

# A Quick Introduction to GAP

Michael Wu\*

January 25, 2006

---

\*The author is an undergraduate at the Franklin W. Olin College of Engineering and completed this work while supported by the National Science Foundation Grant CCLI DUE-0410517 and under supervision by Professor Sarah Spence Adams.

# Contents

<b>1</b>	<b>Getting Started with GAP</b>	<b>3</b>
1.1	Installing GAP . . . . .	3
1.2	Installing Guava . . . . .	3
1.3	Ready, Set, Go . . . . .	4
1.4	Using Help in GAP . . . . .	4
<b>2</b>	<b>General Operations in GAP</b>	<b>5</b>
2.1	Variables and Objects . . . . .	5
2.2	Arithmetic Operators . . . . .	6
2.3	Comparison Operators . . . . .	6
<b>3</b>	<b>Data Structures</b>	<b>7</b>
3.1	Lists . . . . .	7
3.2	Vectors . . . . .	8
3.3	Range . . . . .	8
3.4	Matrices . . . . .	9
3.5	Characters and Strings . . . . .	10
<b>4</b>	<b>Scripts</b>	<b>10</b>
4.1	Creating Scripts . . . . .	11
4.2	Print . . . . .	11
4.3	Running Scripts from Other Directories . . . . .	11
<b>5</b>	<b>Functions</b>	<b>12</b>
<b>6</b>	<b>Conditionals</b>	<b>12</b>
<b>7</b>	<b>Loops</b>	<b>13</b>
7.1	For Loops . . . . .	13
7.2	While Loops . . . . .	14
<b>8</b>	<b>Exercises</b>	<b>15</b>

# 1 Getting Started with GAP

GAP is a system for computational discrete algebra. This software is specialized for working with groups, rings, and vector spaces. It will aid and expand our exploration into both coding theory and cryptology. The GAP package GUAVA is specialized for working with algebraic error-control codes, which will allow even deeper use of GAP for the coding theory portion of the course.

## 1.1 Installing GAP

This document assumes you are operating Microsoft Windows. For detailed installation instructions and support information on other operating systems, such as MacOS, and UNIX, please consult <http://www.gap-system.org/>.

To install GAP on windows, please follow the instructions below.

1. Download `gap4r4p6.zoo` from <http://www.gap-system.org/Download/index.html>
2. Download `unzoo.exe` from <http://www.gap-system.org/Download/formats.html>
3. Move `gap4r4p6.zoo` and `unzoo.exe` to `C:\` (if you did not download the files directly to your C drive.)
4. Click on start menu, click on run, type in `C:\unzoo.exe -x gap4r4p6.zoo`, and press enter.

You will see several lines of code flashing on a screen for one to two minutes. This process will create a new folder (directory) named `gap4r4` on `C:\`. (You can look under your C drive to see this new folder.) Start GAP by clicking on the start menu, then run, then entering `C:\gap4r4\bin\gapw95.exe`

## 1.2 Installing Guava

The installation process assumes you are running GAP (the version downloaded above) on Microsoft Windows. For detailed installation instructions and support information on other operating systems, please visit

<http://cadigweb.ew.usna.edu/~wdj/gap/GUAVA/>.

To install Guava on Microsoft Windows, please follow the instructions below.

1. Towards the bottom of the webpage <http://cadigweb.ew.usna.edu/~wdj/gap/GUAVA/>, you will see a section called "Package Archives for Download." Download beta version GUAVA version 2.4 by clicking on the first link listed: `guava2.4.tar.gz` Save this file to the folder `C:\gap4r4\pkg`.
2. You now must unzip the `guava2.4.tar.gz` file into the folder `C:\gap4r4\pkg`. Since WinZip often will not work, we suggest downloading the free trial of WinRAR 3.50 from <http://www.rarlab.com/download.htm>. Download WinRAR 3.50 to your C drive. Then, right click on the `guava2.4.tar.gz` file inside the `C:\gap4r4\pkg` folder and extract it (using WinRAR) to the folder `C:\gap4r4\pkg`. This completes the installation of Guava.
3. To start Guava, first start GAP and type in the command below.

```
gap> LoadPackage("guava");
```

4. RECOMMENDED: To run **Guava** automatically on the startup of **GAP**

- (a) Open (using Notepad or equivalent) the file `PackageInfo.g` located in the folder `C:\gap4r4\pkg\guava2.4`. This will look ugly, but that is ok.
- (b) Search `PackageInfo.g` for the word `Autoload`, and in its last instance, change the value of `Autoload` from `Autoload := false` to `Autoload := true`.

### 1.3 Ready, Set, Go

Start **GAP** by running `C:\gap4r4\bin\gapw95.exe`. After **GAP** finishes initializing, you should see a **GAP** banner and the following prompt:

```
gap>
```

The prompt, `gap>` indicates that **GAP** is ready to receive commands. For example, we can use **GAP** to evaluate `1+1`.

```
gap>1+1;  
2
```

Notice that we use a semi-colon to end our command. Using two semi-colons will suppress the output.

The following tips will help you become comfortable using **GAP**:

1. **GAP** is case sensitive. This means that `a` is not the same as `A`.
2. Often, errors will cause **GAP** to go into a break loop. The prompt will change from `gap>` to `brk>`. To exit from the break loop, press `Control+D` on your keyboard or type `quit;` at the prompt.
3. We can also quit **GAP** by pressing `Control+D` or typing `quit;` at the prompt, and then closing the inactive window.
4. Since **GAP** is a interpreted language, when possible, built-in **GAP** commands should be used.
5. The up and down arrow keys on your keyboard can be used to navigate pervious commands.
6. Keep in the mind the famous quote, "I hate this 'orrible computer, I really ought to sell it: It never does what I want, but only what I tell it."

### 1.4 Using Help in GAP

**GAP** has a built-in help system that explains and provides usage syntaxes for common commands. To access the help system, enter `?`, a space, followed by the command name. For example, to get more information about the `print` command, enter the text below at the prompt.

```
gap> ? print
```

`? print` returns the help information about `print`. Often, the returned text spans over multiple screens. To navigate the returned text, press `space` to go to the next page, press `n` for the next line, `b` for the previous page, `p` for previous line, `q` to quit.

Sometimes, there are multiple entries for a topic. For example, enter the text below at the prompt.

```

gap> ? print
Help: several entries match this topic - type ?2 to get match [2]
[1] Tutorial: Print
[2] Reference: Print
[3] Tutorial: PrintTo
[4] Reference: Printing Presentations
[5] Reference: Printing Tables of Marks
[6] Reference: Printing Character Tables
[7] Reference: Printing Class Functions
[8] Reference: PrintObj!for tables of marks
[9] Reference: PrintObj!for character tables
[10] Reference: PrintObj!for character tables
[11] Reference: PrintObj
[12] Reference: PrintTo
[13] Reference: PrintTo!for streams
[14] Reference: PrintFormattingStatus
[15] Reference: PrintFactorsInt
[16] Reference: PrintArray
[17] Reference: PrintCharacterTable
[18] Reference: PrintAmbiguity
[19] New Features: PrintHashWithNames

```

If we want to select `Printing Tables of Marks`, we can simply type in `?5` for the 5th entry.

## 2 General Operations in GAP

### 2.1 Variables and Objects

Objects in `GAP` are considered anything that can be assigned to an variable. The assignment operator is `:=` (Note that it is proper form and sometimes necessary to have a a space before and after `:=`). For example, to assign 3 to a variable `a`, enter the following:

```

gap> a := 3;
3

```

We can also assign variables to variables. This will assign the object assigned to an variable to the other variable as well. For example:

```

gap> a := 3;
3
gap> b := 4;
4
gap> b := a;
3

```

The variable `b` and `a` are now both equivalent to 3.

**Type** refers to what set a variable belongs to. For example, 3 and 5 belong to the same type, since both are integers. However, a field element and a polynomial are not of the same type.

Notice that each line is followed by `;`, this tells `GAP` where a command ends. If a command does not have `;`, it will not evaluate the command. For example,

```
gap> 3+4
>;
```

The expression is calculated only after `;` is entered.  
Note that `a ;;` can be used to suppress output, For example,

```
gap> a := 3+4;;
```

The integer value 7 is still assigned to `a`. However, unlike the pervious example, 7 does not appear on screen.

## 2.2 Arithmetic Operators

Basic calculations can be done in `GAP`. When possible, objects in `GAP` are exact. This means that the fraction  $1/3$  is represented as  $0.3333\dots$  with infinite precision, not as  $0.33333$  or any other decimal with finite precision.

Standard operators, such as addition, `+`, subtraction, `-`, multiplication, `*`, and division, `/`, can be used. For example, to evaluate  $7+8/5+3$ , enter the following.

```
gap> 7+8/5+3;
58/5
```

Notice `GAP` knows the order of operation. Do note that implicit multiplication such as  $3(8)$  results in an error. We can use parentheses if we want `GAP` to change the order of evaluation. For example, if the first operation should be  $(7+8)$ , enter the following:

```
gap> (7+8)/5+3;
6
```

Modulo is an operation that is used very frequently in both cryptology and coding theory. The Modulus is defined as the remainder,  $r$ , when a dividend is divided,  $a$ , by a divisor,  $b$ , as in:  $a \bmod b = r$ . For example,  $3 \bmod 2 = 1$ ,  $-1 \bmod 5 = 4$ . The Modulus is always between 0 and the divisor. `GAP` can do this operation by using the `mod` operator. For example, to calculate  $19 \bmod 5$ , enter the following:

```
gap> 19 mod 5;
4
```

## 2.3 Comparison Operators

Objects and variables in `GAP` can be compared. Two objects can be compared by using a operator. For example, suppose we want to know if  $8/3$  is greater than 3 or not, enter the following.

```
gap> 8/3 > 3;
false
```

The return value, `false`, is a boolean value. Boolean values are either `true` or `false`. `GAP` will return `false` if the expression is false, and `true` otherwise. Here are some other comparison operators.

```
=    equal to
<    less than
>    greater than
<=   less or equal to
>=   greater or equal to
```

## 3 Data Structures

Data structures can be thought as a way of grouping objects together.

### 3.1 Lists

A list is one of the most common data structures. It is basically a collection of objects in order. Let's construct a list.

```
gap> a := [3, 4, 5];;
```

The order of the objects is indicated by the order which the objects appear. Hence, the first element of  $a$  is 3, the second element of  $a$  is 4, and the third element of  $a$  is 5.

Any object in list can be accessed or modified. For example, we can view what the first object of  $a$ .

```
gap> a[1];
3
```

We can also modify any objects in  $a$ .

```
gap> a[2] := 7;;
gap> a;
[ 3, 7, 5 ]
```

Often, we want to see if an object is in a list or not. GAP has a built-in membership test for lists, the `in` operator. For example:

```
gap> a := [1, 2, 3];
gap> 1 in a;
true
gap> 5 in a;
false
```

GAP also allows the user to append objects to the end of a list. For example, another object can be appended to our list,  $a$ .

```
gap> a := [1, 2, 3];
gap> Append(a, [3]);
gap> Append(a, [1 3 4]);
gap> a;
[1, 2, 3, 3, 1, 3, 4];
```

Notice multiple objects can be appended to a list at a time.

## 3.2 Vectors

Lists of integers and rationals are considered vectors. For example

```
v1 := [1, 2, 3];  
v2 := [1/2, 1/4, 1/5];
```

Lists of field elements over a finite field and lists of polynomials over a polynomial ring are also considered vectors.

Certain operations can be applied to a vector if the operation is defined for that vector. For example, for integer and rational vectors, several operations are defined.

vector $\pm$ vector	sum or difference of vectors
vector $\pm$ scalar	sum or difference of vector and scalar
vector * scalar	product of scalar and vector
vector * vector	dot product

Here are a couple examples:

```
gap> a := [1, 2, 3];;  
gap> b := [1/2, 1/3, 1/4];;  
gap> c := a+b;  
[ 3/2, 7/3, 13/4 ]  
gap> c := a + 7;  
[ 8, 9, 10 ]  
gap> c := a * 7;  
[ 7, 14, 21 ]  
gap> c := a*b;  
23/12
```

## 3.3 Range

Suppose we want to create a vector that goes from 1 to 100 incrementing by 1. Instead of typing the entire vector by hand, we can use range to create a vector from 1 to 100. For example, we can create an vector that goes from 1 to 100 incrementing by 1.

```
gap> t := [1 .. 100];  
[ 1 .. 100 ]  
gap> t[55];  
55  
gap> t[20];  
20
```

Range accepts other increments besides 1. To count up or down in other increments, the first element and second element of the range need to be defined. For example, suppose we want all odd numbers up to 27 that greater than 0. The first number is 1. Since the increment is 2, the second element is 3.

```
gap> t := [1, 3 .. 27];  
[ 1, 3 .. 27 ]  
gap> t[5];  
9
```

Negative increments are acceptable as well.

```
gap> t := [1, -1 .. -27];
[ 1, -1 .. -27 ]
gap> t[5];
-7
```

Note that the last element of the range must be an element in the range. For example, 26 is not an odd number. Thus, it can not be the last element of a range of odd numbers. The command `[1,3 .. 26]` will result in an error.

### 3.4 Matrices

Matrices are defined as lists of lists. For example, we can defined a  $3 \times 3$  identity matrix:

```
a := [[1, 0, 0], [0, 1, 0], [0, 0, 1]];
```

The matrix `a` is the same as:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that the elements in the matrix must be of the same type. In other words, a matrix should not have both polynomials and field elements.

Like vectors, certain operations can be applied to a matrix if the operation is defined for that type of matrix.

matrix $\pm$ matrix	sum or difference of matrices
matrix $\pm$ vector	sum and difference of matrix and vector
matrix $\pm$ scalar	sum and difference of matrix and scalar
matrix * scalar	product of scalar and matrix
matrix * vector	matrix multiplication
matrix * matrix	matrix multiplication

Here is an example:

```
gap> a := [[3, 0, 2], [1, 1, 0], [5, 0, 1]];;
gap> b := [[1, 1, 1], [2, 2, 2], [3, 3, 3]];;
gap> a+b;
[ [ 4, 1, 3 ], [ 3, 3, 2 ], [ 8, 3, 4 ] ]
gap> a-c;
[ [ -5, -9, -8 ], [ -7, -8, -10 ], [ -3, -9, -9 ] ]
gap> a*c;
[ 44, 17, 50 ]
```

### 3.5 Characters and Strings

Characters are the printed representation of letters, numbers, punctuation, spaces, and a variety of other markings. To store a character, surround the character with ' single quotes. To store the lowercase letter a into variable c1, enter:

```
gap> c1 := 'a';  
'a'
```

A string is simply a list of characters, and can be entered as such.

```
gap> s1 := ['a', 'b', 'c', 'd'];  
"abcd"
```

Alternatively, and much more easily, strings can be entered surrounded by " double quotes

```
gap> s2 := "efgh";  
"efgh"
```

Without regard for how the string was entered, individual characters can be accessed and modified as with lists.

```
gap> s2[3];  
'g'
```

```
gap> s2[3] := 'm';  
'm'
```

```
gap> s2;  
"efmh"
```

Strings and characters cannot undergo most operations, but have a special set of operations of their own, such as concatenation, and length functions that work on all lists. For instance, to concatenate two strings s1 and s2 from above, or to find the length of string s1, in characters,

```
gap> Concatenation(s1, s2);  
"abcdefmh"
```

```
gap> Length(s1);  
4
```

The characters ' , " , and \ are all problematic in **GAP** as they are used to designate strings and characters (and special characters). To use these characters, one must place a backslash \ immediately before the character. To store an apostrophe or single quote to variable c2, enter:

```
gap> c2 := '\'';  
,,
```

## 4 Scripts

There are other ways of doing calculations in **GAP**. Instead of typing commands line by line at the prompt, it is possible to execute a list of commands created in an external text editor.

## 4.1 Creating Scripts

To create a script, first we need to create a script file. One way to do this is to use a text editor such as notepad. Let's enter the commands below into notepad. Note that `RootInt(n,k)`, where `n` and `k` are integers, returns the `k`th root of `n`.

```
a := 3;
b := 4;
c := RootInt((a^2 + b^2), 2);
```

Save the file as `test.txt` in the folder where main program of GAP resides. The default directory path is `C:\gap4r4\bin`. We can run the script by calling 'test.txt' from the GAP prompt.

```
gap> Read("test.txt");
```

Notice that there is no output. However, the variable `c` now has a value.

```
gap> c;
5
```

## 4.2 Print

Often, we want the script to print out intermediate and final values. We can use `Print` to print values onto the screen. `Print` can print multiple variables and objects on one line as long as the variables and objects are separated by `,`. The command `\n` can be used to create a new line. Enter the text below in notepad.

```
a := 3;
Print("a is equal to: ", a, "\n");
b := 4;
Print("b is equal to: ", b, "\n");
c := RootInt((a^2 + b^2), 2);
Print("The root is equal to: ", c, "\n");
```

Save the file as "test.txt" and run the script in GAP.

```
gap> Read("test.txt");
a is equal to: 3
b is equal to: 4
The root is equal to: 5
```

## 4.3 Running Scripts from Other Directories

It is rather inconvenient to put all the work file in the `C:\gap4r4\bin`. Here is one solution to this problem in Microsoft Windows. Let's suppose there are a number of scripts in `C:\ECC` I want to run.

1. Create a shortcut to `C:\gap4r4\bin\gapw95.exe`.
2. Right click on the shortcut, and choose properties.
3. Change the text in the `start in` field to `C:\ECC`.

Now the command `Read` will read files from `C:\ECC`.

## 5 Functions

Often, different values need to be processed in similar ways. We have seen an example of a function, `RootInt(n,k)`. The variables `n`, and `k` are arguments of the function `RootInt`, the objects needed by the function. `RootInt(n,k)` returns the answer. The return value in this case, is the `k`th root of `n`.

We can construct our own functions. All functions have the same basic format.

```
name := function{arguments}
  function body;
end;
```

The keyword `function` declares "name" as a function object. The name of the function, 'name', can be replaced. Arguments should be variables needed by the function and should be separated by commas. The function body is the block of code that does a certain task. `End` indicates the the function declaration is over.

Let's write a function that will take two sides of a triangle, and return the hypotenuse. Enter the following function in notepad, or an equivalent plain text editor.

```
hypo := function(a, b)
  local c;
  c := RootInt((a^2 + b^2), 2);
  return c;
end;
```

Save the text file as `hypo.txt`. Note any new variable needs to be declared as a local variable before they can be used. `Return` returns the result to the main loop, since the value `c` only exists within the function. In other words, if we did not include the return statement, we will not get a value.

We can use the `read` command to load the function.

```
gap> read("hypo.txt");
gap> hypo(3, 4);
5
```

Notice the `GAP` functions are objects. Hence, functions can be used in place of objects. For example,

```
gap> a: = hypo(hypo(3, 4), 6);
7
```

You might have noticed that  $\sqrt{(5^2 + 6^2)} \neq 7$ . This is because `IntRoot` only returns an integer.

## 6 Conditionals

Often the behavior of a program needs to change on certain conditions. The command `if` evaluates an expression. If the expression is true, the corresponding block of code is evaluated. Otherwise, the block of code is skipped. For example, lets write a function that return true if the argument is 0. Enter the following in notepad.

```

checkzero := function(a)
  if a = 0 then
    return true;
  else
    return false;
  fi;
end;

```

Save the function as `checkzero.txt`. Load the function into GAP.

```

gap> Read("checkzero.txt");
gap> checkzero(1);
false
gap> checkzero(0);
true

```

The first statement `a = 0` check to see if  $a$  is equal to 0 or not, if  $a$  is equal to 0, then the following block of code is executed. In this case, the function will return true. Otherwise, it will go to the default case, which returns false. the statement `fi` marks the end of the conditionals.

It is possible to check for multiple conditionals. For example, the code below checks the sign of a variable.

```

sign_check := function(a)
  if a > 0 then
    return 1;
  elif a < 0 then
    return -1;
  else return 0;
  fi;
end;

```

Save the function as `sign_check.txt`. Load the function into GAP.

```

gap> read("sign_check.txt");
gap> sign_check(-3);
-1
gap> sign_check(3);
1
gap> sign_check(0);
0

```

In this case, if the function `a > 0`, then return 1, if that's not true, then it check to see if `a < 0`, then it return `-1`. Otherwise, it returns to the default case, which returns 0.

## 7 Loops

Loops execute a block of code multiple times.

### 7.1 For Loops

For loop is also an easy way of traversing a list or a vector. For example, let's suppose we want to add up all the elements in a list or a vector. Enter the following into notepad.

```

listsum := function(r)
  local s, e;
  s := 0;
  for e in r do
    s := s + e;
  od;
  return s;
end;

```

Save the function as "listsum.txt", load the function into GAP.

```

gap> Read("listsum.txt");
gap> listsum([1, 2, 3]);
6

```

The command `for e in r` takes elements of the list `r` one by one and assign the it to `e`. The command `do` marks the beginning of the statement, while `od` marks the end of the statement. The statements between `do` and `od` is executed each time the loop is run until there are no elements left in the list or the vector. In our example, each element in list `l` is assign to `e`, and is continuously added to `s`. When `e` cycles through all the elements in `r`, `s` is the sum of all the elements in `r`.

Instead of using a predetermined list, we can use create a range vector at the beginning of the loop. For example, if we want to create a loop that counts from 1 to 10. We can do the following.

```

gap> for e in [1..10] do
> Print(e, "\n");
> od;
1 2 3 4 5 6 7 8 9 10

```

## 7.2 While Loops

While loops executes a block of code until the conditional expression evaluates false. All while loops have the following syntax:

```

while condition do
  statements
od;

```

`do`, and `od` mark the start and end of the statement. The statement between `do` and `od` is evaluated until the condition evaluates to false. For example, we can write a function that counts to 10.

```

gap> n := 0;;
gap> while n < 10 do
> n := n + 1;
> od;
gap> n;
10

```

Only when `n` is greater than 10, will the loop end. Note the condition is check at the beginning of the loop.

## 8 Exercises

These exercises will utilize what you have learned thus far.

1. Write a function that will switch the objects assigned to two variables  $a$  and  $b$ .
2. Do the following calculations in **GAP**.

(a) 
$$\begin{bmatrix} 3 & 6 & 7 \\ 1 & 1 & 2 \\ 3 & 4 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \\ 7 & 1 & 2 \\ 2 & 1 & 1 \end{bmatrix}$$

(b)  $[1 \ 3 \ 4] + [1 \ 3 \ 4] * 3;$

(c) Add up all even numbers greater than 0 but smaller than 500.

3. Write a function that will return the dot product of two vectors.
4. Write a function that will return the result of  $a \bmod b$  without using built-in function `mod`.
5. Write a function that will calculate  $x!$ , given  $x$ .
6. \*Write a function that will calculate  $x!$ , given  $x$ , without using loops. Hint: a function can call itself.