MAGMA FÜRS LEHRAMT

WERNER BLEY

Zusammenfassung.

Very preliminary version

1. Die ersten Schritte

1.1. **Die allerersten Schritte.** Von der Konsole wird MAGMA mit dem Kommando `magma` gestartet. Man kann MAGMA wie einen Taschenrechner benutzen. Jedes Kommando muss mit einem Semikolon abgeschlossen werden. Probieren Sie

```
2+3;
2*(-3);
(-3)^117;
3/7+2/17;
(1/13)/(-5/19);
```

MAGMA wird mit dem Befehl

```
quit;
```

verlassen.

1.2. **Strukturen von MAGAM.** Jedes Objekt hat in MAGMA einen wohldefinierten *Typ* und gehört einer darüber liegenden Struktur, dem *Parent*, an. Diese Informationen können wir folgt ermittelt werden.

```
> Type(5);
RngIntElt
> Parent(5);
Integer Ring
> Type(3/2);
FldRatElt
> Parent(3/2);
Rational Field
```

Die Ausgaben sind weitgehend selbsterklärend. Die Befehle

```
Z := Integers();
Q := Rationals();
```

erzeugen die ganzen Zahlen bzw. die rationalen Zahlen. Versuchen Sie die folgenden Befehle.

```
a := 9/3;
IsPrime(a);
IsPrime(Z!a);
```

Date: 13. Oktober 2015.

Wie der Name erraten läßt, testet die Funktion `IsPrime`, ob eine ganze Zahl eine Primzahl ist. Die Eingabe für `IsPrime` muss also vom Typ `RngIntElt` sein. Bei der Zuweisung `a := 9/3;` erzeugt aber MAGMA ein Objekt vom Typ `FldRatElt`, daher die Ausgabe *false*, obwohl ja $a = 3$ scheinbar offensichtlich eine Primzahl ist. Will man, daß MAGMA a als Element der ganzen Zahlen, so muss man dies MAGMA mittels dem Typen-Umwandlungsoperator `!` mitteilen. Versuchen Sie

```
Type(a);
Type(Z!a);
Z ! (3/4);
```

1.3. **Intrinsics.** MAGMA hat eine große Anzahl an vordefinierten Funktionen, sogenannten *intrinsics*. Wir haben schon mehrere davon kennen gelernt: `Integers()`, `Rationals()`, `IsPrime(n)`. Intrinsics haben eine bestimmte Anzahl von Eingabeobjekten von einem bestimmten Typ. Hier sind weitere Beispiele:

```
Divisors(2020);
PrimeDivisors(2020);
Factorization(2020);
GCD(12, 33);
```

Sowohl die Befehle als auch die Ausgaben sind weitgehend selbsterklärend. Wenn man sich darüber informieren will, wie die verlangte Eingabe ist und wie die Ausgabe zu interpretieren ist, so hilft neben dem MAGMA-Handbuch auch die Eingabe des Intrinsic-Namens abgeschlossen mit Semikolon weiter. Zum Beispiel

```
Divisors;
IsPrime;
```

Sie sehen, daß MAGMA den selben Namen für unterschiedliche Eingabetypen verwendet; MAGMA kann dann anhand der Anzahl und der Typen der Eingabeparameter erkennen, welche Funktion zu verwenden ist.

2. Aussagenlogik

MAGMA kennt den Typ `BoolElt`, wie wir schon bei der Funktion `IsPrime` gesehen haben. Einfache Beispiele sind

```
a := true;
b := false;
a or b;
a and b;
not a;
```

Sehr oft benötigt man für Vergleichstests `eq` für $=$, `lt` für $<$, `le` für $\leq$, `gt` für $>$ und `ge` für $\geq$.

3. Konditionalausdrücke

Konditionalausdrücke haben die Gestalt

```
if BoolAusdruck1 then
    Anweisungen 1;
elif BoolAusdsruck 2 then
    Anweisungen 2;
else
    Anweisungen 3;
```

```
end if;
```

Dabei kann es beliebig viele `elif`-Teile geben. Betrachten wir das folgende Beispiel. Wir wollen die Funktion $f\colon \mathbb{N}_{>0} \longrightarrow \mathbb{N}_{>0}$ mit

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

realisieren. Dazu betrachten wir

```
n := 35;
if IsEven(n) then n := Integers()!(n/2); else n := 3*n+1; end if; n;
```

Man kann einen solchen Ausdruck also auch in eine Zeile schreiben; beim Gebrauch von MAGMA als Taschenrechner hat dies gewisse Vorteile, ist aber eher unleserlich. Wenn wir später eigene Funktionen schreiben, werden wir daher (meist) eine Schreibeweise wie folgt bevorzugen

```
if IsEven(n) then
    n := Integers()!(n/2);
else
    n := 3*n+1;
end if;
```

Da man mit ↑ Befehle zurückholen kann, können Sie die Kurzversion des obigen Konditionalausdrucks nun wieder und wieder durchführen. Was beobachten Sie?

3.1. **Mengen.** Die Verwendung von Mengen in MAGMA ist sehr intuitiv. Wir wollen uns daher hier nur auf das notwendigste beschränken. Versuchen Sie

```
M := {1,2,3,4,5,6,7,8,9};
N := {2..5};
Type(M);
```

Die Kardinalität einer Menge ermittelt man mit `#`, `join` ergibt die Vereinigung und `meet` den Schnitt. Hier einige triviale Beispiele, allesamt selbsterklärend:

```
A := {1,2,3};
B := {3,4,5,6};
#A; #B;
A join B;
A meet B;
A diff B;
4 in A;
4 in B;
4 notin B;
{1,2} subset {1..10};
```

MAGMA erlaubt auch eine Art der Definition von Mengen, die dem Mathematiker intuitiv sofort klar ist. Wir erläutern dieses Konzept an Beispielen:

```
M := {1..20};
X := {a : a in M | IsEven(a)};
Y := {a : a in M | IsOdd(a) and a lt 11};
```

X ist also die Menge der Elemente $a \in M$ mit der Eigenschaft, daß a gerade ist. Y ist die Menge der $a \in M$, die ungerade und kleiner als 11 sind.

Wir wollen die Menge M der Primzahlen p bstimmen, die sich als Summe dreier Primzahlen a, b, c mit $3800 \leq a, b, c \leq 4100$ schreiben lassen. Sodann wollen wir

entscheiden, ob es eine Primzahl zwischen $\min(M)$ und $max(M)$ gibt, die nicht in M liegt.

```
M := {a+b+c : a,b,c in {3800..4100} | IsPrime(a) and IsPrime(b) and IsPrime(c)
                                      and IsPrime(a+b+c)};
N := {n : n in {Min(M)..Max(M)} | IsPrime(n)};
N subset M;
M subset N;
N diff M;
```

Aufgabe 3.1. Verstehen Sie diese Zeilen. Warum dauert das relativ lang?

3.2. **Sequenzen.** *Sequenzen* funktionieren ähnlich wie Mengen, allerdings spielt jetzt die Reihenfolge eine Rolle und es sind Wiederholungen möglich. Der Datentyp heißt `SeqEnum`. Hier einige Beispiele:

```
S := [1,2,2,3,3,3,4,4,4,4,5,5,5,5,5];
#S;
S[1]; S[2];
S[1] := 17;
7 in S;
Type(S);
T := [100,101,102];
S cat T;
```

Der Befehl `cat` hängt also zwei Sequenzen aneinander, natürlich unter Beachtung der Reihenfolge.

Die Befehle `SetToSequence` bzw. `SequenceToSet` konvertieren zwischen Mengen und Sequenzen.

```
SetToSequence( {1,2,3,4} );
SequenceToSet( [1,2,2,3,3,3,4,4,4,4] );
```

4. Funktionen und Intrinsics

In diesem Abschnitt wollen wir lernen, wie man die Funktionalität von MAGMA erweitern kann, indem man eigene Funktionen und Intrinsics schreibt und einbindet

Die Struktur von Funktioen ist wie folgt:

```
function_name := function(input_1, input_2,....)
    Anweisungen;
    return output_1, output_2, ....;
end function;
```

Hier ein einfaches Beispiel. Wir wollen eine Funktion f definieren, die $x \mapsto x^3$ realisiert.

```
f := function(x)
    return x^3;
end function;
f(2);
f(1.1);
```

Komplexere Funktionen will man natürlich nicht bei jedem Start von MAGMA neu schreiben. Wir erzeugen daher eine Datei MyFunctions.m (oder Sie wählen sich

einen anderen Namen) und schreiben die Funktion in diese Datei. Mit `load MMyFunctions.m` kann man dann die Funktionen in dieser Datei laden und in MAGMA benutzen.

Wir wollen die Funktion

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

als Function realisieren. Editieren Sie dazu die Datei `MyFunctions.m` und speichern Sie sie mit dem Inhalt

```
f := function(n)
    if IsEven(n) then
        n := Integers()!(n/2);
    else
        n := 3*n+1;
    end if;
    return n;
end function;
```

Nach `load MyFunctions.m` können wir nun die Funktion f benutzen. Schleifen werden wir zwar erst kennen lernen, aber intuitiv ist klar, was in den folgenden Zeilen passiert.

```
while n ne 1 do
    n := f(n);
    n;
end while;
```

Für die nächste Aufgabe möchten Sie vielleicht eine `for`-Schleife verwenden. Wir greifen deshalb etwas vor und geben hier die einfachste Form einer solchen Schleife an

```
for i:=n to m do
    Anweisungen;
end for;
```

Hier sollten n, m ganze Zahlen sein. Zum Beispiel können Sie mit der folgenden Schleife die 20 Primzahlen $> N$ in eine Sequenz `P` anfügen. wobei N eine natürliche Zahl ist.

```
P := [];
N := 2000000000000000000;
for i:=1 to 20 do
    N := NextPrime(N);
    Append(~P, N);
end for;
```

Aufgabe 4.1. Bekanntlich sind die Fibinacci-Zahlen rekursiv definiert durch

$$F_0 := 0, F_1 := 1; F_n := F_{n-1} + F_{n-2}, \text{ falls } n \geq 2.$$

Schreiben Sie eine Funktion `MyFibonacci(n)`, die die n-te Fibonacci-Zahl berechnet. Testen Sie Ihre Implementierung durch Vergleich mit der MAGMA-Funktion `Fibonacci`.

TO DO: Intrinsics

5. Schleifen

Wir wollen uns hier auf `for`- und `while`-Schleifen beschränken und diese anhand von Beispielen kennen lernen. Zunächst zur `for`-Schleife.

```
for i:=0 to 99 do
    MyFibonacci(i) eq Fibonacci(i);
end for;
```

vergleicht Ihre Implementierung mit der MAGMA-eigenen Implementierung anhand der ersten hundert Fibonacci-Zahlen. Wenn Ihre Funktion richtig arbeitet, so bekommen Sie 100 mal den wahrheitswert `true`. Äquivalent dazu sind

```
for i in [0..99] do
    MyFibonacci(i) eq Fibonacci(i);
end for;
```

oder

```
for i in {0..99} do
    MyFibonacci(i) eq Fibonacci(i);
end for;
```

Auch `for`-Schleifen der folgenden Art sind erlaubt

```
for i,j in {1,2,3} do
    print "i = ", i, " j = ", j;
end for;
```

Hier haben wir eine etwas primitive Art der Ausgabe benutzt, die weitgehend selbsterklärend ist. Wenn nicht, so konsultieren Sie bitte das Handbuch.

Die Form einer `while`-Schleife ist wie folgt

```
while BoolElt do
    Anweisungen;
end while;
```

Wir betrachten wieder einige Beispiele. Wir wollen eine Liste der Primzahlen $\leq N$ erzeugen.

```
N := 200;
p := 2;
P := [];
while p le N do
    Append(~P, p);
    p := NextPrime(p);
end while;
```

Die folgende Schleife sucht die erste Primzahl $\geq N$, die kongruent zu a modulo m ist. Dabei sollte $ggT(a, m) = 1$ gelten.

```
N := 500;
a := 1;
m := 133;
p := NextPrime(N-1);
found := p mod m eq a;
while not found do
    p := NextPrime(p);
    found := p mod m eq a;
end while;
```

```
p;
```

Aufgabe 5.1. Beweisen Sie, dass die Bedingung $ggT(m, a) = 1$ notwendig ist. Das die Bedingung hinreichend für die Existenz einer Primzahl p mit $p \equiv a \pmod{m}$ ist, ist nicht-trivial. Tatsächlich gibt es unendlich viele solcher Primzahlen. Schreiben Sie ein `intrinsic MyPrime(N :: RngIntElt, a :: RngIntElt, m :: RngIntElt) -> RngIntElt`, das die kleinste Primzahl $p \geq N$ mit $p \equiv a \pmod{m}$ liefert. $\leq N$ zurück liefert.

Aufgabe 5.2. Schreiben Sie ein `intrinsic MyPrimes(N :: RngIntElt) -> SeqEnum`, das eine Sequenz mit den Primzahlen $\leq N$ zurück liefert.

6. Der Chinesische Restsatz (für den Ring $\mathbb{Z}$)

Der Chinesische Restsatz ist vielleicht aus der linearen Algebra bekannt. Wenn nicht, so ist dies an dieser Stelle kein Nachteil, da wir den Satz hier formulieren und beweisen werden. Der Beweis ist, wie wir sehen werden, konstruktiv und wir können ihn fast wortwörtlich in eine MAGMA-Funktion umsetzen.

Theorem 6.1. *Sei $n \geq 1$ eine natürliche Zahl und $a_1, \ldots, a_n \in \mathbb{Z}$. Seien $m_1, \ldots, m_n \in \mathbb{N}$ paarweise teilerfremde, natürliche Zahlen. Dann gibt es eine ganze Zahl $x \in \mathbb{Z}$ mit*

$$x \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, n.$$

Die Zahl x ist modulo $m_1 \cdots m_n$ eindeutig bestimmt, d.h., falls $y \in \mathbb{Z}$ ebenfalls die simulatanen Kongruenzen

$$y \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, n,$$

erfüllt, so gilt $x \equiv y \pmod{(m_1 \cdots m_n)}$.

Proof. Dem Beweis schicken wir die folgende Tatsache voraus.

Tatsache: Falls $a, b \in \mathbb{Z}$ und $d := \mathrm{ggT}(a, b)$, so gibt es $p, q \in \mathbb{Z}$ mit $pa + qb = d$.

Den Beweis hierzu werden wir (in größerer Allgemeinheit, nämlich für sogenannte Hauptidealringe) in der Algebra führen. Die explizite Berechnung von p und q ist algorithmisch in sogenannten Euklidischen Ringen und erfolgt über den erweiterten Euklidischen Algorithmus.

Nun zum Beweis des Chinesischen Restsatzes. Der Beweis erfolgt mittels Induktion über n. Falls $n = 1$, so setze $x := a_1$. Da wir den Fall $n = 2$ im Induktionsschritt brauchen werden, zeigen wir die Behauptung zunächst in diesem Fall. Da a_1 und a_2 nach Voraussetzung teilerfremd sind, gibt es $\tilde{p}, \tilde{q} \in \mathbb{Z}$ mit $1 = \tilde{p}m_1 + \tilde{q}m_2$. Multiplikation mit $a_1 - a_2$ liefert

$$a_1 - a_2 = pm_1 + qm_2 \text{ mit } p = (a_1 - a_2)\tilde{p}, q = (a_1 - a_2)\tilde{q}.$$

Für $x := a_1 - pm_1 = a_2 + qm_2$ gilt wie gefordert

$$x \equiv a_1 \pmod{m_1}, \quad x \equiv a_2 \pmod{m_2}.$$

Wir kommen nun zum Induktionsschritt $n - 1 \longrightarrow n$. Dazu setzen wir $M := m_1 \cdots m_{n-1}$. Nach Induktion gibt es eine Lösung $y \in \mathbb{Z}$ von

$$y \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, n - 1.$$

Da $\mathrm{ggT}(M, m_n) = 1$, können wir wie im Fall $n = 2$ die simultanen Kongruenzen

$$x \equiv y \pmod{M}, \quad x \equiv a_n \pmod{m_n}$$

lösen. Da $m_i \mid M$ folgt $x \equiv y \equiv a_i \pmod{m_i}$ für $i = 1, \ldots, n-1$.

Zur Eindeutigkeit: Es gelte sowohl

$$x \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, n$$

als auch

$$y \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, n.$$

Dann folgt $x \equiv y \pmod{m_i}, i = 1, \ldots, n$, was äquivalent zu $m_i \mid (x-y), i = 1, \ldots, n$, ist. Da die m_i paarweise teilerfremd sind, folgt

$$m_1 \cdots m_n \mid (x - y),$$

beziehungsweise äquivalent hierzu

$$x \equiv y \pmod{m_1 \cdots m_n}.$$

□

Wir wollen diesen Beweis nun umsetzen in eine MAGMA-Funktion. Dazu nehmen wir die intrinsic XGCD zur Hilfe. Der Aufruf *d,p,q := XGCD(a,b)* liefert die Zahlen d, p, q wie in der obigen Tatsache.

Der Input für die Funktion seien zwei Listen $a := [a_1, \ldots, a_n], m := [m_1, \ldots, m_n]$. Wir verzichten auf jegliche Konsistenzüberprüngen wie etwa *#a eq #m*, da es uns hier nur um die Implementierung der wesentlichen Teile des Algorithmus ankommt.

```
MyCR := function(a, m)
    M := 1;
    x := a[1];
    for i:=2 to #a do
        y := x;
        M := M * m[i-1];
        d,p,q := XGCD(M, m[i]);
        if d ne 1 then
            print "FEHLER: die m_i sind nicht paarweise teilerfremd.";
            return -1;
        end if;
        p := (y - a[i])*p;
        q := (y - a[i])*q;
        x := y - p*M;
    end for;
    return x mod &*m ;
end function;
```

Aufgabe 6.2. Bevor Sie die Funktion implementieren, spielen Sie bitte selbst Algorithmus und führen die Funktion auf dem Papier am Beispiel

$$\begin{aligned} x &\equiv 1 \pmod{3}, \\ x &\equiv 2 \pmod{5}, \\ x &\equiv 3 \pmod{7} \end{aligned}$$

durch. Vergleichen Sie Beweis und Implementierung.

Aufgabe 6.3. Überprüfen Sie die Sinnhaftigkeit der Fehlermeldung. Zeigen Sie dazu: Die m_i sind genau dann paarweise teilerfremd, wenn für alle $i \in \{2, \ldots, n\}$ gilt: $\mathrm{ggT}(m_1 \cdots m_{i-1}, m_i) = 1$.

Wie wir in der Vorlesung lernen werden, kann man den Chinesischen Restsatz völlig analog in beliebigen sogenannten Euklidischen Ringen formulieren. Der Beweis ist praktisch identisch. Unsere Standardbeispiele für Eukldische Ringe sind $\mathbb{Z}$ und Polynomringe über einem Körper. Da MAGMA in Funktionen keine Typenüberprüfung vornimmt, können wir die Funktion MyCR auch auf Polynome anwenden. Versuchen Sie die folgenden Eingaben.

```
QX<x> := PolynomialRing(Rationals());
a := [x-1,2*x,1,2,3];
m := [x^2 - 5, x^3 - 1, x^3, x^17 - x, x^15 -x^10 - 7];
f := MyCR(a,m);
// Fehler, da die m_i nicht paarweise teilerfremd sind.


a := [x-1,2*x,1,2,3];
m := [x^2 - 5, x^3 - 1, x^3, x^17 - x+1, x^15 -x^10 - 7];
f := MyCR(a,m);
f;
// Kontrolle
[f mod m[i] eq a[i] mod m[i] : i in [1..#a]];
```

Der (erweiterte) Euklidische Algorithmus spielt in der algorithmischen Zahlentheorie eine bedeutende Rolle. Auch im folgenden Satz ist er das wesentliche Hilfsmittel im Beweis, der sich wieder eins zu eins in eine MAGMA-Funktion umsetzen läst. Genaueres hierzu nach dem Satz und Beweis.

Theorem 6.4. *Sei $m \geq 2$ eine natürliche Zahl und $a \in \mathbb{Z}$. Dann gilt:*

$$\overline{a} \in (\mathbb{Z}/m\mathbb{Z})^\times \iff \mathrm{ggT}(a,m) = 1.$$

Proof. Falls a modulo m invertierbar ist, so gibt es ein $b \in \mathbb{Z}$ mit $ab \equiv 1 \pmod{m}$. Also gibt es ein $v \in \mathbb{Z}$ mit $ab - 1 = vm$, oder äquivalent dazu, $ab - vm = 1$. Wäre nun $d := \mathrm{ggT}(a,m) > 1$, so wäre $d > 1$ auch ein Teiler von 1, was absurd ist. Sei umgekehrt $\mathrm{ggT}(a,m) = 1$. Dann gibt es $p, q \in \mathbb{Z}$, so dass $1 = pa + qm$. Lesen wir diese Gleichung modulo m, so erhalten wir $\overline{p} \cdot \overline{a} = \overline{1}$, d.h. $\overline{p}$ ist das Inverse von $\overline{a}$. □

Aufgabe 6.5. Schreiben Sie eine Funktion MyIsInvertibe(a,m), die zu a und m entscheidet, ob a modulo m invertierbar ist und gegebenenfalls das Inverse $\overline{b}$ berechnet. Die Ausgabe soll also aus zwei Werten bestehen, einem BoolElt und einem RngIntElt. Geben Sie für $\overline{b}$ einen Vertreter $b \in \mathbb{Z}$ mit $1 \leq b \leq m-1$ zurück.

Der obige Satz liefert das folgende Korollar.

Corollary 6.6. *Sei $m \geq 2$ eine natürliche Zahl. Dann gilt:*

$$\mathbb{Z}/m\mathbb{Z} \text{ ist ein Körper} \iff m \text{ ist eine Primzahl.}$$

Aufgabe 6.7. Führen Sie die folgende Schleife aus:

```
p := 2;
while p lt 20 do
    p := NextPrime(p);
    print "p = ", p, "    ", [MyIsInvertible(a,p) : a in [1..p-1]];
end while;
```

Beweisen Sie nun das Korollar.

Umlaute ä, ß,.....

7. Gruppentheorie, Teil 1

Wir wollen in diesem Abschnitt verschiedene Typen von Gruppen kennen lernen und verstehen, wie sie in MAGMA dargestellt werden und wie man mit ihnen umgehen kann.

Wir beginnen mit den endlichen abelschen Gruppen. Wie in der Vorlesung bezeichne C_n die zyklische Gruppe der Ordnung n. Hierzu beachte man, daß je zwei zyklische Gruppen der Ordnung n isomorph sind, so daß die Notation gerechtfertigt ist. In MAGMA wird die C_n durch den Befehl `CyclicGroup(GrpAb, n)` erzeugt. Auch der Aufruf `CyclicGroup(n)` ist korrekt, liefert aber keine Gruppe vom Typ `GrpAb`, sondern eine Gruppe vom Typ `GrpPerm`. Hierzu kommen wir später; zunächst wollen wir den Typ `GrpAb` genauer betrachten. Geben Sie

```
C := CyclicGroup(GrpAb, 14); C;
```

ein. MAGMA liefert uns eine abelsche Gruppe isomorph zu $\mathbb{Z}/14\mathbb{Z}$ zurück. Der Erzeuger kann mit `C.1` angesprochen werden. Schöner ist es unter Umständen, wenn man

```
C<a> := CyclicGroup(GrpAb, 14); C;
```

benutzt. Dann kann man auf den Erzeuger mittels `a` zugreifen. Versuchen sie die folgenden Eingaben, überlegen Sie sich aber bereits zuvor das Ergebnis.

```
Order(a);
Order(2*a);
Order(7*a);
8*a eq 22*a;
Order(3*a) eq Order(a);
{x : x in C} eq {i*a : i in [5..18]};
```

Wir betrachten nun abelsche Gruppen der Form $C_{n_1} \times \ldots \times C_{n_l}$ mit $l, n_1, \ldots, n_l \in \mathbb{N}$. Dies ist eine abelsche Gruppe der Ordnung $n_1 \cdots n_l$. Der Aufruf

```
AbelianGroup([$n_1, \ldots, n_l$])
```

erzeugt diese Gruppe. Wie wir in der Vorlesung lernen werden, ist jede endliche abelsche Gruppe bis auf Isomorphie von dieser Form. Mit dem Kommando

```
A := AbelianGroup([2,3,5]);
```

erzeugen wir also eine abelsche Gruppe, die isomorph zu $\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/3\mathbb{Z} \times \mathbb{Z}/5\mathbb{Z}$ ist. Die Erzeuger heißen `A.1, A.2` und `A.3`. Versuchen sie die folgenden Eingaben, überlegen Sie sich aber wieder vorab das Ergebnis.

```
[Order(A.1), Order(A.2), Order(A.3)];
A<a,b,c> := AbelianGroup([2,3,5]);
a eq A.1; b eq A.2; c eq A.3;
Order(a+b);
a+b eq a + 4*b;
Order(A);
```

Der Aufrauf `AbelianGroup([n])` erzeugt ebensfalls die zyklische Gruppe C_n. Hierzu folgendes Beispiel:

```
A<a> := AbelianGroup([100]);
B<b> := CyclicGroup(GrpAb, 100);
bool, f := IsIsomorphic(A,B);
bool; f;
f(a);
```

Sicherlich haben Sie sich gewundert, warum `CyclicGroup` zwei Argumente hat. Wie bereits weiter oben erwähnt, ist auch `CyclicGroup(n)` korrekt, liefert aber eine Gruppe vom Typ `GrpPerm`. Versuchen Sie folgende Eingaben:

```
C<c> := CyclicGroup(100);
C;
Type(C);
IsIsomorphic(A,C);
```

MAGMA weigert sich also Gruppen von verschiedenem Typ auf Isomorphie zu testen. Mit dem Befehl `PermutationGroup(A)` kann man aus einer Gruppe A vom Typ `GrpAb` eine Gruppe vom Typ `GrpPerm` machen. Hier ein Beispiel

```
C<c> := CyclicGroup(100);
C;
Type(C);
IsIsomorphic(PermutationGroup(A), C);
```

`GrpPerm` steht für endliche Permutationsgruppen. Dies ist der Typ von Gruppen, für die es die meisten Algorithmen gibt. Natürlich kann man diese Probleme auch für endliche abelsche Gruppen lösen. Hier sind die Probleme jedoch meist trivial oder haben eine einfache Lösung. Wir beschäftigen uns nun also mit Permutationsgruppen. Dies sind Untergruppen einer symmetrischen Gruppe S_n. Ohne Einschränkung stellen wir uns die S_n stets als die Gruppe der Permutationen der Menge $\{1, \ldots, n\}$ vor. Wie wir in der in der Vorlesung sehen werden, kann man jede endliche Gruppe G als Permutationsgruppe darstellen (mit einem geeigneten n). Hier ist die Beweisidee, die wir später an Beispielen umsetzen wollen. Sei $\sigma \in G$. Dann defieren wir

$$f_\sigma \colon G \longrightarrow G, \quad \tau \to \sigma\tau.$$

Die Abbildung

$$G \longrightarrow S(G), \quad \sigma \to f_\sigma$$

ist eine Einbettung (d.h. ein injektiver Gruppenhomomorphismus) und wir könne G mit seinem Bild identifizieren. Hierbei bezeichnet $S(G)$ die Gruppe der Permutationen der Menge G. Numerieren wir die Gruppenelemente, so können wir natürlich $S(G)$ und $S_{|G|}$ identifizieren.

Wir müssen also zunächst die symmetrischen Gruppen S_n untersuchen. Sie werden in MAGMA durch `SymmetricGroup(n)` oder kurz durch `Sym(n)` erzeugt. Versuchen Sie

```
S := Sym(3);
Set(S);
```

Wie wir sehen verwendet MAGMA für die Elemente der symmetrischen Gruppen die übliche Zyklenschreibweise. Im Gegensatz zur Vorlesung wird in MAGMA jedoch $\sigma\tau$ nicht wie "σ nach τ" interpretiert, sondern wie "τ nach σ". Grund hierfür ist die MAGMA-Konvention, daß Abbildungen stets von rechts wirken. Es gilt also zum Beispiel $(1,2)(2,3) = (1,3,2)$. Die entsprechende Eingabe in MAGMA ist

```
s := S ! (1,2)(2,3);
s;
```

Betrachten wir nun das folgende Beispiel.

```
C<a> := CyclicGroup(4); C;
C ! (1,2)(3,4);
C ! (1,3)(2,4);
```

C wird also dargestellt als Untergruppe der S_4 erzeugt von $a = (1, 2, 3, 4)$. Da

$$\langle a \rangle = \{id, (1, 2, 3, 4), (1, 3)(2, 4), (1, 4, 3, 2)\}$$

erklären sich die obigen Ausgaben von MAGMA.

Weitere wichtige Gruppen, die in MAGMA als Permutationsgruppen dargestellt werden, sind die Diedergruppen. Das intrinsic `DihedralGroup(n)` erzeugt die Diedergruppe D_n mit $D_n = 2n$. Mit dem Befehl `AlternatingGroup(n)` erzeugt man die alternierende Guppe A_n.

Aufgabe 7.1. Testen Sie mit MAGMA, ob $D_3 \simeq S_3$ und $C_3 \simeq A_3$. Geben Sie gegebenenfalls eine expliziten Isomorphismus an.

Als letzten Typ von Gruppen wollen wir Matrixgruppen kennen lernen. Wir beschränken uns heir auf die Konstruktion einer $\mathrm{Gl}_n(\mathbb{F}_p)$ und ihrer Untergruppen. Hierbei bezeichnet $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ den Körper mit p Elementen. Ferner werden wir die Quaternionengruppe der Ordnung 8 zunächst als Matrixgruppe, und darauf aufbauend als Permutationsgruppe realisieren.

Wir betrachten folgende Sequenz.

```
QX<x> := PolynomialRing( Rationals());
F<i> := NumberField(x^2+1);
M := GL(2,F);
E := M![1,0,0,1];
I := M![i,0,0,-i];
J := M![0,-1,1,0];
K := M![0,-i,-i,0];
Q := MatrixGroup<2,F|[I,J,K]>;
```

Zunächst definieren wir den Polynomring $\mathbb{Q}[x]$ über den rationalen Zahlen. Der zweite Befehl erzeugt die Gaußschen Zahlen $\mathbb{Q}(i)$. `M` ist die Gruppe der invertierbaren 2×2-Matrizen mit Einträgen in $F = \mathbb{Q}(i)$. Durch die folgenden Zeilen werden die Matrizen

$$E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, I = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}, J = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, K = \begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix}$$

definiert, und schließlich in der letzten Zeile die Untergruppe der $\mathrm{Gl}_2(F)$ die durch I, J, K erzeugt wird.

Wie wir aus den Übungen wissen, ist Q die Quaternionengruppe der Ordnung 8. Die Befehle

```
IsAbelian(Q);
#Q;
```

bestätigen dies. Aus den Übungen wissen wir außerdem, daß es genau zwei nichtabelsche Gruppen der Ordnung 8 (bis auf Isomorphie) gibt. Versuchen Sie

```
D := DihedralGroup(4);
IsIsomorphic(Q, D);
```

Wie zu erwarten war, weigert sich MAGMA, da `Q` vom Typ `GrpMat` ist und nicht wie `D` vom Typ `GrpPerm`. Wir wollen daher die Quaternionengruppe auch noch als Permutationsgruppe realisieren und verwenden dazu die obige Beweisskizze. Betrachten Sie

```
S8 := Sym(8);
Qelts := [g : g in Q];
v := [Index(Qelts, I*g) : g in Qelts]; v;
sigma := S8!v;
w := [Index(Qelts, J*g) : g in Qelts];
tau := S8!w;
H := sub<S8 | [sigma, tau]>;
#H;
D := DihedralGroup(4);
#D;
IsIsomorphic(D, H);
IsAbelian(H);
IsAbelian(D);
#D eq #H;
```

Aufgabe 7.2. Versuchen Sie die obige Befehlssequenz zu verstehen. Stellen Sie dazu den Zusammenhang zur obigen Beweisskizze her.

Wir wollen nun eine Funktion schreiben, die die D_4 als Permutationsgruppe erzeugt. Dazu stellen wir uns die D_4 als die Gruppe der Isometrien eines Quadrates vor. Wir numerieren die Ecken von 1 bis 4. Jede Isometrie ist durch die zugehörige Permutation der Ecken eindeutig bestimmt. Also können wir die D_4 als Untergruppe der S_4 darstellen. Aus der Vorlesung wissen wir, daß die D_4 durch die Drehung σ um 90 Grad und eine Spiegelung τ erzeugt wird. Dies sollte ausreichen, um die folgende Funktion zu verstehen.

```
MyD4 := function()
    S4 := Sym(4);
    sigma := S4 ! (1,2,3,4);
    tau := S4 ! (1,2)(3,4);
    return sub<S4 | [sigma, tau]>;
end function;
```

Wir testen die Funktion durch

```
G := MyD4();
G;
IsIsomorphic(G, DihedralGroup(4));
```

Aufgabe 7.3. Schreiben Sie Funktionen MyD5() und MyD6(), und allgemeiner eine Funktion MyDn(n).

Aufgabe 7.4. Schreiben Sie eine Funktion MyQ8(), die die Quaternionengruppe der Ordnung 8 als Permuatuonsgruppe erzeugt.

8. Zyklische Untergruppen, Untergruppen, Normalteiler

In diesem Abschnitt wollen wir uns mit Untergruppen einer gegegebenen Gruppe G beschäftigen. Mit dem Befehl `sub< G | [`$g_1, \ldots, g_n$`] >` kann man die von

den Elementen $g_1, \ldots, g_n \in G$ erzeugte Untergruppe erzeugen. Geben Sie folgende Zeilen ein.

```
G := Sym(5);
Gens := Generators(G);
tau := Gens[1];
sigma := Gens[2];
U1 := sub<G | [tau]>;
U2 := sub<G | [sigma]>;
U := sub<G | [sigma, tau]>
```

U_1 und U_2 sind also jeweils von einem Element erzeugt, d.h. dies sind die von τ bzw. σ erzeugten zyklischen Untergruppen von G. Da für eine zyklische Untergruppe $\langle\gamma\rangle \leq G$ ganz allgemein (unabhängig von unserer speziellen Situation im Beispiel) mit einem $\gamma \in G$ von endlicher Ordnung gilt, daß

$$\langle\gamma\rangle = \{\gamma, \gamma^2, \ldots, \gamma^{\mathrm{ord}(\gamma)} = 1\}$$

ist in unserem Beispiel $|U_1| = \mathrm{ord}(\tau) = 2$ und $|U_2| = \mathrm{ord}(\sigma) = 5$. Da ferner σ und τ G erzeugen, gilt $U = G$. Wir überprüfen dies mittels

```
Order(tau) eq #U1;
Order(sigma) eq #U2;
U eq G;
```

Wir wollen nun selbst eine einfache Funktion schreiben, die die Ordnung eines Elements $g \in G$ für eine endliche Gruppe G bestimmt.

```
Ordnung := function(g)
    G := Parent(g);
    a := g;
    o := 1;
    while a ne One(G) do
        o := o + 1;
        a := a*g;
    end while;
    return o;
end function;
```

Hier zwei Erläuterungen: Jedes Gruppenelement 'lebt' in einer Gruppe; diese erhält man durch den Befehl `Parent(g)`. Mit `One(G)` haben wir Zugriff auf das neutrale Element von G.

Wir wollen nun in $G := \mathrm{Gl}(2, \mathbb{F}_q)$, $q = p^n$, die fogenden zwei Untergruppen

$$\begin{aligned} D &= \left\{ \begin{pmatrix} \alpha & 0 \\ 0 & \beta \end{pmatrix} \mid \alpha, \beta \in \mathbb{F}_q^\times \right\}, \\ B &= \left\{ \begin{pmatrix} 1 & \alpha \\ 0 & 1 \end{pmatrix} \mid \alpha, \beta \in \mathbb{F}_q^\times \right\}, \end{aligned}$$

als Untergruppen der $\mathrm{Gl}(2, \mathbb{F}_q)$ realisieren. Dazu die folgenden Vorbemerkungen. Den endlichen Körper mit q Elementen erzeugt man mit dem Befehl `GaloisField(q)`, oder kurz mit `GF(q)`. In der Vorlesung werden wir zeigen, daß endliche Untergruppen der multiplikativen Gruppe eines Körpers stets zyklisch sind. Die folgende Sequenz berechnet ein Element $w \in \mathbb{F}_q^\times$ mit $\langle w\rangle = \mathbb{F}_q^\times$ für $q = 11$.

```
q := 11;
F := GF(q);
U, j := UnitGroup(F); U; j;
w := j(U.1);
```

Durch die folgenden Befehle

```
G := GL(2, F);
A := G ! [w,0,0,1];
H := sub<G | [A]>;
```

wird zunächst die Matrix $A = \begin{pmatrix} w & 0 \\ 0 & 1 \end{pmatrix}$ und sodann die Untergruppe

$$H = \left\{ \begin{pmatrix} \alpha & 0 \\ 0 & 1 \end{pmatrix} \mid \alpha \in \mathbb{F}_q^\times \right\}$$

erzeugt.

Aufgabe 8.1. Schreiben Sie eine Funktion, die die beiden Untergruppen D und B erzeugt.

Wir wollen nun zu einer Untergruppe U von G und $g \in G$ die Linksnebenklasse gU sowie die Rechtsnebenklasse Ug bestimmen. Wir werden dazu intrinsics benutzen, da dadurch, wie wir gleich sehen werden, der Aufruf sehr intuitiv wird. Erzeugen Sie ein File, zum Beispiel mit dem Namen *MyIntrinsics.m*, und schreiben Sie die beiden intrinsics

```
intrinsic Nebenklasse(g :: GrpPermElt, U :: GrpPerm) -> SetEnum
{Berechnet die Linksnebenklasse gU.}
    return {g*u : u in U};
end intrinsic;

intrinsic Nebenklasse(U :: GrpPerm, g :: GrpPermElt) -> SetEnum
{Berechnet die Linksnebenklasse gU.}
    return {u*g : u in U};
end intrinsic;
```

Die Datei wird nun durch

```
Attach("MyIntrinsics.m");
```

zu den MAGMA-Funktionen hinzugefügt. MAGMA handhabt nun die Funktion `Nebenklasse` wie die MAGMA-eigenen intrinsics, so funktioniert jetzt zum Beispiel

```
Nebenklasse;
```

Testen Sie die Funktion, etwa mit folgendem Beispiel

```
S := Sym(3);
sigma := S.1;
U := sub<S | [S!(1,2)]>;
sigmaU := Nebenklasse(sigma, U);
Usigma := Nebenklasse(U, sigma);
```

Nachdem wir diese Funktionen zur Verfügung haben, wollen wir sie verwenden, um die Menge der Linksnebenklassen G/U sowie die Menge der Rechtsnebenklassen $U\backslash G$ berechnet.

```
intrinsic Linksnebenklassen(G :: GrpPerm, U :: GrpPerm) -> SeqEnum, SeqEnum
{Berechnet G/U.}
    Gset := Set(G);
    R := [Id(G)];
    Q := Nebenklasse(Id(G), U);
    Nk := [ Q ];
    while #Q lt #G do
        g := [g : g in Gset diff Q][1];
        gU := Nebenklasse(g, U);
        Q := Q join gU;
        Append(~R, g);
        Append(~Nk, gU);
    end while;
    return Nk, R;
end intrinsic;

intrinsic Rechtsnebenklassen(G :: GrpPerm, U :: GrpPerm) -> SeqEnum, SeqEnum
{Berechnet U\G.}
    Gset := Set(G);
    R := [Id(G)];
    Q := Nebenklasse(U, Id(G));
    Nk := [ Q ];
    while #Q lt #G do
        g := [g : g in Gset diff Q][1];
        Ug := Nebenklasse(U, g);
        Q := Q join Ug;
        Append(~R, g);
        Append(~Nk, Ug);
    end while;
    return Nk, R;
end intrinsic;
```

Aufgabe 8.2. Spielen Sie die Funktionen am obigen Beispiel durch.

Nachdem Sie sich klar gemacht haben, wie die Funktionen arbeiten, versuchen wir die folgenden Beispiele.

```
G := Sym(3);
U := sub<G | [G!(1,2)]>;
L := Linksnebenklassen(G, U);
R := Rechtsnebenklassen(G, U);
L eq R;

H := AlternatingGroup(4);
V := sub<H | [H!(1,2)(3,4), H!(1,3)(2,4)]>;
L := Linksnebenklassen(H, V);
R := Rechtsnebenklassen(H, V);
L eq R;
```

Wie das erste Beispiel zeigt, stimmen im Allgemeinen Rechts- und Linksnebenklassen nicht überein. Wie wir in der Vorlesung sehen, ist dies aber die Voraussetzung

dafür, daß die von G auf G/U induzierte Multiplikation wohldefiniert ist. Für zwei Nebenklassen g_1U, g_2U möchte man

$$g_1U \cdot g_2U := g_1g_2U$$

definieren. Dies soll unabhängig von der Wahl von Vertretern g_1 bzw. g_2 sein.

Aufgabe 8.3. Finden Sie im ersten Beispiel oben zwei Nebenklassen g_1U und g_2U, so dass die so definierte Multiplikation nicht wohldefiniert ist.

Wir erinnern an die folgende Definition.

Definition 8.4. Sei G eine Gruppe und N eine Untegruppe von G. Dann ist N ein Normalteiler, falls für alle $g \in G$ gilt: $gU = Ug$.

Offensichtlich ist die definierende Bedingung äquivalent zu $g^{-1}Ug = U$ für alle $g \in G$.

Aufgabe 8.5. Zeigen Sie, daß es reicht, die Normalteilereigenschaft für die Erzeugenden von G zu überprüfen. Schreiben Sie eine intrinsic IstNormal (G, U).

Aufgabe 8.6. Bestimmen Sie die sämtlichen Untergruppen der D_4, Q_8 und der A_4. Zeichnen Sie jeweils ein Untergruppendiagramm. Welche der Untergruppen sind jeweils Normalteiler.

Literatur

Institut für Mathematik der Universität Kassel, Heinrich-Plett-Str. 40, 34132 Kassel, Germany
E-mail address: bley@mathematik.uni-kassel.de