GAP

Release 4.3

06 May 2002

Programming in GAP 4

The GAP Group

http://www.gap-system.org

Acknowledgement

We would like to thank the many people who have made contributions of various kinds to the development of GAP since 1986, in particular:

Isabel M. Araújo, Robert Arthur, Hans Ulrich Besche, Thomas Bischops, Oliver Bonten, Thomas Breuer, Frank Celler, Gene Cooperman, Bettina Eick, Volkmar Felsch, Franz Gähler, Greg Gamble, Willem de Graaf, Burkhard Höfling, Jens Hollmann, Derek Holt, Erzsébet Horváth, Alexander Hulpke, Ansgar Kaup, Susanne Keitemeier, Steve Linton, Frank Lübeck, Bohdan Majewski, Johannes Meier, Thomas Merkwitz, Wolfgang Merkwitz, Jürgen Mnich, Robert F. Morse, Scott Murray, Joachim Neubüser, Max Neunhöffer, Werner Nickel, Alice Niemeyer, Dima Pasechnik, Götz Pfeiffer, Udo Polis, Ferenc Rákóczi, Sarah Rees, Edmund Robertson, Ute Schiffer, Martin Schönert, Ákos Seress, Andrew Solomon, Heiko Theißen, Rob Wainwright, Alex Wegner, Chris Wensley and Charles Wright.

The following list gives the authors, indicated by `A`, who designed the code in the first place as well as the current maintainers, indicated by `M` of the various modules of which GAP is composed.

Since the process of modularization was started only recently, there might be omissions both in scope and in contributors. The compilers of the manual apologize for any such errors and promise to rectify them in future editions.

Kernel
: Frank Celler (A), Steve Linton (AM), Frank Lübeck (AM), Werner Nickel (AM), Martin Schönert (A)

Automorphism groups of finite pc groups
: Bettina Eick (AM)

Binary Relations
: Robert Morse (AM), Andrew Solomon (A)

Classes in nonsolvable groups
: Alexander Hulpke (AM)

Classical Groups
: Thomas Breuer (AM), Frank Celler (A), Stefan Kohl (AM), Frank Lübeck (AM), Heiko Theißen (A)

Congruences of magmas, semigroups and monoids
: Robert Morse (AM), Andrew Solomon (A)

Cosets and Double Cosets
Alexander Hulpke (AM)

Cyclotomics
Thomas Breuer (AM)

Dixon-Schneider Algorithm
Alexander Hulpke (AM)

Documentation Utilities
Frank Celler (A), Heiko Theißen (A), Alexander Hulpke (A), Willem de Graaf (A), Steve Linton (A), Werner Nickel (A), Greg Gamble (AM)

Factor groups
Alexander Hulpke (AM)

Finitely presented groups
Volkmar Felsch (AM), Alexander Hulpke (AM), Martin Schoenert (A)

Finitely presented monoids and semigroups
Isabel Araújo (AM), Derek Holt (A), Alexander Hulpke (A), Götz Pfeiffer (A), Andrew Solomon (AM)

Group actions
Heiko Theißen (A) and Alexander Hulpke (AM)

Homomorphism search
Alexander Hulpke (AM)

Homomorphisms for finitely presented groups
Alexander Hulpke (AM)

Intersection of subgroups of finite pc groups
Frank Celler (A), Bettina Eick (AM)

Irreducible Modules over finite fields for finite pc groups
Bettina Eick (AM)

Isomorphism testing with random methods
Hans Ulrich Besche (AM), Bettina Eick (AM)

Multiplier and Schur cover
Werner Nickel (AM), Alexander Hulpke (AM)

One-Cohomology and Complements
Frank Celler (A) and Alexander Hulpke (AM)

Partition Backtrack algorithm
Heiko Theißen (A), Alexander Hulpke (M)

Permutation group composition series
Ákos Seress (AM)

Permutation group homomorphisms
Ákos Seress (AM), Heiko Theißen (A), Alexander Hulpke (M)

Permutation Group Pcgs
Heiko Theißen (A), Alexander Hulpke (M)

Primitive groups library
Heiko Theißen (A), Alexander Hulpke (M)

Properties and attributes of finite pc groups
Frank Celler (A), Bettina Eick (AM)

Random Schreier-Sims
Ákos Seress (AM)

Rational Functions
Frank Celler (A) and Alexander Hulpke (AM)

Semigroup relations
Isabel Araujo (A), Robert F. Morse (AM), Andrew Solomon (A)

Special Pcgs for finite pc groups
Bettina Eick (AM)

Stabilizer Chains
Ákos Seress (AM), Heiko Theißen (A), Alexander Hulpke (M)

Strings and Characters
Martin Schönert (A), Frank Celler (A), Thomas Breuer (A), Frank Lübeck (AM)

Subgroup lattice
Martin Schönert (A), Alexander Hulpke (AM)

Subgroup lattice for solvable groups
Alexander Hulpke (AM)

Subgroup presentations
Volkmar Felsch (AM)

The Help System
Frank Celler (A), Frank Lübeck (AM)

Tietze transformations
Volkmar Felsch (AM)

Transformation semigroups
Isabel Araujo (A), Robert Arthur (A), Robert F. Morse (AM), Andrew Solomon (A)

Transitive groups library
Alexander Hulpke (AM)

Two-cohomology and extensions of finite pc groups
Bettina Eick (AM)

Lie algebras
Thomas Breuer (A), Craig Struble (A), Juergen Wisliceny (A), Willem A. de Graaf (AM)

GAP for MacOS
Burkhard Höfling (AM)

Contents

Copyright Notice

If you modify any part of GAP and redistribute it, you must supply a `README` document. This should specify what modifications you made in which files. We do not want to take credit or be blamed for your modifications.

Of course we are interested in all of your modifications. In particular we would like to see bug-fixes, improvements and new functions. So again we would appreciate it if you would inform us about all modifications you make.

1 About Programming in GAP

This is one of four parts of the GAP documentation, the others being the GAP **Tutorial**, a beginner's introduction to GAP, the GAP **Reference Manual**, which contains the official definitions of GAP, and **Extending** GAP, which explains how to create files and functions that will work together with mechanisms built in GAP, how to write documentation, and so on.

This manual is divided into chapters. Each chapter is divided into sections, and within each section, important definitions are numbered. References therefore are triples.

The chapters 2 and 3 of this manual describe how the knowledge about GAP objects is used by the system, via the so-called method selection mechanism, and how such knowledge resp. objects with such knowledge can be created.

Chapter 4 gives some simple examples of how to add new functionality to the system.

A more involved example for the design of new GAP objects can be found in Chapter 5. In particular, see Sections 5.1 and 5.2 for finding out whether this manual is useful for you at all. One more example is discussed in Chapter 6.

Pages are numbered consecutively in each of the four manuals. For manual conventions, see Section 1.1 in the Reference Manual.

2 Method Selection

This chapter explains how GAP decides which function to call for which types of objects. It assumes that you have read the chapters about objects (Chapter 12) and types (Chapter 13) in the Reference Manual.

An **operation** is a special GAP function that bundles a set of functions, its **methods**.

All methods of an operation compute the same result. But each method is installed for specific types of arguments.

If an operation is called with a tuple of arguments, one of the applicable methods is selected and called.

Special cases of methods are partial methods, immediate methods, and logical implications.

2.1 Operations and Methods

Operations are functions in the category `IsOperation` (see 5.4.2 in the Reference Manual).

So on the one hand, **operations** are GAP functions, that is, they can be applied to arguments and return a result or cause a side-effect.

On the other hand, operations are more. Namely, an operation corresponds to a set of GAP functions, called the **methods** of the operation.

Each call of an operation causes a suitable method to be selected and then called. The choice of which method to select is made according to the types of the arguments, the underlying mechanism is described in the following sections.

Examples of operations are the binary infix operators `=`, `+` etc., and `PrintObj` is the operation that is called for each argument of `Print`.

Also all attributes and properties are operations. Each attribute has a special method which is called if the attribute value is already stored; this method of course simply returns this value.

The setter of an attribute is called automatically if an attribute value has been computed. Attribute setters are operations, too. They have a default method that ignores the request to store the value. Depending on the type of the object, there may be another method to store the value in a suitable way, and then set the attribute tester for the object to `true`.

2.2 Method Installation

In order to describe what it means to select a method of an operation, we must describe how the methods are connected to their operations.

1 ▶ `InstallMethod(` *opr*`[,`*info*`][,`*famp*`],`*args-filts*`[,`*val*`],`*method* `)` F

installs a function method *method* for the operation *opr*; *args-filts* should be a list of requirements for the arguments; if supplied *info* should be a short but informative string that describes for what situation the method is installed, *famp* should be a function to be applied to the families of the arguments, each entry being a filter, and *val* is an integer that measures the priority of the method.

The default values for *info*, *famp*, and *val* are the empty string, the function `ReturnTrue`, and the integer zero, respectively.

The exact meaning of the arguments *famp*, *args-filts*, and *val* is explained in Section 2.3.

opr expects its methods to require certain filters for their arguments. For example, the argument of a method for the operation `Zero` must be in the category `IsAdditiveElementWithZero`. It is not possible to use `InstallMethod` to install a method for which the entries of *args-filts* do not imply the respective requirements of the operation *opr*. If one wants to override this restriction, one has to use `InstallOtherMethod` instead.

2 ▶ `InstallOtherMethod(` *opr*`[,`*info*`][,`*famp*`],`*args-filts*`[,`*val*`],`*method* `)` F

installs a function method *method* for the operation *opr*, in the same way as for `InstallMethod` (see 2.2.1), but without the restriction that the number of arguments must match the declaration of *opr* and without the restriction that *args-filts* imply the respective requirements of the operation *opr*.

For attributes and properties there is `InstallImmediateMethod` (see 2.6.1).

For declaring that a filter is implied by other filters there is `InstallTrueMethod` (see 2.7.1).

2.3 Applicable Methods and Method Selection

A method installed as above is **applicable** for an arguments tuple if the following conditions are satisfied.

The number of arguments equals the length of the list *args-filts*, the *i*-th argument lies in the filter *args-filts*`[`*i*`]`, and *famp* returns `true` when applied to the families of the arguments.

So *args-filt* describes conditions for each argument, and *famp* describes a relation between the arguments.

For unary operations such as attributes and properties, there is no such relation to postulate, *famp* is `ReturnTrue` for these operations, a function that always returns `true`. For binary operations, the usual value of *famp* is `IsIdenticalObj` (see 12.5.1 in the Reference Manual), which means that both arguments must lie in the same family.

Note that any properties which occur among the filters in the filter list will **not** be tested by the method selection if they are not yet known. (More exact: if *prop* is a property then the filter implicitly uses not *prop* but `Has`*prop* `and` *prop*.) If this is desired you must explicitly enforce a test (see section 2.5) below.

If no method is applicable, the error message `no method found` is signaled.

Otherwise, the applicable method with highest **rank** is selected and then called. This rank is given by the sum of the ranks of the filters in the list *args-filt*, **including involved filters**, plus the number *val* used in the call of `InstallMethod`. So the argument *val* can be used to raise the priority of a method relative to other methods for *opr*.

Note that from the applicable methods, an efficient one shall be selected. This is a method that needs only little time and storage for the computations.

It seems to be impossible for GAP to select an optimal method in all cases. The present ranking of methods is based on the assumption that a method installed for a special situation shall be preferred to a method installed for a more general situation.

For example, a method for computing a Sylow subgroup of a nilpotent group is expected to be more efficient than a method for arbitrary groups. So the more specific method will be selected if GAP knows that the group given as argument is nilpotent.

Of course there is no obvious way to decide between the efficiency of incommensurable methods. For example, take an operation with one method for permutation groups, another method for nilpotent groups, but no method for nilpotent permutation groups, and call this operation with a permutation group known to be nilpotent.

2.4 Partial Methods

1 ► `TryNextMethod()`

After a method has been selected and called, the method may recognize that it cannot compute the desired result, and give up by calling `TryNextMethod()`.

In effect, the execution of the method is terminated, and the method selection calls the next method that is applicable w.r.t. the original arguments. In other words, the applicable method is called that is subsequent to the one that called `TryNextMethod`, according to decreasing rank of the methods.

For example, since every finite group of odd order is solvable, one may install a method for the property `IsSolvableGroup` that checks whether the size of the argument is an odd integer, returns `true` if so, and gives up otherwise.

Care is needed if a partial method might modify the type of one of its arguments, for example by computing an attribute or property. If this happens, and the type has really changed, then the method should not exit using `TryNextMethod()` but should call the operation again, as the new information in the type may cause some methods previously judged inapplicable to be applicable. For example, if the above method for `IsSolvableGroup` actually computes the size, (rather than just examining a stored size), then it must take care to check whether the type of the group has changed.

2.5 Redispatching

As mentioned above the method selection will not test unknown properties. In situations, in which algorithms are only known (or implemented) under certain conditions, however such a test might be actually desired.

One way to achieve this would be to install the method under weaker conditions and explicitly test the properties first, exiting via `TryNextMethod` (see 2.4.1) if some of them are not fulfilled. A problem of this approach however is that such methods then automatically are ranked lower and that the code does not look nice.

A much better way is to use redispatching: Before deciding that no method has been found one tests these properties and if they turn out to be true the method selection is started anew (and will then find a method).

This can be achieved via the following function:

1 ► `RedispatchOnCondition(` *oper*`,` *fampred*`,` *reqs*`,` *cond*`,` *val* `)` F

This function installs a method for the operation *oper* under the conditions *fampred* and *reqs* which has absolute value *val*; that is, the value of the filters *reqs* is disregarded. *cond* is a list of filters. If not all the values of properties involved in these filters are already known for actual arguments of the method, they are explicitly tested and if they are fulfilled **and** stored after this test, the operation is dispatched again. Otherwise the method exits with `TryNextMethod` (see 2.4.1). This can be used to enforce tests like `IsFinite` in situations when all existing methods require this property. The list *cond* may have unbound entries in which case the corresponding argument is ignored for further tests.

2.6 Immediate Methods

Usually a method is called only if its operation has been called and if this method has been selected.

For attributes and properties, one can install also **immediate methods**. An immediate method is called automatically as soon as it is applicable to an object, provided that the value is not yet known. Afterwards the attribute setter is called in order to store the value.

Note that in such a case GAP executes a computation for which it was not explicitly asked by the user. So one should install only those methods as immediate methods that are **extremely cheap**. To emphasize this, immediate methods are also called **zero cost methods**. The time for their execution should really be approximately zero.

An immediate method *method* for the attribute or property *attr* with requirement *req* is installed via

```
1 ► InstallImmediateMethod( attr, req, val, method )
```

where *val* is an integer value that measures the priority of *method* among the immediate methods for *attr*.

Note the difference to `InstallMethod` (see 2.2.1) that no family predicate occurs because *attr* expects only one argument, and that *req* is not a list of requirements but the argument requirement itself.

For example, the size of a permutation group can be computed very cheaply if a stabilizer chain of the group is known. So it is reasonable to install an immediate method for `Size` with requirement `IsGroup and Tester( stab )`, where *stab* is the attribute corresponding to the stabilizer chain.

Another example would be the implementation of the conclusion that every finite group of prime power order is nilpotent. This could be done by installing an immediate method for the attribute `IsNilpotentGroup` with requirement `IsGroup and Tester( Size )`. This method would then check whether the size is a finite prime power, return `true` in this case and otherwise call `TryNextMethod()` (see 2.4.1). But this requires factoring of an integer, which cannot be guaranteed to be very cheap, so one should not install this method as an immediate method.

Immediate methods are thought of as a possibility for objects to gain useful knowledge. They must not be used to force the storing of "defining information" in an object. In other words, GAP should work even if all immediate methods are invalidated.

2.7 Logical Implications

It may happen that a filter *newfil* shall be implied by another filter *filt*, which is usually a meet of other properties, or the meet of some properties and some categories. Such a logical implication can be installed as an immediate method for *newfil* that requires *filt* and that always returns `true`. It should be installed via

```
1 ► InstallTrueMethod( newfil, filt )
```

This has the effect that *newfil* becomes an implied filter of *filt*, see 13.2 in the Reference Manual.

For example, each cyclic group is abelian, each finite vector space is finite dimensional, and each division ring is integral. The first of these implications is installed as follows.

```
InstallTrueMethod( IsCommutative, IsGroup and IsCyclic );
```

Contrary to other immediate methods, logical implications cannot be switched off. This means that after the above implication has been installed, one can rely on the fact that every object in the filter `IsGroup and IsCyclic` will also be in the filter `IsCommutative`.

2.8 Operations and Mathematical Terms

Usually an operation stands for a mathematical concept, and the name of the operation describes this uniquely. Examples are the property `IsFinite` and the attribute `Size`. But there are cases where the same mathematical term is used to denote different concepts, for example `Degree` is defined for polynomials, group characters, and permutation actions, and `Rank` is defined for matrices, free modules, p-groups, and transitive permutation actions.

It is in principle possible to install methods for the operation `Rank` that are applicable to the different types of arguments, corresponding to the different contexts. But this is not the approach taken in the GAP library. Instead there are operations such as `RankMat` for matrices and `DegreeOfCharacter` (in fact these are attributes) which are installed as methods of the "ambiguous" operations `Rank` and `Degree`.

The idea is to distinguish between on the one hand different ways to compute the same thing (e.g. different methods for `\=`, `Size`, etc.), and on the other hand genuinely different things (such as the degree of a polynomial and a permutation action).

The former is the basic purpose of operations and attributes. The latter is provided as a user convenience where mathematical usage forces it on us **and** where no conflicts arise. In programming the library, we use the underlying mathematically precise operations or attributes, such as `RankMat` and `RankOperation`. These should be attributes if appropriate, and the only role of the operation `Rank` is to decide which attribute the user meant. That way, stored information is stored with "full mathematical precision" and is less likely to be retrieved for a wrong purpose later.

One word about possible conflicts. A typical example is the mathematical term "centre", which is defined as $\{x \in M | a * x = x * a \forall a \in M\}$ for a magma M, and as $\{x \in L | l * x = 0 \forall l \in L\}$ for a Lie algebra L. Here it is **not** possible to introduce an operation `Centre` that delegates to attributes `CentreOfMagma` and `CentreOfLieAlgebra`, depending on the type of the argument. This is because any Lie algebra in GAP is also a magma, so both `CentreOfMagma` and `CentreOfLieAlgebra` would be defined for a Lie algebra, with different meaning if the characteristic is 2. So we cannot achieve that one operation in GAP corresponds to the mathematical term "centre".

"Ambiguous" operations such as `Rank` are declared in the library file `overload.g`.

3 Creating New Objects

This chapter is divided into three parts.

In the first part, it is explained how to create filters (see 3.1, 3.2, 3.3, 3.4), operations (see 3.5), families (see 3.6), types (see 3.7), and objects with given type (see 3.8).

In the second part, first a few small examples are given, for dealing with the usual cases of component objects (see 3.9) and positional objects (see 3.10), and for the implementation of new kinds of lists (see 3.11 and 3.12). Finally, the external representation of objects is introduced (see 3.13), as a tool for representation independent access to an object.

The third part deals with some rules concerning the organization of the GAP library; namely, some commands for creating global variables are explained (see 3.15) that correspond to the ones discussed in the first part of the chapter, and the idea of distinguishing declaration and implementation part of GAP packages is outlined (see 3.16).

See also Chapter 5 for examples how the functions from the first part are used, and why it is useful to have a declaration part and an implementation part.

3.1 Creating Categories

1 ▶ `NewCategory(` *name*`,` *super* `)`

`NewCategory` returns a new category *cat* that has the name *name* and is contained in the filter *super*, see 13.2 in the Reference Manual. This means that every object in *cat* lies automatically also in *super*. We say also that *super* is an implied filter of *cat*.

For example, if one wants to create a category of group elements then *super* should be `IsMultiplicativeElementWithInverse` or a subcategory of it. If no specific supercategory of *cat* is known, *super* may be `IsObject`.

@Eventually tools will be provided to display hierarchies of categories etc., which will help to choose *super* **appropriately.@**

The incremental rank (see 13.2 in the Reference Manual) of *cat* is 1.

Two functions that return special kinds of categories are of importance.

2 ▶ `CategoryCollections(` *cat* `)`

For a category *cat*, `CategoryCollections` returns the **collections category** of *cat*. This is a category in that all collections of objects in *cat* lie.

For example, a permutation lies in the category `IsPerm`, and every dense list of permutations and every domain of permutations lies in the collections category of `IsPerm`.

3 ▶ `CategoryFamily(` *cat* `)`

For a category *cat*, `CategoryFamily` returns the **family category** of *cat*. This is a category in that all families lie that know from their creation that all their elements are in the category *cat*, see 3.6.

For example, a family of tuples is in the category `CategoryFamily( IsTuple )`, and one can distinguish such a family from others by this category. So it is possible to install methods for operations that require one argument to be a family of tuples.

`CategoryFamily` is quite technical, and in fact of minor importance.

3.2 Creating Representations

1 ► `NewRepresentation( name, super, slots )`

`NewRepresentation` returns a new representation *rep* that has the name *name* and is a subrepresentation of the representation *super*. This means that every object in *rep* lies automatically also in *super*. We say also that *super* is an implied filter of *rep*.

Each representation in GAP is a subrepresentation of exactly one of the four representations `IsInternalRep`, `IsDataObjectRep`, `IsComponentObjectRep`, `IsPositionalObjectRep`. The data describing objects in the former two can be accessed only via GAP kernel functions, the data describing objects in the latter two is accessible also in library functions, see 3.9 and 3.10 for the details.

The third argument *slots* is a list either of integers or of strings. In the former case, *rep* must be `IsPositionalObjectRep` or a subrepresentation of it, and *slots* tells what positions of the objects in the representation *rep* may be bound. In the latter case, *rep* must be `IsComponentObjectRep` or a subrepresentation of, and *slots* lists the admissible names of components that objects in the representation *rep* may have. The admissible positions resp. component names of *super* need not be be listed in *slots*.

The incremental rank (see 13.2 in the Reference Manual) of *rep* is 1.

Note that for objects in the representation *rep*, of course some of the component names and positions reserved via *slots* may be unbound.

Examples for the use of `NewRepresentation` can be found in 3.9, 3.10, and also in 5.3.

3.3 Creating Attributes and Properties

1 ► `NewAttribute( name, filt )`
► `NewAttribute( name, filt, rank )`

`NewAttribute` returns a new attribute *attr* with name *name* (see also 13.5 in the Reference Manual). The filter *filt* describes the involved filters of *attr* (see 13.2 in the Reference Manual). That is, the argument for *attr* is expected to lie in *filt*.

Each method for *attr* that does **not** require its argument to lie in *filt* must be installed using `InstallOtherMethod`.

Contrary to the situation with categories and representations, the tester of *attr* does **not** imply *filt*. This is exactly because of the possibility to install methods that do not require *filt*.

For example, the attribute `Size` was created with second argument a list or a collection, but there is also a method for `Size` that is applicable to a character table, which is neither a list nor a collection.

The optional third argument *rank* denotes the incremental rank (see 13.2 in the Reference Manual) of the tester of *attr*, the default value is 1.

2 ► `NewAttribute( name, filt, "mutable" )`
► `NewAttribute( name, filt, "mutable", rank )`

If the third argument is the string `"mutable"`, the stored values of the new attribute are not forced to be immutable. This is useful for an attribute whose value is some partial information that may be completed later. For example, there is an attribute `ComputedSylowSubgroups` for the list holding those Sylow subgroups of a group that have been computed already by the function `SylowSubgroup`, and this list is mutable because one may want to enter groups into it as they are computed.

3 ► `NewProperty( name, filt )`
► `NewProperty( name, filt, rank )`

`NewProperty` returns a new property *prop* with name *name* (see also 13.7 in the Reference Manual). The filter *filt* describes the involved filters of *prop*. As in the case of attributes, *filt* is not implied by *prop*.

The optional third argument *rank* denotes the incremental rank (see 13.2 in the Reference Manual) of the property *prop* itself, i.e. **not** of its tester, the default value is 1.

Each method that is installed for an attribute or a property via `InstallMethod` must require exactly one argument, and this must lie in the filter *filt* that was entered as second argument of `NewAttribute` resp. `NewProperty`.

As for any operation (see 3.5), for attributes and properties one can install a method taking an argument that does not lie in *filt* via `InstallOtherMethod`, or a method for more than one argument; in the latter case, clearly the result value is **not** stored in any of the arguments.

3.4 Creating Other Filters

1 ▶ `NewFilter(` *name* `)`
▶ `NewFilter(` *name*`,` *rank* `)`

`NewFilter` returns a simple filter with name *name* (see 13.8 in the Reference Manual). The optional second argument *rank* denotes the incremental rank (see 13.2 in the Reference Manual) of the filter, the default value is 1.

In order to change the value of *filt* for an object *obj*, one can use logical implications (see 2.7) or the functions

2 ▶ `SetFilterObj(` *obj*`,` *filt* `)`
▶ `ResetFilterObj(` *obj*`,` *filt* `)`

`SetFilterObj` sets the value of *filt* (and of all filters implied by *filt*) for *obj* to `true`,

`ResetFilterObj` sets the value of *filt* for *obj* to `false` (but implied filters of *filt* are not touched. This might create inconsistent situations if applied carelessly).

The default value of *filt* for each object is `false`.

3.5 Creating Operations

1 ▶ `NewOperation(` *name*`,` *args-filts* `)`

`NewOperation` returns an operation *opr* with name *name*. The list *args-filts* describes requirements about the arguments of *opr*, namely the number of arguments must be equal to the length of *args-filts*, and the i-th argument must lie in the filter *args-filts*`[`i`]`.

Each method that is installed for *opr* via `InstallMethod` must require that the i-th argument lies in the filter *args-filts*`[`i`]`.

One can install methods for other arguments tuples via `InstallOtherMethod`, this way it is also possible to install methods for a different number of arguments than the length of *args-filts*.

3.6 Creating Families

Families are probably the least obvious part of the GAP type system, so some remarks about the role of families are necessary. When one uses GAP as it is, one will (better: should) not meet families at all. The two situations where families come into play are the following.

First, since families are used to describe relations between arguments of operations in the method selection mechanism (see Chapter 2 in this manual, and also Chapter 13 in the Reference Manual), one has to prescribe such a relation in each method installation (see 2.2); usual relations are `ReturnTrue` (which means that any relation of the actual arguments is admissible), `IsIdenticalObj` (which means that there are two arguments that lie in the same family), and `IsCollsElms` (which means that there are two arguments, the first being a collection of elements that lie in the same family as the second argument).

Second —and this is the more complicated situation— whenever one creates a new kind of objects, one has to decide what its family shall be. If the new object shall be equal to existing objects, for example if it is just represented in a different way, there is no choice: The new object must lie in the same family as all objects that shall be equal to it. So only if the new object is different (w.r.t. the equality "=") from all other GAP objects, we are likely to create a new family for it. Note that enlarging an existing family by such new objects may be problematic because of implications that have been installed for all objects of the family in question. The choice of families depends on the applications one has in mind. For example, if the new objects in question are not likely to be arguments of operations for which family relations are relevant (for example binary arithmetic operations), one could create one family for all such objects, and regard it as "the family of all those GAP objects that would in fact not need a family". On the other extreme, if one wants to create domains of the new objects then one has to choose the family in such a way that all intended elements of a domain do in fact lie in the same family. (Remember that a domain is a collection, see Chapter 12.4 in the Reference Manual, and that a collection consists of elements in the same family, see Chapter 28 and Section 13.1 in the Reference Manual.)

Let us look at an example. Suppose that no permutations are available in GAP, and that we want to implement permutations. Clearly we want to support permutation groups, but it is not a priori clear how to distribute the new permutations into families. We can put all permutations into one family; this is how in fact permutations are implemented in GAP. But it would also be possible to put all permutations of a given degree into a family of their own; this would for example mean that for each degree, there would be distinguished trivial permutations, and that the stabilizer of the point `5` in the symmetric group on the points $1, 2, \ldots, 5$ is not regarded as equal to the symmetric group on $1, 2, 3, 4$. Note that the latter approach would have the advantage that it is no problem to construct permutations and permutation groups acting on arbitrary (finite) sets, for example by constructing first the symmetric group on the set and then generating any desired permutation group as a subgroup of this symmetric group.

So one aspect concerning a reasonable choice of families is to make the families large enough for being able to form interesting domains of elements in the family. But on the other hand, it is useful to choose the families small enough for admitting meaningful relations between objects. For example, the elements of different free groups in GAP lie in different families; the multiplication of free group elements is installed only for the case that the two operands lie in the same family, with the effect that one cannot erroneously form the product of elements from different free groups. In this case, families appear as a tool for providing useful restrictions.

As another example, note that an element and a collection containing this element never lie in the same family, by the general implementation of collections; namely, the family of a collection of elements in the family *Fam* is the collections family of *Fam* (see 3.6.2). This means that for a collection, we need not (because we cannot) decide about its family.

1 ▶ `NewFamily(` *name* `)`
▶ `NewFamily(` *name*`,` *req* `)`
▶ `NewFamily(` *name*`,` *req*`,` *imp* `)`
▶ `NewFamily(` *name*`,` *req*`,` *imp*`,` *famfilter* `)`

`NewFamily` returns a new family *fam* with name *name*. The argument *req*, if present, is a filter of which *fam* shall be a subset. If one tries to create an object in *fam* that does not lie in the filter *req*, an error message is printed. Also the argument *imp*, if present, is a filter of which *fam* shall be a subset. Any object that is created in the family *fam* will lie automatically in the filter *imp*.

The filter *famfilter*, if given, specifies a filter that will hold for the family *fam* (not for objects in *fam*).

Families are always represented as component objects (see 3.9). This means that components can be used to store and access useful information about the family.

There are a few functions in GAP that construct families. As an example, consider (see also 28.1 in the Reference Manual)

2 ► `CollectionsFamily( fam )`

`CollectionsFamily` is an attribute that takes a family *fam* as argument, and returns the family of all collections over *fam*, that is, of all dense lists and domains that consist of objects in *fam*.

The `NewFamily` call in the standard method of `CollectionsFamily` is executed with second argument `IsCollection`, since every object in the collections family must be a collection, and with third argument the collections categories of the involved categories in the implied filter of *fam*.

If *fam* is a collections family then

3 ► `ElementsFamily( fam )`

returns the unique family with collections family *fam*; note that by definition, all elements in a collection lie in the same family, so `ElementsFamily( fam )` is the family of each element in any collection that has the family *fam*.

3.7 Creating Types

1 ► `NewType( fam, filt )`
► `NewType( fam, filt, data )`

`NewType` returns the type given by the family *fam* and the filter *filt*. The optional third argument *data* is any object that denotes defining data of the desired type.

For examples where `NewType` is used, see 3.9, 3.10, and the example in Chapter 5.

3.8 Creating Objects

1 ► `Objectify( type, data )` F

New objects are created by `Objectify`. *data* is a list or a record, and *type* is the type that the desired object shall have. `Objectify` turns *data* into an object with type *type*. That is, *data* is changed, and afterwards it will not be a list or a record unless *type* is of type list resp. record.

If *data* is a list then `Objectify` turns it into a positional object, if *data* is a record then `Objectify` turns it into a component object (for examples, see 3.9 and 3.10).

`Objectify` does also return the object that it made out of *data*.

For examples where `Objectify` is used, see 3.9, 3.10, and the example in Chapter 5.

Attribute assignments will change the type of an object. If you create many objects, code of the form

```
o:=Objectify(type,rec());
SetMyAttribute(o,value);
```

will take a lot of time for type changes. You can avoid this by setting the attributes immediately while the object is created, via:

2 ► `ObjectifyWithAttributes(`*obj*`,`*type*`,`*Attr1*`,`*val1*`[,`*Attr2*`,`*val2*`...])` F

which changes the type of object *obj* to type *type* and sets attribute *Attr1* to *val1*, sets attribute *Attr2* to *val2* and so forth.

If the filter list of *type* includes that these attributes are set (and the properties also include values of the properties) and if no special setter methods are installed for any of the involved attributes then they are set simultaneously without type changes which can produce a substantial speedup.

If the conditions of the last sentence are not fulfilled, an ordinary `Objectify` with subsequent `Setter` calls for the attributes is performed, instead.

3.9 Component Objects

A **component object** is an object in the representation `IsComponentObjectRep` or a subrepresentation of it. Such an object *cobj* is built from subobjects that can be accessed via *cobj*`!.`*name*, similar to components of a record. Also analogously to records, values can be assigned to components of *cobj* via *cobj*`!.`*name*`:=` *val*. For the creation of component objects, see 3.8.

1 ▶ `NamesOfComponents(` *comobj* `)` F

For a component object *comobj*, `NamesOfComponents` returns a list of strings, which are the names of components currently bound in *comobj*.

One must be **very careful** when using the `!.` operator, in order to interpret the component in the right way, and even more careful when using the assignment to components using `!.`, in order to keep the information stored in *cobj* consistent.

First of all, in the access or assignment to a component as shown above, *name* must be among the admissible component names for the representation of *cobj*, see 3.2. Second, preferably only few low level functions should use `!.`, whereas this operator should not occur in "user interactions".

Note that even if *cobj* claims that it is immutable, i.e., if *cobj* is not in the category `IsMutable`, access and assignment via `!.` work. This is necessary for being able to store newly discovered information in immutable objects.

The following example shows the implementation of an iterator (see 28.7 in the Reference Manual) for the domain of integers, which is represented as component object. See 3.10 for an implementation using positional objects.

The used succession of integers is $0, 1, -1, 2, -2, 3, -3, \ldots$, that is, $a_n = n/2$ if n is even, and $a_n = (1-n)/2$ otherwise.

```
IsIntegersIteratorCompRep := NewRepresentation( "IsIntegersIteratorRep",
    IsComponentObjectRep, [ "counter" ] );
```

The above command creates a new representation (see 3.2.1) `IsIntegersIteratorCompRep`, as a subrepresentation of `IsComponentObjectRep`, and with one admissible component `counter`. So no other components than `counter` will be needed.

```
InstallMethod( Iterator,
    "method for 'Integers'",
    [ IsIntegers ],
    function( Integers )
    return Objectify( NewType( IteratorsFamily,
                                  IsIterator
                               and IsIntegersIteratorCompRep ),
                      rec( counter := 0 ) );
    end );
```

After the above method installation, one can already ask for `Iterator( Integers )`. Note that exactly the domain of integers is described by the filter `IsIntegers`.

By the call to `NewType`, the returned object lies in the family containing all iterators, which is `IteratorsFamily`, it lies in the category `IsIterator` and in the representation `IsIntegersIteratorCompRep`; furthermore, it has the component `counter` with value `0`.

What is missing now are methods for the two basic operations of iterators, namely `IsDoneIterator` and `NextIterator`. The former must always return `false`, since there are infinitely many integers. The latter must return the next integer in the iteration, and update the information stored in the iterator, that is, increase the value of the component `counter`.

```
InstallMethod( IsDoneIterator,
    "method for iterator of 'Integers'",
    [ IsIterator and IsIntegersIteratorCompRep ],
    ReturnFalse );

InstallMethod( NextIterator,
    "method for iterator of 'Integers'",
    [ IsIntegersIteratorCompRep ],
    function( iter )
    iter!.counter:= iter!.counter + 1;
    if iter!.counter mod 2 = 0 then
      return iter!.counter / 2;
    else
      return ( 1 - iter!.counter ) / 2;
    fi;
    end );
```

3.10 Positional Objects

A **positional object** is an object in the representation `IsPositionalObjectRep` or a subrepresentation of it. Such an object *pobj* is built from subobjects that can be accessed via *pobj*`![`*pos*`]`, similar to positions in a list. Also analogously to lists, values can be assigned to positions of *pobj* via *pobj*`![`*pos*`]:=` *val*. For the creation of positional objects, see 3.8.

One must be **very careful** when using the `![]` operator, in order to interpret the position in the right way, and even more careful when using the assignment to positions using `![]`, in order to keep the information stored in *pobj* consistent.

First of all, in the access or assignment to a position as shown above, *pos* must be among the admissible positions for the representation of *pobj*, see 3.2. Second, preferably only few low level functions should use `![]`, whereas this operator should not occur in "user interactions".

Note that even if *pobj* claims that it is immutable, i.e., if *pobj* is not in the category `IsMutable`, access and assignment via `![]` work. This is necessary for being able to store newly discovered information in immutable objects.

The following example shows the implementation of an iterator (see 28.7 in the Reference Manual) for the domain of integers, which is represented as positional object. See 3.9 for an implementation using component objects, and more details.

```
IsIntegersIteratorPosRep := NewRepresentation( "IsIntegersIteratorRep",
    IsPositionalObjectRep, [ 1 ] );
```

The above command creates a new representation (see 3.2.1) `IsIntegersIteratorPosRep`, as a subrepresentation of `IsComponentObjectRep`, and with only the first position being admissible for storing data.

```
InstallMethod( Iterator,
    "method for 'Integers'",
    [ IsIntegers ],
    function( Integers )
    return Objectify( NewType( IteratorsFamily,
                                   IsIterator
                               and IsIntegersIteratorRep ),
                      [ 0 ] );
    end );
```

After the above method installation, one can already ask for `Iterator( Integers )`. Note that exactly the domain of integers is described by the filter `IsIntegers`.

By the call to NewType, the returned object lies in the family containing all iterators, which is IteratorsFamily, it lies in the category IsIterator and in the representation IsIntegersIteratorPosRep; furthermore, the first position has value 0.

What is missing now are methods for the two basic operations of iterators, namely IsDoneIterator and NextIterator. The former must always return false, since there are infinitely many integers. The latter must return the next integer in the iteration, and update the information stored in the iterator, that is, increase the value stored in the first position.

```
InstallMethod( IsDoneIterator,
    "method for iterator of 'Integers'",
    [ IsIterator and IsIntegersIteratorPosRep ],
    ReturnFalse );

InstallMethod( NextIterator,
    "method for iterator of 'Integers'",
    [ IsIntegersIteratorPosRep ],
    function( iter )
    iter![1]:= iter![1] + 1;
    if iter![1] mod 2 = 0 then
      return iter![1] / 2;
    else
      return ( 1 - iter![1] ) / 2;
    fi;
    end );
```

It should be noted that one can of course install both the methods shown in Section 3.9 and 3.10. The call Iterator(Integers) will cause one of the methods to be selected, and for the returned iterator, which will have one of the representations we constructed, the right NextIterator method will be chosen.

3.11 Implementing New List Objects

This section gives some hints for the quite usual situation that one wants to implement new objects that are lists. More precisely, one either wants to deal with lists that have additional features, or one wants that some objects also behave as lists.

A **list** in GAP is an object in the category IsList. Basic operations for lists are Length, \[\], and IsBound\[\] (see 21.2 in the Reference Manual).

Note that the access to the position *pos* in the list *list* via *list*[*pos*] is handled by the call \[\](*list*, *pos*) to the operation \[\]. To explain the somewhat strange name \[\] of this operation, note that non-alphanumeric characters like [and] may occur in GAP variable names only if they are escaped by a \ character.

Analogously, the check IsBound(*list*[*pos*]) whether the position *pos* of the list *list* is bound is handled by the call IsBound\[\](*list*, *pos*) to the operation IsBound\[\].

For mutable lists, also assignment to positions and unbinding of positions via the operations \[\]\:\= and Unbind\[\] are basic operations. The assignment *list*[*pos*]:= *val* is handled by the call \[\]\:\=(*list*, *pos*, *val*), and Unbind(*list*[*pos*]) is handled by the call Unbind\[\](*list*, *pos*).

All other operations for lists, e.g., Add, Append, Sum, are based on these operations. This means that it is sufficient to install methods for the new list objects only for the basic operations.

So if one wants to implement new list objects then one creates them as objects in the category IsList, and installs methods for Length, \[\], and IsBound\[\]. If the new lists shall be mutable, one needs to install also methods for \[\]\:\= and Unbind\[\].

One application for this is the implementation of **enumerators** for domains. An enumerator for the domain D is a dense list whose entries are in bijection with the elements of D. If D is large then it is not useful to write down all elements. Instead one can implement such a bijection implicitly. This works also for infinite domains.

In this situation, one implements a new representation of the lists that are already available in GAP, in particular the family of such a list is the same as the family of the domain D.

But it is also possible to implement new kinds of lists that lie in new families, and thus are not equal to lists that were available in GAP before. An example for this is the implementation of matrices whose multiplication via "`*`" is the Lie product of matrices.

In this situation, it makes no sense to put the new matrices into the same family as the original matrices. Note that the product of two Lie matrices shall be defined but not the product of an ordinary matrix and a Lie matrix. So it is possible to have two lists that have the same entries but that are not equal w.r.t. "`=`" because they lie in different families.

3.12 Arithmetic Issues in the Implementation of New Kinds of Lists

When designing a new kind of list objects in GAP, defining the arithmetic behaviour of these objects is an issue.

There are situations where arithmetic operations of list objects are unimportant in the sense that adding two such lists need not be represented in a special way. In such cases it might be useful either to support no arithmetics at all for the new lists, or to enable the default arithmetic methods. The former can be achieved by not setting the filters `IsGeneralizedRowVector` and `IsMultiplicativeGeneralizedRowVector` in the types of the lists, the latter can be achieved by setting the filter `IsListDefault`. (for details, see 21.12 in the GAP Reference Manual). An example for "wrapped lists" with default behaviour are vector space bases; they are lists with additional properties concerning the computation of coefficients, but arithmetic properties are not important. So it is no loss to enable the default methods for these lists.

However, often the arithmetic behaviour of new list objects is important, and one wants to keep these lists away from default methods for addition, multiplication etc. For example, the sum and the product of (compatible) block matrices shall be represented as a block matrix, so the default methods for sum and product of matrices shall not be applicable, although the results will be equal to those of the default methods in the sense that their entries at corresponding positions are equal.

So one does not set the filter `IsListDefault` in such cases, and thus one can implement one's own methods for arithmetic operations. (Of course "can" means on the other hand that one **must** implement such methods if one is interested in arithmetics of the new lists.)

The specific binary arithmetic methods for the new lists will usually cover the case that both arguments are of the new kind, and perhaps also the interaction between a list of the new kind and certain other kinds of lists may be handled if this appears to be useful.

For the last situation, interaction between a new kind of lists and other kinds of lists, GAP provides already a setup. Namely, there are the categories `IsGeneralizedRowVector` and `IsMultiplicativeGeneralizedRowVector`, which are concerned with the additive and the multiplicative behaviour, respectively, of lists. For lists in these filters, the structure of the results of arithmetic operations is prescribed (see 21.13 and 21.14 in the GAP Reference Manual).

For example, if one implements block matrices in `IsMultiplicativeGeneralizedRowVector` then automatically the product of such a block matrix and a (plain) list of such block matrices will be defined as the obvious list of matrix products, and a default method for plain lists will handle this multiplication. (Note that this method will rely on a method for computing the product of the block matrices, and of course no default method is available for that.) Conversely, if the block matrices are not in `IsMultiplicativeGeneralizedRowVector` then the product of a block matrix and a (plain) list of block matrices is not defined. (There is no default method for it, and one can define the result and provide a method for computing it.)

Thus if one decides to set the filters `IsGeneralizedRowVector` and `IsMultiplicativeGeneralizedRowVector` for the new lists, on the one hand one loses freedom in defining arithmetic behaviour, but on the other hand one gains several default methods for a more or less natural behaviour.

If a list in the filter `IsGeneralizedRowVector` (`IsMultiplicativeGeneralizedRowVector`) lies in `IsAttributeStoringRep`, the values of additive (multiplicative) nesting depth is stored in the list and need not be calculated for each arithmetic operation. One can then store the value(s) already upon creation of the lists, with the effect that the default arithmetic operations will access elements of these lists only if this is unavoidable. For example, the sum of two plain lists of "wrapped matrices" with stored nesting depths are computed via the method for adding two such wrapped lists, and without accessing any of their rows (which might be expensive). In this sense, the wrapped lists are treated as black boxes.

3.13 External Representation

An operation is defined for elements rather than for objects in the sense that if the arguments are replaced by objects that are equal to the old arguments w.r.t. the equivalence relation "`=`" then the result must be equal to the old result w.r.t. "`=`".

But the implementation of many methods is representation dependent in the sense that certain representation dependent subobjects are accessed.

For example, a method that implements the addition of univariate polynomials may access coefficients lists of its arguments only if they are really stored, while in the case of sparsely represented polynomials a different approach is needed.

In spite of this, for many operations one does not want to write an own method for each possible representations of each argument, for example because none of the methods could in fact take advantage of the actually given representations of the objects. Another reason could be that one wants to install first a representation independent method, and then add specific methods as they are needed to gain more efficiency, by really exploiting the fact that the arguments have certain representations.

For the purpose of admitting representation independent code, one can define an **external representation** of objects in a given family, install methods to compute this external representation for each representation of the objects, and then use this external representation of the objects whenever they occur.

We cannot provide conversion functions that allow us to first convert any object in question to one particular "standard representation", and then access the data in the way defined for this representation, simply because it may be impossible to choose such a "standard representation" uniformly for all objects in the given family.

So the aim of an external representation of an object *obj* is a different one, namely to describe the data from which *obj* is composed. In particular, the external representation of *obj* is **not** one possible ("standard") representation of *obj*, in fact the external representation of *obj* is in general different from *obj* w.r.t. "`=`", first of all because the external representation of *obj* does in general not lie in the same family as *obj*.

For example the external representation of a rational function is a list of length two or three, the first entry being the zero coefficient, the second being a list describing the coefficients and monomials of the numerator, and the third, if bound, being a list describing the coefficients and monomials of the denominator. In particular, the external representation of a polynomial is a list and not a polynomial.

The other way round, the external representation of *obj* encodes *obj* in such a way that from this data and the family of *obj*, one can create an object that is equal to *obj*. Usually the external representation of an object is a list or a record.

Although the external representation of *obj* is by definition independent of the actually available representations for *obj*, it is usual that a representation of *obj* exists for which the computation of the external representation is obtained by just "unpacking" *obj*, in the sense that the desired data is stored in a component or a position of *obj*, if *obj* is a component object (see 3.9) or a positional object (see 3.10).

To implement an external representation means to install methods for the following two operations.

1 ▶ `ExtRepOfObj(` *obj* `)`
▶ `ObjByExtRep(` *fam*`,` *data* `)`

`ExtRepOfObj` returns the external representation of its argument, and `ObjByExtRep` returns an object in the family *fam* that has external representation *data*.

Of course, `ObjByExtRep( FamilyObj(` *obj* `), ExtRepOfObj(` *obj* `) )` must be equal to *obj*. But it is **not** required that equal objects have equal external representations.

Note that if one defines a new representation of objects for which an external representation does already exist then one **must** install a method to compute this external representation for the objects in the new representation.

3.14 Mutability and Copying

Any GAP Object is either mutable or immutable. This can be tested with the Operation `IsMutable`. The intended meaning of (im)mutability is a mathematical one: an immutable Object should never change in such a way that it represents a different Element. Objects **may** change in other ways, for instance to store more information, or represent an element in a different way.

Immutability is enforced in different ways for built-in objects (like records, or lists) and for external objects (made using `Objectify`).

For built-in objects which are immutable, the kernel will prevent you from changing them. Thus

```
gap> l := [1,2,4];
[ 1, 2, 4 ]
gap> MakeImmutable(l);
gap> l[3] := 5;
Lists Assignment: <list> must be a mutable list
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' and ignore the assignment to continue
brk>
```

For external Objects, the situation is different. An external Object which claims to be immutable (i.e. its Type does not contain `IsMutable`) should not admit any Methods which change the Element it represents. The kernel does **not** prevent the use of `!.` and `![` to change the underlying data structure. This is used for instance by the code that stores Attribute values for reuse. In general, these `!` operations should only be used in Methods which depend on the Representation of the Object. Furthermore, we would **not** recommend users to install Methods which depend on the Representations of Objects created by the library or by GAP packages, as there is certainly no guarantee of the representations being the same in future versions of GAP.

Here we see an immutable Object (the group S_4), in which we improperly install a new component.

```
gap> g := SymmetricGroup(IsPermGroup,4);
Sym( [ 1 .. 4 ] )
gap> IsMutable(g);
false
gap> NamesOfComponents(g);
[ "GeneratorsOfMagmaWithInverses", "Size", "MovedPoints", "NrMovedPoints" ]
gap> g!.silly := "rubbish";
"rubbish"
gap> NamesOfComponents(g);
[ "GeneratorsOfMagmaWithInverses", "Size", "MovedPoints", "NrMovedPoints",
  "silly" ]
```

```
gap> g!.silly;
"rubbish"
```

On the other hand, if we form an immutable externally represented list, we find that there is no Method for changing it using list assignment

```
gap> e := Enumerator(g);
<enumerator of perm group>
gap> IsMutable(e);
false
gap> IsList(e);
true
gap> e[3];
(1,4)(2,3)
gap> e[3] := false;
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 1st choice method found for 'ASS_LIST' on 3 arguments called from
... lines omitted here ...
```

When we consider copying Objects, another filter `IsCopyable`, enters the game and we find that `ShallowCopy` and `StructuralCopy` behave quite differently. Objects can be divided for this purpose into three: mutable Objects, immutable but copyable Objects, and non-copyable objects (called constants).

A mutable or copyable Object should have a Method for the Operation `ShallowCopy`, which should make a new mutable Object, sharing its top-level subobjects with the original. The exact definition of top-level subobject may be defined by the implementor for new kinds of Object.

`ShallowCopy` applied to a constant simply returns the constant.

`StructuralCopy` is expected to be much less used than `ShallowCopy`. Applied to a mutable object, it returns a new mutable object which shares no mutable sub-objects with the input. Applied to an immutable Object (even a copyable one), it just returns the object. It is not an Operation (indeed, it's a rather special kernel function).

```
gap> e1 := StructuralCopy(e);
<enumerator of perm group>
gap> IsMutable(e1);
false
gap> e2 := ShallowCopy(e);
[ (), (1,2)(3,4), (1,4)(2,3), (1,3)(2,4), (2,3,4), (1,2,4), (1,4,3), (1,3,2),
  (2,4,3), (1,2,3), (1,4,2), (1,3,4), (3,4), (1,2), (1,4,2,3), (1,3,2,4),
  (2,3), (1,2,4,3), (1,4), (1,3,4,2), (2,4), (1,2,3,4), (1,4,3,2), (1,3) ]
gap>
```

There are two other related functions: `Immutable`, which makes a new immutable object which shares no mutable subobjects with its input and `MakeImmutable` which changes an object and its mutable subobjects **in place** to be immutable. It should only be used on "new" Objects that you have just created, and which cannot share mutable subobjects with anything else.

Both `Immutable` and `MakeImmutable` work on external objects by just resetting the `IsMutable` filter in the Object's type. This should make ineligible any methods that might change the Object. As a consequence, you must allow for the possibility of immutable versions of any objects you create.

So, if you are implementing your own external Objects. The rules amount to the following:

1. You decide if your Objects should be mutable or copyable or constants, by fixing whether their Type includes `IsMutable` or `IsCopyable`.

2. You install Methods for your objects respecting that decision:
   - for constants – no methods change the underlying elements;
   - for copyables – you provide a method for `ShallowCopy`;
   - for mutables – you may have methods that change the underlying elements and these should explicitly require `IsMutable`.

3.15 Global Variables in the Library

Global variables in the GAP library are usually read-only in order to avoid their being overwritten accidentally.

1 ▶ `BindGlobal(` *name*`,` *val* `)` F

sets the global variable named by the string *name* to the value *val*, and makes it read-only. An error is given if the global variable corresponding to *name* already had a value bound.

2 ▶ `DeclareAttribute(` *name*`,` *filt*`[, "mutable"][,` *rank*`] )` F
▶ `DeclareCategory(` *name*`,` *super* `)` F
▶ `DeclareFilter(` *name*`,` *rank* `)` F
▶ `DeclareProperty(` *name*`,` *filt*`[,` *rank*`] )` F
▶ `DeclareRepresentation(` *name*`,` *super*`,` *slots* `)` F

The different types of filters (see Sections 3.1, 3.2, 3.3, 3.4) that are used in the GAP library are assigned by the above `Declare`*Something* functions which make the variable with name *name* (a string) automatically read-only. The only other difference between `New`*Something* and `Declare`*Something* is that `DeclareAttribute` and `DeclareProperty` also bind read-only global variables with names `Has`*name* and `Set`*name* for the tester and setter of the attribute (see Section 13.6 in the Reference Manual). For the meaning of the other arguments of `Declare`*Something*, see 3.3.1, 3.1.1, 3.4.1, 3.3.3, and 3.2.1.

3 ▶ `DeclareOperation(` *name*`,` *args-filts* `)` F
▶ `DeclareGlobalFunction(` *name* `)` F

declare operations and other global functions used in the GAP library, respectively, are assigned to the read-only variable with name *name* (a string). For the meaning of the other arguments of `DeclareOperation`, see 3.5.1.

GAP functions that are not operations and that are intended to be called by users should be notified to GAP in the declaration part of the respective package (see Section 3.16) via `DeclareGlobalFunction`, which returns a function that serves as a place holder for the function that will be installed later, and that will print an error message if it is called. See also 3.15.7.

4 ▶ `InstallGlobalFunction(` *gvar*`,` *func* `)` F

A global function declared with `DeclareGlobalFunction` can be given its value *func* via `InstallGlobalFunction`; *gvar* is the global variable (**not** a string) named with the *name* argument of the call to `DeclareGlobalFunction`. For example, a declaration like

```
DeclareGlobalFunction( "SumOfTwoCubes" );
```

in the "declaration part" (see Section 3.16) might have a corresponding "implementation part" of:

```
InstallGlobalFunction( SumOfTwoCubes, function(x, y) return x^3 + y^3; end);
```

Note: *func* must be a function which has **not** been declared as a `GlobalFunction` itself. Otherwise completion files (see 3.5 in the reference manual) get confused!

5 ▶ `DeclareGlobalVariable( name[, description] )` F

For global variables that are **not** functions, instead of using `BindGlobal` one can also declare the variable with `DeclareGlobalVariable` which creates a new global variable named by the string *name*. If the second argument *description* is entered then this must be a string that describes the meaning of the global variable. `DeclareGlobalVariable` shall be used in the declaration part of the respective package (see 3.16), values can then be assigned to the new variable with `InstallValue` or `InstallFlushableValue`, in the implementation part (again, see 3.16).

6 ▶ `InstallValue( gvar, value )` F
▶ `InstallFlushableValue( gvar, value )` F

`InstallValue` assigns the value *value* to the global variable *gvar*. `InstallFlushableValue` does the same but additionally provides that each call of `FlushCaches` (see 3.15.9) will assign a structural copy of *value* to *gvar*.

`InstallValue` does **not** work if *value* is an "immediate object" (i.e., an internally represented small integer or finite field element). Furthermore, `InstallFlushableValue` works only if *value* is a list. (Note that `InstallFlushableValue` makes sense only for **mutable** global variables.)

7 ▶ `DeclareSynonym( name, value )` F

assigns the string *name* to a global variable as a synonym for *value*. Two typical intended usages are to declare an "and-filter", e.g.

```
DeclareSynonym( "IsGroup", IsMagmaWithInverses and IsAssociative );
```

and (mainly for compatibility reasons) to provide a previously declared global function with an alternative name, e.g.

```
DeclareGlobalFunction( "SizeOfSomething" );
DeclareSynonym( "OrderOfSomething", SizeOfSomething );
```

Note: Before using `DeclareSynonym` in the way of this second example, one should determine whether the synonym is really needed. Perhaps an extra index entry in the documentation would be sufficient.

When declaring a synonym that is to be an attribute `DeclareSynonymAttr` should be used.

8 ▶ `DeclareSynonymAttr( name, value )` F

assigns the string *name* to an attribute global variable as a synonym for *value*. Two typical intended usages are to provide a previously declared attribute or property with an alternative name, e.g.

```
DeclareAttribute( "GeneratorsOfDivisionRing", IsDivisionRing );
DeclareSynonymAttr( "GeneratorsOfField", GeneratorsOfDivisionRing );
```

and to declare an attribute that is an "and-filter", e.g.

```
DeclareSynonymAttr( "IsField", IsDivisionRing and IsCommutative );
```

Also see 3.15.7. (The comments made there also pertain to `DeclareSynonymAttr`.)

9 ▶ `FlushCaches()` O

`FlushCaches` resets the value of each global variable that has been declared with `DeclareGlobalVariable` and for which the initial value has been set with `InstallFlushableValue` to this initial value.

`FlushCaches` should be used only for debugging purposes, since the involved global variables include for example lists that store finite fields and cyclotomic fields used in the current GAP session, in order to avoid that these fields are constructed anew in each call to `GF` and `CF` (see 57.3.1 and 58 in the Reference Manual).

3.16 Declaration and Implementation Part

Each package of GAP code consists of two parts, the **declaration part** that defines the new categories and operations for the objects the package deals with, and the **implementation part** where the corresponding methods are installed. The declaration part should be representation independent, representation dependent information should be dealt with in the implementation part.

GAP functions that are not operations and that are intended to be called by users should be notified to GAP in the declaration part via `DeclareGlobalFunction`. Values for these functions can be installed in the implementation part via `InstallGlobalFunction`.

Calls to the following functions belong to the declaration part.

`DeclareAttribute`, `DeclareCategory`, `DeclareFilter`, `DeclareOperation`, `DeclareGlobalFunction`, `DeclareSynonym`, `DeclareSynonymAttr`, `DeclareProperty`, `InstallTrueMethod`.

See 3.15.2, 3.15.2, 3.15.2, 3.15.3, 3.15.3, 3.15.7, 3.15.8, 3.15.2, 2.7.1.

Calls to the following functions belong to the implementation part.

`DeclareRepresentation`, `InstallGlobalFunction`, `InstallMethod`, `InstallImmediateMethod`, `InstallOtherMethod`, `NewFamily`, `NewType`, `Objectify`.

See 3.15.2, 3.15.4, 2.2.1, 2.6.1, 2.2.2, 3.6.1, 3.7.1, 3.8.1.

Whenever both a `New`*Something* and a `Declare`*Something* variant of a function exist (see 3.15), the use of `Declare`*Something* is recommended because this protects the variables in question from being overwritten. Note that there are **no** functions `DeclareFamily` and `DeclareType` since families and types are created dynamically, hence usually no global variables are associated to them. Further note that `DeclareRepresentation` is regarded as belonging to the implementation part, because usually representations of objects are accessed only in very few places, and all code that involves a particular representation is contained in one file; additionally, representations of objects are often not interesting for the user, so there is no need to provide a user interface or documentation about representations.

It should be emphasized that "declaration" means only an explicit notification of mathematical or technical terms or of concepts to GAP. For example, declaring a category or property with name `IsInteresting` does of course not tell GAP what this shall mean, and it is necessary to implement possibilities to create objects that know already that they lie in `IsInteresting` in the case that it is a category, or to install implications or methods in order to compute for a given object whether `IsInteresting` is `true` or `false` for it in the case that `IsInteresting` is a property.

4 Examples of Extending the System

This chapter gives a few examples of how one can extend the functionality of GAP.

They are arranged in ascending difficulty. We show how to install new methods, add new operations and attributes and how to implement new features using categories and representations. (As we do not introduce completely new kinds of objects in these example it will not be necessary to declare any families.) Finally we show a simple way how to create new objects with an own arithmetic.

The examples given are all very rudimentary – no particular error checks are performed and the user interface sometimes is quite clumsy.

Even more complex examples that create whole classes of objects anew will be given in the following two chapters 5 and 6.

4.1 Addition of a Method

The easiest case is the addition of a new algorithm as a method for an existing operation for the existing structures.

For example, assume we wanted to implement a better method for computing the exponent of a nilpotent group (it is the product of the exponents of the Sylow subgroups).

The first task is to find which operation is used by GAP (it is `Exponent`) and how it is declared. We can find this in the reference manual (in our particular case in section 37.15) and the declaration in the library file `lib/grp.gd` (The easiest way to find the place of the declaration is usually to `grep` over all `.gd` and `.g` files, see section 3 of "Extending Gap".)

In our example the declaration in the library is:

```
DeclareAttribute("Exponent",IsGroup);
```

Similarly we find that the filter `IsNilpotentGroup` represents the concept of being nilpotent.

We then write a function that implements the new algorithm which takes the right set of arguments and install it as a method. In our example this installation would be:

```
InstallMethod(Exponent,"for nilpotent groups",
  [IsGroup and IsNilpotent],
function(G)
  [function body omitted]
end);
```

We have left out the optional rank argument of `InstallMethod`, which normally is a wise choice – GAP automatically uses an internal ranking based on the filters that is only offset by the given rank. So our method will certainly be "better" than a method that has been installed for mere groups or for solvable groups but will be ranked lower than the library method for abelian groups.

That's all. Using `ApplicableMethod` (see 7.2.1) we can check for an nilpotent group that indeed our new method will be used.

When testing, remember that the method selection will not check for properties that are not known. (This is done internally by checking the property tester first.) Therefore the method would not be applicable for the group `g` in the following definition but only for the – mathematically identical but endowed with more knowledge by GAP – group `h`. (Section 4.3 shows a way around this.)

```
gap> g:=Group((1,2),(1,3)(2,4));;
gap> h:=Group((1,2),(1,3)(2,4));;
gap> IsNilpotentGroup(h); # enforce test
true
gap> HasIsNilpotentGroup(g);
false
gap> HasIsNilpotentGroup(h);
true
```

Lets now look at a slightly more complicated example: We want to implement a better method for computing normalizers in a nilpotent permutation group. (Such an algorithm can be found for example in [LRW97].)

We already know `IsNilpotentGroup`, the filter `IsPermGroup` represent the concepts of being a group of permutations.

GAP uses `Normalizer` to compute normalizers, however the declaration is a bit more complicated. In the library we find

```
InParentFOA( "Normalizer", IsGroup, IsObject, NewAttribute );
```

The full mechanism of `InParentFOA` is described in chapter 6 of "Extending GAP", however for our purposes it is sufficient to know that for such a function the actual work is done by an operation `NormalizerOp` (and all the complications are just there to be able to remember certain results) and that the declaration of this operation is given by the first arguments, it would be:

```
DeclareOperation( "NormalizerOp", [IsGroup, IsObject] );
```

This time we decide to enter a non-default family predicate in the call to `InstallMethod`. We could just leave it out as in the previous call; this would yield the default value, the function `ReturnTrue` of arbitrary many arguments which always returns `true`. However, then the method might be called in some cases of inconsistent input (for example matrix groups in different characteristics) that ought to fall through the method selection to raise an error.

In our situation, we want the second group to be a subgroup of the first, so necessarily both must have the same family and we can use `IsIdenticalObj` as family predicate.

Now we can install the method. Again this manual is lazy and does not show you the actual code:

```
InstallMethod(NormalizerOp,"for nilpotent permutation groups",IsIdenticalObj,
  [IsPermGroup and IsNilpotentGroup,
   IsPermGroup and IsNilpotentGroup],
function(G,U)
  [ function body omitted ]
end);
```

4.2 Extending the Range of Definition of an Existing Operation

It might be that the operation has been defined so far only for a set of objects that is too restrictive for our purposes (or we want to install a method that takes another number of arguments). If this is the case, the call to `InstallMethod` causes an error message. We can avoid this by using `InstallOtherMethod` instead of `InstallMethod`.

4.3 Enforcing Property Tests

As mentioned above, GAP does not check unknown properties to test whether a method might be applicable. In some cases one wants to enforce this, however, because the gain from knowing the property outweighs the cost of its determination.

In this situation one has to install a method **without** the additional property (so it can be tried even if the property is not yet known) and at high rank (so it will be used before other methods). The first thing to do in the actual function then is to test the property and to bail out with `TryNextMethod()` (see 2.4.1) if it turns out to be `false`.

The above `Exponent` example thus would become:

```
InstallMethod(Exponent,"test abelianity", [IsGroup],
  50,# enforced high rank
function(G)
  if not IsAbelian(G) then
    TryNextMethod();
  fi;
  [remaining function body omitted]
end);
```

The value "50" used in this example is quite arbitrary. A better way is to use values that are given by the system inherently: We want this method still to be ranked as high, **as if it had** the `IsAbelian` requirement. So we have GAP compute the extra rank of this:

```
InstallMethod(Exponent,"test abelianity", [IsGroup],
  # enforced absolute rank of 'IsGroup and IsAbelian' installation: Subtract
  # the rank of 'IsGroup' and add the rank of 'IsGroup and IsAbelian':
  SIZE_FLAGS(FLAGS_FILTER(IsGroup and IsAbelian))
  -SIZE_FLAGS(FLAGS_FILTER(IsGroup)),
function(G)
```

the slightly complicated construction of addition and subtraction is necessary because `IsGroup` and `IsAbelian` might imply the **same** elementary filters which we otherwise would count twice.

A somehow similar situation occurs with matrix groups. Most methods for matrix groups are only applicable if the group is known to be finite.

However we should not enforce a finiteness test early (someone else later might install good methods for infinite groups while the finiteness test would be too expensive) but just before GAP would give a "no method found" error. This is done by redispatching, see 2.5. For example to enforce such a final finiteness test for normalizer calculations could be done by:

```
RedispatchOnCondition(NormalizerOp,IsIdenticalObj,
  [IsMatrixGroup,IsMatrixGroup],[IsFinite,IsFinite],0);
```

4.4 Adding a new Operation

The next step is to add own operations. As an example we take the Sylow normalizer in a group of a given prime. This operation gets two arguments, the first has to be a group, the second a prime number.

There is a function `IsPrimeInt`, but no property for being prime (which would be pointless as integers cannot store property values anyhow). So the second argument gets specified only as positive integer:

```
SylowNormalizer:=NewOperation("SylowNormalizer",[IsGroup,IsPosInt]);
```

(Note that we are using `NewOperation` (see 3.5.1) instead of `DeclareOperation` (see 3.15.3) as used in the library. The only difference other than that `DeclareOperation` saves some typing, is that it also protects the variables against overwriting. When testing code (when one probably wants to change things) this might be restricting. If this does not bother you, you can use

```
DeclareOperation("SylowNormalizer",[IsGroup,IsPosInt]);
```

as well.)

The filters `IsGroup` and `IsPosInt` given are **only** used to test that `InstallMethod` (see 2.2.1) installs methods with suitable arguments and will be completely ignored when using `InstallOtherMethod` (see 2.2.2). Technically one could therefore simply use `IsObject` for all arguments in the declaration. The main point of using more specific filters here is to help documenting with which arguments the function is to be used (so for example a call `SylowNormalizer(5,G)` would be invalid).

Of course initially there are no useful methods for newly declared operations; you will have to write and install them yourself.

If the operation only takes one argument and has reproducible results without side effects, it might be worth declaring it as an attribute instead; see the next section (4.5).

4.5 Adding a new Attribute

Now we look at an example of how to add a new attribute. As example we consider the set of all primes that divide the size of a group.

First we have to declare the attribute:

```
PrimesDividingSize:=NewAttribute("PrimesDividingSize",IsGroup);
```

(See 3.3.1). This implicitly declares attribute tester and setter, it is convenient however to assign these to variables as well:

```
HasPrimesDividingSize:=Tester(PrimesDividingSize);
SetPrimesDividingSize:=Setter(PrimesDividingSize);
```

Alternatively, there is a declaration command `DeclareAttribute` (see 3.15.2) that executes all three assignments simultaneously and protects the variables against overwriting:

```
DeclareAttribute("PrimesDividingSize",IsGroup);
```

Next we have to install method(s) for the attribute that compute its value. (This is not strictly necessary. We could use the attribute also without methods only for storing and retrieving information, but calling it for objects for which the value is not known would produce a "No method found" error.) For this purpose we can imagine the attribute simply as an one-argument operation:

```
InstallMethod(PrimesDividingSize,"for finite groups",
  [IsGroup and IsFinite],
function(G)
  if Size(G)=1 then return [];
  else return Set(Factors(Size(G)));fi;
end);
```

The function installed **must** always return a value (or call `TryNextMethod`; see 2.4.1). If the object is in the representation `IsAttributeStoringRep` this return value once computed will be automatically stored and retrieved if the attribute is called a second time. We don't have to call setter or tester ourselves. (This storage happens by GAP internally calling the attribute setter with the return value of the function. Retrieval is by a high-ranking method which is installed under the condition `HasPrimesDividingSize`. This method was installed automatically when the attribute was declared.)

4.6 Adding a new Representation

Next, we look at the implementation of a new representation of existing objects. In most cases we want to implement this representation only for efficiency reasons while keeping all the existing functionality.

For example, assume we wanted (following [Wie69]) to implement permutation groups defined by relations.

Next, we have to decide a few basics about the representation. All existing permutation groups in the library are attribute storing and we probably want to keep this for our new objects. Thus the representation must be a subrepresentation of `IsComponentObjectRep and IsAttributeStoringRep`. Furthermore we want each object to be a permutation group and we can imply this directly in the representation.

We also decide that we store the degree (the largest point that might be moved) in a component `degree` and the defining relations in a component `relations` (we do not specify the format of relations here. In an actual implementation one would have to design this as well, but it does not affect the declarations this chapter is about).

```
IsPermutationGroupByRelations:=NewRepresentation(
  "IsPermutationGroupByRelations",
  IsComponentObjectRep and IsAttributeStoringRep and IsPermGroup,
  ["degree","relations"]);
```

(If we wanted to implement sparse matrices we might for example rather settle for a positional object in which we store a list of the nonzero entries.)

We can make the new representation a subrepresentation of an existing one. In such a case of course we have to provide all structure of this "parent" representation as well.

Next we need to check in which family our new objects will be. This will be the same family as of every other permutation group, namely the `CollectionsFamily(PermutationsFamily)` (where the family `PermutationsFamily=FamilyObj((1,2,3))` has been defined already in the library).

Now we can write a function to create our new objects. Usually it is helpful to look at functions from the library that are used in similar situations (for example `GroupByGenerators` in our case) to make sure we have not forgotten any further requirements in the declaration we might have to add here. However in most cases the function is straightforward:

```
PermutationGroupByRelations:=function(degree,relations)
local g
  g:=Objectify(NewType(CollectionsFamily(PermutationsFamily),
       IsPermutationGroupByRelations),
               rec(degree:=degree,relations:=relations));
end;
```

It also is a good idea to install a `Print` (possibly also a `View`) method – otherwise testing becomes quite hard:

```
InstallMethod(PrintObj,"for perm grps. given by relations",
  [IsPermutationGroupByRelations],
function(G)
  Print("PermutationGroupByRelations(", G!.degree,",",G!.relations,")");
end);
```

Next we have to write enough methods for the new representation so that the existing algorithms can be used. In particular we will have to implement methods for all operations for which library or kernel provides methods for the existing (alternative) representations. In our particular case there are no such methods. (If we would have implemented sparse matrices we would have had to implement methods for the list access and assignment functions, see 21.2 in the reference manual.) However the existing way permutation groups

are represented is by generators. To be able to use the existing machinery we want to be able to obtain a generating set also for groups in our new representation. This can be done (albeit not very effectively) by a stabilizer calculation in the symmetric group given by the `degree` component. The operation function to use is probably a bit complicated and will depend on the format of the `relations` (we have not specified in this example). In the following method we use `operationfunction` as a placeholder;

```
InstallMethod(GeneratorsOfGroup,"for perm grps. given by relations",
  [IsPermutationGroupByRelations],
function(G)
local S,U;
  S:=SymmetricGroup(G!.degree);
  U:=Stabilizer(S,G!.relations,  operationfunction );
  return GeneratorsOfGroup(U);
end);
```

This is all we **must** do. Of course for performance reasons one might want to install methods for further operations as well.

4.7 Components versus Attributes

In the last section we introduced two new components, `G!.degree` and `G!.relations`. Technically, we could have used attributes instead. There is no clear distinction which variant is to be preferred: An attribute expresses part of the functionality available to certain objects (and thus could be computed later and probably even for a wider class of objects), a component is just part of the internal definition of an object.

So if the data is "of general interest", if we want the user to have access to it, attributes are preferable. They provide a clean interface and their immutability makes it safe to hand the data to a user who potentially could corrupt a components entries.

On the other hand more "technical" data (say the encoding of a sparse matrix) is better hidden from the user in a component, as declaring it as an attribute would not give any advantage.

Resource-wise, attributes need more memory (the attribute setter and tester are implicitly declared, and two filter bits are required), the attribute access is one further function call in the kernel, thus components might be an immeasurable bit faster.

4.8 Adding new Concepts

Finally we look how to implement a new concept for existing objects and fit this in the method selection. Three examples that will be made more explicit below would be groups for which a "length" of elements (as a word in certain generators) is defined, groups that can be decomposed as a semidirect product and M-groups.

In each case we have two possibilities for the declaration. We can either declare it as a property or as a category. Both are eventually filter and in this way indistinguishable for the method selection. The distinction is rather conceptual and mainly reflects whether we want existing objects to be part of our new concept or not.

Property:
: Properties also are attributes: If a property value is not known for an object, GAP tries to find a method to compute the property value. If no suitable method is found, an error is raised.

Category:

An object is in a category if it has been created in it. Testing the category for an object simply returns this value. Existing objects cannot enter a new category later in life. This means that in most cases one has to write own code to create objects in a new category.
If we want to implement a completely new concept so that new operations are defined only for the new objects – for example bialgebras for which a second scalar multiplication is defined – usually a category is chosen.
Technically, the behaviour of the category `IsXYZ`, declared as subcategory of `IsABC` is therefore exactly the same as if we would declare `IsXYZ` to be a property for `IsABC` and install the following method:

```
InstallMethod(IsXYZ,"return false if not known",[IsABC],ReturnFalse);
```

(The words `category` also has a well-defined mathematical meaning, but this does not need to concern us at this point. The set of objects which is defined to be a (GAP)-category does not need to be a category in the mathematical sense, vice versa not every mathematical category is declared as a (GAP) category.)

Eventually the choice between category and property often becomes a matter of taste or style.

Sometimes there is even a third possibility (if you have GAP 3 experience this might reflect most closely "an object whose operations record is `XYOps`"): We might want to indicate this new concept simply by the fact that certain attributes are set. In this case we could simply use the respective attribute tester(s).

The examples given below each give a short argument why the respective solution was chosen, but one could argue as well for other choices.

4.9 Example: M-groups

M-groups are finite groups for which all irreducible complex representations are induced from linear representations of subgroups, it turns out that they are all solvable and that every supersolvable group is an M-group. See [Isa76] for further details.

Solvability and supersolvability both are testable properties. We therefore declare `IsMGroup` as a property for solvable groups:

```
IsMGroup:=NewProperty("IsMGroup",IsSolvableGroup);
```

The filter `IsSolvableGroup` in this declaration **only** means that methods for `IsMGroup` by default can only be installed for groups that are (and know to be) solvable (though they could be installed for more general situations using `InstallOtherMethod`). It does not yet imply that M-groups are solvable. We must do this deliberately via an implication and we use the same technique to imply that every supersolvable group is an M-group.

```
InstallTrueMethod(IsSolvableGroup,IsMGroup);
InstallTrueMethod(IsMGroup,IsSupersolvableGroup);
```

Now we might install a method that tests for solvable groups whether they are M-groups:

```
InstallMethod(IsMGroup,"for solvable groups",[IsSolvableGroup],
function(G)
  [... code omitted. The function must return 'true' or 'false' ...]
end);
```

4.10 Example: Groups with a word length

Our second example is that of groups for whose elements a `word length` is defined. (We assume that the word length is only defined in the context of the group with respect to a preselected generating set but not for single elements alone. However we will not delve into any details of how this length is defined and how it could be computed.)

Having a word length is a feature which enables other operations (for example a "word length" function). This is exactly what categories are intended for and therefore we use one.

First, we declare the category. All objects in this category are groups and so we inherit the supercategory `IsGroup`:

```
DeclareCategory("IsGroupWithWordLength",IsGroup);
```

We also define the operation which is "enabled" by this category, the word length of a group element, which is defined for a group and an element (remember that group elements are described by the category `IsMultiplicativeElementWithInverse`):

```
DeclareOperation("WordLengthOfElement",[IsGroupWithWordLength,
  IsMultiplicativeElementWithInverse]);
```

We then would proceed by installing methods to compute the word length in concrete cases and might for example add further operations to get shortest words in cosets.

4.11 Example: Groups with a decomposition as semidirect product

The third example is groups which have a (nontrivial) decomposition as a semidirect product. If this information has been found out, we want to be able to use it in algorithms. (Thus we do not only need the fact **that** there is a decomposition, but also the decomposition itself.)

We also want this to be applicable to every group and not only for groups which have been explicitly constructed via `SemidirectProduct`.

Instead we simply declare an attribute `SemidirectProductDecomposition` for groups. (again, in this manual we don't go in the details of how such an decomposition would look like).

```
DeclareAttribute("SemidirectProductDecomposition",IsGroup);
```

If a decomposition has been found, it can be stored in a group using `SetSemidirectProductDecomposition`. (At the moment all groups in GAP are attribute storing.)

Methods that rely on the existence of such a decomposition then get installed for the tester filter `HasSemidirectProductDecomposition`.

4.12 Creating Own Arithmetic Objects

Finally let's look at a way to create new objects with a user-defined arithmetic such that one can form for example groups, rings or vector spaces of these elements. This topic is discussed in much more detail in chapter 6, in this section we present a simple approach that may be useful to get started but does not permit you to exploit all potential features.

The basic design is that the user designs some way to represent her objects in terms of GAPs built-in types, for example as a list or a record. We call this the "defining data" of the new objects. Also provided are functions that perform arithmetic on this "defining data", that is they take objects of this form and return objects that represent the result of the operation. The function `ArithmeticElementCreator` then is called to provide a wrapping such that proper new GAP-objects are created which can be multiplied etc. with the default infix operations such as `\*`.

1 ▶ `ArithmeticElementCreator( spec )` F

offers a simple interface to create new arithmetic elements by providing functions that perform addition, multiplication and so forth, conforming to the specification *spec*. `ArithmeticElementCreator` creates a new category, representation and family for the new arithmetic elements being defined, and returns a function which takes the "defining data" of an element and returns the corresponding new arithmetic element.

spec is a record with one or more of the following components:

`ElementName`
: a string used to identify the new type of object. A global identifier `Is`*ElementName* will be defined to indicate a category for these now objects. (Therefore it is not clever to have blanks in the name). Also a collections category is defined. (You will get an error message if the identifier `Is`*ElementName* is already defined.)

`Equality`, `LessThan`, `One`, `Zero`, `Multiplication`, `Inverse`, `Addition`, `AdditiveInverse`
: functions defining the arithmetic operations. The functions interface on the level of "defining data", the actual methods installed will perform the unwrapping and wrapping as objects. Components are optional, but of course if no multiplication is defined elements cannot be multiplied and so forth. There are default methods for `Equality` and `LessThan` which simply calculate on the defining data. If one is defined, it must be ensured that the other is compatible (so that $a < b$ implies not$(a = b)$)

`Print`
: a function which prints the object. By default, just the defining data is printed.

`MathInfo`
: filters determining the mathematical properties of the elements created. A typical value is for example `IsMultiplicativeElementWithInverse` for group elements.

`RepInfo`
: filters determining the representational properties of the elements created. The objects created are always component objects, so in most cases the only reasonable option is `IsAttributeStoringRep` to permit the storing of attributes.

All components are optional and will be filled in with default values (though of course an empty record will not result in useful objects).

Note that the resulting objects are **not equal** to their defining data (even though by default they print as only the defining data). The operation `UnderlyingElement` can be used to obtain the defining data of such an element.

As the first example we look at subsets of $\{1 \ldots, 4\}$ and define an "addition" as union and "multiplication" as intersection. These operations are both commutative and we want the resulting elements to know this.

We therefore use the following specification:

```
gap> # the whole set
gap> w := [1,2,3,4];
[ 1, 2, 3, 4 ]
gap> PosetElementSpec :=rec(
>   # name of the new elements
>   ElementName := "PosetOn4",
>   # arithmetic operations
>   One := a -> w,
>   Zero := a -> [],
>   Multiplication := function(a, b) return Intersection(a, b); end,
>   Addition := function(a, b) return Union(a, b); end,
>   AdditiveInverse := a -> Filtered(w, x->(not x in a)),
>   # Mathematical properties of the elements
```

```
>   MathInfo := IsCommutativeElement and IsAdditivelyCommutativeElement
> );;
gap> mkposet := ArithmeticElementCreator(PosetElementSpec);
function( x ) ... end
```

Now we can create new elements, perform arithmetic on them and form domains:

```
gap> a := mkposet([1,2,3]);
[ 1, 2, 3 ]
gap> CategoriesOfObject(a);
[ "IsExtAElement", "IsNearAdditiveElement", "IsNearAdditiveElementWithZero",
  "IsNearAdditiveElementWithInverse", "IsExtLElement", "IsExtRElement",
  "IsMultiplicativeElement", "IsMultiplicativeElementWithOne",
  "IsAdditivelyCommutativeElement", "IsCommutativeElement", "IsPosetOn4" ]
gap> a=[1,2,3];
false
gap> UnderlyingElement(a)=[1,2,3];
true
gap> b:=mkposet([2,3,4]);
[ 2, 3, 4 ]
gap> a+b;
[ 1, 2, 3, 4 ]
gap> a*b;
[ 2, 3 ]
gap> s:=Semigroup(a,b);
<semigroup with 2 generators>
gap> Size(s);
3
```

The categories `IsPosetOn4` and `IsPosetOn4Collection` can be used to install methods specific to the new objects.

```
gap> IsPosetOn4Collection(s);
true
```

5 An Example – Residue Class Rings

In this chapter, we give an example how GAP can be extended by new data structures and new functionality. In order to focus on the issues of the implementation, the mathematics in the example chosen is trivial. Namely, we will discuss computations with elements of residue class rings $\mathbb{Z}/n\mathbb{Z}$.

The first attempt is straightforward (see Section 5.1), it deals with the implementation of the necessary arithmetic operations. Section 5.2 deals with the question why it might be useful to use an approach that involves creating a new data structure and integrating the algorithms dealing with these new GAP objects into the system. Section 5.3 shows how this can be done in our example, and Section 5.4, the question of further compatibility of the new objects with known GAP objects is discussed. Finally, Section 5.5 gives some hints how to improve the implementation presented before.

5.1 A First Attempt to Implement Elements of Residue Class Rings

Suppose we want to do computations with elements of a ring $\mathbb{Z}/n\mathbb{Z}$, where n is a positive integer.

First we have to decide how to represent the element $k + n\mathbb{Z}$ in GAP. If the modulus n is fixed then we can use the integer k. More precisely, we can use any integer k' such that $k - k'$ is a multiple of n. If different moduli are likely to occur then using a list of the form $[k, n]$, or a record of the form `rec( residue :=` k`, modulus :=` n `)` is more appropriate. In the following, let us assume the list representation $[k, n]$ is chosen. Moreover, we decide that the residue k in all such lists satisfies $0 \leq k < n$, i.e., the result of adding two residue classes represented by $[k_1, n]$ and $[k_2, n]$ (of course with same modulus n) will be $[k, n]$ with $k_1 + k_2$ congruent to k modulo n and $0 \leq k < n$.

Now we can implement the arithmetic operations for residue classes. Note that the result of the `mod` operator is normalized as required. The division by a noninvertible residue class results in `fail`.

```
resclass_sum := function( c1, c2 )
    if c1[2] <> c2[2] then Error( "different moduli" ); fi;
    return [ ( c1[1] + c2[1] ) mod c1[2], c1[2] ];
end;

resclass_diff := function( c1, c2 )
    if c1[2] <> c2[2] then Error( "different moduli" ); fi;
    return [ ( c1[1] - c2[1] ) mod c1[2], c1[2] ];
end;

resclass_prod := function( c1, c2 )
    if c1[2] <> c2[2] then Error( "different moduli" ); fi;
    return [ ( c1[1] * c2[1] ) mod c1[2], c1[2] ];
end;
```

```
resclass_quo := function( c1, c2 )
    local quo;
    if c1[2] <> c2[2] then Error( "different moduli" ); fi;
    quo:= QuotientMod( c1[1], c2[1], c1[2] );
    if quo <> fail then
      quo:= [ quo, c1[2] ];
    fi;
    return quo;
end;
```

With these functions, we can in principle compute with residue classes.

```
gap> list:= List( [ 0 .. 3 ], k -> [ k, 4 ] );
[ [ 0, 4 ], [ 1, 4 ], [ 2, 4 ], [ 3, 4 ] ]
gap> resclass_sum( list[2], list[4] );
[ 0, 4 ]
gap> resclass_diff( list[1], list[2] );
[ 3, 4 ]
gap> resclass_prod( list[2], list[4] );
[ 3, 4 ]
gap> resclass_prod( list[3], list[4] );
[ 2, 4 ]
gap> List( list, x -> resclass_quo( list[2], x ) );
[ fail, [ 1, 4 ], fail, [ 3, 4 ] ]
```

5.2 Why Proceed in a Different Way?

It depends on the computations we intended to do with residue classes whether or not the implementation described in the previous section is satisfactory for us.

Probably we are mainly interested in more complex data structures than the residue classes themselves, for example in matrix algebras or matrix groups over a ring such as $\mathbb{Z}/4\mathbb{Z}$. For this, we need functions to add, multiply, invert etc. matrices of residue classes. Of course this is not a difficult task, but it requires to write additional GAP code.

And when we have implemented the arithmetic operations for matrices of residue classes, we might be interested in domain operations such as computing the order of a matrix group over $\mathbb{Z}/4\mathbb{Z}$, a Sylow 2 subgroup, and so on. The problem is that a residue class represented as a pair $[k, n]$ is not regarded as a group element by GAP. We have not yet discussed how a matrix of residue classes shall be represented, but if we choose the obvious representation of a list of lists of our residue classes then also this is not a valid group element in GAP. Hence we cannot apply the function `Group` to create a group of residue classes or a group of matrices of residue classes. This is because GAP assumes that group elements can be multiplied via the infix operator `*` (equivalently, via the operation `\*`). Note that in fact the multiplication of two lists `[ k_1, n ]`, `[ k_2, n ]` is defined, but we have `[ k_1, n ] * [ k_2, n ] = k_1 * k_2 + n * n`, the standard scalar product of two row vectors of same length. That is, the multiplication with `*` is not compatible with the function `reclass_prod` introduced in the previous section. Similarly, ring elements are assumed to be added via the infix operator `+`; the addition of residue classes is not compatible with the available addition of row vectors.

What we have done in the previous section can be described as implementation of a "standalone" arithmetic for residue classes. In order to use the machinery of the GAP library for creating higher level objects such as matrices, polynomials, or domains over residue class rings, we have to "integrate" this implementation into the GAP library. The key step will be to create a new kind of GAP objects. This will be done in the following sections; there we assume that residue classes and residue class rings are not yet available in GAP; in fact they are available, and their implementation is very close to what is described here.

5.3 A Second Attempt to Implement Elements of Residue Class Rings

Faced with the problem to implement elements of the rings $\mathbb{Z}/n\mathbb{Z}$, we must define the **types** of these elements as far as is necessary to distinguish them from other GAP objects.

As is described in Chapter 13 in the Reference Manual, the type of an object comprises several aspects of information about this object; the **family** determines the relation of the object to other objects, the **categories** determine what operations the object admits, the **representation** determines how an object is actually represented, and the **attributes** describe knowledge about the object.

First of all, we must decide about the **family** of each residue class. A natural way to do this is to put the elements of each ring $\mathbb{Z}/n\mathbb{Z}$ into a family of their own. This means that for example elements of $\mathbb{Z}/3\mathbb{Z}$ and $\mathbb{Z}/9\mathbb{Z}$ lie in different families. So the only interesting relation between the families of two residue classes is equality; binary arithmetic operations with two residue classes will be admissible only if their families are equal. Note that in the naive approach in Section 5.1, we had to take care of different moduli by a check in each function; these checks may disappear in the new approach because of our choice of families.

Note that we do not need to tell GAP anything about the above decision concerning the families of the objects that we are going to implement, that is, the **declaration part** (see 3.16) of the little GAP package we are writing contains nothing about the distribution of the new objects into families. (The actual construction of a family happens in the function `MyZmodnZ` shown below.)

Second, we want to describe methods to add or multiply two elements in $\mathbb{Z}/n\mathbb{Z}$, and these methods shall be not applicable to other GAP objects. The natural way to do this is to create a new **category** in which all elements of all rings $\mathbb{Z}/n\mathbb{Z}$ lie. This is done as follows.

```
DeclareCategory( "IsMyZmodnZObj", IsScalar );
DeclareCategoryCollections( "IsMyZmodnZObj" );
```

So all elements in the rings $\mathbb{Z}/n\mathbb{Z}$ will lie in the category `IsMyZmodnZObj`, which is a subcategory of `IsScalar`. The latter means that one can add, subtract, multiply and divide two such elements that lie in the same family, with the obvious restriction that the second operand of a division must be invertible. (The name `IsMyZmodnZObj` is chosen because `IsZmodnZObj` is already defined in GAP, for an implementation of residue classes that is very similar to the one developed in this manual chapter. Using this different name, one can simply enter the GAP code of this chapter into a GAP session, either interactively or by reading a file with this code, and experiment after each step whether the expected behaviour has been achieved, and what is still missing.)

The second line of GAP code above declares the category `CategoryCollections( IsMyZmodnZObj )`, which will be needed later; it is important to create this category before collections of the objects in `IsMyZmodnZObj` arise.

Note that the only difference between `DeclareCategory` and `NewCategory` is that in a call to `DeclareCategory`, a variable corresponding to the first argument is set to the new category, and this variable is read-only (see 3.15). The same holds for `DeclareRepresentation` and `NewRepresentation` etc.

There is no analogue of categories in the implementation in Section 5.1, since there it was not necessary to distinguish residue classes from other GAP objects. Note that the functions there assumed that their arguments were residue classes, and the user was responsible not to call them with other arguments. Thus an important aspect of types is to describe arguments of functions explicitly.

Third, we must decide about the **representation** of our objects. This is something we know already from Section 5.1, where we chose a list of length two. Here we may choose between two essentially different representations for the new GAP objects, namely as "component object" (record–like) or "positional object" (list–like). We decide to store the modulus of each residue class in its family, and to encode the element $k + n\mathbb{Z}$ by the unique residue in the range `[ 0 .. n-1 ]` that is congruent to k modulo n, and the object itself is chosen to be a positional object with this residue at the first and only position (see 3.10).

```
DeclareRepresentation( "IsMyModulusRep", IsPositionalObjectRep, [ 1 ] );
```

The fourth ingredients of a type, **attributes**, are usually of minor importance for element objects. In particular, we do not need to introduce special attributes for residue classes.

Having defined what the new objects shall look like, we now declare a global function (see 3.16), to create an element when family and residue are given.

```
DeclareGlobalFunction( "MyZmodnZObj" );
```

Now we have declared what we need, and we can start to implement the missing methods resp. functions; so the following command belongs to the **implementation part** of our package (see 3.16).

The probably most interesting function is the one to construct a residue class.

```
InstallGlobalFunction( MyZmodnZObj, function( Fam, residue )
    return Objectify( NewType( Fam, IsMyZmodnZObj and IsMyModulusRep ),
                      [ residue mod Fam!.modulus ] );
end );
```

Note that we normalize `residue` explicitly using `mod`; we assumed that the modulus is stored in `Fam`, so we must take care of this below. If `Fam` is a family of residue classes, and `residue` is an integer, `MyZmodnZObj` returns the corresponding object in the family `Fam`, which lies in the category `IsMyZmodnZObj` and in the representation `IsMyModulusRep`.

`MyZmodnZObj` needs an appropriate family as first argument, so let us see how to get our hands on this. Of course we could write a handy function to create such a family for given modulus, but we choose another way. In fact we do not really want to call `MyZmodnZObj` explicitly when we want to create residue classes. For example, if we want to enter a matrix of residues then usually we start with a matrix of corresponding integers, and it is more elegant to do the conversion via multiplying the matrix with the identity of the required ring $\mathbb{Z}/n\mathbb{Z}$; this is also done for the conversion of integral matrices to finite field matrices. (Note that we will have to install a method for this.) So it is often sufficient to access this identity, for example via `One( MyZmodnZ(` n `) )`, where `MyZmodnZ` returns a domain representing the ring $\mathbb{Z}/n\mathbb{Z}$ when called with the argument n. We decide that constructing this ring is a natural place where the creation of the family can be hidden, and implement the function. (Note that the declaration belongs to the declaration part, and the installation belongs to the implementation part, see 3.16).

```
DeclareGlobalFunction( "MyZmodnZ" );

InstallGlobalFunction( MyZmodnZ, function( n )

    local F, R;

    if not IsPosInt( n ) then
      Error( "<n> must be a positive integer" );
    fi;

    # Construct the family of element objects of our ring.
    F:= NewFamily( Concatenation( "MyZmod", String( n ), "Z" ),
                   IsMyZmodnZObj );

    # Install the data.
    F!.modulus:= n;
```

```
    # Make the domain.
    R:= RingWithOneByGenerators( [ MyZmodnZObj( F, 1 ) ] );
    SetIsWholeFamily( R, true );
    SetName( R, Concatenation( "(Integers mod ", String(n), ")" ) );

    # Return the ring.
    return R;
end );
```

Note that the modulus `n` is stored in the component `modulus` of the family, as is assumed by `MyZmodnZ`. Thus it is not necessary to store the modulus in each element. When storing `n` with the `!.` operator as value of the component `modulus`, we used that all families are in fact represented as component objects (see 3.9).

We see that we can use `RingWithOneByGenerators` to construct a ring with one if we have the appropriate generators. The construction via `RingWithOneByGenerators` makes sure that `IsRingWithOne` (and `IsRing`) is `true` for each output of `MyZmodnZ`. So the main problem is to create the identity element of the ring, which in our case suffices to generate the ring. In order to create this element via `MyZmodnZObj`, we have to construct its family first, at each call of `MyZmodnZ`.

Also note that we may enter known information about the ring. Here we store that it contains the whole family of elements; this is useful for example when we want to check the membership of an element in the ring, which can be decided from the type of the element if the ring contains its whole elements family. Giving a name to the ring causes that it will be printed via printing the name. (By the way: This name `(Integers mod` n`)` looks like a call to `\mod` with the arguments `Integers` and n; a construction of the ring via this call seems to be more natural than by calling `MyZmodnZ`; later we shall install a `\mod` method in order to admit this construction.)

Now we can read the above code into GAP, and the following works already.

```
gap> R:= MyZmodnZ( 4 );
(Integers mod 4)
gap> IsRing( R );
true
gap> gens:= GeneratorsOfRingWithOne( R );
[ <object> ]
```

But of course this means just to ask for the information we have explicitly stored in the ring. Already the questions whether the ring is finite and how many elements it has, cannot be answered by GAP. Clearly we know the answers, and we could store them in the ring, by setting the value of the property `IsFinite` to `true` and the value of the attribute `Size` to n (the argument of the call to `MyZmodnZ`). If we do not want to do so then GAP could only try to find out the number of elements of the ring via forming the closure of the generators under addition and multiplication, but up to now, GAP does not know how to add or multiply two elements of our ring.

So we must install some methods for arithmetic and other operations if the elements are to behave as we want.

We start with a method for showing elements nicely on the screen. There are different operations for this purpose. One of them is `PrintObj`, which is called for each argument in an explicit call to `Print`. Another one is `ViewObj`, which is called in the read-eval-print loop for each object. `ViewObj` shall produce short and human readable information about the object in question, whereas `PrintObj` shall produce information that may be longer and is (if reasonable) readable by GAP. We cannot satisfy the latter requirement for a `PrintObj` method because there is no way to make a family GAP readable. So we decide to display the expression `( k mod n )` for an object that is given by the residue `k` and the modulus `n`, which would be fine as a `ViewObj` method. Since the default for `ViewObj` is to call `PrintObj`, and since no other `ViewObj` method is applicable to our elements, we need only a `PrintObj` method.

```
InstallMethod( PrintObj,
"for element in Z/nZ (ModulusRep)",
[ IsMyZmodnZObj and IsMyModulusRep ],
function( x )
Print( "( ", x![1], " mod ", FamilyObj(x)!.modulus, " )" );
end );
```

So we installed a method for the operation `PrintObj` (first argument), and we gave it a suitable information message (second argument), see 7.2.1 and 7.3 for applications of this information string. The third argument tells GAP that the method is applicable for objects that lie in the category `IsMyZmodnZObj` and in the representation `IsMyModulusRep`. and the fourth argument is the method itself. More details about `InstallMethod` can be found in 2.2.

Note that the requirement `IsMyModulusRep` for the argument `x` allows us to access the residue as `x![1]`. Since the family of `x` has the component `modulus` bound if it is constructed by `MyZmodnZ`, we may access this component. We check whether the method installation has some effect.

```
gap> gens;
[ ( 1 mod 4 ) ]
```

Next we install methods for the comparison operations. Note that we can assume that the residues in the representation chosen are normalized.

```
InstallMethod( \=,
    "for two elements in Z/nZ (ModulusRep)",
    IsIdenticalObj,
    [ IsMyZmodnZObj and IsMyModulusRep, IsMyZmodnZObj and IsMyModulusRep ],
    function( x, y ) return x![1] = y![1]; end );

InstallMethod( \<,
    "for two elements in Z/nZ (ModulusRep)",
    IsIdenticalObj,
    [ IsMyZmodnZObj and IsMyModulusRep, IsMyZmodnZObj and IsMyModulusRep ],
    function( x, y ) return x![1] < y![1]; end );
```

The third argument used in these installations specifies the required relation between the families of the arguments (see 13.1 in the Reference Manual). This argument of a method installation, if present, is a function that shall be applied to the families of the arguments. `IsIdenticalObj` means that the methods are applicable only if both arguments lie in the same family. (In installations for unary methods, obviously no relation is required, so this argument is left out there.)

Up to now, we see no advantage of the new approach over the one in Section 5.1. For a residue class represented as `[` k`,` n `]`, the way it is printed on the screen is sufficient, and equality and comparison of lists are good enough to define equality and comparison of residue classes if needed. But this is not the case in other situations. For example, if we would have decided that the residue k need not be normalized then we would have needed functions in Section 5.1 that compute whether two residue classes are equal, and which of two residue classes is regarded as larger than another. Note that we are free to define what "larger" means for objects that are newly introduced.

Next we install methods for the arithmetic operations, first for the additive structure.

```
InstallMethod( \+,
    "for two elements in Z/nZ (ModulusRep)",
    IsIdenticalObj,
    [ IsMyZmodnZObj and IsMyModulusRep, IsMyZmodnZObj and IsMyModulusRep ],
    function( x, y )
    return MyZmodnZObj( FamilyObj( x ), x![1] + y![1] );
    end );

InstallMethod( ZeroOp,
    "for element in Z/nZ (ModulusRep)",
    [ IsMyZmodnZObj ],
    x -> MyZmodnZObj( FamilyObj( x ), 0 ) );

InstallMethod( AdditiveInverseOp,
    "for element in Z/nZ (ModulusRep)",
    [ IsMyZmodnZObj and IsMyModulusRep ],
    x -> MyZmodnZObj( FamilyObj( x ), AdditiveInverse( x![1] ) ) );
```

Here the new approach starts to pay off. The method for the operation `\+` allows us to use the infix operator `+` for residue classes. The method for `ZeroOp` is used when we call this operation or the attribute `Zero` explicitly, and `ZeroOp` it is also used when we ask for `0 *` *rescl*, where *rescl* is a residue class.

(Note that `Zero` and `ZeroOp` are distinguished because `0 *` *obj* is guaranteed to return a **mutable** result whenever a mutable version of this result exists in GAP –for example if *obj* is a matrix– whereas `Zero` is an attribute and therefore returns **immutable** results; for our example there is no difference since the residue classes are always immutable, nevertheless we have to install the method for `ZeroOp`. The same holds for `AdditiveInverse`, `One`, and `Inverse`.)

Similarly, `AdditiveInverseOp` can be either called directly or via the unary `-` operator; so we can compute the additive inverse of the residue class *rescl* as `-`*rescl*.

It is not necessary to install methods for subtraction, since this is handled via addition of the additive inverse of the second argument if no other method is installed.

Let us try what we can do with the methods that are available now.

```
gap> x:= gens[1];  y:= x + x;
( 1 mod 4 )
( 2 mod 4 )
gap> 0 * x;  -x;
( 0 mod 4 )
( 3 mod 4 )
gap> y = -y;  x = y;  x < y;  -x < y;
true
false
true
false
```

We might want to admit the addition of integers and elements in rings $\mathbb{Z}/n\mathbb{Z}$, where an integer is implicitly identified with its residue modulo n. To achieve this, we install methods to add an integer to an object in `IsMyZmodnZObj` from the left and from the right.

```
InstallMethod( \+,
    "for element in Z/nZ (ModulusRep) and integer",
    [ IsMyZmodnZObj and IsMyModulusRep, IsInt ],
    function( x, y )
    return MyZmodnZObj( FamilyObj( x ), x![1] + y );
    end );

InstallMethod( \+,
    "for integer and element in Z/nZ (ModulusRep)",
    [ IsInt, IsMyZmodnZObj and IsMyModulusRep ],
    function( x, y )
    return MyZmodnZObj( FamilyObj( y ), x + y![1] );
    end );
```

Now we can do also the following.

```
gap> 2 + x;  7 - x;  y - 2;
( 3 mod 4 )
( 2 mod 4 )
( 0 mod 4 )
```

Similarly we install the methods dealing with the multiplicative structure. We need methods to multiply two of our objects, and to compute identity and inverse. The operation `OneOp` is called when we ask for *rescl*`^0`, and `InverseOp` is called when we ask for *rescl*`^-1`. Note that the method for `InverseOp` returns `fail` if the argument is not invertible.

```
InstallMethod( \*,
    "for two elements in Z/nZ (ModulusRep)",
    IsIdenticalObj,
    [ IsMyZmodnZObj and IsMyModulusRep, IsMyZmodnZObj and IsMyModulusRep ],
    function( x, y )
    return MyZmodnZObj( FamilyObj( x ), x![1] * y![1] );
    end );

InstallMethod( OneOp,
    "for element in Z/nZ (ModulusRep)",
    [ IsMyZmodnZObj ],
    elm -> MyZmodnZObj( FamilyObj( elm ), 1 ) );

InstallMethod( InverseOp,
    "for element in Z/nZ (ModulusRep)",
    [ IsMyZmodnZObj and IsMyModulusRep ],
    function( elm )
    local residue;
    residue:= QuotientMod( 1, elm![1], FamilyObj( elm )!.modulus );
    if residue <> fail then
      residue:= MyZmodnZObj( FamilyObj( elm ), residue );
    fi;
    return residue;
    end );
```

To be able to multiply our objects with integers, we need not (but we may, and we should if we are going for efficiency) install special methods. This is because in general, GAP interprets the multiplication of an

integer and an additive object as abbreviation of successive additions, and there is one generic method for such a multiplication that uses only additions and —in the case of a negative integer— taking the additive inverse. Analogously, there is a generic method for powering by integers that uses only multiplications and taking the multiplicative inverse.

Note that we could also interpret the multiplication with an integer as a shorthand for the multiplication with the corresponding residue class. We are lucky that this interpretation is compatible with the one that is already available. If this would not be the case then of course we would get into trouble by installing a concurrent multiplication that computes something different from the multiplication that is already defined, since GAP does not guarantee which of the applicable methods is actually chosen (see 2.3).

Now we have implemented methods for the arithmetic operations for our elements, and the following calculations work.

```
gap> y:= 2 * x;  z:= (-5) * x;
( 2 mod 4 )
( 3 mod 4 )
gap> y * z;  y * y;
( 2 mod 4 )
( 0 mod 4 )
gap> y^-1;  y^0;
fail
( 1 mod 4 )
gap> z^-1;
( 3 mod 4 )
```

There are some other operations in GAP that we may want to accept our elements as arguments. An example is the operation `Int` that returns, e.g., the integral part of a rational number or the integer corresponding to an element in a finite prime field. For our objects, we may define that `Int` returns the normalized residue.

Note that we **define** this behaviour for elements but we **implement** it for objects in the representation `IsMyModulusRep`. This means that if someone implements another representation of residue classes then this person must be careful to implement `Int` methods for objects in this new representation compatibly with our definition, i.e., such that the result is independent of the representation.

```
InstallMethod( Int,
    "for element in Z/nZ (ModulusRep)",
    [ IsMyZmodnZObj and IsMyModulusRep ],
    z -> z![1] );
```

Another example of an operation for which we might want to install a method is `\mod`. We make the ring print itself as `Integers` mod the modulus, and then it is reasonable to allow a construction this way, which makes the `PrintObj` output of the ring GAP readable.

```
InstallMethod( PrintObj,
    "for full collection Z/nZ",
    [ CategoryCollections( IsMyZmodnZObj ) and IsWholeFamily ],
    function( R )
    Print( "(Integers mod ",
           ElementsFamily( FamilyObj(R) )!.modulus, ")" );
    end );
```

```
InstallMethod( \mod,
    "for 'Integers', and a positive integer",
    [ IsIntegers, IsPosRat and IsInt ],
    function( Integers, n ) return MyZmodnZ( n ); end );
```

Let us try this.

```
gap> Int( y );
2
gap> Integers mod 1789;
(Integers mod 1789)
```

Probably it is not necessary to emphasize that with the approach of Section 5.1, installing methods for existing operations is usually not possible or at least not recommended. For example, installing the function `resclass_sum` defined in Section 5.1 as a `\+` method for adding two lists of length two (with integer entries) would not be compatible with the general definition of the addition of two lists of same length. Installing a method for the operation `Int` that takes a list `[` k`,` n `]` and returns k would in principle be possible, since there is no `Int` method for lists yet, but it is not sensible to do so because one can think of other interpretations of such a list where different `Int` methods could be installed with the same right.

As mentioned in Section 5.2, one advantage of the new approach is that with the implementation we have up to now, automatically also matrices of residue classes can be treated.

```
gap> r:= Integers mod 16;
(Integers mod 16)
gap> x:= One( r );
( 1 mod 16 )
gap> mat:= IdentityMat( 2 ) * x;
[ [ ( 1 mod 16 ), ( 0 mod 16 ) ],
  [ ( 0 mod 16 ), ( 1 mod 16 ) ] ]
gap> mat[1][2]:= x;;
gap> mat;
[ [ ( 1 mod 16 ), ( 1 mod 16 ) ],
  [ ( 0 mod 16 ), ( 1 mod 16 ) ] ]
gap> Order( mat );
16
gap> mat + mat;
[ [ ( 2 mod 16 ), ( 2 mod 16 ) ],
  [ ( 0 mod 16 ), ( 2 mod 16 ) ] ]
gap> last^4;
[ [ ( 0 mod 16 ), ( 0 mod 16 ) ],
  [ ( 0 mod 16 ), ( 0 mod 16 ) ] ]
```

Such matrices, if they are invertible, are valid as group elements. So we can also construct matrix groups over residue class rings.

```
gap> mat2:= IdentityMat( 2 ) * x;;
gap> mat2[2][1]:= x;;
gap> g:= Group( mat, mat2 );;
gap> Size( g );
3072
gap> Factors( last );
[ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3 ]
gap> syl3:= SylowSubgroup( g, 3 );;
gap> gens:= GeneratorsOfGroup( syl3 );
```

```
[ [ [ ( 8 mod 16 ), ( 5 mod 16 ) ], [ ( 11 mod 16 ), ( 7 mod 16 ) ] ] ]
gap> Order( gens[1] );
3
```

By the way, also groups of (invertible) residue classes can be formed, but this may be of minor interest.

```
gap> g:= Group( x );  Size( g );
<group with 1 generators>
1
gap> g:= Group( 3*x );  Size( g );
<group with 1 generators>
4
```

Having done enough for the elements, we may install some more methods for the rings if we want to use them as arguments. These rings are finite, and there are many generic methods that will work if they are able to compute the list of elements of the ring, so we install a method for this.

```
InstallMethod( Enumerator,
    "for full collection Z/nZ",
    [ CategoryCollections( IsMyZmodnZObj ) and IsWholeFamily ],
    function( R )
    local F;
    F:= ElementsFamily( FamilyObj(R) );
    return List( [ 0 .. Size( R ) - 1 ], x -> MyZmodnZObj( F, x ) );
    end );
```

Note that this method is applicable only to full rings $\mathbb{Z}/n\mathbb{Z}$, for proper subrings it would return a wrong result. Furthermore, it is not required that the argument is a ring; in fact this method is applicable also to the additive group formed by all elements in the family, provided that it knows to contain the whole family.

Analogously, we install methods to compute the size, a random element, and the units of full rings $\mathbb{Z}/n\mathbb{Z}$.

```
InstallMethod( Random,
    "for full collection Z/nZ",
    [ CategoryCollections( IsMyZmodnZObj ) and IsWholeFamily ],
    R -> MyZmodnZObj( ElementsFamily( FamilyObj(R) ),
                    Random( [ 0 .. Size( R ) - 1 ] ) ) );

InstallMethod( Size,
    "for full ring Z/nZ",
    [ CategoryCollections( IsMyZmodnZObj ) and IsWholeFamily ],
    R -> ElementsFamily( FamilyObj(R) )!.modulus );

InstallMethod( Units,
    "for full ring Z/nZ",
    [     CategoryCollections( IsMyZmodnZObj )
      and IsWholeFamily and IsRing ],
    function( R )
    local F;
    F:= ElementsFamily( FamilyObj( R ) );
    return List( PrimeResidues( Size(R) ), x -> MyZmodnZObj( F, x ) );
    end );
```

The `Units` method has the disadvantage that the result is returned as a list (in fact this list is also strictly sorted). We could improve the implementation by returning the units as a group; if we do not want to take

the full list of elements as generators, we can use the function `GeneratorsPrimeResidues` (see 15.1.4 in the Reference Manual).

```
InstallMethod( Units,
    "for full ring Z/nZ",
    [     CategoryCollections( IsMyZmodnZObj )
      and IsWholeFamily and IsRing ],
    function( R )
    local G, gens;

    gens:= GeneratorsPrimeResidues( Size( R ) ).generators;
    if not IsEmpty( gens ) and gens[ 1 ] = 1 then
      gens:= gens{ [ 2 .. Length( gens ) ] };
    fi;
    gens:= Flat( gens ) * One( R );
    return GroupByGenerators( gens, One( R ) );
    end );
```

Each ring $\mathbb{Z}/n\mathbb{Z}$ is finite, and we could install a method that returns `true` when `IsFinite` is called with $\mathbb{Z}/n\mathbb{Z}$ as argument. But we can do this more elegantly via installing a **logical implication**.

```
InstallTrueMethod( IsFinite,
    CategoryCollections( IsMyZmodnZObj ) and IsDomain );
```

In effect, every domain that consists of elements in `IsMyZmodnZObj` will automatically store that it is finite, even if `IsFinite` is not called for it.

5.4 Compatibility of Residue Class Rings with Prime Fields

The above implementation of residue classes and residue class rings has at least two disadvantages. First, if p is a prime then the ring $\mathbb{Z}/p\mathbb{Z}$ is in fact a field, but the return values of `MyZmodnZ` are never regarded as fields because they are not in the category `IsMagmaWithInversesIfNonzero`. Second, and this makes the example really interesting, there are already elements of finite prime fields implemented in GAP, and we may want to identify them with elements in $\mathbb{Z}/p\mathbb{Z}$.

To be more precise, elements of finite fields in GAP lie in the category `IsFFE`, and there is already a representation, `IsInternalRep`, of these elements via discrete logarithms. The aim of this section is to make `IsMyModulusRep` an alternative representation of elements in finite prime fields.

Note that this is only one step towards the desired compatibility. Namely, after having a second representation of elements in finite prime fields, we may wish that the function `GF` (which is the usual function to create finite fields in GAP) is able to return `MyZmodnZ( p )` when `GF( p )` is called for a prime p. Moreover, then we have to decide about a default representation of elements in `GF( p )` for primes p for which both representations are available. Of course we can force the new representation by explicitly calling `MyZmodnZ` and `MyZmodnZObj` whenever we want, but it is not a priori clear in which situation which representation is preferable.

The same questions will occur when we want to implement a new representation for non-prime fields. The steps of this implementation will be the same as described in this chapter, and we will have to achieve compatibility with both the internal representation of elements in small finite fields and the representation `IsMyModulusRep` of elements in arbitrary prime fields.

But let us now turn back to the task of this section. We first adjust the setup of the declaration part of the previous section, and then repeat the installations with suitable modifications.

```
DeclareCategory( "IsMyZmodnZObj", IsScalar );
```

```
DeclareCategory( "IsMyZmodnZObjNonprime", IsMyZmodnZObj );

DeclareSynonym( "IsMyZmodpZObj", IsMyZmodnZObj and IsFFE );

DeclareRepresentation( "IsMyModulusRep", IsPositionalObjectRep, [ 1 ] );

DeclareGlobalFunction( "MyZmodnZObj" );

DeclareGlobalFunction( "MyZmodnZ" );
```

As in the previous section, all (newly introduced) elements of rings $\mathbb{Z}/n\mathbb{Z}$ lie in the category `IsMyZmodnZObj`. But now we introduce two subcategories, namely `IsMyZmodnZObjNonprime` for all elements in rings $\mathbb{Z}/n\mathbb{Z}$ where n is not a prime, and `IsMyZmodpZObj` for elements in finite prime fields. All objects in the latter are automatically known to lie in the category `IsFFE` of finite field elements.

It would be reasonable if also those internally represented elements in the category `IsFFE` that do in fact lie in a prime field would also lie in the category `IsMyZmodnZObj` (and thus in fact in `IsMyZmodpZObj`). But this cannot be achieved because internally represented finite field elements do in general not store whether they lie in a prime field.

As for the implementation part, again let us start with the definitions of `MyZmodnZObj` and `MyZmodnZ`.

```
InstallGlobalFunction( MyZmodnZObj, function( Fam, residue )
if IsFFEFamily( Fam ) then
  return Objectify( NewType( Fam,     IsMyZmodpZObj
                                  and IsMyModulusRep ),
                    [ residue mod Characteristic( Fam ) ] );
else
  return Objectify( NewType( Fam,     IsMyZmodnZObjNonprime
                                  and IsMyModulusRep ),
                    [ residue mod Fam!.modulus ] );
fi;
end );


InstallGlobalFunction( MyZmodnZ, function( n )

    local F, R;

    if not ( IsInt( n ) and IsPosRat( n ) ) then
      Error( "<n> must be a positive integer" );

    elif IsPrimeInt( n ) then

      # Construct the family of element objects of our field.
      F:= FFEFamily( n );

      # Make the domain.
      R:= FieldOverItselfByGenerators( [ MyZmodnZObj( F, 1 ) ] );
      SetIsPrimeField( R, true );

    else
```

```
      # Construct the family of element objects of our ring.
      F:= NewFamily( Concatenation( "MyZmod", String( n ), "Z" ),
                     IsMyZmodnZObjNonprime );

      # Install the data.
      F!.modulus:= n;

      # Make the domain.
      R:= RingWithOneByGenerators( [ MyZmodnZObj( F, 1 ) ] );
      SetIsWholeFamily( R, true );
      SetName( R, Concatenation( "(Integers mod ",String(n),")" ) );

    fi;

    # Return the ring resp. field.
    return R;
end );
```

Note that the result of `MyZmodnZ` with a prime as argument is a field that does not contain the whole family of its elements, since all finite field elements of a fixed characteristic lie in the same family. Further note that we cannot expect a family of finite field elements to have a component `modulus`, so we use `Characteristic` to get the modulus. Requiring that `Fam!.modulus` works also if `Fam` is a family of finite field elements would violate the rule that an extension of GAP should not force changes in existing code, in this case code dealing with families of finite field elements.

```
InstallMethod( PrintObj,
    "for element in Z/nZ (ModulusRep)",
    [ IsMyZmodnZObjNonprime and IsMyModulusRep ],
    function( x )
    Print( "( ", x![1], " mod ", FamilyObj(x)!.modulus, " )" );
    end );

InstallMethod( PrintObj,
    "for element in Z/pZ (ModulusRep)",
    [ IsMyZmodpZObj and IsMyModulusRep ],
    function( x )
    Print( "( ", x![1], " mod ", Characteristic(x), " )" );
    end );

InstallMethod( \=,
    "for two elements in Z/nZ (ModulusRep)",
    IsIdenticalObj,
    [ IsMyZmodnZObj and IsMyModulusRep,
      IsMyZmodnZObj and IsMyModulusRep ],
    function( x, y ) return x![1] = y![1]; end );
```

The above method to check equality is independent of whether the arguments have a prime or nonprime modulus, so we installed it for arguments in `IsMyZmodnZObj`. Now we install also methods to compare objects in `IsMyZmodpZObj` with the "old" finite field elements.

```
InstallMethod( \=,
    "for element in Z/pZ (ModulusRep) and internal FFE",
    IsIdenticalObj,
    [ IsMyZmodpZObj and IsMyModulusRep, IsFFE and IsInternalRep ],
    function( x, y )
    return DegreeFFE( y ) = 1 and x![1] = IntFFE( y );
    end );

InstallMethod( \=,
    "for internal FFE and element in Z/pZ (ModulusRep)",
    IsIdenticalObj,
    [ IsFFE and IsInternalRep, IsMyZmodpZObj and IsMyModulusRep ],
    function( x, y )
    return DegreeFFE( x ) = 1 and IntFFE( x ) = y![1];
    end );
```

The situation with the operation `\<` is more difficult. Of course we are free to define the comparison of objects in `IsMyZmodnZObjNonprime`, but for the finite field elements, the comparison must be compatible with the predefined comparison of the "old" finite field elements. The definition of the `\<` comparison of internally represented finite field elements can be found in Chapter 57 in the Reference Manual. In situations where the documentation does not provide the required information, one has to look it up in the GAP code; for example, the comparison in our case can be found in the appropriate source code file of the GAP kernel.

```
InstallMethod( \<,
    "for two elements in Z/nZ (ModulusRep, nonprime)",
    IsIdenticalObj,
    [ IsMyZmodnZObjNonprime and IsMyModulusRep,
      IsMyZmodnZObjNonprime and IsMyModulusRep ],
    function( x, y ) return x![1] < y![1]; end );

InstallMethod( \<,
    "for two elements in Z/pZ (ModulusRep)",
    IsIdenticalObj,
    [ IsMyZmodpZObj and IsMyModulusRep,
      IsMyZmodpZObj and IsMyModulusRep ],
    function( x, y )
    local p, r;      # characteristic and primitive root
    if x![1] = 0 then
      return y![1] <> 0;
    elif y![1] = 0 then
      return false;
    else
      p:= Characteristic( x );
      r:= PrimitiveRootMod( p );
      return LogMod( x![1], r, p ) < LogMod( y![1], r, p );
    fi;
    end );

InstallMethod( \<,
    "for element in Z/pZ (ModulusRep) and internal FFE",
    IsIdenticalObj,
    [ IsMyZmodpZObj and IsMyModulusRep, IsFFE and IsInternalRep ],
```

```
    function( x, y )
    return x![1] * One( y ) < y;
    end );

InstallMethod( \<,
    "for internal FFE and element in Z/pZ (ModulusRep)",
    IsIdenticalObj,
    [ IsFFE and IsInternalRep, IsMyZmodpZObj and IsMyModulusRep ],
    function( x, y )
    return x < y![1] * One( x );
    end );
```

Now we install the same methods for the arithmetic operations `\+`, `ZeroOp`, `AdditiveInverseOp`, `\-`, `\*`, and `OneOp` as in the previous section, without listing them below. Also the same `Int` method is installed for objects in `IsMyZmodnZObj`. Note that it is compatible with the definition of `Int` for finite field elements. And of course the same method for `\mod` is installed.

We have to be careful, however, with the methods for `InverseOp`, `\/`, and `\^`. These methods and the missing methods for arithmetic operations with one argument in `IsMyModulusRep` and the other in `IsInternalRep` are given below.

```
InstallMethod( \+,
    "for element in Z/pZ (ModulusRep) and internal FFE",
    IsIdenticalObj,
    [ IsMyZmodpZObj and IsMyModulusRep, IsFFE and IsInternalRep ],
    function( x, y ) return x![1] + y; end );

InstallMethod( \+,
    "for internal FFE and element in Z/pZ (ModulusRep)",
    IsIdenticalObj,
    [ IsFFE and IsInternalRep, IsMyZmodpZObj and IsMyModulusRep ],
    function( x, y ) return x + y![1]; end );

InstallMethod( \*,
    "for element in Z/pZ (ModulusRep) and internal FFE",
    IsIdenticalObj,
    [ IsMyZmodpZObj and IsMyModulusRep, IsFFE and IsInternalRep ],
    function( x, y ) return x![1] * y; end );

InstallMethod( \*,
    "for internal FFE and element in Z/pZ (ModulusRep)",
    IsIdenticalObj,
    [ IsFFE and IsInternalRep, IsMyZmodpZObj and IsMyModulusRep ],
    function( x, y ) return x * y![1]; end );

InstallMethod( InverseOp,
    "for element in Z/nZ (ModulusRep, nonprime)",
    [ IsMyZmodnZObjNonprime and IsMyModulusRep ],
    function( x )
    local residue;
    residue:= QuotientMod( 1, x![1], FamilyObj(x)!.modulus );
    if residue <> fail then
      residue:= MyZmodnZObj( FamilyObj(x), residue );
```

```
    fi;
    return residue;
    end );

InstallMethod( InverseOp,
    "for element in Z/pZ (ModulusRep)",
    [ IsMyZmodpZObj and IsMyModulusRep ],
    function( x )
    local residue;
    residue:= QuotientMod( 1, x![1], Characteristic( FamilyObj(x) ) );
    if residue <> fail then
      residue:= MyZmodnZObj( FamilyObj(x), residue );
    fi;
    return residue;
    end );
```

The operation `DegreeFFE` is defined for finite field elements, we need a method for objects in `IsMyZmodpZObj`. Note that we need not require `IsMyModulusRep` since no access to representation dependent data occurs.

```
InstallMethod( DegreeFFE,
    "for element in Z/pZ",
    [ IsMyZmodpZObj ],
    z -> 1 );
```

The methods for `Enumerator`, `Random`, `Size`, and `Units`, that we had installed in the previous section had all assumed that their argument contains the whole family of its elements. So these methods make sense only for the nonprime case. For the prime case, there are already methods for these operations with argument a field.

```
InstallMethod( Enumerator,
    "for full ring Z/nZ",
    [ CategoryCollections( IsMyZmodnZObjNonprime ) and IsWholeFamily ],
    function( R )
    local F;
    F:= ElementsFamily( FamilyObj( R ) );
    return List( [ 0 .. Size( R ) - 1 ], x -> MyZmodnZObj( F, x ) );
    end );

InstallMethod( Random,
    "for full ring Z/nZ",
    [ CategoryCollections( IsMyZmodnZObjNonprime ) and IsWholeFamily ],
    R -> MyZmodnZObj( ElementsFamily( FamilyObj( R ) ),
                   Random( [ 0 .. Size( R ) - 1 ] ) ) );

InstallMethod( Size,
    "for full ring Z/nZ",
    [ CategoryCollections( IsMyZmodnZObjNonprime ) and IsWholeFamily ],
    R -> ElementsFamily( FamilyObj( R ) )!.modulus );
```

```
InstallMethod( Units,
    "for full ring Z/nZ",
    [     CategoryCollections( IsMyZmodnZObjNonprime )
      and IsWholeFamily and IsRing ],
    function( R )
    local G, gens;

    gens:= GeneratorsPrimeResidues( Size( R ) ).generators;
    if not IsEmpty( gens ) and gens[ 1 ] = 1 then
      gens:= gens{ [ 2 .. Length( gens ) ] };
    fi;
    gens:= Flat( gens ) * One( R );
    return GroupByGenerators( gens, One( R ) );
    end );

InstallTrueMethod( IsFinite,
    CategoryCollections( IsMyZmodnZObjNonprime ) and IsDomain );
```

5.5 Further Improvements in Implementing Residue Class Rings

There are of course many possibilities to improve the implementation.

With the setup as described above, subsequent calls `MyZmodnZ( n )` with the same n yield incompatible rings in the sense that elements of one ring cannot be added to elements of an other one. The solution for this problem is to keep a global list of all results of `MyZmodnZ` in the current GAP session, and to return the stored values whenever possible. Note that this approach would admit `PrintObj` methods that produce GAP readable output.

One can improve the `Units` method for the full ring in such a way that a group is returned and not only a list of its elements; then the result of `Units` can be used, e. g., as input for the operation `SylowSubgroup`.

To make computations more efficient, one can install methods for `\-`, `\/`, and `\^`; one reason for doing so may be that this avoids the unnecessary construction of the additive or multiplicative inverse, or of intermediate powers.

```
InstallMethod( \-, "two elements in Z/nZ (ModulusRep)", ... );
InstallMethod( \-, "Z/nZ-obj. (ModulusRep) and integer", ... );
InstallMethod( \-, "integer and Z/nZ-obj. (ModulusRep)", ... );
InstallMethod( \-, "Z/pZ-obj. (ModulusRep) and internal FFE", ... );
InstallMethod( \-, "internal FFE and Z/pZ-obj. (ModulusRep)", ... );
InstallMethod( \*, "Z/nZ-obj. (ModulusRep) and integer", ... );
InstallMethod( \*, "integer and Z/nZ-obj. (ModulusRep)", ... );
InstallMethod( \/, "two Z/nZ-objs. (ModulusRep, nonprime)", ... );
InstallMethod( \/, "two Z/pZ-objs. (ModulusRep)", ... );
InstallMethod( \/, "Z/nZ-obj. (ModulusRep) and integer", ... );
InstallMethod( \/, "integer and Z/nZ-obj. (ModulusRep)", ... );
InstallMethod( \/, "Z/pZ-obj. (ModulusRep) and internal FFE", ... );
InstallMethod( \/, "internal FFE and Z/pZ-obj. (ModulusRep)", ... );
InstallMethod( \^, "Z/nZ-obj. (ModulusRep, nonprime) & int.", ... );
InstallMethod( \^, "Z/pZ-obj. (ModulusRep), and integer", ... );
```

The call to `NewType` in `MyZmodnZObj` can be avoided by storing the required type, e.g., in the family. But note that it is **not** admissible to take the type of an existing object as first argument of `Objectify`. For example, suppose two objects in `IsMyZmodnZObj` shall be added. Then we must not use the type of one of

the arguments in a call of `Objectify`, because the argument may have knowledge that is not correct for the result of the addition. One may think of the property `IsOne` that may hold for both arguments but certainly not for their sum.

For comparing two objects in `IsMyZmodpZObj` via "`<`", we had to install a quite expensive method because of the compatibility with the comparison of finite field elements that did already exist. In fact GAP supports finite fields with elements represented via discrete logarithms only up to a given size. So in principle we have the freedom to define a cheaper comparison via "`<`" for objects in `IsMyZmodpZObj` if the modulus is large enough. This is possible by introducing two categories `IsMyZmodpZObjSmall` and `IsMyZmodpZObjLarge`, which are subcategories of `IsMyZmodpZObj`, and to install different `\<` methods for pairs of objects in these categories.

6 An Example – Designing Arithmetic Operations

In this chapter, we give a –hopefully typical– example of extending GAP by new objects with prescribed arithmetic operations.

6.1 New Arithmetic Operations vs. New Objects

A usual procedure in mathematics is the definition of new operations for given objects; here are a few typical examples. The Lie bracket defines an interesting new multiplicative structure on a given (associative) algebra. Forming a group ring can be viewed as defining a new addition for the elements of the given group, and extending the multiplication to sums of group elements in a natural way. Forming the exterior algebra of a given vector space can be viewed as defining a new multiplication for the vectors in a natural way.

GAP does **not** support such a procedure. The main reason for this is that in GAP, the multiplication in a group, a ring etc. is always written as `*`, and the addition in a vector space, a ring etc. is always written as `+`. Therefore it is not possible to define the Lie bracket as a "second multiplication" for the elements of a given algebra; in fact, the multiplication in Lie algebras in GAP is denoted by `*`. Analogously, constructing the group ring as sketched above is impossible if an addition is already defined for the elements; note the difference between the usual addition of matrices and the addition in the group ring of a matrix group! (See Chapter 63 in the Reference Manual for an example.) Similarly, there is already a multiplication defined for row vectors (yielding the standard scalar product), hence these vectors cannot be regarded as elements of the exterior algebra of the space.

In situations such as the ones mentioned above, GAP's way to deal with the structures in question is the following. Instead of defining **new** operations for the **given** objects, **new** objects are created to which the **given** arithmetic operations `*` and `+` are then made applicable.

With this construction, matrix Lie algebras consist of matrices that are different from the matrices with associative multiplication; technically, the type of a matrix determines how it is multiplied with other matrices (see 24.1.1 in the Reference Manual). A matrix with the Lie bracket as its multiplication can be created with the function `LieObject` from a matrix with the usual associative multiplication.

Group rings (more general: magma rings, see Chapter 63 in the Reference Manual) can be constructed with `FreeMagmaRing` from a coefficient ring and a group. The elements of the group are not contained in such a group ring, one has to use an embedding map for creating a group ring element that corresponds to a given group element.

It should be noted that the GAP approach to the construction of Lie algebras from associative algebras is generic in the sense that all objects in the filter `IsLieObject` use the same methods for their addition, multiplication etc., by delegating to the "underlying" objects of the associative algebra, no matter what these objects actually are. Analogously, also the construction of group rings is generic.

6.2 Designing new Multiplicative Objects

The goal of this section is to implement objects with a prescribed multiplication. Let us assume that we are given a field F, and that we want to define a new multiplication $*$ on F that is given by $a*b = ab-a-b+2$; here ab denotes the ordinary product in F.

By the discussion in Section 6.1, we know that we cannot define a new multiplication on F itself but have to create new objects.

We want to distinguish these new objects from all other GAP objects, in order to describe for example the situation that two of our objects shall be multiplied. This distinction is made via the **type** of the objects. More precisely, we declare a new **filter**, a function that will return `true` for our new objects, and `false` for all other GAP objects. This can be done by calling `DeclareFilter` (see 3.15.2), but since our objects will know about the value already when they are constructed, the filter can be created with `DeclareCategory` (see 3.15.2 and 3.1.1).

```
DeclareCategory( "IsMyObject", IsObject );
```

The idea is that the new multiplication will be installed only for objects that "lie in the category `IsMyObject`".

The next question is what internal data our new objects store, and how they are accessed. The easiest solution is to store the "underlying" object from the field F. GAP provides two general possibilities how to store this, namely record-like and list-like structures (for examples, see 3.9 and 3.10). We decide to store the data in a list-like structure, at position 1. This **representation** is declared as follows.

```
DeclareRepresentation( "IsMyObjectListRep", IsPositionalObjectRep, [ 1 ] );
```

Of course we can argue that this declaration is superfluous because **all** objects in the category `IsMyObject` will be represented this way; it is possible to proceed like that, but often (in more complicated situations) it turns out to be useful that several representations are available for "the same element".

For creating the type of our objects, we need to specify to which **family** (see 13.1 in the Reference Manual) the objects shall belong. For the moment, we need not say anything about relations to other GAP objects, thus the only requirement is that all new objects lie in the **same** family; therefore we create a **new** family. Also we are not interested in properties that some of our objects have and others do not have, thus we need only one type, and store it in a global variable.

```
MyType:= NewType( NewFamily( "MyFamily" ),
                  IsMyObject and IsMyObjectListRep );
```

The next step is to write a function that creates a new object. It may look as follows.

```
MyObject:= val -> Objectify( MyType, [ Immutable( val ) ] );
```

Note that we store an **immutable copy** of the argument in the returned object; without doing so, for example if the argument would be a mutable matrix then the corresponding new object would be changed whenever the matrix is changed (see 12.6 in the Reference Manual for more details about mutability).

Having entered the above GAP code, we can create some of our objects.

```
gap> a:= MyObject( 3 );  b:= MyObject( 5 );
<object>
<object>
gap> a![1];  b![1];
3
5
```

But clearly a lot is missing. Besides the fact that the desired multiplication is not yet installed, we see that also the way how the objects are printed is not satisfactory.

Let us improve the latter first. There are two GAP functions `View` and `Print` for showing objects on the screen. `View` is thought to show a short and human readable form of the object, and `Print` is thought to show a not necessarily short form that is GAP readable whenever this makes sense. We decide to show `a` as *3* by `View`, and to show the construction `MyObject( 3 )` by `Print`; the methods are installed for the underlying operations `ViewObj` and `PrintObj`.

```
InstallMethod( ViewObj,
    "for object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    function( obj )
    Print( "<", obj![1], ">" );
    end );

InstallMethod( PrintObj,
    "for object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    function( obj )
    Print( "MyObject( ", obj![1], " )" );
    end );
```

This is the result of the above installations.

```
gap> a; Print( a, "\n" );
<3>
MyObject( 3 )
```

And now we try to install the multiplication.

```
InstallMethod( \*,
    "for two objects in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep,
      IsMyObject and IsMyObjectListRep ],
    function( a, b )
    return MyObject( a![1] * b![1] - a![1] - b![1] + 2 );
    end );
```

When we enter the above code, GAP runs into an error. This is due to the fact that the operation `\*` is declared for two arguments that lie in the category `IsMultiplicativeElement`. One could circumvent the check whether the method matches the declaration of the operation, by calling `InstallOtherMethod` instead of `InstallMethod`. But it would make sense if our objects would lie in `IsMultiplicativeElement`, for example because some generic methods for objects with multiplication would be available then, such as powering by positive integers via repeated squaring. So we want that `IsMyObject` implies `IsMultiplicativeElement`. The easiest way to achieve such implications is to use the implied filter as second argument of the `DeclareCategory` call; but since we do not want to start anew, we can also install the implication afterwards.

```
InstallTrueMethod( IsMultiplicativeElement, IsMyObject );
```

Afterwards, installing the multiplication works without problems. Note that `MyType` and therefore also `a` and `b` are **not** affected by this implication, so we construct them anew.

```
gap> MyType:= NewType( NewFamily( "MyFamily" ),
>                      IsMyObject and IsMyObjectListRep );;
gap> a:= MyObject( 3 );;  b:= MyObject( 5 );;
gap> a*b;  a^27;
<9>
<134217729>
```

Powering the new objects by negative integers is not possible yet, because GAP does not know how to compute the inverse of an element a, say, which is defined as the unique element a' such that both aa' and $a'a$ are "the unique multiplicative neutral element that belongs to a".

And also this neutral element, if it exists, cannot be computed by GAP in our current situation. It does, however, make sense to ask for the multiplicative neutral element of a given magma, and for inverses of elements in the magma.

But before we can form domains of our objects, we must define when two objects are regarded as equal; note that this is necessary in order to decide about the uniqueness of neutral and inverse elements. In our situation, equality is defined in the obvious way. For being able to form sets of our objects, also an ordering via `\<` is defined for them.

```
InstallMethod( \=,
    "for two objects in 'IsMyObject'",
    [ IsMyObject and IsMyObjectListRep,
      IsMyObject and IsMyObjectListRep ],
    function( a, b )
    return a![1] = b![1];
    end );

InstallMethod( \<,
    "for two objects in 'IsMyObject'",
    [ IsMyObject and IsMyObjectListRep,
      IsMyObject and IsMyObjectListRep ],
    function( a, b )
    return a![1] < b![1];
    end );
```

Let us look at an example. We start with finite field elements because then the domains are finite, hence the generic methods for such domains will have a chance to succeed.

```
gap> a:= MyObject( Z(7) );
<Z(7)>
gap> m:= Magma( a );
<magma with 1 generators>
gap> e:= MultiplicativeNeutralElement( m );
<Z(7)^2>
gap> elms:= AsList( m );
[ <Z(7)>, <Z(7)^2>, <Z(7)^5> ]
gap> ForAll( elms, x -> ForAny( elms, y -> x*y = e and y*x = e ) );
true
gap> List( elms, x -> First( elms, y -> x*y = e and y*x = e ) );
[ <Z(7)^5>, <Z(7)^2>, <Z(7)> ]
```

So a multiplicative neutral element exists, in fact all elements in the magma `m` are invertible. But what about the following.

```
gap> b:= MyObject( Z(7)^0 );  m:= Magma( a, b );
<Z(7)^0>
<magma with 2 generators>
gap> elms:= AsList( m );
[ <Z(7)^0>, <Z(7)>, <Z(7)^2>, <Z(7)^5> ]
gap> e:= MultiplicativeNeutralElement( m );
<Z(7)^2>
gap> ForAll( elms, x -> ForAny( elms, y -> x*y = e and y*x = e ) );
false
gap> List( elms, x -> b * x );
[ <Z(7)^0>, <Z(7)^0>, <Z(7)^0>, <Z(7)^0> ]
```

Here we found a multiplicative neutral element, but the element `b` does not have an inverse. If an addition would be defined for our elements then we would say that `b` behaves like a zero element.

When we started to implement the new objects, we said that we wanted to define the new multiplication for elements of a given field F. In principle, the current implementation would admit also something like `MyObject( 2 ) * MyObject( Z(7) )`. But if we decide that our initial assumption holds, we may define the identity and the inverse of the object `<a>` as `<2*e>` and `<a/(a-e)>`, respectively, where `e` is the identity element in F and `/` denotes the division in F; note that the element `<e>` is not invertible, and that the above definitions are determined by the multiplication defined for our objects. Further note that after the installations shown below, also `One( MyObject( 1 ) )` is defined.

(For technical reasons, we do not install the intended methods for the attributes `One` and `Inverse` but for the operations `OneOp` and `InverseOp`. This is because for certain kinds of objects –mainly matrices– one wants to support a method to compute a **mutable** identity or inverse, and the attribute needs only a method that takes this object, makes it immutable, and then returns this object. As stated above, we only want to deal with immutable objects, so this distinction is not really interesting for us.)

A more interesting point to note is that we should mark our objects as likely to be invertible, since we add the possibility to invert them. Again, this could have been part of the declaration of `IsMyObject`, but we may also formulate an implication for the existing category.

```
InstallTrueMethod( IsMultiplicativeElementWithInverse, IsMyObject );

InstallMethod( OneOp,
    "for an object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    a -> MyObject( 2 * One( a![1] ) ) );

InstallMethod( InverseOp,
    "for an object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    a -> MyObject( a![1] / ( a![1] - One( a![1] ) ) ) );
```

Now we can form groups of our (nonzero) elements.

```
gap> MyType:= NewType( NewFamily( "MyFamily" ),
>                   IsMyObject and IsMyObjectListRep );;
gap>
gap> a:= MyObject( Z(7) );
<Z(7)>
gap> b:= MyObject( 0*Z(7) );  g:= Group( a, b );
<0*Z(7)>
<group with 2 generators>
gap> Size( g );
```

```
6
```

We are completely free to define an **addition** for our elements, a natural one is given by `<a> + <b> = <a+b-1>`. As we did for the multiplication, we first change `IsMyObject` such that the additive structure is also known.

```
InstallTrueMethod( IsAdditiveElementWithInverse, IsMyObject );
```

Next we install the methods for the addition, and those to compute the additive neutral element and the additive inverse.

```
InstallMethod( \+,
    "for two objects in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep,
      IsMyObject and IsMyObjectListRep ],
    function( a, b )
    return MyObject( a![1] + b![1] - 1 );
    end );

InstallMethod( ZeroOp,
    "for an object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    a -> MyObject( One( a![1] ) ) );

InstallMethod( AdditiveInverseOp,
    "for an object in ‘IsMyObject’",
    [ IsMyObject and IsMyObjectListRep ],
    a -> MyObject( a![1] / ( a![1] - One( a![1] ) ) ) );
```

Let us try whether the addition works.

```
gap> MyType:= NewType( NewFamily( "MyFamily" ),
>                   IsMyObject and IsMyObjectListRep );;
gap> a:= MyObject( Z(7) );;  b:= MyObject( 0*Z(7) );;
gap> m:= AdditiveMagma( a, b );
<additive magma with 2 generators>
gap> Size( m );
7
```

Similar as installing a multiplication automatically makes powering by integers available, multiplication with integers becomes available with the addition.

```
gap> 2 * a;
<Z(7)^5>
gap> a+a;
<Z(7)^5>
gap> MyObject( 2*Z(7)^0 ) * a;
<Z(7)>
```

In particular we see that this multiplication does **not** coincide with the multiplication of two of our objects, that is, an integer **cannot** be used as a shorthand for one of the new objects in a multiplication.

(It should be possible to create a **field** with the new multiplication and addition. Currently this fails, due to missing methods for computing several kinds of generators from field generators, for computing the characteristic in the case that the family does not know this in advance, for checking with `AsField` whether a domain is in fact a field, for computing the closure as a field.)

It should be emphasized that the mechanism described above may be not suitable for the situation that one wants to consider many different multiplications "on the same set of objects", since the installation of a new multiplication requires the declaration of at least one new filter and the installation of several methods. But the design of GAP is not suitable for such dynamic method installations.

Turning this argument the other way round, the implementation of the new arithmetics defined by the above multiplication and addition is available for any field F, one need not repeat it for each field one is interested in.

Similar to the above situation, the construction of a magma ring RM from a coefficient ring R and a magma M is implemented only once, since the definition of the arithmetic operations depends only on the given multiplication of M and not on M itself. So the addition is not implemented for the elements in M or –more precisely– for an isomorphic copy. In some sense, the addition is installed "for the multiplication", and as mentioned in Section 6.1, there is only one multiplication `\*` in GAP.

Bibliography

[Isa76] I. M. Isaacs. *Character theory of finite groups*, volume 69 of *Pure and applied mathematics*. Academic Press, New York, 1976. xii+303 pp., ISBN 0-12-374550-0.

[LRW97] Eugene M. Luks, Ferenc Rákóczi, and Charles R. B. Wright. Some algorithms for nilpotent permutation groups. *J. Symbolic Comput.*, 23(4):335–354, 1997.

[Wie69] Helmut Wielandt. Permutation groups through invariant relations and invariant functions. Lecture notes, Department of Mathematics, The Ohio State University, 1969.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "`PermutationCharacter`" comes before "permutation group".