Mathematics 567 **Introduction to** GAP A. Hulpke

Some of the examples in this course will involve the computer algebra system GAP.
You are also welcome to use GAP (or any other computer algebra system) for calculations, as long as they do not render the problem trivial. For example, if a problem asks to calculate a GCD, you could use GAP to do the division with remainders. If a problem was to calculate a Smith normal form, you could use GAP for gcd computations, etc.

You can download a GAP installer from: `http://www.math.colostate.edu/~hulpke/CGT/education.html`.
You can start GAP with the GGAP icon on the screen or as `gap` on Unix.. The program will start up and you will get a prompt/worksheet with a prompt that lets you enter commands:

```
>
```

You now can type in commands (followed by a semicolon) and GAP will return the result. If you type a partical command name and then type `TAB`, GAP will give you a list of all possible completions.
The worksheet works similarly to Maple. While you can return to previous lines and change (or re-execute) the command there, this does not time travel and change the sequence of commands executed so far, but simply executes this as new command.
Editing and cut/paste ought to work as usual – if not please let me know!
You can use the online help to get documentation

```
> ?gcd
Help: Showing 'Reference: Gcd'
> Gcd( <R>, <r1>, <r2>, ... )[...]
```

(For example `?Line Editing` to get from the online help a list of editing commands.)
You might want to create a transcript of your session in a text file. On the laboratory machines the best place for this is the `My Documents` folder. You create the log by issuing the command

```
LogTo("C:/Documents and Settings/m467/My Documents/logfile.txt");
```

Two caveats:

- You must type *forward* slashes (/) even though the standard Windows convention is for backslashes.
- You all share the same account – files with the same name will overwrite each other. A good idea therefore is to have files include e.g. your eName to be unique, e.g. call a log file *yourname*`log.txt`.

You can end logging with the command

```
LogTo();
```

If you want to enter a set of commands (or even write your own code) it is convenient to write this in a text file (`.txt`), for example using `NotePad`. (Careful with using `Word`, which usually creates more complicated files!)
You can read in such a file using the `Read` command:

```
> Read("C:/Documents and Settings/m467/My Documents/myprogram.txt");
```

(Again careful with name clashes, if you are using the common class account!)
If you want to continue work later, you can save a *workspace*, which saves also the internal state of GAP. Loading this workspace gets you back to the point where you were before. (Saving a *worksheet* only saves the text of input and output without the GAP objects.)

Numbers

GAP knows integers of arbitrary length and rational numbers:

```
> 17 - 23;
-6
> 123456/7891011+1;
2671489/2630337
```

GAP knows a precedence between operators that may be overridden by parentheses and can compare objects:

```
> (9 - 7) * 5 = 9 - 7  * 5;
false
> 5/3<2;
true
```

You can assign numbers (or more general: every GAP object) to variables, by using the assignment operator `:=`. Once a variable is assigned to, you can refer to it as if it was a number. The special variables `last`, `last2`, and `last3` contain the results of the last three commands.

```
> a:=2^16-1;
65535
> b:=a/(2^4+1);
3855
```

The following commands show some useful integer calculations related to quotient and remainder:

```
> Int(8/3); # round down
2
> QuoInt(76,23); # integral part of quotient
3
> QuotientRemainder(76,23);
[ 3, 7 ]
```

```
> 76 mod 23; # remainder (note the blanks)
7
> 1/5 mod 7;
3
> Gcd(64,30);
2
> rep:=GcdRepresentation(64,30);
[ -7, 15 ]
> rep[1]*64+rep[2]*30;
2

> Factors(2^64-1);
[ 3, 5, 17, 257, 641, 65537, 6700417 ]
```

Polynomials

To create polynomials, we need to create the variable first. Note that the name we give is the printed name (which could differ from the variable we assign it to). As GAP does not know the real numbers we do it over the rationals

```
> x:=X(Rationals,"x");
x

> f:=x^7+3*x^6+x^5+3*x^3-4*x^2-4*x;
x^7+3*x^6+x^5+3*x^3-4*x^2-4*x
> g:=x^5+2*x^4-x^2-2*x;
x^5+2*x^4-x^2-2*x
> f+g;f*g;
x^7+3*x^6+2*x^5+2*x^4+3*x^3-5*x^2-6*x
x^12+5*x^11+7*x^10+x^9-2*x^8-5*x^7-14*x^6-11*x^5-2*x^4+12*x^3+8*x^2
```

The same operations as for integers hold for polynomials

```
> QuotientRemainder(f,g);
[ x^2+x-1, 3*x^4+6*x^3-3*x^2-6*x ]
> f mod g;
3*x^4+6*x^3-3*x^2-6*x
> Gcd(f,g);
x^3+x^2-2*x
> GcdRepresentation(f,g);
[ -1/3*x, 1/3*x^3+1/3*x^2-1/3*x+1 ]
> rep:=GcdRepresentation(f,g);
[ -1/3*x, 1/3*x^3+1/3*x^2-1/3*x+1 ]
> rep[1]*f+rep[2]*g;
```

```
x^3+x^2-2*x
> Factors(g);
[ x-1, x, x+2, x^2+x+1 ]
```

By creating further variables, we get multivariate polynomials:

```
> y:=X(Rationals,"y");
y
> p:=x^2-y^2;
x^2-y^2
> Factors(p);
[ x-y, x+y ]
```

`Value` can be used to evaluate polynomials. If the polynomial is univariate, the variable is assumed to be the natural one:

```
> Value(f,2);
352
> Value(f,[x],[2]);
352
> Value(f,[y],[2]);
x^7+3*x^6+x^5+3*x^3-4*x^2-4*x
> Value(p,[x],[3]);
-y^2+9
> Value(p,[y],[2]);
x^2-4
> Value(p,[x,y],[3,2]);
5
> Value(p,[x,y],[y,x]);
-x^2+y^2
```

Lists

Objects separated by commas and enclosed in square brackets form a list.
Collections of objects are represented by such lists. Lists are also used to represent sets.

```
> l:=[5,3,99,17,2]; # create a list
[ 5, 3, 99, 17, 2 ]
> l[4]; # access to list entry
17
> l[3]:=22; # assignment to list entry
22
> l;
[ 5, 3, 22, 17, 2 ]
```

```
> Length(l);
5
> 3 in l; # element test
true
> 4 in l;
false
> Position(l,2);
5
> Add(l,17); # extension of list at end
> l;
[ 5, 3, 22, 17, 2, 17 ]
> s:=Set(l); # new list, sorted, duplicate free
[ 2, 3, 5, 17, 22 ]
```

Results that consist of several numbers are represented as list.

```
> DivisorsInt(96);
[ 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 96 ]
```

There are powerful list functions that often can save programming loops: `List`, `Filtered`, `ForAll`, `ForAny`, `First`. The notation `i -> xyz` is a shorthand for a one parameter function.

```
> l:=[5,3,99,17,2];
[ 5, 3, 99, 17, 2 ]
> List(l,IsPrime);
[ true, true, false, true, true ]
> List(l,i -> i^2);
[ 25, 9, 9801, 289, 4 ]
> Filtered(l,IsPrime);
[ 5, 3, 17, 2 ]
> ForAll(l,i -> i>10);
false
> ForAny(l,i -> i>10);
true
> First(l,i -> i>10);
99
```

A special case of lists are *ranges*, indicated by double dots.

```
> l:=[10..100];
[ 10 .. 100 ]
> Length(l);
91
```

For example to solve homework problem 2b, we want to find years, for which $d = 1$ and $e = 0$. To do so, we first solve the value for a from the defining formula $d = (19a + 24) mod 30$. We then search for years which give this $a = Y mod 19$ value.

```
> a:=(1-24)/19 mod 30;
13
> years:=Filtered([1900..2099],y->y mod 19=a);
[ 1913, 1932, 1951, 1970, 1989, 2008, 2027, 2046, 2065, 2084 ]
```

We now calculate for each of these years the value of e, and check when $e = 0$:

```
> Filtered(years,y-> (2*(y mod 4)+4*(y mod 7) +6+5) mod 7=0);
[ 1913, 2008 ]
```

Matrices

Lists are also used to form vectors and matrices:
A *vector* is simply a list of numbers. A list of (row) vectors is a matrix. GAP knows matrix arithmetic.

```
> mat:=[[1,2,3],[5,6,7],[9,10,12]];;
> mat^5;
[ [ 289876, 342744, 416603 ], [ 766848, 906704, 1102091 ],
  [ 1309817, 1548698, 1882429 ] ]
> DeterminantMat(mat);
-4
```

The command `Display` can be used to get a nicer output:

```
> 3*mat^2-mat;
[ [ 113, 130, 156 ], [ 289, 342, 416 ], [ 492, 584, 711 ] ]
> Display(last);
[ [  113,  130,  156 ],
  [  289,  342,  416 ],
  [  492,  584,  711 ] ]
```

The command `SmithNormalFormIntegerMat` will compute the Smith Normal form of a matrix. `SmithNormalFormIntegerMatTransforms` also computes transforming matrices:

```
> snf:=SmithNormalFormIntegerMatTransforms(mat);
rec( normal := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 4 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ 3, -13, 7 ], [ 1, -2, 1 ], [ 7, -23, 12 ] ],
  colC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 1, 1 ] ],
  colQ := [ [ 1, 0, -2 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ], rank := 3,
  signdet := -1, rowtrans := [ [ 3, -13, 7 ], [ 1, -2, 1 ], [ 7, -23, 12 ] ],
```

```
  coltrans := [ [ 1, 0, -2 ], [ 0, 1, -1 ], [ 0, 1, 0 ] ] )
> snf.rowtrans*mat*snf.coltrans=snf.normal;
true
```

Modulo arithmetic

If we want to compute in the integers modulo n (what we called $\mathbb{Z}_n$ in the lecture) without need to always type `mod` we can create obhjects that immediately reduce their arithmetic modulo n:

```
> im:=Integers mod 6; # represent numbers for ‘‘modulo 6’’ calculations
(Integers mod 6)
```

To convert “ordinary” integers to residue classes, we have to multiply them with the“One” of these residue classes, the command `Int` converts back to ordinary integers:

```
> a:=5*One(im);
ZmodnZObj( 5, 6 )
> b:=3*One(im);
ZmodnZObj( 3, 6 )
> a+b;
ZmodnZObj( 2, 6 )
> Int(last);
2
```

(If one wants one can get all residue classes or – for example test which are invertible).

```
> Elements(im);
[ ZmodnZObj(0,6),ZmodnZObj(1,6),ZmodnZObj(2,6),ZmodnZObj(3,6),
  ZmodnZObj(4,6),ZmodnZObj(5,6) ]
> Filtered(last,x->IsUnit(x));
[ ZmodnZObj( 1, 6 ), ZmodnZObj( 5, 6 ) ]
> Length(last);
2
```

If we calculate modulo a *prime* the output looks a bit different[1], we cannot immediately read of what each object correspoinds to, but need to use ‘Int’ for this.

```
> im:=Integers mod 7;
GF(7)
> One(im);
Z(7)^0
> a:=6*One(im);
```

[1] **Why?** Every nonzero element is invertible, and there is a better way to represent elements internally – in effect, GAP will avoid having to do division-with-remainder when doing arithmetic. You will learn about this in a more advanced algebra course.

```
Z(7)^3
> b:=3*One(im);
Z(7)
> a+b;
Z(7)^2
> Int(a+b);
2
```