Manual de GAP

0. Contenido

I. El sistema GAP

El desarrollo de GAP comenzó en Lehrstuhl D für Mathematik, RWTH-Aachen, bajo la dirección del profesor Joachim Neubüser en 1985. La versión 2.4 fue distribuida en 1988 y la 3.1 en 1992. La distribución completa de GAP 3, versión 3.4, se hizo en 1994. En 1997, el profesor Neubüser se retiró, y la coordinación del desarrollo de GAP, mucho más internacional que en un principio, fue asignada St Andrews. Se realizó un rediseño completo y se reescribió el sistema, y la versión 4.1, distribuida en julio de 1999, fue la primera versión del nuevo sistema, libre de restricciones de uso. La versión 4.2 fue distribuida en primavera de 2000 y la versión 4.3 ha sido distribuida en mayo de 2002.

El sistema GAP consiste en un sistema central y una serie de paquetes. El sistema central está compuesto de cuatro partes principales:

1. **Un núcleo**, escrito en C, cuyas funciones son:

    - Gestión automática de almacenamiento, transparente al usuario.

    - Un conjunto de funciones básicas como operaciones para enteros, campos finitos, permutaciones así como operaciones naturales sobre listas y registros.

    - Un intérprete del lenguaje GAP, que es un lenguaje imperativo no tipado con funciones como objetos de primera clase y algunos tipos de datos extra, como permutaciones y elementos de campos finitos. El lenguaje soporta una forma de programación orientada a objetos, similar a la que soportan lenguajes como C++ y Java aunque con importantes diferencias.

    - Un pequeño conjunto de funciones del sistema que permiten al programador de GAP manejar ficheros y ejecutar programas externos de un modo uniforme independientemente del sistema operativo que se use.

    - Un conjunto de herramientas para la prueba, el depurado y el cronometrado de algoritmos.

    - Una interfaz de usuario de tipo "lectura-evaluación-respuesta".

2. Una **librería de funciones GAP** mucho mayor que el núcleo, que implementa operaciones algebraicas y otras operaciones. Esta librería está escrita integramente escrita en el lenguaje de GAP. De este modo, el usuario puede facilmente investigar y moficar los algoritmos de la librería y añadir nuevas funciones al sistema.

3. **Librerías específicas** que permiten extender la funcionalidad del GAP, como GRAPE para la teoría de grafos, GUAVA para la teoría de codificación y un gran número de paquetes para la teoría de grupos.

4. **Documentación** disponible como ayuda en línea durante una sesión, así como en ficheros de diferentes formatos (HTML, pdf...).

Marcas en este manual

Las siguientes marcas en este manual pretenden resaltar las funciones que el lector necesitará para el manejo de GUAVA y la realización de las prácticas:

∏ - Capítulo básico para el manejo de GUAVA.
√ - Función necesaria para el manejo de GUAVA.

II. Una primera sesión con GAP.

1. El comienzo.

Al iniciar una sesión con GAP, se obtiene un prompt llamado *return key* o *newline key* (gap>). Para concluir una sesión se emplea el comando *quit;.* Para guardar una sesión en un fichero se debe escribir al principio de la misma *LogTo("logfile");*. La longitud de la línea de escritura se establece mediante el comando *SizeScreen([80,]);*, instrucción que establece la longitud de línea a 80 caracteres. De este modo, se sugiere que una sesión comience mediante las siguientes instrucciones:

```
gap> SizeScreen( [80, ] ); LogTo( "logfile1");
```

2. El bucle "Lectura Evaluación Respuesta"

La interacción del sistema GAP con el usuario es mediante un entorno de lectura-evaluación-respuesta, lo que quiere decir que a cada instrucción le corresponderá una respuesta de resultado por parte del sistema. Esta respuesta puede ser el resultado de la evaluación de una instrucción o un informe de error.
Ejemplo:

```
gap> ( 9 - 7);
2
gap> (9 - 7) * (5 + 6)
>;
22
gap> (9 - 7) * (5 + 6 ;
Syntax error: ) expected
(9 - 7) * (5 + 6;
             ^
```

Si se introuducen varias sentencias en una misma línea, se obtienen los resultados de la evaluación de cada uno de ellas en el mismo orden en que fueron escritas:

```
gap> -3; 17 - 23;
-3
-6
```

3. Constantes y operadores

- Operadores arimtéticos: + - * / mod ^
Ejemplo:

```
gap> 17 mod 3;
2
gap> (5 / 2)^4;
625/16
gap>
```

Nótese que gap calcula para cada fracción su equivalente irreducibles:

```
gap> 12345 / 25;
2469 / 5
```

- Operadores de comparación: =, <>, <, <=, >, >=

El resultado de emplear un operador de comparación es un valor booleano:

```
gap> 3^10 > 2^10;
true
```

- Operadores lógicos: not, and, or

Los operadores booleanos actúan sobre valores booleanos (true o false):

```
gap> true or (2 > 3);
true
```

- Permutaciones. Se escriben en notación cíclica y pueden ser multiplicadas:

```
gap> (1,2,3);
(1,2,3)
gap> (1,2,3) * (1,2);
(2,3)
```

La permutación inversa de (1,2,3) se denota mediante (1,2,3) ^ -1. Por otro lado el operador ^ se usa para determinar la imagen de un punto bajo una permutación y para conjugar una permutación con otra:

```
gap> (1,2,3) ^ -1;
(1,3,2)
gap> 2^(1,2,3);
3
```

```
gap> (1,2,3) ^ (1,2);
(1,3,2)
```

- Campos finitos como Z(8) y raíces complejas de unidad como E(4).

- Caracteres. Se encierran entre comillas simples:

```
gap> 'a';
'a'
```

Los caracteres pueden ser comparados mediante los comparadores de comparación.

```
gap> 'a' < 'b';
true
```

4. Variables y Objetos

Una variable se define como una secuencia de letras y dígitos que tienen asignado un valor. Para asignar un valor a una variable, se emplea el operador := del siguiente modo:

```
gap> v := 3;
3
```

Como se puede ver, el bucle lectura-evaluación-resultado devuelve el nuevo valor de la variable. De hecho, a partir de ahora escribir el nombre de la variable, es equivalente a escribir el valor, y se podrá manipular como si fuese dicho valor:

```
gap> v;
3
```

NOTA: En GAP, las variables no almacenan propiamente dicho su valor, sino que las variables emplean un puntero que apunta al objeto que realmente almacena el valor. Un objeto tiene tipo, y por ello un objeto entero, solo puede almacenar enteros. Sin embargo, las variables, que pueden apuntar a unos u otros objetos, no tienen tipo. De este modo, una variable entera puede pasar a ser booleana en la siguiente expresión:

```
gap> v := 3;
3
gap> v := true;
true
```

Existe un modo de interrumpir el normal funcionamiento del bucle lectura-evaluación-resultado, añadiendo ; al final de la expresión completa:

```
gap> v := 3 ;;
gap>
```

Los nombres de las variables deben incluir al menos una letra, y no pueden ser igual que ninguna palabra reservada del sistema, como por ejemplo `quit`. GAP distingue entre mayúsculas y minúsculas:

```
gap> Var1 := 1;
1
gap> var1 := 2;
2
gap> var1 = Var1;
false
```

GAP contiene un buffer en el que se almacena el valor de las últimas expresiones evaluadas. De este modo es posible recuperarlas mediante los comandos `last`, `last2` y *`last3`*.

```
gap> v := 1;
1

gap> 3 * 2;
6
gap> v := last;
6
gap> v := last3;
1
```

Para conocer las variables que el usuario a ido definiendo en una sesión, se emplea el comando `NamesUserGVars()`. También se pueden conocer todas las variables predefinidas en GAP mediante el comando `NamesGVars()`.

5. Objetos y elementos.

Como ya se ha dicho, en GAP una variable apunta a un objeto que contiene un valor. Dos objetos diferentes, poseen localizaciones en memoria diferentes aunque sus valores sean iguales. De este modo, en la siguiente secuencia de expresiones se obtienen dos variables de igual valor, a pesar de que los objetos a los que apuntan sean diferentes:

```
gap> a := 1;
```

```
1
gap> b := 1;
1
```

Cuando dos variables apuntan al mismo objeto, se dice que son iguales y que además son idénticas:

```
gap> a := 1;
1
gap> b := a;
1
```

Para comprobar si dos variables son idénticas se emplea el comando `IsIdenticalObj(var1, var2)`:

```
gap> a := 1;
1
gap> b := a;; IsIdenticalObj(a, b);
true
```

El operador = define una relación de equivalencia que comprende a todos los objetos de GAP. De este modo a = b cuando el valor de a y de b son iguales. Los componentes de las clases resultantes de la relación de equivalencia establecida por el operador =, son llamados elementos. De este modo, un mismo elemento puede estar representado por varios objetos de GAP.

6. Funciones

Una función en GAP es un programa escrito en el lenguaje de GAP. Las funciones son objetos especiales de GAP. Las funciones son aplicadas a objetos y normalmente devuelven un nuevo objeto dependiendo de la entrada. Sin embargo existen un tipo de funciones que aunque no devuelven ningún resultado producen efectos laterales que pueden ser de interés, como por ejemplo la función `Print`:

```
gap> Factorial(3);
6
gap> Print(last,"\n");
6
```

Nótese que en la segunda expresión no se devuelve ningún objeto, sino que simplemente se publica por pantalla el valor de la variable *last*.

NOTA: Algunas funciones pueden llegar a cambiar los objetos que toman como argumento como efecto lateral, como por ejemplo la función Sort. Sin embargo, esto no

es lo más común.
Para definir una función sencilla se emplea la siguiente sintaxis: nombre_función := objeto_entrada -> objeto_salida.

```
gap> cubo := x -> x^3;
function(x) ... end
gap> cubo(3);
27
```

Funciones con más de un argumento no pueden ser definidas de este modo. En ello entraremos más adelante.

7. Ayuda

Para obtener ayuda en línea se debe colocar el signo ? delante del tema a preguntar:

```
gap> ?sets
Help: several entries match this topic - type ?2 to get match [2]

[1] Tutorial: Sets
[2] Reference: Sets
[3] Reference: sets
[4] Reference: Sets of Subgroups
[5] Reference: setstabilizer
```

Bastará entonces con introducir el número del libro de ayuda deseado. Si en vez de ? se emplea ?? se listarán todas los libros donde aparece la palabra indicada.

III Listas y Registros

La matemática moderna, se basa en gran medida en la teoría de conjuntos. Los ordenadores almacenan los conjuntos en forma de listas. En concreto, GAP entiende los conjuntos como un tipo de listas, sin elementos vacíos ni duplicados y cuyas entradas se ordenan según la relación de precedencia <.

1. Listas planas

Una lista es una colección de objetos separados por comas y encerrados entre corchetes:

```
gap> primos := [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Para insertar un elemento al final de una lista se emplea `Add`:

```
gap> Add(primos, 31);
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

Para agregar una lista a otra se usa la función `Append`:

```
gap> Append(primos, [37, 41]);
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
```

Para obtener el i-ésimo elemento de una lista, se emplea la expresión `lista[i]`:

```
gap> primos[7];
17
```

El i-ésimo elemento de una lista se puede emplear como cualquier otro elemento, de modo que puede ser asignado a otra variable, empleado como argumento de una función o formar parte de una expresión como si fuese una variable independiente de su lista. En el sentido contrario, al elemento i-ésimo de una lista se le puede asignar un valor independientemente del resto de la lista. De hecho, esto último puede hacerse independientemente del tamaño de la lista, es decir que si la lista contiene 3 elementos, se puede introducir directamente un elemento en la posición 7 de la lista:

```
gap> l := [1,2,3];
[1,2,3]
gap> l[2] + 1;
```

```
3
gap> l[5] := 5;
5
gap> l;
[1, 2, 3, , 5]
```

Se puede conocer la longitud de una lista mediante la función `Length:`

```
gap> Length(l);
5
```

Una lista puede tener longitud 0, es decir, ser la lista vacía. Esta lista se consigue asignando la lista [ ] a una variable.
También es posible obtener la posición de un elemento en una lista, si es que se encuentra en ella:

```
gap> l := [1, 2 , 3];
[1, 2, 3]
gap> Position(l,3);
3
gap> Position(l,4);
fail
```

Para eliminar un elemento de una lista (eliminar la entrada de un elemento) se emplea la función `Unbind:`

```
gap> l := [1, 2, 3];
[1, 2, 3]
gap> Unbind( l[2] ); l;
[1, , 3]
```

"Aplanar" una lista consiste en poner todos los elementos de la misma (incluidos los que pertenezcan a sublistas) en el mismo nivel. Esto se consigue mediante la función `Flat( l ) :`

```
gap> Flat( [ 1, [ 2, 3 ], [ [ 1, 2 ], 3 ] ] );
[ 1, 2, 3, 1, 2, 3 ]
gap> Flat( [] );
[ ]
```

Así mismo, para invertir una lista, se emplea la función `Reverse( l ):`

```
gap> Reversed( [ 1, 4, 9, 5, 6, 7 ] );
[ 7, 6, 5, 9, 4, 1 ]
```

La concatenación de dos listas, es decir, la yuxtaposición de sus elementos, se realiza mediante la función `Concatenation( l1, l2 )` o `Concatenation( [l1,`

l2]) :

```
gap> Concatenation( [1, 2, 3], [4, 5] );
[ 1, 2, 3, 4, 5 ]
gap> Concatenation([[1,2,3], [4,5]]);
[ 1, 2, 3, 4, 5 ]
```

Los elementos de una lista se pueden ordenar, según el operador < o en base a un criterio definido por el usuario mediante una función, a través de la función `Sort( l [, f ])`.
La función *f* debe tener dos parámetros de entrada y devolverá una constante booleana:

```
gap> l := [ 2, 3, 1 ];; Sort( l ); l;
[ 1, 2, 3 ]
gap> Sort( l, function(v,w) return v*v < w*w; end); l;
[ 1, 2, 3 ]
```

Los elementos máximo y mínimo de una lista se pueden obtener con las funciones `Maximum( l )` y `Minimum( l )`:

```
gap> Maximum( [ -123, 700, 123, 0, -1000 ] );
700
gap> Minimum( [ -123, 700, 123, 0, -1000 ] );
-1000
```

Los objetos de una lista no tienen por qué ser del mismo tipo.

```
gap> l := [true, "Cadena de caracteres", , 3];
[true, "Cadena de caracteres", , 3]
```

Una lista puede contener otras listas como elementos. De hecho puede contenerse a sí misma. En tal caso, la lista pasa a ser cíclica, y su escritura sería infinita, por lo que se emplea el carácter ~ para indicar que en dicho lugar se encuentra lo que está actualmente escrito:

```
gap> l := [1, 2, 3];
[1, 2, 3]
gap> l[4] := [1, 2]; l[6] := "hola";
[1, 2]
"hola"
gap> l[5] := l;
[1, 2, 3, [1,2], ~, "hola"]
```

Las cadenas de caracteres son realmente listas de caracteres, de modo que GAP hace uso indistinto de ambas representaciones. De hecho si se introduce una lista de caracteres, se obtiene una cadena de caracteres:

```
gap> cad := [ 'h', 'o', 'l', 'a', '.'];
"hola."
```

Para extraer una sublista de una lista, se emplea el operador *lista{ posiciones }* del siguiente modo:

```
gap> l := [1, 4, 5, 4, 5];
[1, 2, 3, 4, 5]
gap> sl := l { [2,3] };
[4, 5]
```

Este operador permite también asignar varios elementos simultáneamente a una lista (insertar una sublista):

```
gap> l := [1, 2, 3, 4, 5];
[1, 2, 3, 4, 5]
gap> l { [2, 3] } := ["hola", 5];
["hola", 5]
gap> l;
[1, "hola", 5, 4, 5]
```

Por último, para obtener un elemento pseudo-aleatorio de una lista, se puede emplear la función `RandomList( l ):`

```
gap> l := [1, 2, 3, 4, 5];; RandomList( l ); RandomList( l );
5
4
```

2. Listas idénticas

En este punto volvemos a la relación de equivalencia que establece =. Dos listas son iguales cuando todos sus elementos son iguales, aunque los objetos a los que se refieren sean diferentes.

```
gap> l1 := [1, 2];
[1, 2]
gap> l2 := [1,2];
[1, 2]
gap> l1 = l2;
true;
```

En el ejemplo anterior, l1 y l2 apuntan a objetos diferentes. Sin embargo, la expresión *l2 := l1* hace que sí apunten al mismo elemento. En este caso son listas idénticas además de iguales. Con esto hay que tener cuidado, puesto cuando dos listas son idénticas, un cambio en una de ellas, afecta a la otra, ya que en realidad se trata del mismo objeto:

```
gap> l1 := [1, 2];
[1, 2]
gap> l2 := l1;
[1, 2]
gap> l1 = l2;
true
gap> l2[2] := 3;; l1 = l2;
true
```

Cuando se desea tener una copia de una lista, y no la propia lista (como ocurre en el ejemplo anterior), hay que emplear la función `ShallowCopy( lista )`*:*

```
gap> l1 := [1, 2];
[1, 2]
gap> l2 := ShallowCopy( l1 );; l1 = l2;
true
gap> l2[2] := 3;; l1 = l2;
false
```

3. Inmutabilidad

Para que no ocurran errores inesperados como el comentado en la sección anterior, se puede definir una lista como inmutable, lo que quiere decir que sus elementos no pueden cambiarse. En realidad, la lista no se hace inmutable, sino que se hace una copia recursiva de la misma (todos los elementos de la lista son copiados y no referenciados, y si existe un elemento de la lista que sea una lista, el procedimiento de copia es el mismo que con la lista principal). Una vez que se ha realizado la copia inmutable de la lista, existe la lista original (que sigue siendo mutable) y la lista inmutable (la copia). Para lleva esto a cabo se emplea la función `Immutable`*:*

```
gap> l := [1, 2];; lcopy := Immutable( l );
[1, 2]
gap> l[2] := 3;  lcopy[2] := 3;
3
Lists Assignment: <list> must be a mutable list
```

```
not in any function
```

Existe un modo de obtener una lista mutable a partir de una lista inmutable, esto es empleando la función anteriormente comentada `ShallowCopy`. Sin embargo esta función realiza solo una copia de la lista, y no de sus elementos, de modo que éstos son simplemente colocados en la nueva lista. Por ello, si la lista original (antes de crear la copia inmutable) era inmutable, entonces la lista resultante de aplicar `ShallowCopy` será mutable solo en el primer nivel, ya que los elementos de la lista segirán siendo inmutables.

```
gap> l := [1 , [2 , 3]];; cl := Immutable( Immutable( l ) );;
     lcl := ShallowCopy( cl );
[1, [2, 3]]
gap> lcl[3] := 3; lcl[2][2] := 4;
3
Lists Assignment: <list> must be a mutable list
not in any function
```

4. Conjuntos

En GAP, un conjunto es un tipo de lista que no posee "huecos" y cuyos elementos están estrictamente ordenados. Un conjunto no puede tener duplicados. En GAP, un conjunto puede denominarse como lista estrictamente ordenada y esta propiedad es probada mediante la función `IsSSortedList`. Un conjunto se construye a partir de una lista a la que se le aplica la función *Set:*

```
gap> nat3 := [1, 2, 3];
[1, 2, 3]
gap> IsSSortedList(nat3);
false
gap> nat3 := Set(nat3);; IsSSortedList(nat3);
true
```

La pertencia de un elemento a un conjunto se prueba mediante el operador *in:*

```
gap> 3 in nat3;
true
```

Para insertar un elemento en un conjunto se emplea la función `AddSet:`

```
gap> Add(nat3, 4);
gap> nat3;
[1, 2, 3, 4]
```

La intersección, la unión y la diferencia entre conjuntos se obtiene mediante las funciones `Intersection`, `Union` y `Difference` respectivamente:

```
gap> l:= Set([1, 2]);; l2:= Set([2,3]);; lu := Union(l, l2);
li := Intersection(l,l2); l   d := Difference(l,l2);
[1, 2, 3]
[2]
[1]
```

NOTA: Existen dos funciones, `IntersectSet` y `UniteSet` que aunque también realizan la intersección y la unión de los conjuntos, no devuelven ningún valor, y modifican el primer argumento asignándole el resultado.

---

5. Rangos

Un rango es una progresión aritmética finita de enteros. Es al igual que los conjuntos un tipo especial de lista. Para definir los elementos de esta lista, se emplea el operador .. :

```
gap> [1 .. 1000];
[1 .. 1000]
```

Un rango puede tener un valor de incremento diferente a 1 (como es el caso del ejemplo anterior). Para indicar el valor de incremento, se especifican los dos primeros elementos, cuya distancia será el valor de incremento, del siguiente modo:

```
gap> [1, 3 .. 999];
[1, 3 .. 999]
```

En el ejemplo anterior, el valor de incremento es 2.
Un rango, es reconocible como tal aunque no presente la representación compacta que se indica en los ejemplos anteriores, y por ello existe una función destinada a probar si una lista es un rango o no. Dicha función es `IsRange`. Así mismo, una lista que sea rango, puede ser expresada de modo compacto mediante la función `ConvertToRange`:

```
gap> a := [1, 2, 3, 4];; IsRange(a);
true
gap> ConvertToRange(a);; a;
[1 .. 4]
```

6. Bucles For y While

La sintaxis para for es *for valor in lista do cuerpo od;*

```
gap> pp := [1, 2, 3];;
gap> prod := 1;;
gap> for p in pp do prod := prod * p; od;
gap> prod;
6
```

La lista que se emplea para enumerar, pude ser un rango. De este modo, la sentencia anterior queda del siguiente modo:

```
gap> prod := 1;; for p in [1 .. 3] do prod := prod *p; od;
gap> prod;
6
```

La sintaxis del bucle while la siguiente: *while condición do cuerpo od;* donde condición debe ser una expresión booleana:

```
gap> res := 1;; num := 10;; while (num > 0) do
>       res := res * num; num := num - 1; od;
3628800
```

En el cuerpo de un bucle pueden haber cualquier tipo de sentencia como las que hemos visto anteriormente: operaciones aritméticas, sobre lista, sobre conjuntos...

7. Operaciones sobre listas

A las listas se le pueden aplicar funciones. Existen dos maneras de aplicarle una función a una lista. La primera es es aplicar una función sobre todos los elementos obteniendo un único resultado. La segunda aplicar la función a cada elemento, obteniendo una nueva lista donde cada elemento es el resultado. El primer caso es el que se produce al emplear una lista como argumento de una función como *Sum* o *Product*:

```
gap> Sum( [1, 2, 3] );
6
gap> Product( [1, 2, 4]);
8
```

El segundo caso se obtiene a través de cierto tipo de funciones que reciben la lista y la función como argumentos. Las funciones `List` y `Filtered` reciben como

argumentos una lista y una función, aunque su filosofía sea bien diferente:

```
gap> List ( [2 .. 10], x -> x^3 );
[ 8 , 27 , 64 , 125 , 216 , 343 , 512 , 729, 1000 ]
gap> Filtered ([2 .. 10], x -> x < 5);
[ 2 , 3 , 4 ]
```

En la función `Filtered` la función de filtrado, devuelve un valor booleano para un elemento de entrada. Dicha función puede estar predifinida:

```
gap> Filtered( [1..20], IsPrime);
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> Filtered( [ 1, 2, 3, 4, 5, 6, 7, 8 ],
> n -> n mod 4 = 0);
[0, 4, 8]
```

La función `Iterated`, aplica una función de un modo muy útil en ciertas ocasiones. Dada una lista [e1, e2, ..., en] y una función f, el resultado de aplicarles la función `Iterated` es f(... f (f (e1, e2), e3), ... , en):

```
gap> Iterated( [ 126, 66, 105 ], Gcd );
3
```

Otra forma de emplear las listas como argumentos es a través del operador { }, para obtener sublistas, lo cual fue anteriormente comentado:

```
gap> l := [1, 3, 6, 23, 12, 7];; l { [1 .. 4] };
[1, 3, 6, 23]
```

Para concluir, los cuantificadores universal `ForAll` y existencial `ForAny` toman como argumento una lista y una función booleana, y devuelven verdadero o falso en el caso de que se cumpla el predicado para cada elemento de la lista:

```
gap> ForAll( [1, 2, 3, 4], x -> x > 0);
true
gap> ForAny([1..20], IsPrime);
true
```

8. Vectores y matrices

Un vector es una lista densa (sin huecos) de elementos de un mismo 'campo'. Así mismo, una matriz es una lista de vectores de un mismo 'campo' y de igual longitud. Para comprobar si una lista es un vector, se puede emplear la función `IsRowVector( lista )`.

```
gap> IsRowVector([1, 2, 3]); IsRowVector([1, 'a', 3]);
true
false
```

Se pueden realizar operaciones entre vectores y escalares, o matrices y escalares, como la suma y el producto:

```
gap> [1, 2, 4/3] * 4;
[4, 8, 16/3]
gap> [[1,2], [3, 4]] * 2;
[[2, 4], [6, 8]]
```

También se pueden realizar multiplicaciones entre vectores y matrices:

```
gap> [ 1, 2, 3 ] * [[1, 2], [2, 3], [3, 4]];
[14, 20]
```

Para obtener una submatriz, se emplean dos operadores { } (recuérdese que una matriz es una lista) con una expresión como la siguiente:

```
gap> sm := m{ [1, 2] }{ [2, 3] };
[ [-1, 1], [0, -1] ]
```

Una matriz, puede multiplicarse escalarmente por un un campo finito. Un producto de este tipo da como resultado otra matriz cuyos enteros pertenecen al campo finito. Un campo finito se expresa como *`Z(número):`*

```
gap> [[1, 15], [9, 12]] * Z(2) + [[18, 10], [5, 9]] * Z(4);
[[ Z(2)^0, Z(2)^0], [Z(2^2)^2, Z(2^2)]]
```

Para realizar la transposición de una matriz mutable, se emplea la función `TransposedMat( m )`:

```
gap> TransposedMat([[1,2,3],[4,5,6],[7,8,9]]);
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
```

El uso de las matrices no se restringe al álgebra lineal, sino que también se emplean como elementos de grupos, siempre que sean invertibles. Se puede calcular el orden de un grupo por medio de la función *Order:*

```
gap> m := [[1, -1, 1], [2, 0 , -1],  [1, 1,  1]];;Order( m );
infinity
```

En matrices cuyas entradas son objetos más complejos, como por ejemplo funciones racionales, las funciones de cálculo de orden de GAP no son capaces de probar que la matriz tiene un orden infinito, y se obtiene un mensaje indicándolo (*Order: warning, order of <mat> might be infinite*).

Se puede probar que el orden de m es infinito por medio del mínimo polinomio de m sobre los racionales:

```
gap> Factors( MinimalPolynomial( Rationals, m) );
[ -2+x_1, 3+x_1^2 ]
```

El primer factor irreducible (x - 2) revela que 2 es un eigen valor de m, por lo que su orden no puede ser finito.

9. Registros planos

Los registros aportan otra forma de construir estructuras de datos. Al igual que una lista, un registro contiene subobjetos. Los objetos son llamados componentes o campos del registro, y no están indexados por su posición, sino por su nombre. Los registros cambian por medio de la asignación a sus componentes. Inicialmente un registro es definido como una lista de asignaciones separadas por comas:

```
gap> fecha := rec( anno := 1997, mes := "Jul", dia := 14);
rec( anno := 1997, mes := "Jul", dia := 14)
```

Los registros se pueden anidar unos dentro de otros:

```
gap> fecha := rec( anno := 1997, mes := "Jul", dia := 14,
> hora := rec(hora := 19, minuto := 23) );
rec( anno := 1997, mes := "Jul", dia := 14,
> hora := rec( hora := 19, minuto := 23))
```

Para conocer la estructura de un registro, se puede emplear la función `RecNames:`

```
gap> RecNames( fecha );
["anno", "mes", "dia", "hora"]
```

Para obtener el valor de un campo concreto de un registro, se utiliza la notación r*egistro.campo.* Así mismo se pueden realizar asignaciones sobre un campo concreto del registro:

```
gap> fecha.anno; fecha.anno := 1998;
1997
1998
gap> fecha.anno;
1998
```

IV. Funciones

En esta sección se va a mostrar como construir funciones. La estructura básica de una función en GAP es la siguiente:

function (argumentos) sentencias end

Tómese como primer ejemplo la función que imprime un pequeño texto:

```
gap> saludar := function ( )
> Print(" Hola mundo. \n ");
> end;
function ( ) ... end
```

El resultado de crear una nueva función muestra la parte más importante de la función, es decir, su lista de parámetros (que en este caso es vacía). Desde este momento, *saludar* es una función de GAP, de modo que su valor puede ser obtenido empleando la función `Print:`

```
gap> Print(saludar, "\n");
function ( )
     Print( "Hola mundo. \n ");
     return;
end
```

Para ejecutar la función saludar, solo hay que seguir el mismo procedimiento empleado hasta ahora:

```
gap> saludar( );
Hola mundo.
```

Se debe tener en cuenta que esta función no es un ejemplo correcto de devolución de algún valor, ya que lo que se produce es un efecto lateral provocado por la función `Print.`

2. Sentencia if

El sentido de la sentencia if en GAP no es diferente del que posee en otros lenguajes de programación, y por ello no se profundizará en su filosofía. Lo que sí posee cierta diferencia es su sintaxis, la cual puede ser comprendida mejor mediante un ejemplo que mediante una descripción formal:

```
gap> signo := function(n)
>    if n < 0 then return -1;
```

```
>    elif n = 0 then return 0;
>    else return 1;
>    fi;
> end;
function ( n ) ... end
```

La función *signo* devuelve el signo de un número. Como se puede comprobar, la sentencia if difiere tan solo en detalles con respecto a otros lenguajes, y llama especialmente la atención la partícula *elif*, habitualmente llamada *elsif*, y la cláusula *fi*.

La recursividad es un recurso soportado por GAP, y su modo de empleo es similar al de otros lenguajes. La clásica función de Fibonacci puede ilustrarlo de modo sencillo:

```
gap> fib := function ( n )
>    if n in [1, 2] then return 1;
>    else return fib(n-1) + fib(n-2);
>    fi;
> end;
```

3. Variables locales

En ciertas ocasiones, el nombre de las variables que se emplean en el interior de las funciones y el nombre de las variables globales pueden coincidir (ya sean predefinidas del sistema o definidas por el usuario). Para evitar esta situación no deseable se emplea la cláusula `local` delante de la variable que será local:

```
gap> mcd := function (a, b)
>    local c;
>    while b <> 0 do
>         c := b;
>         b := a mod b;
>         a := c;
>    od;
>    return c;
> end;
function ( a, b ) ... end
```

De este modo entra en juego el concepto de ámbito de la función (en inglés *scope*) que se trata de una parte léxica del código de un programa. Está el ámbito global, que acoge todo el código del programa y están los ámbitos locales que se localizan entre la palabra `function` y la palabra `end`. En un ámbito, cada identificador corresponde a una única variable de forma que cuando se crea un nuevo ámbito local, entran en juego las variables de la lista formal de argumentos de la función, y

las variables locales, de modo que el uso de un identificador en el código del programa se refiere a la variable en el ámbito más interno que tiene ese identificador como nombre.

4. Recursión.

En determinadas ocasiones la recursión por si misma no basta para resolver algunos problemas recursivos. Este es el caso del problema de la partición de un entero positivo, consistente en obtener una lista de números descendentes, cuya suma es el número dado. Así por ejemplo [4, 2, 1, 1] es la partición de 8. El número de particiones de n con elementos menores de m es la suma del número de particiones de n-i con elementos menores que i, donde i es menor que m y n. La función siguiente traduce a código dicha descripción:

```
gap> nrparts := function (n)
>    local np;
>    np:= funtion(n, m)
>         local i, res;
>         if n = 0 then return 1;
>         fi;
>         res := 0;
>         for i in [1 .. Minimum(n,m)]do
>             res:=res+np(n-i, i);
>         od;
>         return res;
>    end;
> end;
function ( n ) ... end
```

Obsérvese que se define una función local dentro de otra función. Esto es necesario puesto que partiendo del argumento n, es preciso considerar diferentes m. Lo principal de este ejemplo, es que los argumentos de la función np, son considerados por el sistema como variables locales.

V. Dominios

1. Introducción

En GAP un dominio es un conjunto estructurado. Así el anillo de los enteros Z[i] es un ejemplo de dominio. GAP predefine algunos dominios. Por ejemplo el anillo de los enteros gausianos está predefinido como *GaussianIntegers* y el campo de los racionales como *Rationals*. La mayoría de los dominios son construidos mediante funciones llamadas constructores de dominios. Así por ejemplo el campo finito de 16 elementos es construido por *GaloisField(16)*.

En general, los dominios pueden emplearse en dos sentidos. El primero de ellos es de forma directa, es decir, realizar cálculos sobre el dominio en sí, como por ejemplo calcular su tamaño. En tal caso existen funciones específicas que aceptan un dominio como argumento. Para el ejemplo empleado, la respuesta sería *Size(d)* donde *d* es el dominio a estudiar. El segundo sentido en que se puede emplear un dominio es para realizar operaciones y computaciones que ocurren dentro de ese dominio, como por ejemplo factorizar 10 en el anillo de los enteros gausianos. De modo que existen funciones por defecto actuarán sobre el anillo de los enteros, salvo que se les especifique otro. La respuesta a la pregunta formulada sería *Factors(GaussianIntegers, 10).*

2. Igualdad y comparación de dominios

Dos dominios son considerados iguales cuando los conjuntos de sus elementos computados por `AsSSortedList` son iguales. De este modo, = se comporta como si cada operando de un dominio fuera reemplazado por su elemento del conjunto. Sin embargo algunas veces, no siempre, = trabaja bien con dominios infinitos, para los que GAP no puede calcular el conjunto de sus elementos. Por diferentes razones, GAP no implementa el operador < para dominios.

3. Operaciones sobre dominios

En ciertas ocasiones, es útil trabajar con objetos que son similares a dominios pero que no son colecciones en el sentido de GAP, porque sus elementos pueden pertenecer a diferentes familias. Tales objetos son llamados dominios generalizados. Tal condición se revela mediante `IsGeneralizedDomain( d )`, `IsDomain( d )` o `IsCollection( d )`:

```
gap> IsDomain(GF(2^3));
true
```

Para obtener todos los elementos de un dominio, se emplea la función `GeneratorsOfDomain` o la función `Elements:`

```
gap> GeneratorsOfDomain(GF(2^3)); Elements(GF(2^3));
[0*Z(2),Z(2)^0,Z(2^3),Z(2^3)^2,Z(2^3)^3,Z(2^3)^4,Z(2^3)^5,Z(2^3)^6]
[0*Z(2),Z(2)^0,Z(2^3),Z(2^3)^2,Z(2^3)^3,Z(2^3)^4,Z(2^3)^5,Z(2^3)^6]
```

Un dominio puede ser definido mediante la lista de elementos que lo definen mediante la función `Domain`:

```
gap>D:=Domain(GeneratorsOfDomain(GF(2^3)));
<object>
gap>GeneratorsOfDomain(D);
[0*Z(2),Z(2)^0,Z(2^3),Z(2^3)^2,Z(2^3)^3,Z(2^3)^4,
Z(2^3)^5,Z(2^3)^6]
gap> D2 := Domain( [1, 2, 3, 4]); GeneratorsOfDomain(D2);
<object>
[ 1, 2, 3, 4 ]
```

Dado un dominio D, su característica se define, cuando D es cerrado sobre la suma y tiene un elemento cero (`Zero(D)`), como el entero positivo menor p, tal que `p * x = Zero(D)` para todos los elementos x en D, o el entero 0 en otro caso. La característica de un dominio se calcula mediante la función `Characteristic( d )`. Nótese que d puede ser un dominio o un elemento aditivo:

```
gap> Characteristic( Elements(GF(2^3)) );
2
```

El tamaño de un dominio se puede calcular mediante la función *Size*:

```
gap> Size( [1, 2, 3] ); Size( GF(2^3)); Size( Rationals);
3
8
infinity
```

Dada la naturaleza de los dominios en GAP, comentada anteriormente, se pueden emplear las operaciones introducidas anteriormente para conjuntos, es decir, la unión, la intersección y la diferencia.

```
gap> Difference( Domain( [0, 1, 2, 3] ), Domain( [3, 4,
5] ) );
[ 0, 1, 2 ]
```

Por último se mencionará una función que puede ser muy útil, llamada `Random`. Esta función devuelve un elemento aleatorio perteneciente al dominio especificado:

```
     gap> Random( Domain( [ 0, 1, 2, 3] )); Random( Domain( [ 0,
1, 2, 3] ));
     3
     2
```

VI. Campos finitos ($\prod$)

1 Introducción

Los campos finitos se construyen a partir de un elemento generador, llamado también característica del campo. Los elementos de un campo finito se obtiene mediante las potencias de su generador p, y se dice que el campo tiene un tamaño p^n para algún entero n.

En GAP, se puede saber si un elemento pertenece o no a un grupo finito mediante la función *IsFFE*:

```
gap> IsFEE(Z(2^4));
true
```

Todos los elementos de campos finitos de las misma característica forman una familia en GAP. Para generar elementos de un campo finito se puede usar la función *Z*. De este modo la llamada `Z(p^d)` devuelve el generador del grupo multiplicativo del campo finito con p^d elementos (donde p debe ser un número primo, y por limitaciones del sistema p^d menor o igual a 65536). El elemento devuelto por *Z*, es el generador del grupo multiplicativo del campo finito con p^d elementos, que es cíclico. Todo elemento distinto de cero en el campo finito con p^d elementos puede ser escrito en GAP como `Z(p^d)^i`. Obsérvese que esta es también la forma en que GAP devuelve estos elementos.

```
gap> r := Z(2^5); r^3;
Z(2^5)
Z(2^5)^3
```

El elemento neutro para la suma se representa como `0*Z(p)`. Este valor es diferente del entero 0. El elemento neutro para el producto se representa como `Z(p)^0`. Nótese que este valor tampoco es exactamente igual que el entero 1. Ambas diferencias pueden comprobarse mediante las funciones `IsInt` y `IsFFE`:

```
gap> IsInt(0); IsInt(0*Z(2)); IsFFE(0); IsFFE(0*Z(2));
true
false
false
true
gap> IsInt(1); IsInt(Z(2)^0); IsFFE(1); IsFFE(Z(2)^0);
true
false
false
true
```

Los elementos de un campo finito se pueden comparar mediante los operadores = y <. De este modo *a* = *b* devuelve true si y solo si los elementos de campo finito a y b son iguales. *a* < *b* prueba cual de los dos elementos es menor. Lo elementos de los campos finitos se ordenan del siguiente modo: si dos elementos pertenecen a campos finitos diferentes, el que pertenece al campo finito con menor característica es menor. Si los dos elementos pertenecen a diferentes campos de la misma característica, el que pertenece al menor campo es menor. Si los dos elementos pertenecen al mismo campo y uno es cero y el otro no, el cero es el menor elemento. Si los dos elementos pertenecen al mismo campo y los dos son diferentes de cero, y se representan como $Z(p^d)^{i1}$ y $Z(p^d)^{i2}$, y se cumple que i1 < i2, entonces el primer elemento es menor que el segundo.

```
gap> Z(16)^10 = Z(4)^2;
true
gap> Z(256) > Z(101);
false
gap> 0 < 0*Z(101);
true
```

2. Operaciones sobre campos finitos

Partiendo de un elemento de un campo finito, se puede obtener la característica del campo finito, la unidad y el cero, el inverso y el orden, mediante las funciones *Characteristic, One, Zero, Inverse* y *AditiveInverse*, y *Order*:

```
gap> Zero(Z(9));
0*Z(3)
gap> Order(Z(9));
8
gap> Characteristic(Z(2^3));
2
gap> Z(2^3)^2 * Inverse(Z(2^3)^2);
Z(2)^0
gap> AdditiveInverse(Z(2^3)^3);
Z(2^3)^3
Gap> Z(2^3)^3+AdditiveInverse(Z(2^3)^3);
0*Z(2)
```

La función *DegreeFFE* devuelve el grado de la campo finito menor que contiene al elemento en cuestión. El argumento de esta función puede ser un elemento de un campo finito, un vector o una matriz sobre un campo finito:

```
gap> DegreeFFE( Z(16)^10);
```

```
2
gap> DegreeFFE(Z(16)^11);
4
gap> DegreeFFE( [ Z(2^13), Z(2^10) ] );
130
```

La función `LogFEE` devuelve el logaritmo discreto del elemento en cuestión en un campo finito respecto a la raiz r. Nótese que no tiene solución (y por tanto genera un error) si el elemento es cero o no es una potencia de r:

```
gap> LogFFE( Z(409)^116, Z(409));
116
gap> LogFFE(Z(2^3)^5,Z(8)^2);
-15
```

La función `IntFFE` devuelve el entero correspondiente al elemento en cuestión, que debe pertenecer a un campo finito. Esto es, dado z `IntFFE` devuelve el menor entero i no negativo tal que `i * One(z) = z`. La correspondencia entre elementos de un campo finito de característica p y los enteros entre 0 y p-1 se define eligiendo z(p) como elemento correspondiente a la raíz primitiva más pequeña módulo p:

```
gap> IntFFE( Z(409) );
2
```

3. Creando campos finitos

Partiendo de un conjunto de elementos es posible determinar el campo finito mínimo que contiene esos elementos mediante la función `DefaultField`:

```
gap> DefaultField( [ Z(4), Z(4)^2 ] );
GF(2^2)
gap> DefaultField( [ Z(4), Z(8) ] );
GF(2^6)
```

La función `GaloisField` (en modo abreviado *GF*) devuelve un campo finito. Toma como argumentos dos enteros p y d, que se pueden expresar de dos modos diferentes: `GaloisField(p,d)` o `GaloisField(p^q)`:

```
gap> f1 := GF(2^4); Size(f1);
GF(2^4)
16
```

Para saber si un elemento es un campo finito (es decir que se construyó mediante `GaloisField`) se puede emplear la función `IsFiniteField` que devuelve true

en tal caso y false en caso contrario.

Para generar un campo de característica p y que tiene un polinomio irreductible f(x) se escribirá GF(p,L(f)) donde L(f) es una lista con los coeficientes de f(x). Por ejemplo, para generar un campo con característica 3 y que tiene el polinomio irreductible $2 + x + x^2$ debe escribirse lo siguiente:

```
gap> S:= GF(3,[ Z(3), Z(3)^0, Z(3)^0 ]);
GF(3^2)
```

Para obtener el polinomio mínimo de un elemento sobre un campo finito, se emplea la función MinimalPolynomial, en el que se indica el campo y el elemento:

```
gap> MinimalPolinomial(GF(5^2),Z(5^2));
Z(5^2)^13+x_1
```

VII. Anillos

1. Introducción

Un anillo es un grupo aditivo cerrado para la multiplicación. Cuando en este tipo de dominio la multiplicación y la suma son distributivas, se le llama anillo en GAP. Cada anillo con división, campo o álgebra es un anillo. Ejemplos de ello son los enteros y los anillos de las matrices. Ciertas funciones para elementos de anillos, como `IsPrime` o `Factors`, se definen para un anillo concreto que se puede introducir como parámetro opcional. En caso contrario se define un *anillo por defecto* formado por los elementos de anillo usados como argumentos.

2. Creando anillos

Un anillo en GAP es un grupo aditivo que también es magma (se llama así a los dominios con multiplicación no necesariamente asociativa). Para verificar tal condición se emplea la función `IsRing` :

```
gap> IsRing( GF(2^3) );
true
```

Para crear un anillo a partir de sus elementos se emplea la función `Ring`. Esta función se puede aplicar de dos modos: la primera es especificando una serie de elementos, y en tal caso la función `Ring` devolverá el anillo menor que incluya a todos esos elementos. La segunda forma es especificando una colección de elementos, en cuyo caso la función `Ring` devolverá el menor anillo que contenga dicha colección de elementos. Los anillos de polinomios, se pueden crear mediante la función PolynomialRing(F), con F el campo sobre el que se construye:

```
gap> Ring(2, 3); Ring(2, E(4)); Ring(GF(2^3));
Integers
<ring with 2 generators>
<ring with 1 generators>
gap> PolynomialRing(GF(5^2));
PolynomialRing(…, [x_1])
```

La función `IsPrime` indica si un elemento es primo dentro de un anillo concreto (aunque si éste no se indica, se utiliza el anillo por defecto):

```
gap> IsPrime(3,Integers);
true
```

Para factorizar en un anillo, se emplea la función `Factors`:

```
gap> Factors(10);
[ 2, 5 ]
gap> Factors(GaussianIntegers,10);
[ -1 -E(4), 1+ E(4), 1+2*E(4), 2+ E(4) ]
```

Para verificar que un anillo es conmutativo (y en general aplicable para todo tipo de magmas) se emplea la función `IsCommutative`:

```
gap> IsCommutative( GaussianIntegers );
true
```

Un anillo R se llama euclidiano cuando es un anillo integral y existe una función *delta*, llamada grado euclidiano, de R+ en Z+, tal que cada par r y s, donde r pertenece a R y s pertenece a R+, existe un elemento q tal que r - qs = 0(R) ó delta(r - qs) < delta(s). En GAP el grado euclidiano delta está implícito a un anillo y no se puede cambiar. La existencia de esta división con resíduo implica que el algoritmo de Euclides puede ser aplicado para computar el máximo común divisor de dos elementos, lo implica a su vez que R es un anillo de factorización única. Para verificar esta condición, se emplea la función `IsEuclideanRing` :

```
gap> IsEuclideanRing( GaussianIntegers );
true
```

Para calcular el grado euclidiano de un elemento de un anillo R, se emplea la función `EuclideanDegree`, en la cual se puede especificar un anillo concreto :

```
gap> EuclideanDegree( GaussianIntegers, 3 );
EuclideanDegree( 3 );
9
3
```

Por otro lado, para calcular el cociente euclidiano de dos elementos r y m de un anillo, se emplea la función `EuclideanQuotient`, en la cual puede también especificar un anillo concreto (que tambien puede ser el anillo de los polinomios):

```
gap> EuclideanQuotient( 8, 3);
2
gap> EuclideanQuotient( GaussianIntegers, 8, 3 );
3
```

Por último (en referencia a la condición de anillo euclidiano) se puede obtener el resto de un elemento r módulo m de un anillo, se emplea la función `EuclideanRemainder`, la que al igual que las anteriores acepta como argumento un anillo concreto :

```
gap> EuclidenaRemainder(8, 3);
2
```

```
gap> EuclideanRemainder( GaussianIntegers, 8, 3 );
     -1
```

La función *QuotientRemainder* devuelve simultáneamente tanto el cociente como el resto de dos elementos r y m de un anillo :

```
gap> QuotientRemainder( 8, 3);
[ 2, 2 ]
gap> QuotienteRemainder( GaussianIntegers, 8, 3);
[ 3, -1 ]
```

El máximo común múltiplo (LCM) y el mínimo común divisor (GCD) se puede obtener mediante las funciones `Lcm` y `Gcd` respectivamente. Ambas funciones pueden responder frente a dos formatos de argumentos. La primera es una secuencia de elementos (r1, r2, r3...) y la segunda es una lista de elementos ([r1, r2, r3, ...]). También es preciso decir que ambas funciones admiten como argumento opcional un anillo concreto sobre el que se realizarán las computaciones, que en caso de no estar presente se empleará el anillo por defecto (que es el anillo mínimo que contiene los elementos indicados):

```
gap> Lcm( 30, 55, 81); Lcm( [30, 55, 81] );
8910
8910
gap> Lcm(GaussianIntegers, 30, 55, 81);
8910
gap> Lcm(GaussianIntegers,[30, 55, 81]);
8910
gap> Gcd(30, 55, 40 + E(4)); Gcd([30, 55, 40 + E(4));
1
1
gap> Gcd(GaussianIntegers, 30, 55, 40 + E(4));
1
gap> Gcd(GaussianIntegers, [30, 55, 40 + E(4));
1
```

La representación g del máximo común divisor de los elementos r1, r2, ... del anillo R, es una lista de elementos s1, s2, ... tal que g = s1r1 + s2r2 + ... De este modo la representación del máximo común divisor de 30 y 50 es [2, -1] ya que 2*30 - 1*50 = 10. La representación del mcd se obtiene mediante la función `GcdRepresentation` :

```
gap> GcdRepresentation( 30, 50 );
[ 2, -1 ]
gap> GcdRepresentation( 30 + E(4), 50 + E(4) );
[ 25 - 2*E(4), -15 + E(4) ]
```

VIII. Enteros

El anillo de los enteros está representado por la variable *Integers*. Los semianillos

de los enteros positivos y los enteros no negativos están representados por `PositiveIntegers` y `NonnegativeIntegers` respectivamente:

```
gap> Size( Integers ); 2 in Integers;
infinity
true
```

Dado un número se puede determinar si se trata de un entero o no mediante la función *IsInt* :

```
gap> x:=3; IsInt( x );
true
```

Así mismo se puede determinar si un número es un entero positivo mediante la función `IsPosInt`.
La función *Int* devuelve un entero i cuyo valor depende del tipo del elemento pasado como parámetro. Si el elemento es un racional entonces el entero es la parte entera del cociente del numerador y el denominador. Si se trata de un elemento e de un campo finito primo entonces el resultado es el entero no negativo más pequeño tal que `e = i * One( e )`. Si el elemento es una cadena cuyo significado es un número entero entonces el resultado es el entero correspondiente a dicha cadena (nótese que la operación inversa, de entero a cadena, se consigue mediante la fucnón *String*) :

```
gap> Int( 4/3 ); Int( -2/3 );
1
0

gap> int := Int( Z(5) ); int * One( Z(5) );
2
Z(5)
gap> Int( "12345" ); Int( "-27" ); Int( "-27/3" );
12345
-27
fail
```

Para obtener el valor absoluto de un entero n, se emplea la función `AbsInt` :

```
gap> AbsInt( -23 ); AbsInt( 23 );
23
23
```

El signo de un entero se obtiene con la función `SignInt` :

```
gap> SignInt( 33 ); SignInt( -33 );
1
-1
```

`LogInt` *`(n,base)`* devuelve la parte entera del logaritmo del entero positivo n con respecto a la base especificada :

```
gap> LogInt( 1030, 2 );
10
gap> LogInt( 1, 10 );
0
```

`RootInt (n, k)` devuelve la parte entera de la raíz k-ésima del entero n. Si no se especifica el argumento k se toma como defecto el valor 2 :

```
gap> RootInt( 17000, 5 );
7
gap> RootInt( 361 );
19
```

Por otro lado la función `SmallestRootInt` devuelve la menor raíz del entero n, es decir el entero r de menor valor absoluto para el que existe un entero positivo tal que n = r^k

```
gap> SmallestRootInt( 2^30 );
2
gap> SmallestRootInt( -(2^30) );
-4
```

La función `Random( Integers )` devuelve enteros pseudoaleatorios entre -10 y 10 distribuidos de acuerdo a una distribución binomial:

```
gap> Random( Integers ); Random( Integers );
1
-4
```

Para obtener cociente y resto entero de una división, se emplean las funciones `QuoInt( n, m )` y `RemInt( n, m )` respectivamente :

```
gap> 2 * QuoInt( 7, 2 ) + RemInt( 7, 2 );
7
```

`ChineseRem( modul, res)` devuelve la combinación de *res* módulo *modul*, es decir el único entero *c* entre *[0 .. Lcm(modul) -1]* tal que *c = res[i] módulo modul[i]* para todo i, si existe. En caso de que no exista tal combinación se señala un error. Dicha combinación existe si y solo si *res[i] = res[k] mod Gcd(modul[i], modul[k])* para

cada par i y k:

```
gap> ChineseRem( [ 1, 2, 3, 5, 7 ], [ 1, 2, 3, 4 ] );
53
gap> ChineseRem( [ 6, 10, 14 ], [ 1, 3, 5 ] );
103
```

La lista *Primes* es un lista estrictamente ordenada de los 168 primos menores que 1000, de modo que `Primes[1] = 2` y `Primes[168] =997`. Para confirmar si un número es o no primo, se emplea la función `IsPrimeInt.` Por convención, se asume que 0 y 1 no son primos. También es preciso mencionar el hecho de que la respuesta de `IsPrimeInt` a un número mayor a 10^{13} solo responde a un "test de primalidad probable". Por último, es preciso mencionar que la función `IsPrimeInt` es un método para la función general `IsPrime` :

```
gap> IsPrimeInt(Primes[23]);
true
```

La función `FactorsInt` devuelve una lista de factores primos de un entero. Esta función emplea el test de primalidad probable comentado anteriormente:

```
gap> FactorsInt( 10^41 -1);
[ 29,101,281,9901,226549,121499449,4458192223320340849 ]
```

Por último, la función `DivisorsInt` devuelve una lista con los enteros divisores de n. La lista está ordenada, de modo que empieza con 1 y termina con el entero en cuestión. Los divisores de -n se definen igual que los de n:

```
gap> DivisorsInt( 1 );DivisorsInt( 20 ); DivisorsInt( 541 );
[ 1 ]
[ 1, 2, 4, 5, 10, 20 ]
[ 1, 541 ]
```

IX. Racionales

El anillo de los racionales está representado por la variable `Rationals`. Sobre los números racionales se introducirán sólamente 4 funciones: `Rat`, `IsRat`, `NumeratorRat` y `DenominatorRat` :

La función `Rat` devuelve un número racional en función del significado de la cadena de caracteres que se le pasa como argumento de modo que devuelve su racional equivalente reducido.

```
gap> Rat("3.14159");
314159/100000
gap> Rat("35/14");
5/2
```

La función `IsRat` devuelve true en el caso de que un número sea racional y false en caso contario.

```
gap> IsRat( 3 );
true
gap> IsRat( IsRat );
false
```

Las funciones `NumeratorRat` y `DenominatorRat` devuelven el numerador y denominador del racional que se le pasa como argumento:

```
gap> NumeratorRat( 2/3 ); NumeratorRat(17/-13);
2
-17
gap> DenominatorRat(2/3); DenominatorRat(17/ -13);
3
13
```

Por último, se mencionará la función `Random` para enteros que es equivalente al número racional compuesto de dos números enteros pseudo aleatorios como numerador y denominador.

X. Polinomios ($\prod$)

En esta brevísima sección se introducirán las funciones que se emplearán en GUAVA. GAP cubre extensamente la temática de los polinomios, sin embargo este manual no ha sido concebido como una referencia de GAP, sino como una introducción a las herramientas de GAP que se emplean en GUAVA. Aun así, algunas cuestiones sobre polinomios e incluso sobre temas anteriores serán ampliadas durante el desarrollo del manual de GUAVA.

En versiones anteriores a GAP 4, los polinomios se definian sobre campos. De este modo, un polinomio sobre GF(3) era diferente de un polinomio sobre GF(9). En la versión GAP 4, un polinomio se define sobre una característica (de un campo) de modo que se puden hacer cosas como multiplicar un polinomio sobre GF(3) con un polinomio sobre GF(9) sin necesidad de hacer conversiones al campo mayor.

Para definir un polinomio se debe emplear la función `UnivariatePolynomial`, en la que se indica el anillo sobre el que se define el polinomio y sus coeficientes:

```
gap> p := UnivariatePolynomial(Rationals,[1, 2, 3, 4]);
1+2*x+3*x^2+4*x^3
gap>p2:=UnivariatePolynomial(GF(3),Z(3)^0 * [1, 0, 1, 2] );
Z(3)^0 + x_1^2 -x_1^3
```

Del mismo modo, se puede obtener la lista de los coeficientes de un polinomio mediante la función `PolynomialCoefficientsOfPolynomial` :

```
gap> PolynomialCoefficientsOfPolynomial( p );
[1, 2, 3, 4]
```

Empleando la función `Indeterminate( R )(`√`)`, se establece una variable indeterminada sobre el anillo *R.* De este modo se pueden definir polinomios empelando una o varias variables indeterminadas que se se ubiquen en un anillo concreto:

```
gap> x := Indeterminate( GF(3)); Pol := x^2 + 1;
x_1
gap> P := x^2 + 1;
Z(3)^0+x_1^2
```

Dada la filosofía en que GAP gestiona variables y objetos, una variable indeterminada `x` solo es un “puntero” al auténtico objeto indeterminado. De este modo, mediante la función `Indeterminate` indicamos que el valor de `x`, no es un entero, ni un racional, etc., sino que no se conoce su valor. La función Indeterminate devolverá un objeto indeterminado que se identifica por el primer nombre del tipo `x_i`, que esté libre, donde `i` por defecto será 1, a no ser que se indique lo contrario. Esto se puede hacer añadiendo un argumento entero a `Indeterminate` con el valor de i:

```
gap> x := Indeterminate( GF(3), 2);
x_2
```

También es posible asignar un nombre a las variables indeterminadas sobre un determinado anillo, de modo que todo objeto indeterminado sobre dicho anillo será identificado por el símbolo especificado (`ind` en el siguiente ejemplo):

```
gap> Indeterminate( GF(2), ”ind”);
ind
gap> x := Indeterminate( GF(2));
ind
gap> y := Indeterminate( GF(2));
ind
```

En cualquier caso, siempre es posible cambiar el nombre que identifica a un objeto indeterminado por medio de la función `SetName`:

```
gap> a := Indeterminate(GF(3),5);
x_5
gap> a + a^5;
x_5 + x_5^5
gap> SetName(a,”ah”);
gap> a + a^5;
ah + ah^5
```

La función `IsPolynomial` indica si una función es o no un polinomio. Se define polinomio como una función racional cuyo denominador es 1:

```
gap> IsPolynomial((x*y+x^2*y^3)/y);
trae
gap> IsPolynomial((x*y+x^2)/y);
false
```

La función `Degree` devuelve el grado de una variable o de un polinomio:

```
gap> x := Indeterminate(GF(3));
x_1
gap> y := x^3;
x_1^3
gap> Degree(y);
3
gap> p := y^2 + 1;; Degree(p);
6
```

Es posible evaluar un polinomio mediante la función `Value` especificando un polinomio, e indicando las variables que se desean evaluar y sus valores:

```
gap> x := Indeterminate(Integers,1);
x_1
gap> y := Indeterminate(Integers,2);
x_2
gap> z := Indeterminate(Integers,3);
x_3
gap> p:= x*y*z + x*y + z + 1;
1 + x_3 + x_1*x_2 + x_1*x_2*x_3
gap> p2 := Value(p,[x,y,z],[2,3,4]);
35
```

Las operaciones, +, -, * y / son aplicables a los polinomios:

```
gap> p := x*y*z + x*y + z + 1;;
gap> p2 := x*y + 1;
gap> p + p2;
2+x_3+2*x_1*x_2+x_1*x_2*x_3
gap> p * p2;
1+x_3+2*x_1*x_2+2*x_1*x_2*x_3+x_1^2*x_2^2+x_1^2*x_2^2*x_3
gap> p / p2;
1 + x_3
```

La factorización de un polinomio de una variable se puede realizar por medio de la función `Factors(√)`, que devuelve los factores irreducibles del mencionado polinomio:

```
gap> p := x + x^2 + 2*x^3;; Factors(p);
[2*x_1,1/2+1/2*x_1+x_1^2]
```

Para obtener la derivada parcial de un polinomio se puede emplear la función `Derivative( pol, ind )`, donde *pol* es el polinomio en cuestión e *ind* es el índice de la variable a derivar, que por defecto es 1. En caso de que solo exista una variable no es necesario especificar *ind*:

```
gap> x := Indeterminate(Integers);; P:=x^2 -1;
-1+x_1^2
gap> Derivative(P);
2*x_1
```

Las raíces de un polinomio se pueden obtener mediante la función `RootsOfUPol(√)`:

```
gap> RootsOfUPol(x^2 – 1);
[ 1, -1 ]
```

Dado un polinomio $P = a_0+a_1X+...+a_nX^n$, el polinomio recíproco es $P' = a_n+a_{n-1}X+ ...+a_0X^n$. Éste se puede obtener empleando la función

`ReciprocalPolynomial(P[,n])` . En esta función, si se especifica *n*, se indica que grado debe tener el polinomio resultante, de modo que se multiplica por x^k para que alcance dicho grado:

```
gap> P := UnivariatePolynomial( GF(3),Z(3)^0*[1,0,1,2] );
Z(3)^0+x_1^2-x_1^3
gap> RecP := ReciprocalPolynomial( P );
-Z(3)^0+x_1+x_1^3
gap> ReciprocalPolynomial( RecP ) = P;
true
gap> RecPn := ReciprocalPolynomial(P, 6);
-x_1^3+x_1^4+x_1^6
```

Sesión interactiva con GAP.

El objetivo de esta sesión es que el usuario se familiarice con la construcción y manipulación de diferentes dominios en GAP, concretamente los dominios de los campos finitos y de los polinomios sobre campos finitos.
Para realizar esta práctica es conveniente dominar el contenido de la asignatura de *Matemática Discreta*.

Al final de esta sesión, se deberá presentar un fichero llamado *sesiongap.log* con las respuestas a las preguntas. Para llevar a cabo esta tarea, se puede guardar la sesión empleando el comando `LogTo("sesiongap.log");` y posteriormente numerar las respuestas y añadir los comentarios precisos.

Apartado 1: Campos finitos.

1. Construir tres campos finitos, G2, G3 y G5 con 2, 3 y 5 elementos respectivamente. Mostrar por pantalla los elementos de cada uno de estos campos. Expresar estos elementos como elementos de $Z_2=\{0,1\}$, $Z_3=\{0,1,2\}$ y $Z_5=\{0,1,2,3,4\}$ respectivamente. ¿Es $Z_4=\{0,1,2,3\}$ un campo?¿Por qué?

2. Definir los campos finitos siguientes: G9=Z3(x)/2 + 2x + x^2, G16=Z2[x]/1 + x + x^4, G25 = Z5[x]/2 + 4x +x^2 y G9b = Z3[x]/x^2 + 2x + 1 ¿Por qué en el último caso tuvo problemas? ¿Son irreductibles los polinomios 2 + 2x + $x^2 \in$ Z3[x], 1 + x + $x^4 \in$ Z2[x] y 2 +4x + $x^2 \in$ Z5[x]?

3. Definir un campo de 16 elementos, G16b, sin especificar un polinomio irreductible.

4. Mostrar por pantalla el elemento neutro para la suma y para el producto del campo G16.

5. Construir campos finitos de 2, 4, 8 y 16 elementos. Mostrar por pantalla todos los elementos de cada uno de estos campos. ¿Qué estructura de subcampos existe entre estos campos?

6. Encontrar el orden de todos los elementos del campo G25 distintos de cero. Utilizando el comando `Filtered`, construir un una lista con todos los elementos primitivos de dicho campo. ¿Cuántos elementos primitivos tiene? En general, ¿Cuántos elementos primitivos tiene un campo de p^n elementos, donde p es el primero?

7. Sea α un elemento primitivo del campo G25. Hallar la característica y el grado de α.

8. Calcular $2\alpha^3 - \alpha^5 + \alpha^6/(\alpha+2)$ y su inverso si existe, donde α es el elemento primitivo $Z(5^2)$ DE G(25). ¿Qué elementos de G25 son inversibles?

9. Calcular el polinomio mínimo del elemento $\alpha = Z(5^2)$. ¿Sobre qué campo está definido?¿ Cuáles son sus raíces en Z_5 y en G_{25}?

10. Definir el dominio $GF(3^{11})$. ¿Por qué se produce un mensaje de error?

Apartado 2: Polinomios sobre campos finitos.

1. Definir el anillo de los polinomios con coeficientes sobre el campo finito G25. ¿Es finito?

2. Introducir los polinomios siguientes:

   $p1 = Z(5^2)^2 + z(5^2)x + Z(5)x^2$

   $p2 = Z(5)^0 + Z(5^2)x$

3. Simplificar la notación empleada al visualizar un polinomio con coeficientes sobre G25, utilizando el comando `Indeterminate`. Visualizar el polinomio p1 para constatar el efecto producido.
   Observación: El resultado de visualizar p1 debería ser el siguiente:

   $Z(5)*x^2 + Z(5^2)*x + Z(5^2)^2$

   si se toma como indeterminada el nombre x.

4. Calcular la suma y el producto de los polinomios p1 y p2.

5. Calcular el cociente y el residuo que se obtiene al dividir p1 entre p2.

6. Calcular el máximo común divisor y el mínimo común múltiplo de ambos polinomios. ¿Son p1 y p2 coprimos?¿Por qué?

7. Representa el máximo común divisor como una combinación lineal de p1 y p2.

8. Factoriza el polinomio siguiente sobre G25 y calcula sus raices. Comprueba el resultado obtenido con la función `Value`

   $1/\alpha^4 - \alpha^3x + \alpha^5x^2$

   con $\alpha = Z(5^2)$

Glosario de funciones