
Konjugationsproblem, 25
Konkatenation, 8, 20
Kopie, 17

Matrixgruppe, 34
maximales Ideal, 44
Monome, 41
Monomordnung, 43
Morphismus, 37

nano, 16
Normalform, 28

Operation
  treue, 33
operiert, 33

Parent, 3
PC-Gruppen, 35
Permutationsdarstellung
  treue, 33
Permutationsgruppe, 31, 33
Polynomring
  univariater, 41
power-conjugation, 35
Präsentation, 24, 39
Projektion, 3
Prozedur, 17

Referenz, 17
Reidemeister–Schreier Rewriting, 27
Rekursion, 16
Relationen, 24
rewrite system, 28

Sequenz, 13
Signatur, 6
Strings, 20
Struktur, 3
symmetrische Gruppe, 30

Terme, 41
Termersetzungssystem, 28
  konfluent, 28
  konvergent, 28
  terminierend, 28
Todd–Coxeter Algorithmus, 27
Tupel, 14
Typ, 3

universellen Eigenschaft, 22
Universum, 11

while-Schleife, 20
Wort, 24
Wortproblem, 25, 28