Worksheet 1

Version 2015-10-14

1 Basic GAP commands

The following is a long list of basic things you can do in GAP. You don't need to learn all of them at once, but this sheet may be useful as a reference in the future.

1.1 Numbers, variables, assignment

GAP supports computing with arbitrarily large integers and rational numbers.

```
gap> 10 + 40 - 5;
45
gap> 5/3; 34/14;
5/3
17/7
```

As you see, one can put more than one command on a line. You can also compute powers and remainders, and many other things.

```
gap> 17^39;
97164570157532388251963534291362258993980749195 3
gap> 17 mod 5;
2
gap> Fibonacci(10);
55
```

Variables can be assigned using :=, while comparison can be done using = for equality, <> for inequality, and <, >, <=, >= for less, greater, less-or-equal, greater-or-equal, respectively.

```
gap> a;
Error, Variable: 'a' must have a value
not in any function at line 1 of *stdin*
gap> a := Int(17^39 / 3^7);
444282442421272922962796224468963232711388885
gap> b := 17^39 mod 3^7;
458
gap> a * 3^7 + b = 17^39;
true
gap> a < b;
false
```

Output can be suppressed by ending the line with a double semicolon – try without it!

```
gap> 1234^567;;
gap>
```

1.2 Lists

Lists are ordered collections of arbitrary other GAP objects.

```
gap> l := [1, 3, -100, 3^7/2^9, [ 5 ] ];
[ 1, 3, -100, 2187/512, [ 5 ] ]
gap> l[1];
1
gap> l[1]:=42;
42
gap> l;
[ 42, 3, -100, 2187/512, [ 5 ] ]
```

You can ask a list for its length; reading beyond that gives an error, but you can write to arbitrary positions:

```
gap> Length(l);
5
gap> l[6];
Error, List Element: <list>[6] must have an assigned value
not in any function at line 15 of *stdin*
you can 'return;' after assigning a value
brk> quit;
gap> l[6] := 12;;
gap> l[10] := -9;;
gap> l;
[ 1, 3, -100, 2187/512, [ 5 ], 12,,,, -9 ]
```

A special type of list are ranges:

```
gap> r:=[2..10];
[ 2 .. 10 ]
gap> Length(r);
9
gap> r[1];
2
gap> Flat(r);
[ 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
gap> Flat([1,3..11]);
[ 1, 3, 5, 7, 9, 11 ]
```

That last one was a range with a step size different from one. Play a bit with them!

The operator `in` can be used to test whether an element is contained in a collection:

```
gap> 11 in r;
false
gap> -17 in PositiveIntegers;
false
```

Exercise 1.2.1

Consider this example.

```
gap> l := [1, 3, -100, 3^7/2^9, [ 5 ] ];;
gap> x := l[5];;
gap> x[1] := 100;;
gap> l;
[ 1, 3, -100, 2187/512, [ 100 ] ]
```

Did you expect this? Why did `l` change?

1.3 Other data types

There are many other data types in GAP: We already saw strings on the previous sheet for the `Read` and `Print` commands. The special sequence `\n` is used to indicate where a new line should begin.

```
gap> Print("Hello, world!\n\nIt's a wonderful morning, don't you agree?\n");
Hello, world!

It's a wonderful morning, don't you agree?
```

Then there are also floating point numbers (also known as “floats”), for “decimal” numbers like 3.141592, as well as “records” (don’t worry, you don’t need to know more about them right now). Beyond that, there are also many more complex data types, e.g. for various kinds of groups. We will get to know some of them over time.

1.4 Conditionals: if-then-else-fi

You can use conditionals on the prompt:

```
gap> if 3^3 < 4^4 then Print("OK\n"); fi;
```

General syntax

```
if CONDITION then
    INSTRUCTION; ...
elif CONDITION then  # elif block is optional
    INSTRUCTION; ...
else                 # else block is optional, too
    INSTRUCTION; ...
fi;
```

A “condition” can be any expression taking on a boolean value, i.e. `true` or `false`. This could be for example the result of a comparison such as $a < b$. You can use `and`, `or` and `not` to combine boolean values as expected, e.g. `(a < b) and (a <> 2)`.

Here is a non-trivial example.

test.g

```
a := Random([-100..100]);
b := Random([-100..100]);
if a < b and b < 0 then
    Print("a < b < 0");
elif a = b then
    Print("a = b");
else
    Print("a > b or b >= 0");
fi;
```

Exercise 1.4.1

Write a program which computes a random integer between 1 and 100, then tests whether it is divisible by 7 and prints “yes” if it is, and “no” otherwise.

1.5 Loops

There are three kinds of loops in GAP.

1.5.1 “while” loops

These repeat a set of instructions *while* the given condition holds true:[1]

```
gap> x := 5;; while x > 0 do x := x - 1; Print(x); od;
43210
```

General syntax

```
while CONDITION do
    INSTRUCTION; ...
od;
```

1.5.2 ‘repeat-until” loops

These are similar to while loops, but check the condition at the end, and run *until* the condition is true.

```
gap> x := 2;; repeat x := x * x; until x > 100; x;
256
```

[1] By the way, can you explain the output of this example?

General syntax

```
repeat
    INSTRUCTION; ...
until CONDITION;
```

1.5.3 “for” loops

These iterate over the elements in a list, and execute the given instructions for each element.

```
gap> sum := 0;; for i in [1..10] do sum := sum + i; od; sum;
55
```

General syntax

```
for VARIABLE in LIST do
    INSTRUCTION; ...
od;
```

Note that the above loop is essentially equivalent to the following while-loop:

```
index := 1;
while index <= Length(LIST) do
  VARIABLE := LIST[index];
  INSTRUCTION; ...
  index := index + 1;
od;
```

Thus, it is safe to append elements to the LIST that is being iterated over.

Exercise 1.5.1

Write a program which prints the the cubes of the integers from 5 to 15 (inclusive), as well as the sum of these cubes.

1.5.4 Modifying loop behavior

You can also abort a loop early or skip to the next loop iteration via the `break` and `continue` keywords.

Exercise 1.5.2

Write a program which computes random integers between 1 and 100 and prints them, and which stops once it encounters an integer greater than 90.

1.6 Functions

As in other programming languages, functions in GAP take a list of parameters, do something with them, and possibly return a value.

However, functions in GAP are objects themselves.

```
gap> f := function(n) return n*(n-1)/2; end;;
gap> f(12);
66
```

General syntax

```
MYFUNC := function(A, B, ...)
    local LOCAL_VAR1, ...;
    INSTRUCTION; ...
    return VALUE;
end;
```

The local keyword is followed by a list of variable names you intend to use locally inside the function. Any variable not listed there is assumed to be a global variable (and thus visible outside your function).

Here is a somewhat longer example.

```
sumSquares := function(n)
    local sum, i;
    sum := 0;
    for i in [1..n] do
        sum := sum + i^2;
    od;
    return sum;
end;
```

After entering the above code into GAP (e.g. via `Read`), we can use the function:

```
gap> sumSquares(10);
385
```

Exercise 1.6.1

Implement a function `Checksum(a)` which returns the sum of the decimal digits of an integer a, i.e. `Checksum(123)`$= 1 + 2 + 3 = 6$.

There is also a short hand syntax for simple short functions with a single parameter:

```
gap> square := a -> a^2;;
gap> square(17);
289
```

GAP can even show you the source code for functions (though its formatting may differ from what you originally entered).

```
gap> Display(square);
function ( a )
    return a ^ 2;
end
```

1.7 Code formatting

In this course, we want you to also learn some “best practices”, which, while not strictly necessary, will help you on the long run. Therefore, we require you to take some time to nicely format your code. The text editors we recommended all can assist you with that. Please:

1. Indent your code consistently. E.g. I use 4 spaces per level, but you can also use tabs or a different number of spaces. But do not mix spaces and tabs!
2. Do not put multiple commands on a single line.
3. In general, put “if-CONDITION-then”, “while-CONDITION-do” and “until CONDITION;” on a single line; if it gets too long to be readable, you can wrap it, but be careful how you format it.
4. Put a single space before and after operators like `:=`, `=`, etc., and a single space after (but not before) commas.
5. . . . and more – apply some common sense. And if in doubt, feel free to ask us.

We may refuse to look at (in particular, grade) code which is badly formatted and thus hard to read.

Bad example

```
a:=100;for i in
[1
..100] do a:=a+i;Print("a=",a,"\n");od;
```

Good example

```
a := 100;
for i in [1..100] do
    a := a + i;
    Print("a=", a, "\n");
od;
```

1.8 Additional exercises

1. Consider the following two functions:

```
g1 := function(n)
    if n <= 2 then
        return 1;
    fi;
    return g1(n-1) + g1(n-2);
end;
g2 := function(n)
    local tmp, i;
    tmp := [1,1];
    for i in [3..n] do
        tmp[i] := tmp[i-1] + tmp[i-2];
    od;
    return tmp[n];
end;
```

(a) Without using GAP, determine `g1(i)` and `g2(i)` for $1 \leq i \leq 10$, then compare with GAP.

(b) Do you recognize what these functions compute?

(c) With the `time` variable I measured the runtime in milliseconds of the two commands on my laptop:

```
gap> g1(30);; time;
123
gap> g2(30);; time;
0
```

Determine timings for $30 \leq n \leq 37$ on your computer and sketch (with pen and paper) plots of n against the time. Make an educated guess how long computing `g1(100)` and `g2(100)` might take. Can you explain why `g2` is so much faster than `g1`?

2. Enter the following program.

```
i := 1;
f := function(n)
    local sum;
    sum := 0;
    for i in [1..n] do
        sum := sum + i;
    od;
    return sum;
end;
f(10);
i;
```

What is its output? Is that what you expected? What went wrong?

1.9 Homework

Remark 1.9.1

Homework solutions should be presented to the tutor in the exercise class. You will have to demonstrate that your program works (with some sample inputs provided by the tutor), then show your source code and explain how it works.
Sometimes, homework can be easily solved by calling a function from the GAP library. E.g. to compute the extended euclidean algorithm, you could use `Gcdex`. This is of course *not* the idea – and we will not accept such solutions. Use some common sense. If you are uncertain what you may or may not use for a given exercise, ask!

(s1-h1) Implement a function `Weird(a)` which returns the product of the decimal digits (after adding 1 to each digit) of an integer a, i.e. `Weird(123)`$= (1+1)(2+1)(3+1) = 24$.

(s1-h2) Write a program which loops over the numbers from 1 to 100, and outputs a single line for each number, namely: for each number divisible by 3 it prints “fizz”; for each number divisible by 5 it prints “buzz”; for numbers divisible by both, it prints “fizz buzz”; and in all other cases it prints the number.

(s1-h3) Implement a function `Euclid(a,b)` which computes $\gcd(a, b)$, without using any GAP library functions (in particular, without using `Gcd` or `GcdInt`).

(Hint: Do not start typing right away. First recall how the algorithm worked. Perhaps sketch it with pen and paper).

(s1-h4*) Implement a function `ExtendedEuclid(a,b)` which computes the extended euclidean algorithm. It should return a list `[x,y,g]` such that $\gcd(a, b) = g = ax + by$.