AtlasRep — A GAP 4 Package

(Version 2.1.6)

Robert A. Wilson
Richard A. Parker
Simon Nickerson
John N. Bray
Thomas Breuer

Robert A. Wilson Email: R.A.Wilson@qmul.ac.uk
Homepage: http://www.maths.qmw.ac.uk/~raw

Richard A. Parker Email: richpark@gmx.co.uk

Simon Nickerson
Homepage: http://nickerson.org.uk/groups

John N. Bray Email: J.N.Bray@qmul.ac.uk
Homepage: http://www.maths.qmw.ac.uk/~jnb

Thomas Breuer Email: sam@Math.RWTH-Aachen.De
Homepage: https://www.math.rwth-aachen.de/~Thomas.Breuer

Copyright

Contents

Chapter 1

Introduction to the AtlasRep Package

The aim of the GAP 4 package AtlasRep is to provide a link between GAP and databases such as the ATLAS of Group Representations [WWT+], which comprises generating permutations and matrices for many almost simple groups, and information about their maximal subgroups. This database is available independent of GAP at

`http://atlas.math.rwth-aachen.de/Atlas/v3`.

The AtlasRep package consists of this database (see Section 1.1) and a GAP interface (see Section 1.2); the latter is extended by further information available via the internet (see Section 4.3).

This package manual has the following parts.

A tutorial
gives an overview how the functions of the package can be used, see Chapter 2.

User interface functions
are described in Chapter 3.

Customizations of the package
are described in Chapter 4.

Information how to extend the database
can be found in Chapter 5.

More technical information
can be found in the chapters 6 (concerning GAP objects that are introduced by the package) and 7 (concerning global variables and sanity checks).

1.1 The ATLAS of Group Representations

The ATLAS of Group Representations [WWT+] consists of matrices over various rings, permutations, and shell scripts encoding so-called black box programs (see [Nic06] and Section 6.2). Many of these scripts are straight line programs (see [BSWW01], [SWW00], and (**Reference: Straight Line Programs**)) and straight line decisions (see Section 6.1). These programs can be used to compute certain elements in a group G from its standard generators (see [Wil96] and Section 3.3) for example generators of maximal subgroups of G or representatives of conjugacy classes of G.

The ATLAS of Group Representations has been prepared by Robert Wilson, Peter Walsh, Jonathan Tripp, Ibrahim Suleiman, Richard Parker, Simon Norton, Simon Nickerson, Steve Linton, John Bray, and Rachel Abbott (in reverse alphabetical order).

The information was computed and composed using computer algebra systems such as MeatAxe (see [Rin]), Magma (see [CP96]), and GAP (in reverse alphabetical order). Part of the constructions have been documented in the literature on almost simple groups, or the results have been used in such publications, see for example the bibliographies in [CCN+85] and [BN95] which are available online at `http://www.math.rwth-aachen.de/~Thomas.Breuer/atlasrep/bibl`.

If you use the ATLAS of Group Representations to solve a problem then please send a short email to `R.A.Wilson@qmul.ac.uk` about it. The ATLAS of Group Representations database should be referenced with the entry [WWT+] in the bibliography of this manual.

If your work made use of functions of the GAP interface (see Section 1.2) then you should also reference this interface, using the information printed by the GAP function `Cite` (**Reference: Cite**).

For referencing the GAP system in general, use the entry [GAP19] in the bibliography of this manual, see also `http://www.gap-system.org`.

1.2 The GAP Interface to the ATLAS of Group Representations

The GAP interface to the ATLAS of Group Representations consists of essentially two parts.

- First, there is the *user interface* which allows the user to get an overview of the contents of the database, and to access the data in GAP format; this is described in Chapter 3. Advanced users may add their own data to the database, this is described in Chapter 5.
- Second, there is *administrational information*, which covers also the declaration of GAP objects such as straight line decisions and black box programs. This is important mainly for users interested in the actual implementation (e. g., for modifying the package) or in using it together with the C-MeatAxe standalone (see [Rin]); this is described in Chapter 7.

Information concerning the C-MeatAxe, including the manual [Rin], can be found at

`http://www.math.rwth-aachen.de/~MTX`

The interface and this manual have been provided by Thomas Breuer, except for the interpreter for black box programs (see Section 6.2), which is due to Simon Nickerson. Comments, bug reports, and hints for improving the interface can be sent to `sam@math.rwth-aachen.de`.

1.3 What's New in AtlasRep, Compared to Older Versions?

1.3.1 What's New in Version 2.1.6? (October 2022)

The package now requires the utils package [BGH+22], and uses its `Download` (**Utils: Download**) function for downloading remote files. The former user preference `FileTransferTool` of the AtlasRep package is no longer supported; it had been used in older versions to distinguish between different download tools.

A method for `ConjugacyClasses` (**Reference: ConjugacyClasses attribute**) has been added that uses a straight line program for computing class representatives of a group that has been created with `AtlasGroup` (3.5.8), provided such a program is available. Thanks to Frank Lübeck for suggesting this.

1.3.2 What's New in Version 2.1.5? (August 2022)

Two bugs concerning local file permissions and the handling of download failures were fixed. Thanks to Frank Lübeck and Fabian Zickgraf for reporting these problems.

1.3.3 What's New in Version 2.1.4? (August 2022)

A few changes in the code for downloading files were needed in order to make some CI tests happy.

1.3.4 What's New in Version 2.1.3? (August 2022)

The server address for the core part of the database has changed.

Additional table of contents files are now available, which contain checksums in SHA256 format instead of the checksums computed by `CrcFile` (**Reference: CrcFile**) and `CrcString` (**Reference: CrcString**). Note that the latter values can be interpreted only by GAP.

For 364 representations, the corresponding characters have been identified and can thus be used for accessing these representations with `OneAtlasGeneratingSetInfo` (3.5.6), see `DisplayAtlasInfo` (3.5.1).

1.3.5 What's New in Version 2.1.2? (March 2022)

Not much.

The release of Version 2.1.2 was necessary for technical reasons: Now the testfile mentioned in `PackageInfo.g` exits GAP in the end, and the external links in the package documentation were corrected (the links in version 2.1.1 pointed to a wrong directory).

1.3.6 What's New in Version 2.1.1? (February 2022)

- The new function `EvaluatePresentation` (3.5.11) computes the images of the relators of a presentation (see Section 6.1.7).
- The new function `StandardGeneratorsData` (3.5.12) allows one to compute standard generators from given generators, provided a recipe for that task (a "find" straight line program) for the group in question is available.
- The function `AtlasGroup` (3.5.8) sets known information about the group and the representation, such as `IsPrimitive` (**Reference: IsPrimitive**).

  (Thanks to Steve Linton for suggesting this feature.)
- The function `ResultOfBBoxProgram` (6.2.4) now admits an optional argument, which is used as options record in calls to `RunBBoxProgram` (6.2.3).
- The new user preference `"AtlasRepJsonFilesAddresses"` (see Section 4.2.14) allows one to use Json format data files for matrix representations in characteristic zero, which in turn makes it possible to create the matrices over prescribed fields, for example fields returned by `AlgebraicExtension` (**Reference: AlgebraicExtension**). The information stored in the table of contents file about the field of entries of the matrix representations has been extended by a GAP independent description of this field and the defining polynomial used in the Json format data files.

- When the value of the user preference "AtlasRepDataDirectory" is an empty string then data files that are fetched from remote servers are read into the GAP session without storing the files. (An advantage is that one need not care about where one has permissions for storing files. A disadvantage is of course that one has to fetch a file again whenever it is needed.)

1.3.7 What's New in Version 2.1.0? (May 2019)

The main differences to earlier versions concern extensions of the available data. Up to now, such extensions were possible only in the sense that one could notify certain *locally available files* to the package's functions. With this version, it becomes possible to notify also *remote data files*, i. e., data files which have to be downloaded before they can be read into GAP, in the same way as the data from the ATLAS of Group Representations. Two extensions of this kind become automatically available with this package version, see Section 5.1 for details.

Thus the focus of the package has changed. In earlier versions, it provided a GAP interface to the data in the ATLAS of Group Representations, whereas now this database is regarded as one collection (the "core part") among others. Where applicable, the package manual tries to distinguish between general data available to the AtlasRep functions and the data from the ATLAS of Group Representations.

In order to provide this new functionality, the following changes have been implemented. Note that some are *incompatible changes*, compared with earlier versions of the package.

- The format of the identifier components of the records returned by AtlasGenerators (3.5.3), AtlasProgram (3.5.4), etc., has been changed for those data that belong to extensions, see 7.7. In the new format, the name of the extension is not added to the group name but to the individual filenames; this allows for example the combination of files from the core database and from extensions in one identifier. Functions for converting between the old and the new format are available, see AtlasRepIdentifier (7.7.1).

- The records returned by AtlasGenerators (3.5.3) etc. contain also a component contents, with value the identifier of the part of the database to shich the generators belong.

- The tables of contents of the ATLAS of Group Representations and of extensions are no longer stored in the form of sequences of calls to GAP functions. Instead, each table of contents is defined via a JSON format file, see 6.4. In particular, the file atlasprm.json replaces the former gap/atlasprm.g.

  Two advantages of this change are that there is no danger to call unwanted GAP functions when such files (which are expected to be available in the world wide web) get evaluated, and that the information is independent of GAP –note that MeatAxe format files and straight line programs can be used by other program systems as well.

- The functions ReloadAtlasTableOfContents, StoreAtlasTableOfContents, and ReplaceAtlasTableOfContents are no longer available. They had been intended for updating the table of contents of the ATLAS of Group Representations, but it has turned out that this was in fact not useful.

The second major change concerns the handling of user parameters.

- GAP's general *user preferences* mechanism (see `SetUserPreference` (**Reference: SetUserPreference**)) has been used since version 1.5.1 of the package for dealing with certain customizations of AtlasRep's behaviour, concerning the paths of data directories and two issues with MeatAxe format files.

  Now this mechanism is used in more cases, see Section 4.2 for an overview. The new user preferences replace certain components of the record `AtlasOfGroupRepresentationsInfo` (7.1.5) that were recommended in earlier versions of the package. These components are currently still available but are no longer used by the package's functions. Also the global variable `ATLASREP_TOCFILE` is no longer supported, use the user preference `AtlasRepTOCData` instead, see Section 4.2.3. Analogously, use the user preference `HowToReadMeatAxeTextFiles` instead of the no longer available `CMeatAxe.FastRead`.

  The switch to user preferences is an *incompatible change* if you are used to change the values of these components in your code, for example in your `gaprc` file, see (**Reference: The gap.ini and gaprc files**). All assignments to these components should be changed to calls of `SetUserPreference` (**Reference: SetUserPreference**).

  Another consequence of this change is that the former function `AtlasOfGroupRepresentationsUserParameters` of the package is no longer supported, use `ShowUserPreferences` (**Reference: ShowUserPreferences**) or `BrowseUserPreferences` (BrowseUserPreferences???) with argument `"AtlasRep"` instead.

Finally, the following improvements have been added.

- Straight line programs for computing generators of normal subgroups can now be fetched with `AtlasProgram` (3.5.4), using the argument `"kernel"`. The available programs of this type are shown in the `DisplayAtlasInfo` (3.5.1) overview for a group. More than 200 such programs are available in a new data directory `datapkg` of the package. If fact, this collection of files is part of an extension of the database that is distributed together with the package.

  In earlier versions of the package, this kind of information had been available only implicitly; it had been stored via `AGR.KERPRG`, which is not supported anymore.

- `AtlasProgram` (3.5.4) supports more variants of arguments: `"contents"` can be used to list the available data extensions, `"contents"` and `"version"` can be used to restrict the data under consideration, and one can request a program for computing *standard* generators of some maximal subgroup, not just generators (provided that this information is available).

  The information about the version of straight line programs is shown by `DisplayAtlasInfo` (3.5.1), as well as the availability of straight line programs for computing standard generators of maximal subgroups.

  Making this information more explicit has the side-effect that the access to the AtlasRep data with `BrowseAtlasInfo` (BrowseAtlasInfo???) is both safer and simpler, if at least version 1.8.6 of the Browse package is available. (For that, the function `AGR.InfoPrgs` has been extended such that also the `identifier` records are included in the result.)

- Straight line programs for computing *standard* generators of a maximal subgroup, if available, can now be fetched with `AtlasProgram` (3.5.4), using the argument `"maxstd"`.

- The function `AtlasRepInfoRecord` (3.5.10) now admits a group name as its argument, and then returns information about the group and its maximal subgroups; this information had been used before by `DisplayAtlasInfo` (3.5.1), but it had not been programmatically accessible.

- The sanity checks for the data (see Section 7.9) have been extended, in particular they can be applied also to data extensions. To some extent, these checks can be used also to derive new information; the code for that should be regarded as heuristic and experimental, runtimes and space requirements may be large, depending on the new data to be examined.
- Different header formats are now supported when reading and writing MeatAxe format files, see Section 4.2.8, and one can set a global default for the creation of mode 2 MeatAxe files, see Section 4.2.9.
- The function `MeatAxeString` (7.3.2) admits also an integer matrix as argument.
- The function `CMtxBinaryFFMatOrPerm` (7.3.4) admits an optional argument *base*, in order to write MeatAxe format files that contain either zero based or one based permutations.
- The meaningless lines about p-modular representations of groups with nontrivial p-core have been removed from the file `gap/mindeg.g`.

1.3.8 What's New in Version 1.5.1? (March 2016)

- The paths of the directories where downloaded data files get stored are now customizable, see Section 4.2.2. Up to now, the data were stored in subdirectories of the package directory, which might cause problems with write permissions, depending on the installation of the package. (Note that choosing other data directories can be useful also in order to keep existing local data files when a new version of GAP or of the AtlasRep package gets installed.) Thanks to Bill Allombert for pointing out this problem.
- The information about data files from the ATLAS of Group Representations has been extended by `CrcFile` (**Reference: CrcFile**) values. These values are checked whenever data from such a file are read, and an error is signalled if the checksum does not fit to the expected one. Note that several users may access the same data files, and a user should not suffer from perhaps corrupted files that have been downloaded by other users. Thanks to Frank Lübeck for the idea to introduce this consistency test.
- Whenever `StringFile` (**GAPDoc: StringFile**) is called by functions of the package, this happens in the wrapper function `AGR.StringFile`, in order to replace occasional line breaks of the form `"\r\n"` by `"\n"`. Apparently it may happen that the `"\r"` is silently smuggled in when data files get copied to the local computer. Thanks to Marek Mitros for help with detecting and fixing this problem.
- The function `FFMatOrPermCMtxBinary` (7.3.5) can now read also permutations stored in binary files that have been created with version 2.4 of the C-MeatAxe; note that this format is different from the one that is written by version 2.3. Conversely, `CMtxBinaryFFMatOrPerm` (7.3.4) has been generalized such that both formats can be written. The reference to the C-MeatAxe documentation now points to that of version 2.4. Thanks to Jürgen Müller for pointing out this problem.
- The function `MeatAxeString` (7.3.2) can now encode permutation matrices in different ways. The mode (the first header entry) can be either 2 (then the positions of the nonzero entries are listed) or 1 or 6 (then all entries of the matrix are listed). In previous versions, the function

produced a matrix of mode 2 whenever this was possible, but this behaviour is not useful if the result is not processed by the C-MeatAxe. Thanks to Klaus Lux for pointing out this problem.

- Depending on the terminal capabilities and the user preference `DisplayFunction` (see 4.2.11), some non-ASCII characters may appear in the output shown by `DisplayAtlasInfo` (3.5.1).

1.3.9 What's New in Version 1.5? (July 2011)

- The function `AtlasSubgroup` (3.5.9) now admits also the return value of `OneAtlasGeneratingSetInfo` (3.5.6) or the return value of `AtlasGroup` (3.5.8) as its first argument. The latter is implemented via the new attribute `AtlasRepInfoRecord` (3.5.10), which is set in the groups constructed by `AtlasGroup` (3.5.8).

- Information about transitivity, rank, primitivity, and point stabilizers of many permutation representations is now available. If applicable then this information appears in the records returned by `OneAtlasGeneratingSetInfo` (3.5.6), it is part of the overview shown by `DisplayAtlasInfo` (3.5.1), and it is shown also in the data overview in the web, see Section 4.3.

  Two new manual sections about point stabilizers have been added, see the sections 2.4.6 and 2.4.7.

- Information about the characters afforded by many matrix and permutation representations is now available. If applicable then this information appears in the records returned by `OneAtlasGeneratingSetInfo` (3.5.6), for matrix representations it is part of the overview shown by `DisplayAtlasInfo` (3.5.1), and it is shown also in the data overview in the web, see Section 4.3.

- The functions `Character` (**Reference: Character for a character table and a list**), `Identifier` (**Reference: Identifier for character tables**), `IsPrimitive` (**Reference: IsPrimitive**), `IsTransitive` (**Reference: IsTransitive**), `Transitivity` (**Reference: Transitivity**), and `RankAction` (**Reference: RankAction**) are now supported as input conditions in `DisplayAtlasInfo` (3.5.1) and `OneAtlasGeneratingSetInfo` (3.5.6).

- It is now possible to restrict the data shown by `DisplayAtlasInfo` (3.5.1) or returned by `OneAtlasGeneratingSetInfo` (3.5.6) to private or non-private data.

- A tutorial for beginners was added to the manual, see Chapter 2, and the manual was restructured.

- In the overview shown by `DisplayAtlasInfo` (3.5.1) and in the data overview in the web (see Section 4.3), the ordering of groups was improved such that, e.g., `"A9"` precedes `"A10"`.

- The function `AtlasClassNames` (3.4.2) now admits also a Brauer table as its argument, and works also for character tables of bicyclic extensions of simple groups.

- The group names that are entered in `DisplayAtlasInfo` (3.5.1), `OneAtlasGeneratingSetInfo` (3.5.6), etc., are now case insensitive, and if the package CTblLib is available then the admissible group names for the GAP character table of the group in question can be used in these functions.

- In order to reduce the number of global variables, several functions have been turned into components of the new global variable `AGR` (7.1.4). A few of these functions had been documented in the previous version, the old values are still available if the package files `gap/obsolete.gd` and `gap/obsolete.gi` have been read. These files are read automatically if GAP's user preference `"ReadObsolete"` is `true` when the package gets loaded, see (**Reference: The gap.ini file**).

- A few nicer characters are used by `DisplayAtlasInfo` (3.5.1) if `GAPInfo.TermEncoding` has the value `"UTF-8"` and if `Print` (**Reference: Print**) is not the display function to be used, see Section 4.2.11.

- A bug in the function `ReloadAtlasTableOfContents` was fixed. Thanks to Jack Schmidt for reporting this bug.

1.3.10 What's New in Version 1.4? (June 2008)

- In addition to the group orders that were added in version 1.3 (see Section 1.3.12), also many orders of maximal subgroups are now available. These values occur in the records returned by `AtlasProgram` (3.5.4) (for the case of `"maxes"` type programs) and of the three argument version of `AtlasGenerators` (3.5.3); now a `size` component may be bound. In these cases, the groups returned by `AtlasSubgroup` (3.5.9) have the `Size` (**Reference: Size**) attribute set.

- The information about the number of maximal subgroups, if available, is now used in `DisplayAtlasInfo` (3.5.1).

- In many cases, straight line programs for computing generators of maximal subgroups of a group G, say, can in fact be used to compute also generators of maximal subgroups of downward extensions of G; if not then it may suffice to extend the given straight line programs by additional generators.

  Currently this yields more than 200 new possibilities to compute maximal subgroups, this means a growth by about 25 percent. For example, all maximal subgroups of $12.M_{22}$ and $2.Fi_{22}$ can now be accessed via `AtlasGenerators` (3.5.3).

  (Of course this extension means only that one can access the straight line programs in question automatically via the GAP interface. In principle one could have used them already before, by explicitly applying a straight line program for a factor group to generators of a group, and perhaps adding some element in the kernel of the natural epimorphism.)

  For this feature, information about the compatibility of standard generators of groups and their factor groups was added.

- The bibliographies contained in the ATLAS of Finite Groups [CCN+85] and in the ATLAS of Brauer Characters [JLPW95] are now available as HTML files, as BibXMLext files, and within GAP, see `BrowseBibliographySporadicSimple` (3.6.2).

- If the GAP package Browse (see [BL18]) is loaded then the new functions `BrowseMinimalDegrees` (3.6.1) and `BrowseBibliographySporadicSimple` (3.6.2) are available; these functions can be called also by choosing the corresponding menu entries of the Browse application `BrowseGapData` (BrowseGapData???).

- The function AtlasGroup (3.5.8) now admits also the return value of OneAtlasGeneratingSetInfo (3.5.6) as its argument.

1.3.11 What's New in Version 1.3.1? (October 2007)

This version was mainly released in order to fix a few problems. Now one does not get warnings about unbound variables when the package is loaded and the GAP package IO [Neu14] is not available, and pathological situations in FFMatOrPermCMtxBinary (7.3.5) (concerning extremely short corrupted data files and different byte orderings in binary files) are handled more carefully.

Besides this, the two functions AtlasGroup (3.5.8) and AtlasSubgroup (3.5.9) were introduced, and the extended function QuaternionAlgebra (**Reference: QuaternionAlgebra**) of GAP 4.4.10 can now be used for describing base rings in OneAtlasGeneratingSetInfo (3.5.6) and AllAtlasGeneratingSetInfos (3.5.7). (This is the reason why this version of the package requires at least version 4.4.10 of GAP.)

1.3.12 What's New in Version 1.3? (June 2007)

- The database was extended, see Section 4.2.4 for the number and size of files.
- New data types and corresponding GAP objects have been introduced, for representing semi-presentations, presentations, and programs for finding standard generators. For details, see AtlasProgram (3.5.4), Chapter 6, and Section 7.6.
- The records returned by the functions AtlasGenerators (3.5.3), OneAtlasGeneratingSetInfo (3.5.6), and AllAtlasGeneratingSetInfos (3.5.7) now contain the name and (if known) the order of the group in question, and also components describing the degree in the case of permutation representations or the dimension and the base ring of the natural module in the case of matrix representations.
- For many of the groups, information about the minimal degree of faithful permutation representations and the minimal dimensions of faithful matrix representations in various characteristics is available for DisplayAtlasInfo (3.5.1), OneAtlasGeneratingSetInfo (3.5.6), and AllAtlasGeneratingSetInfos (3.5.7), see also Section 6.3. For these functions, also properties such as IsPrimeInt (**Reference: IsPrimeInt**) can be used to describe the intended restriction of the output.
- One can now use Pager (**Reference: Pager**) functionality in DisplayAtlasInfo (3.5.1), see Section 4.2.11.

  An interactive alternative to DisplayAtlasInfo (3.5.1) is provided by the function BrowseAtlasInfo (BrowseAtlasInfo???) from the new (recommended) GAP package Browse (see [BL18]).
- The functions OneAtlasGeneratingSetInfo (3.5.6) and AllAtlasGeneratingSetInfos (3.5.7) now admit also a list of group names as the first argument.
- The functions for actually accessing the data are more flexible now, see Section 7.2.
- For transferring remote data, the GAP package IO (see [Neu14]) can now be used (and is recommended) as an alternative to wget.

- The address of the data server has changed. The access to the server is no longer possible via `ftp`, thus the mechanism used up to version 1.2, which was based on `ftp`, had to be rewritten.

  The main consequence of this change is that information about updates of the table of contents is now provided at the package's homepage. This means that on the one hand, now package users cannot *compute* the table of contents directly from the server data, but on the other hand the update information can be *downloaded* without the necessity to install `perl`.

  Another consequence is that the system program `ls` is no longer needed, see Section 1.3.14.

- The package manual has been restructured, extended and improved. It is now based on the package GAPDoc (see [LN18]).

1.3.13 What's New in Version 1.2? (November 2003)

Not much.

The release of Version 1.2 became necessary first of all in order to provide a package version that is compatible with GAP 4.4, since some cross-references into the GAP Reference Manual were broken due to changes of section names. Additionally, several web addresses concerning the package itself were changed and thus had to be adjusted.

This opportunity was used

- to upgrade the administrational part for loading the package to the mechanism that is recommended for GAP 4.4,

- to extend the test suite, which now covers more consistency checks using the GAP Character Table Library (see [Bre22]),

- to make the function `ScanMeatAxeFile` (7.3.1) more robust, due to the fact that the GAP function `PermList` (**Reference: PermList**) now returns `fail` instead of raising an error,

- to change the way how representations with prescribed properties are accessed (the new function `OneAtlasGeneratingSetInfo` (3.5.6) is now preferred to the former `OneAtlasGeneratingSet`, and `AllAtlasGeneratingSetInfos` (3.5.7) has been added in order to provide programmatic access in parallel to the human readable descriptions printed by `DisplayAtlasInfo` (3.5.1)),

- and last but not least to include the current table of contents of the underlying database.

For AtlasRep users, the new feature of GAP 4.4 is particularly interesting that due to better kernel support, reading large matrices over finite fields is now faster than it was in GAP 4.3.

1.3.14 What's New in Version 1.1? (October 2002)

The biggest change w. r. t. Version 1.1 is the addition of private extensions (see Chapter 5). It includes a new "free format" for straight line programs (see Section 5.2). Unfortunately, this feature requires the system program `ls`, so it may be not available for example under MS Windows operating systems. [But see Section 1.3.12.]

In order to admit the addition of other types of data, the implementation of several functions has been changed. Data types are described in Section 7.5. An example of a new data type are quaternionic representations (see Section 7.6). The user interface itself (see Chapter 3) remained the same.

As an alternative to `perl`, one can use `wget` now for transferring data files (see 4.2).

Data files can be read much more efficiently in GAP 4.3 than in GAP 4.2. In Version 1.1 of the AtlasRep package, this feature is used for reading matrices and permutations in MeatAxe text format with `ScanMeatAxeFile` (7.3.1). As a consequence, (at least) GAP 4.3 is required for AtlasRep Version 1.1.

The new `compress` component of the global variable `AtlasOfGroupRepresentationsInfo` (7.1.5) allows one to store data files automatically in `gzip`ped form.

For matrix representations in characteristic zero, invariant forms and generators for the centralizer algebra are now accessible in GAP if they are contained in the source files –this information had been ignored in Version 1.0.

Additional information is now available via the internet (see 4.3).

The facilities for updating the table of contents have been extended.

The manual is now distributed also in PDF and HTML format; on the other hand, the PostScript format manual is no longer contained in the archives.

Apart from these changes, a few minor bugs in the handling of MeatAxe files have been fixed, typos in the documentation have been corrected, and the syntax checks for ATLAS straight line programs (see 7.4) have been improved.

1.4 Acknowledgements

- Frank Lübeck and Max Neunhöffer kindly provided the perl script that had been used for fetching remote data until version 1.2. Thanks also to Greg Gamble and Alexander Hulpke for technical hints concerning “standard” perl.
- Ulrich Kaiser helped with preparing the package for MS Windows.
- Klaus Lux had the idea to support data extensions, see Chapter 5, he did a lot of beta testing, and helped to fix several bugs.
- Frank Lübeck contributed the functions `CMtxBinaryFFMatOrPerm` (7.3.4) and `FFMatOrPermCMtxBinary` (7.3.5).
- Frank Lübeck and Max Neunhöffer wrote the GAPDoc package [LN18], which is used for processing the documentation of the AtlasRep package and for processing the bibliographies included in this package (see `BrowseBibliographySporadicSimple` (3.6.2)),
- Max Neunhöffer wrote the GAP package IO [Neu14], which is recommended for transferring data.
- Max Neunhöffer has also suggested the generalization of the data access described in Section 7.2, the admissibility of the function `Character` (**Reference: Character for a character table and a list**) as a filter in `DisplayAtlasInfo` (3.5.1), `OneAtlasGeneratingSetInfo` (3.5.6), and `AllAtlasGeneratingSetInfos` (3.5.7), and the variant of `AtlasRepInfoRecord` (3.5.10) that takes a group name as its input.
- Gunter Malle suggested to make the information about representations of minimal degree accessible, see Section 6.3.

- Andries Brouwer suggested to add a tutorial (see Chapter 2), Klaus Lux suggested several improvements of this chapter.
- The development of this GAP package has been supported by the SFB-TRR 195 “Symbolic Tools in Mathematics and their Applications” (from 2017 until 2022).

Chapter 2

Tutorial for the AtlasRep Package

This chapter gives an overview of the basic functionality provided by the AtlasRep package. The main concepts and interface functions are presented in the first three sections, and Section 2.4 shows a few small examples.

Let us first fix the setup for the examples shown in the package manual.

1. First of all, we load the AtlasRep package. Some of the examples require also the GAP packages CTblLib and TomLib, so we load also these packages.

```
Example
gap> LoadPackage( "AtlasRep", false );
true
gap> LoadPackage( "CTblLib", false );
true
gap> LoadPackage( "TomLib", false );
true
```

2. Depending on the terminal capabilities, the output of `DisplayAtlasInfo` (3.5.1) may contain non-ASCII characters, which are not supported by the LaTeX and HTML versions of GAPDoc documents. The examples in this manual are used for tests of the package's functionality, thus we set the user preference `DisplayFunction` (see Section 4.2.11) to the value `"Print"` in order to produce output consisting only of ASCII characters, which is assumed to work in any terminal.

```
Example
gap> origpref:= UserPreference( "AtlasRep", "DisplayFunction" );;
gap> SetUserPreference( "AtlasRep", "DisplayFunction", "Print" );
```

3. The GAP output for the examples may look differently if data extensions have been loaded. In order to ignore these extensions in the examples, we unload them.

```
Example
gap> priv:= Difference(
>     List( AtlasOfGroupRepresentationsInfo.notified, x -> x.ID ),
>     [ "core", "internal" ] );;
gap> Perform( priv, AtlasOfGroupRepresentationsForgetData );
```

4. If the info level of `InfoAtlasRep` (7.1.1) is larger than zero then additional output appears on the screen. In order to avoid this output, we set the level to zero.

Example

```
gap> globallevel:= InfoLevel( InfoAtlasRep );;
gap> SetInfoLevel( InfoAtlasRep, 0 );
```

2.1 Accessing a Specific Group in AtlasRep

An important database to which the AtlasRep package gives access is the ATLAS of Group Representations [WWT+]. It contains generators and related data for several groups, mainly for extensions of simple groups (see Section 2.1.1) and for their maximal subgroups (see Section 2.1.2).

In general, these data are not part of the package. They are downloaded as soon as they are needed for the first time, see Section 4.2.1.

2.1.1 Accessing a Group in AtlasRep via its Name

Each group that occurs in this database is specified by a *name*, which is a string similar to the name used in the ATLAS of Finite Groups [CCN+85]. For those groups whose character tables are contained in the GAP Character Table Library [Bre22], the names are equal to the `Identifier` (**Reference: Identifier for character tables**) values of these character tables. Examples of such names are `"M24"` for the Mathieu group M_{24}, `"2.A6"` for the double cover of the alternating group A_6, and `"2.A6.2_1"` for the double cover of the symmetric group S_6. The names that actually occur are listed in the first column of the overview table that is printed by the function `DisplayAtlasInfo` (3.5.1), called without arguments, see below. The other columns of the table describe the data that are available in the database.

For example, `DisplayAtlasInfo` (3.5.1) may print the following lines. Omissions are indicated with "...".

Example

```
gap> DisplayAtlasInfo();
group                    |  #  | maxes | cl | cyc | out | fnd | chk | prs
-------------------------+-----+-------+----+-----+-----+-----+-----+----
...
2.A5                     | 26  |     3 |    |     |     |  +  |  +  |  +
2.A5.2                   | 11  |     4 |    |     |     |  +  |  +  |  +
2.A6                     | 18  |     5 |    |     |     |     |     |
2.A6.2_1                 |  3  |     6 |    |     |     |     |     |
2.A7                     | 24  |     2 |    |     |     |     |     |
2.A7.2                   |  7  |       |    |     |     |     |     |
...
M22                      | 58  |     8 |  +  |  +  |     |  +  |  +  |  +
M22.2                    | 46  |     7 |  +  |  +  |     |  +  |  +  |  +
M23                      | 66  |     7 |  +  |  +  |     |  +  |  +  |  +
M24                      | 62  |     9 |  +  |  +  |     |  +  |  +  |  +
McL                      | 46  |    12 |  +  |  +  |     |  +  |  +  |  +
McL.2                    | 27  |    10 |     |  +  |     |  +  |  +  |  +
O7(3)                    | 28  |       |    |     |     |     |     |
O7(3).2                  |  3  |       |    |     |     |     |     |
...
Suz                      | 30  |    17 |    |  +  |   2 |  +  |  +  |
...
```

Called with a group name as the only argument, the function AtlasGroup (3.5.8) returns a group isomorphic to the group with the given name, or fail. If permutation generators are available in the database then a permutation group (of smallest available degree) is returned, otherwise a matrix group.

Example

```
gap> g:= AtlasGroup( "M24" );
Group([ (1,4)(2,7)(3,17)(5,13)(6,9)(8,15)(10,19)(11,18)(12,21)(14,16)
  (20,24)(22,23), (1,4,6)(2,21,14)(3,9,15)(5,18,10)(13,17,16)
  (19,24,23) ])
gap> IsPermGroup( g );  NrMovedPoints( g );  Size( g );
true
24
244823040
gap> AtlasGroup( "J5" );
fail
```

2.1.2 Accessing a Maximal Subgroup of a Group in AtlasRep

Many maximal subgroups of extensions of simple groups can be constructed using the function AtlasSubgroup (3.5.9). Given the name of the extension of the simple group and the number of the conjugacy class of maximal subgroups, this function returns a representative from this class.

Example

```
gap> g:= AtlasSubgroup( "M24", 1 );
Group([ (2,10)(3,12)(4,14)(6,9)(8,16)(15,18)(20,22)(21,24), (1,7,2,9)
  (3,22,10,23)(4,19,8,12)(5,14)(6,18)(13,16,17,24) ])
gap> IsPermGroup( g );  NrMovedPoints( g );  Size( g );
true
23
10200960
gap> AtlasSubgroup( "M24", 100 );
fail
```

The classes of maximal subgroups are ordered w. r. t. decreasing subgroup order. So the first class contains maximal subgroups of smallest index.

Note that groups obtained by AtlasSubgroup (3.5.9) may be not very suitable for computations in the sense that much nicer representations exist. For example, the sporadic simple O'Nan group $O'N$ contains a maximal subgroup S isomorphic with the Janko group J_1; the smallest permutation representation of $O'N$ has degree 122760, and restricting this representation to S yields a representation of J_1 of that degree. However, J_1 has a faithful permutation representation of degree 266, which admits much more efficient computations. If you are just interested in J_1 and not in its embedding into $O'N$ then one possibility to get a "nicer" faithful representation is to call SmallerDegreePermutationRepresentation (**Reference: SmallerDegreePermutationRepresentation**). In the abovementioned example, this works quite well; note that in general, we cannot expect that we get a representation of smallest degree in this way.

Example

```
gap> s:= AtlasSubgroup( "ON", 3 );
<permutation group of size 175560 with 2 generators>
gap> NrMovedPoints( s );  Size( s );
122760
175560
gap> hom:= SmallerDegreePermutationRepresentation( s );;
```

```
gap> NrMovedPoints( Image( hom ) );
1540
```

In this particular case, one could of course also ask directly for the group J_1.

Example

```
gap> j1:= AtlasGroup( "J1" );
<permutation group of size 175560 with 2 generators>
gap> NrMovedPoints( j1 );
266
```

If you have a group G, say, and you are really interested in the embedding of a maximal subgroup of G into G then an easy way to get compatible generators is to create G with `AtlasGroup` (3.5.8) and then to call `AtlasSubgroup` (3.5.9) with first argument the group G.

Example

```
gap> g:= AtlasGroup( "ON" );
<permutation group of size 460815505920 with 2 generators>
gap> s:= AtlasSubgroup( g, 3 );
<permutation group of size 175560 with 2 generators>
gap> IsSubset( g, s );
true
gap> IsSubset( g, j1 );
false
```

2.2 Accessing Specific Generators in AtlasRep

The function `DisplayAtlasInfo` (3.5.1), called with an admissible name of a group as the only argument, lists the ATLAS data available for this group.

Example

```
gap> DisplayAtlasInfo( "A5" );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
 1: G <= Sym(5)                  3-trans., on cosets of A4 (1st max.)
 2: G <= Sym(6)                  2-trans., on cosets of D10 (2nd max.)
 3: G <= Sym(10)                 rank 3, on cosets of S3 (3rd max.)
 4: G <= GL(4a,2)                character 4a
 5: G <= GL(4b,2)                character 2ab
 6: G <= GL(4,3)                 character 4a
 7: G <= GL(6,3)                 character 3ab
 8: G <= GL(2a,4)                character 2a
 9: G <= GL(2b,4)                character 2b
10: G <= GL(3,5)                 character 3a
11: G <= GL(5,5)                 character 5a
12: G <= GL(3a,9)                character 3a
13: G <= GL(3b,9)                character 3b
14: G <= GL(4,Z)                 character 4a
15: G <= GL(5,Z)                 character 5a
16: G <= GL(6,Z)                 character 3ab
17: G <= GL(3a,Field([Sqrt(5)])) character 3a
18: G <= GL(3b,Field([Sqrt(5)])) character 3b
```

```
Programs for G = A5:    (all refer to std. generators 1)
--------------------
- class repres.*
- presentation
- maxes (all 3):
  1:  A4
  2:  D10
  3:  S3
- std. gen. checker:
  (check)
  (pres)
```

In order to fetch one of the listed permutation groups or matrix groups, you can call `AtlasGroup` (3.5.8) with second argument the function `Position` (**Reference: Position**) and third argument the position in the list.

Example

```
gap> AtlasGroup( "A5", Position, 1 );
Group([ (1,2)(3,4), (1,3,5) ])
```

Note that this approach may yield a different group after a data extension has been loaded.

Alternatively, you can describe the desired group by conditions, such as the degree in the case of a permutation group, and the dimension and the base ring in the case of a matrix group.

Example

```
gap> AtlasGroup( "A5", NrMovedPoints, 10 );
Group([ (2,4)(3,5)(6,8)(7,10), (1,2,3)(4,6,7)(5,8,9) ])
gap> AtlasGroup( "A5", Dimension, 4, Ring, GF(2) );
<matrix group of size 60 with 2 generators>
```

The same holds for the restriction to maximal subgroups: Use `AtlasSubgroup` (3.5.9) with the same arguments as `AtlasGroup` (3.5.8), except that additionally the number of the class of maximal subgroups is entered as the last argument. Note that the conditions refer to the group, not to the subgroup; it may happen that the subgroup moves fewer points than the big group.

Example

```
gap> AtlasSubgroup( "A5", Dimension, 4, Ring, GF(2), 1 );
<matrix group of size 12 with 2 generators>
gap> g:= AtlasSubgroup( "A5", NrMovedPoints, 10, 3 );
Group([ (2,4)(3,5)(6,8)(7,10), (1,4)(3,8)(5,7)(6,10) ])
gap> Size( g );  NrMovedPoints( g );
6
9
```

2.3 Basic Concepts used in AtlasRep

2.3.1 Groups, Generators, and Representations

Up to now, we have talked only about groups and subgroups. The AtlasRep package provides access to *group generators*, and in fact these generators have the property that mapping one set of generators to another set of generators for the same group defines an isomorphism. These generators are called *standard generators*, see Section 3.3.

So instead of thinking about several generating sets of a group G, say, we can think about one abstract group G, with one fixed set of generators, and mapping these generators to any set of generators provided by AtlasRep defines a representation of G. This viewpoint had motivated the name "ATLAS of Group Representations" for the core part of the database.

If you are interested in the generators provided by the database rather than in the groups they generate, you can use the function `OneAtlasGeneratingSetInfo` (3.5.6) instead of `AtlasGroup` (3.5.8), with the same arguments. This will yield a record that describes the representation in question. Calling the function `AtlasGenerators` (3.5.3) with this record will then yield a record with the additional component `generators`, which holds the list of generators.

Example

```
gap> info:= OneAtlasGeneratingSetInfo( "A5", NrMovedPoints, 10 );
rec( charactername := "1a+4a+5a", constituents := [ 1, 4, 5 ],
  contents := "core", groupname := "A5", id := "",
  identifier := [ "A5", [ "A5G1-p10B0.m1", "A5G1-p10B0.m2" ], 1, 10 ],
  isPrimitive := true, maxnr := 3, p := 10, rankAction := 3,
  repname := "A5G1-p10B0", repnr := 3, size := 60, stabilizer := "S3",
  standardization := 1, transitivity := 1, type := "perm" )
gap> info2:= AtlasGenerators( info );
rec( charactername := "1a+4a+5a", constituents := [ 1, 4, 5 ],
  contents := "core",
  generators := [ (2,4)(3,5)(6,8)(7,10), (1,2,3)(4,6,7)(5,8,9) ],
  groupname := "A5", id := "",
  identifier := [ "A5", [ "A5G1-p10B0.m1", "A5G1-p10B0.m2" ], 1, 10 ],
  isPrimitive := true, maxnr := 3, p := 10, rankAction := 3,
  repname := "A5G1-p10B0", repnr := 3, size := 60, stabilizer := "S3",
  standardization := 1, transitivity := 1, type := "perm" )
gap> info2.generators;
[ (2,4)(3,5)(6,8)(7,10), (1,2,3)(4,6,7)(5,8,9) ]
```

The record `info` appears as the value of the attribute `AtlasRepInfoRecord` (3.5.10) in groups that are returned by `AtlasGroup` (3.5.8).

Example

```
gap> g:= AtlasGroup( "A5", NrMovedPoints, 10 );;
gap> AtlasRepInfoRecord( g );
rec( charactername := "1a+4a+5a", constituents := [ 1, 4, 5 ],
  contents := "core", groupname := "A5", id := "",
  identifier := [ "A5", [ "A5G1-p10B0.m1", "A5G1-p10B0.m2" ], 1, 10 ],
  isPrimitive := true, maxnr := 3, p := 10, rankAction := 3,
  repname := "A5G1-p10B0", repnr := 3, size := 60, stabilizer := "S3",
  standardization := 1, transitivity := 1, type := "perm" )
```

2.3.2 Straight Line Programs

For computing certain group elements from standard generators, such as generators of a subgroup or class representatives, AtlasRep uses *straight line programs*, see (**Reference: Straight Line Programs**). Essentially this means to evaluate words in the generators, which is similar to `MappedWord` (**Reference: MappedWord**) but can be more efficient.

It can be useful to deal with these straight line programs, see `AtlasProgram` (3.5.4). For example, an automorphism α, say, of the group G, if available in AtlasRep, is given by a straight line program

that defines the images of standard generators of G. This way, one can for example compute the image of a subgroup U of G under α by first applying the straight line program for α to standard generators of G, and then applying the straight line program for the restriction from G to U.

Example

```
gap> prginfo:= AtlasProgramInfo( "A5", "maxes", 1 );
rec( groupname := "A5", identifier := [ "A5", "A5G1-max1W1", 1 ],
  size := 12, standardization := 1, subgroupname := "A4",
  version := "1" )
gap> prg:= AtlasProgram( prginfo.identifier );
rec( groupname := "A5", identifier := [ "A5", "A5G1-max1W1", 1 ],
  program := <straight line program>, size := 12,
  standardization := 1, subgroupname := "A4", version := "1" )
gap> Display( prg.program );
# input:
r:= [ g1, g2 ];
# program:
r[3]:= r[1]*r[2];
r[4]:= r[2]*r[1];
r[5]:= r[3]*r[3];
r[1]:= r[5]*r[4];
# return values:
[ r[1], r[2] ]
gap> ResultOfStraightLineProgram( prg.program, info2.generators );
[ (1,10)(2,3)(4,9)(7,8), (1,2,3)(4,6,7)(5,8,9) ]
```

2.4 Examples of Using the AtlasRep Package

2.4.1 Example: Class Representatives

First we show the computation of class representatives of the Mathieu group M_{11}, in a 2-modular matrix representation. We start with the ordinary and Brauer character tables of this group.

Example

```
gap> tbl:= CharacterTable( "M11" );;
gap> modtbl:= tbl mod 2;;
gap> CharacterDegrees( modtbl );
[ [ 1, 1 ], [ 10, 1 ], [ 16, 2 ], [ 44, 1 ] ]
```

The output of `CharacterDegrees` (**Reference: CharacterDegrees**) means that the 2-modular irreducibles of M_{11} have degrees 1, 10, 16, 16, and 44.

Using `DisplayAtlasInfo` (3.5.1), we find out that matrix generators for the irreducible 10-dimensional representation are available in the database.

Example

```
gap> DisplayAtlasInfo( "M11", Characteristic, 2 );
Representations for G = M11:    (all refer to std. generators 1)
----------------------------
 6: G <= GL(10,2)  character 10a
 7: G <= GL(32,2)  character 16ab
 8: G <= GL(44,2)  character 44a
16: G <= GL(16a,4) character 16a
17: G <= GL(16b,4) character 16b
```

So we decide to work with this representation. We fetch the generators and compute the list of class representatives of M_{11} in the representation. The ordering of class representatives is the same as that in the character table of the ATLAS of Finite Groups ([CCN+85]), which coincides with the ordering of columns in the GAP table we have fetched above.

Example

```
gap> info:= OneAtlasGeneratingSetInfo( "M11", Characteristic, 2,
>                                      Dimension, 10 );;
gap> gens:= AtlasGenerators( info.identifier );;
gap> ccls:= AtlasProgram( "M11", gens.standardization, "classes" );
rec( groupname := "M11", identifier := [ "M11", "M11G1-cclsW1", 1 ],
  outputs := [ "1A", "2A", "3A", "4A", "5A", "6A", "8A", "8B", "11A",
      "11B" ], program := <straight line program>,
  standardization := 1, version := "1" )
gap> reps:= ResultOfStraightLineProgram( ccls.program, gens.generators );;
```

If we would need only a few class representatives, we could use the GAP library function `RestrictOutputsOfSLP` (**Reference: RestrictOutputsOfSLP**) to create a straight line program that computes only specified outputs. Here is an example where only the class representatives of order eight are computed.

Example

```
gap> ord8prg:= RestrictOutputsOfSLP( ccls.program,
>                   Filtered( [ 1 .. 10 ], i -> ccls.outputs[i][1] = '8' ) );
<straight line program>
gap> ord8reps:= ResultOfStraightLineProgram( ord8prg, gens.generators );;
gap> List( ord8reps, m -> Position( reps, m ) );
[ 7, 8 ]
```

Let us check that the class representatives have the right orders.

Example

```
gap> List( reps, Order ) = OrdersClassRepresentatives( tbl );
true
```

From the class representatives, we can compute the Brauer character we had started with. This Brauer character is defined on all classes of the 2-modular table. So we first pick only those representatives, using the GAP function `GetFusionMap` (**Reference: GetFusionMap**); in this situation, it returns the class fusion from the Brauer table into the ordinary table.

Example

```
gap> fus:= GetFusionMap( modtbl, tbl );
[ 1, 3, 5, 9, 10 ]
gap> modreps:= reps{ fus };;
```

Then we call the GAP function `BrauerCharacterValue` (**Reference: BrauerCharacterValue**), which computes the Brauer character value from the matrix given.

Example

```
gap> char:= List( modreps, BrauerCharacterValue );
[ 10, 1, 0, -1, -1 ]
gap> Position( Irr( modtbl ), char );
2
```

2.4.2 Example: Permutation and Matrix Representations

The second example shows the computation of a permutation representation from a matrix representation. We work with the 10-dimensional representation used above, and consider the action on the 2^{10} vectors of the underlying row space.

Example

```
gap> grp:= Group( gens.generators );;
gap> v:= GF(2)^10;;
gap> orbs:= Orbits( grp, AsList( v ) );;
gap> List( orbs, Length );
[ 1, 396, 55, 330, 66, 165, 11 ]
```

We see that there are six nontrivial orbits, and we can compute the permutation actions on these orbits directly using `Action` (**Reference: Action homomorphisms**). However, for larger examples, one cannot write down all orbits on the row space, so one has to use another strategy if one is interested in a particular orbit.

Let us assume that we are interested in the orbit of length 11. The point stabilizer is the first maximal subgroup of M_{11}, thus the restriction of the representation to this subgroup has a nontrivial fixed point space. This restriction can be computed using the AtlasRep package.

Example

```
gap> gens:= AtlasGenerators( "M11", 6, 1 );;
```

Now computing the fixed point space is standard linear algebra.

Example

```
gap> id:= IdentityMat( 10, GF(2) );;
gap> sub1:= Subspace( v, NullspaceMat( gens.generators[1] - id ) );;
gap> sub2:= Subspace( v, NullspaceMat( gens.generators[2] - id ) );;
gap> fix:= Intersection( sub1, sub2 );
<vector space of dimension 1 over GF(2)>
```

The final step is of course the computation of the permutation action on the orbit.

Example

```
gap> orb:= Orbit( grp, Basis( fix )[1] );;
gap> act:= Action( grp, orb );;  Print( act, "\n" );
Group( [ ( 1, 2)( 4, 6)( 5, 8)( 7,10), ( 1, 3, 5, 9)( 2, 4, 7,11) ] )
```

Note that this group is *not* equal to the group obtained by fetching the permutation representation from the database. This is due to a different numbering of the points, thus the groups are permutation isomorphic, that is, they are conjugate in the symmetric group on eleven points.

Example

```
gap> permgrp:= Group( AtlasGenerators( "M11", 1 ).generators );;
gap> Print( permgrp, "\n" );
Group( [ ( 2,10)( 4,11)( 5, 7)( 8, 9), (1,4,3,8)(2,5,6,9) ] )
gap> permgrp = act;
false
gap> IsConjugate( SymmetricGroup(11), permgrp, act );
true
```

2.4.3 Example: Outer Automorphisms

The straight line programs for applying outer automorphisms to standard generators can of course be used to define the automorphisms themselves as GAP mappings.

Example

```
gap> DisplayAtlasInfo( "G2(3)", IsStraightLineProgram );
Programs for G = G2(3):    (all refer to std. generators 1)
-----------------------
- class repres.
- presentation
- repr. cyc. subg.
- std. gen. checker
- automorphisms:
  2
- maxes (all 10):
   1:  U3(3).2
   2:  U3(3).2
   3:  (3^(1+2)+x3^2):2S4
   4:  (3^(1+2)+x3^2):2S4
   5:  L3(3).2
   6:  L3(3).2
   7:  L2(8).3
   8:  2^3.L3(2)
   9:  L2(13)
  10:  2^(1+4)+:3^2.2
gap> prog:= AtlasProgram( "G2(3)", "automorphism", "2" ).program;;
gap> info:= OneAtlasGeneratingSetInfo( "G2(3)", Dimension, 7 );;
gap> gens:= AtlasGenerators( info ).generators;;
gap> imgs:= ResultOfStraightLineProgram( prog, gens );;
```

If we are not suspicious whether the script really describes an automorphism then we should tell this to GAP, in order to avoid the expensive checks of the properties of being a homomorphism and bijective (see Section (**Reference: Creating Group Homomorphisms**)). This looks as follows.

Example

```
gap> g:= Group( gens );;
gap> aut:= GroupHomomorphismByImagesNC( g, g, gens, imgs );;
gap> SetIsBijective( aut, true );
```

If we are suspicious whether the script describes an automorphism then we might have the idea to check it with GAP, as follows.

Example

```
gap> aut:= GroupHomomorphismByImages( g, g, gens, imgs );;
gap> IsBijective( aut );
true
```

(Note that even for a comparatively small group such as $G_2(3)$, this was a difficult task for GAP before version 4.3.)

Often one can form images under an automorphism α, say, without creating the homomorphism object. This is obvious for the standard generators of the group G themselves, but also for generators of a maximal subgroup M computed from standard generators of G, provided that the straight line

programs in question refer to the same standard generators. Note that the generators of M are given by evaluating words in terms of standard generators of G, and their images under α can be obtained by evaluating the same words at the images under α of the standard generators of G.

Example

```
gap> max1:= AtlasProgram( "G2(3)", 1 ).program;;
gap> mgens:= ResultOfStraightLineProgram( max1, gens );;
gap> comp:= CompositionOfStraightLinePrograms( max1, prog );;
gap> mimgs:= ResultOfStraightLineProgram( comp, gens );;
```

The list `mgens` is the list of generators of the first maximal subgroup of $G_2(3)$, `mimgs` is the list of images under the automorphism given by the straight line program `prog`. Note that applying the program returned by `CompositionOfStraightLinePrograms` (**Reference: CompositionOfStraightLinePrograms**) means to apply first `prog` and then `max1`. Since we have already constructed the GAP object representing the automorphism, we can check whether the results are equal.

Example

```
gap> mimgs = List( mgens, x -> x^aut );
true
```

However, it should be emphasized that using `aut` requires a huge machinery of computations behind the scenes, whereas applying the straight line programs `prog` and `max1` involves only elementary operations with the generators. The latter is feasible also for larger groups, for which constructing the GAP automorphism might be too hard.

2.4.4 Example: Using Semi-presentations and Black Box Programs

Let us suppose that we want to restrict a representation of the Mathieu group M_{12} to a non-maximal subgroup of the type $L_2(11)$. The idea is that this subgroup can be found as a maximal subgroup of a maximal subgroup of the type M_{11}, which is itself maximal in M_{12}. For that, we fetch a representation of M_{12} and use a straight line program for restricting it to the first maximal subgroup, which has the type M_{11}.

Example

```
gap> info:= OneAtlasGeneratingSetInfo( "M12", NrMovedPoints, 12 );
rec( charactername := "1a+11a", constituents := [ 1, 2 ],
  contents := "core", groupname := "M12", id := "a",
  identifier := [ "M12", [ "M12G1-p12aB0.m1", "M12G1-p12aB0.m2" ], 1,
      12 ], isPrimitive := true, maxnr := 1, p := 12, rankAction := 2,
  repname := "M12G1-p12aB0", repnr := 1, size := 95040,
  stabilizer := "M11", standardization := 1, transitivity := 5,
  type := "perm" )
gap> gensM12:= AtlasGenerators( info.identifier );;
gap> restM11:= AtlasProgram( "M12", "maxes", 1 );;
gap> gensM11:= ResultOfStraightLineProgram( restM11.program,
>                                           gensM12.generators );
[ (3,9)(4,12)(5,10)(6,8), (1,4,11,5)(2,10,8,3) ]
```

Now we *cannot* simply apply a straight line program for a group to some generators, since they are not necessarily *standard* generators of the group. We check this property using a semi-presentation for M_{11}, see 6.1.7.

Example

```
gap> checkM11:= AtlasProgram( "M11", "check" );
rec( groupname := "M11", identifier := [ "M11", "M11G1-check1", 1, 1 ]
    , program := <straight line decision>, standardization := 1,
  version := "1" )
gap> ResultOfStraightLineDecision( checkM11.program, gensM11 );
true
```

So we are lucky that applying the appropriate program for M_{11} will give us the required generators for $L_2(11)$.

Example

```
gap> restL211:= AtlasProgram( "M11", "maxes", 2 );;
gap> gensL211:= ResultOfStraightLineProgram( restL211.program, gensM11 );
[ (3,9)(4,12)(5,10)(6,8), (1,11,9)(2,12,8)(3,6,10) ]
gap> G:= Group( gensL211 );;  Size( G );  IsSimple( G );
660
true
```

In this case, we could also use the information that is stored about M_{11}, as follows.

Example

```
gap> DisplayAtlasInfo( "M11", IsStraightLineProgram );
Programs for G = M11:    (all refer to std. generators 1)
---------------------
- presentation
- repr. cyc. subg.
- std. gen. finder
- class repres.:
  (direct)
  (composed)
- maxes (all 5):
  1:  A6.2_3
  1:  A6.2_3                                (std. 1)
  2:  L2(11)
  2:  L2(11)                                (std. 1)
  3:  3^2:Q8.2
  4:  S5
  4:  S5                                    (std. 1)
  5:  2.S4
- standardizations of maxes:
  from 1st max., version 1 to A6.2_3, std. 1
  from 2nd max., version 1 to L2(11), std. 1
  from 4th max., version 1 to A5.2, std. 1
- std. gen. checker:
  (check)
  (pres)
```

The entry “std.1” in the line about the maximal subgroup of type $L_2(11)$ means that a straight line program for computing *standard* generators (in standardization 1) of the subgroup. This program can be fetched as follows.

Example

```
gap> restL211std:= AtlasProgram( "M11", "maxes", 2, 1 );;
gap> ResultOfStraightLineProgram( restL211std.program, gensM11 );
[ (3,9)(4,12)(5,10)(6,8), (1,11,9)(2,12,8)(3,6,10) ]
```

We see that we get the same generators for the subgroup as above. (In fact the second approach first applies the same program as is given by `restL211.program`, and then applies a program to the results that does nothing.)

Usually representations are not given in terms of standard generators. For example, let us take the M_{11} type group returned by the GAP function `MathieuGroup` (**Reference: MathieuGroup**).

Example

```
gap> G:= MathieuGroup( 11 );;
gap> gens:= GeneratorsOfGroup( G );
[ (1,2,3,4,5,6,7,8,9,10,11), (3,7,11,8)(4,10,5,6) ]
gap> ResultOfStraightLineDecision( checkM11.program, gens );
false
```

If we want to compute an $L_2(11)$ type subgroup of this group, we can use a black box program for computing standard generators, and then apply the straight line program for computing the restriction.

Example

```
gap> find:= AtlasProgram( "M11", "find" );
rec( groupname := "M11", identifier := [ "M11", "M11G1-find1", 1, 1 ],
  program := <black box program>, standardization := 1,
  version := "1" )
gap> stdgens:= ResultOfBBoxProgram( find.program, Group( gens ) );;
gap> List( stdgens, Order );
[ 2, 4 ]
gap> ResultOfStraightLineDecision( checkM11.program, stdgens );
true
gap> gensL211:= ResultOfStraightLineProgram( restL211.program, stdgens );;
gap> List( gensL211, Order );
[ 2, 3 ]
gap> G:= Group( gensL211 );;  Size( G );  IsSimple( G );
660
true
```

Note that applying the black box program several times may yield different group elements, because computations of random elements are involved, see `ResultOfBBoxProgram` (6.2.4). All what the black box program promises is to construct standard generators, and these are defined only up to conjugacy in the automorphism group of the group in question.

2.4.5 Example: Using the GAP Library of Tables of Marks

The GAP Library of Tables of Marks (the GAP package TomLib, [NMP18]) provides, for many almost simple groups, information for constructing representatives of all conjugacy classes of subgroups. If this information is compatible with the standard generators of the ATLAS of Group Representations then we can use it to restrict any representation from the ATLAS to prescribed subgroups. This is useful in particular for those subgroups for which the ATLAS of Group Representations itself does not contain a straight line program.

Example

```
gap> tom:= TableOfMarks( "A5" );
TableOfMarks( "A5" )
gap> info:= StandardGeneratorsInfo( tom );
[ rec( ATLAS := true, description := "|a|=2, |b|=3, |ab|=5",
      generators := "a, b",
      script := [ [ 1, 2 ], [ 2, 3 ], [ 1, 1, 2, 1, 5 ] ],
      standardization := 1 ) ]
```

The `true` value of the component `ATLAS` indicates that the information stored on `tom` refers to the standard generators of type 1 in the ATLAS of Group Representations.

We want to restrict a 4-dimensional integral representation of A_5 to a Sylow 2 subgroup of A_5, and use `RepresentativeTomByGeneratorsNC` (**Reference: RepresentativeTomByGeneratorsNC**) for that.

Example

```
gap> info:= OneAtlasGeneratingSetInfo( "A5", Ring, Integers, Dimension, 4 );;
gap> stdgens:= AtlasGenerators( info.identifier );
rec( charactername := "4a", constituents := [ 4 ], contents := "core",
  dim := 4,
  generators :=
    [
      [ [ 1, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 1, 0, 0 ],
          [ -1, -1, -1, -1 ] ],
      [ [ 0, 1, 0, 0 ], [ 0, 0, 0, 1 ], [ 0, 0, 1, 0 ],
          [ 1, 0, 0, 0 ] ] ], groupname := "A5", id := "",
  identifier := [ "A5", "A5G1-Zr4B0.g", 1, 4 ],
  repname := "A5G1-Zr4B0", repnr := 14, ring := Integers, size := 60,
  standardization := 1, type := "matint" )
gap> orders:= OrdersTom( tom );
[ 1, 2, 3, 4, 5, 6, 10, 12, 60 ]
gap> pos:= Position( orders, 4 );
4
gap> sub:= RepresentativeTomByGeneratorsNC( tom, pos, stdgens.generators );
<matrix group of size 4 with 2 generators>
gap> GeneratorsOfGroup( sub );
[ [ [ 1, 0, 0, 0 ], [ -1, -1, -1, -1 ], [ 0, 0, 0, 1 ],
      [ 0, 0, 1, 0 ] ],
  [ [ 1, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 1, 0, 0 ],
      [ -1, -1, -1, -1 ] ] ]
```

2.4.6 Example: Index 770 Subgroups in M_{22}

The sporadic simple Mathieu group M_{22} contains a unique class of subgroups of index 770 (and order 576). This can be seen for example using GAP's Library of Tables of Marks.

Example

```
gap> tom:= TableOfMarks( "M22" );
TableOfMarks( "M22" )
gap> subord:= Size( UnderlyingGroup( tom ) ) / 770;
576
gap> ord:= OrdersTom( tom );;
```

```
gap> tomstabs:= Filtered( [ 1 .. Length( ord ) ], i -> ord[i] = subord );
[ 144 ]
```

The permutation representation of M_{22} on the right cosets of such a subgroup S is contained in the ATLAS of Group Representations.

Example

```
gap> DisplayAtlasInfo( "M22", NrMovedPoints, 770 );
Representations for G = M22:    (all refer to std. generators 1)
----------------------------
12: G <= Sym(770) rank 9, on cosets of (A4xA4):4 < 2^4:A6
```

Now we verify the information shown about the point stabilizer and about the maximal overgroups of S in M_{22}.

Example

```
gap> maxtom:= MaximalSubgroupsTom( tom );
[ [ 155, 154, 153, 152, 151, 150, 146, 145 ],
  [ 22, 77, 176, 176, 231, 330, 616, 672 ] ]
gap> List( tomstabs, i -> List( maxtom[1], j -> ContainedTom( tom, i, j ) ) );
[ [ 0, 10, 0, 0, 0, 0, 0, 0 ] ]
```

We see that the only maximal subgroups of M_{22} that contain S have index 77 in M_{22}. According to the ATLAS of Finite Groups, these maximal subgroups have the structure $2^4 : A_6$. From that and from the structure of A_6, we conclude that S has the structure $2^4 : (3^2 : 4)$.

Alternatively, we look at the permutation representation of degree 770. We fetch it from the ATLAS of Group Representations. There is exactly one nontrivial block system for this representation, with 77 blocks of length 10.

Example

```
gap> g:= AtlasGroup( "M22", NrMovedPoints, 770 );
<permutation group of size 443520 with 2 generators>
gap> allbl:= AllBlocks( g );;
gap> List( allbl, Length );
[ 10 ]
```

Furthermore, GAP computes that the point stabilizer S has the structure $(A_4 \times A_4) : 4$.

Example

```
gap> stab:= Stabilizer( g, 1 );;
gap> StructureDescription( stab : nice );
"(A4 x A4) : C4"
gap> blocks:= Orbit( g, allbl[1], OnSets );;
gap> act:= Action( g, blocks, OnSets );;
gap> StructureDescription( Stabilizer( act, 1 ) );
"(C2 x C2 x C2 x C2) : A6"
```

2.4.7 Example: Index 462 Subgroups in M_{22}

The ATLAS of Group Representations contains three degree 462 permutation representations of the group M_{22}.

Example

```
gap> DisplayAtlasInfo( "M22", NrMovedPoints, 462 );
Representations for G = M22:    (all refer to std. generators 1)
----------------------------
7: G <= Sym(462a) rank 5, on cosets of 2^4:A5 < 2^4:A6
8: G <= Sym(462b) rank 8, on cosets of 2^4:A5 < L3(4), 2^4:S5
9: G <= Sym(462c) rank 8, on cosets of 2^4:A5 < L3(4), 2^4:A6
```

The point stabilizers in these three representations have the structure $2^4 : A_5$. Using GAP's Library of Tables of Marks, we can show that these stabilizers are exactly the three classes of subgroups of order 960 in M_{22}. For that, we first verify that the group generators stored in GAP's table of marks coincide with the standard generators used by the ATLAS of Group Representations.

Example

```
gap> tom:= TableOfMarks( "M22" );
TableOfMarks( "M22" )
gap> genstom:= GeneratorsOfGroup( UnderlyingGroup( tom ) );;
gap> checkM22:= AtlasProgram( "M22", "check" );
rec( groupname := "M22", identifier := [ "M22", "M22G1-check1", 1, 1 ]
    , program := <straight line decision>, standardization := 1,
  version := "1" )
gap> ResultOfStraightLineDecision( checkM22.program, genstom );
true
```

There are indeed three classes of subgroups of order 960 in M_{22}.

Example

```
gap> ord:= OrdersTom( tom );;
gap> tomstabs:= Filtered( [ 1 .. Length( ord ) ], i -> ord[i] = 960 );
[ 147, 148, 149 ]
```

Now we compute representatives of these three classes in the three representations `462a`, `462b`, and `462c`. We see that each of the three classes occurs as a point stabilizer in exactly one of the three representations.

Example

```
gap> atlasreps:= AllAtlasGeneratingSetInfos( "M22", NrMovedPoints, 462 );
[ rec( charactername := "1a+21a+55a+154a+231a",
      constituents := [ 1, 2, 5, 7, 9 ], contents := "core",
      groupname := "M22", id := "a",
      identifier :=
        [ "M22", [ "M22G1-p462aB0.m1", "M22G1-p462aB0.m2" ], 1, 462 ],
      isPrimitive := false, p := 462, rankAction := 5,
      repname := "M22G1-p462aB0", repnr := 7, size := 443520,
      stabilizer := "2^4:A5 < 2^4:A6", standardization := 1,
      transitivity := 1, type := "perm" ),
  rec( charactername := "1a+21a^2+55a+154a+210a",
      constituents := [ 1, [ 2, 2 ], 5, 7, 8 ], contents := "core",
      groupname := "M22", id := "b",
      identifier :=
        [ "M22", [ "M22G1-p462bB0.m1", "M22G1-p462bB0.m2" ], 1, 462 ],
      isPrimitive := false, p := 462, rankAction := 8,
      repname := "M22G1-p462bB0", repnr := 8, size := 443520,
```

```
      stabilizer := "2^4:A5 < L3(4), 2^4:S5", standardization := 1,
      transitivity := 1, type := "perm" ),
  rec( charactername := "1a+21a^2+55a+154a+210a",
      constituents := [ 1, [ 2, 2 ], 5, 7, 8 ], contents := "core",
      groupname := "M22", id := "c",
      identifier :=
        [ "M22", [ "M22G1-p462cB0.m1", "M22G1-p462cB0.m2" ], 1, 462 ],
      isPrimitive := false, p := 462, rankAction := 8,
      repname := "M22G1-p462cB0", repnr := 9, size := 443520,
      stabilizer := "2^4:A5 < L3(4), 2^4:A6", standardization := 1,
      transitivity := 1, type := "perm" ) ]
gap> atlasreps:= List( atlasreps, AtlasGroup );;
gap> tomstabreps:= List( atlasreps, G -> List( tomstabs,
> i -> RepresentativeTomByGenerators( tom, i, GeneratorsOfGroup( G ) ) ) );;
gap> List( tomstabreps, x -> List( x, NrMovedPoints ) );
[ [ 462, 462, 461 ], [ 460, 462, 462 ], [ 462, 461, 462 ] ]
```

More precisely, we see that the point stabilizers in the three representations `462a`, `462b`, `462c` lie in the subgroup classes 149, 147, 148, respectively, of the table of marks.

The point stabilizers in the representations `462b` and `462c` are isomorphic, but not isomorphic with the point stabilizer in `462a`.

Example

```
gap> stabs:= List( atlasreps, G -> Stabilizer( G, 1 ) );;
gap> List( stabs, IdGroup );
[ [ 960, 11358 ], [ 960, 11357 ], [ 960, 11357 ] ]
gap> List( stabs, PerfectIdentification );
[ [ 960, 2 ], [ 960, 1 ], [ 960, 1 ] ]
```

The three representations are imprimitive. The containment of the point stabilizers in maximal subgroups of M_{22} can be computed using the table of marks of M_{22}.

Example

```
gap> maxtom:= MaximalSubgroupsTom( tom );
[ [ 155, 154, 153, 152, 151, 150, 146, 145 ],
  [ 22, 77, 176, 176, 231, 330, 616, 672 ] ]
gap> List( tomstabs, i -> List( maxtom[1], j -> ContainedTom( tom, i, j ) ) );
[ [ 21, 0, 0, 0, 1, 0, 0, 0 ], [ 21, 6, 0, 0, 0, 0, 0, 0 ],
  [ 0, 6, 0, 0, 0, 0, 0, 0 ] ]
```

We see:

- The point stabilizers in `462a` (subgroups in the class 149 of the table of marks) are contained only in maximal subgroups in class 154; these groups have the structure $2^4 : A_6$.
- The point stabilizers in `462b` (subgroups in the class 147) are contained in maximal subgroups in the classes 155 and 151; these groups have the structures $L_3(4)$ and $2^4 : S_5$, respectively.
- The point stabilizers in `462c` (subgroups in the class 148) are contained in maximal subgroups in the classes 155 and 154.

We identify the supergroups of the point stabilizers by computing the block systems.

Example

```
gap> bl:= List( atlasreps, AllBlocks );;
gap> List( bl, Length );
[ 1, 3, 2 ]
gap> List( bl, l -> List( l, Length ) );
[ [ 6 ], [ 21, 21, 2 ], [ 21, 6 ] ]
```

Note that the two block systems with blocks of length 21 for `462b` belong to the same supergroups (of the type $L_3(4)$); each of these subgroups fixes two different subsets of 21 points.

The representation `462a` is *multiplicity-free*, that is, it splits into a sum of pairwise nonisomorphic irreducible representations. This can be seen from the fact that the rank of this permutation representation (that is, the number of orbits of the point stabilizer) is five; each permutation representation with this property is multiplicity-free.

The other two representations have rank eight. We have seen the ranks in the overview that was shown by `DisplayAtlasInfo` (3.5.1) in the beginning. Now we compute the ranks from the permutation groups.

Example

```
gap> List( atlasreps, RankAction );
[ 5, 8, 8 ]
```

In fact the two representations `462b` and `462c` have the same permutation character. We check this by computing the possible permutation characters of degree 462 for M_{22}, and decomposing them into irreducible characters, using the character table from GAP's Character Table Library.

Example

```
gap> t:= CharacterTable( "M22" );;
gap> perms:= PermChars( t, 462 );
[ Character( CharacterTable( "M22" ),
  [ 462, 30, 3, 2, 2, 2, 3, 0, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "M22" ),
  [ 462, 30, 12, 2, 2, 2, 0, 0, 0, 0, 0, 0 ] ) ]
gap> MatScalarProducts( t, Irr( t ), perms );
[ [ 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0 ],
  [ 1, 2, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0 ] ]
```

In particular, we see that the rank eight characters are not multiplicity-free.

Chapter 3

The User Interface of the AtlasRep Package

The *user interface* is the part of the GAP interface that allows one to display information about the current contents of the database and to access individual data (perhaps by downloading them, see Section 4.2.1). The corresponding functions are described in this chapter. See Section 2.4 for some small examples how to use the functions of the interface.

Data extensions of the AtlasRep package are regarded as another part of the GAP interface, they are described in Chapter 5. Finally, the low level part of the interface is described in Chapter 7.

3.1 Accessing vs. Constructing Representations

Note that *accessing* the data means in particular that it is *not* the aim of this package to *construct* representations from known ones. For example, if at least one permutation representation for a group G is stored but no matrix representation in a positive characteristic p, say, then `OneAtlasGeneratingSetInfo` (3.5.6) returns `fail` when it is asked for a description of an available set of matrix generators for G in characteristic p, although such a representation can be obtained by reduction modulo p of an integral matrix representation, which in turn can be constructed from any permutation representation.

3.2 Group Names Used in the AtlasRep Package

When you access data via the AtlasRep package, you specify the group in question by an admissible *name*. Thus it is essential to know these names, which are called *the GAP names* of the group in the following.

For a group G, say, whose character table is available in GAP's Character Table Library (see [Bre22]), the admissible names of G are the admissible names of this character table. One such name is the `Identifier` (**Reference: Identifier for character tables**) value of the character table, see (**CTblLib: Admissible Names for Character Tables in CTblLib**). This name is usually very similar to the name used in the ATLAS of Finite Groups [CCN+85]. For example, `"M22"` is a GAP name of the Mathieu group M_{22}, `"12_1.U4(3).2_1"` is a GAP name of $12_1.U_4(3).2_1$, the two names `"S5"` and `"A5.2"` are GAP names of the symmetric group S_5, and the two names `"F3+"` and `"Fi24'"` are GAP names of the simple Fischer group Fi'_{24}.

When a GAP name is required as an input of a package function, this input is case insensitive. For example, both "A5" and "a5" are valid arguments of DisplayAtlasInfo (3.5.1).

Internally, for example as part of filenames (see Section 7.6), the package uses names that may differ from the GAP names; these names are called *ATLAS-file names*. For example, "A5", "TE62", and "F24" are ATLAS-file names. Of these, only "A5" is also a GAP name, but the other two are not; corresponding GAP names are "2E6(2)" and "Fi24'", respectively.

3.3 Standard Generators Used in the AtlasRep Package

For the general definition of *standard generators* of a group, see [Wil96].

Several *different* standard generators may be defined for a group, the definitions for each group that occurs in the ATLAS of Group Representations can be found at

http://atlas.math.rwth-aachen.de/Atlas/v3.

When one specifies the standardization, the i-th set of standard generators is denoted by the number i. Note that when more than one set of standard generators is defined for a group, one must be careful to use *compatible standardization*. For example, the straight line programs, straight line decisions and black box programs in the database refer to a specific standardization of their inputs. That is, a straight line program for computing generators of a certain subgroup of a group G is defined only for a specific set of standard generators of G, and applying the program to matrix or permutation generators of G but w. r. t. a different standardization may yield unpredictable results. Therefore the results returned by the functions described in this chapter contain information about the standardizations they refer to.

3.4 Class Names Used in the AtlasRep Package

For each straight line program (see AtlasProgram (3.5.4)) that is used to compute lists of class representatives, it is essential to describe the classes in which these elements lie. Therefore, in these cases the records returned by the function AtlasProgram (3.5.4) contain a component outputs with value a list of *class names*.

Currently we define these class names only for simple groups and certain extensions of simple groups, see Section 3.4.1. The function AtlasClassNames (3.4.2) can be used to compute the list of class names from the character table in the GAP Library.

3.4.1 Definition of ATLAS Class Names

For the definition of class names of an almost simple group, we assume that the ordinary character tables of all nontrivial normal subgroups are shown in the ATLAS of Finite Groups [CCN+85].

Each class name is a string consisting of the element order of the class in question followed by a combination of capital letters, digits, and the characters ' and - (starting with a capital letter). For example, 1A, 12A1, and 3B' denote the class that contains the identity element, a class of element order 12, and a class of element order 3, respectively.

1. For the table of a *simple* group, the class names are the same as returned by the two argument version of the GAP function ClassNames (**Reference: ClassNames**), cf. [CCN+85, Chapter 7, Section 5]: The classes are arranged w. r. t. increasing element order and for each element order w. r. t. decreasing centralizer order, the conjugacy classes that contain elements of order n are

named *n*A, *n*B, *n*C, ...; the alphabet used here is potentially infinite, and reads A, B, C, ..., Z, A1, B1, ..., A2, B2, ...

For example, the classes of the alternating group A_5 have the names 1A, 2A, 3A, 5A, and 5B.

2. Next we consider the case of an *upward extension* *G.A* of a simple group *G* by a *cyclic* group of order *A*. The ATLAS defines class names for each element *g* of *G.A* only w. r. t. the group *G.a*, say, that is generated by *G* and *g*; namely, there is a power of *g* (with the exponent coprime to the order of *g*) for which the class has a name of the same form as the class names for simple groups, and the name of the class of *g* w. r. t. *G.a* is then obtained from this name by appending a suitable number of dashes '. So dashed class names refer exactly to those classes that are not printed in the ATLAS.

   For example, those classes of the symmetric group S_5 that do not lie in A_5 have the names 2B, 4A, and 6A. The outer classes of the group $L_2(8).3$ have the names 3B, 6A, 9D, and 3B', 6A', 9D'. The outer elements of order 5 in the group $Sz(32).5$ lie in the classes with names 5B, 5B', 5B'', and 5B'''.

   In the group *G.A*, the class of *g* may fuse with other classes. The name of the class of *g* in *G.A* is obtained from the names of the involved classes of *G.a* by concatenating their names after removing the element order part from all of them except the first one.

   For example, the elements of order 9 in the group $L_2(27).6$ are contained in the subgroup $L_2(27).3$ but not in $L_2(27)$. In $L_2(27).3$, they lie in the classes 9A, 9A', 9B, and 9B'; in $L_2(27).6$, these classes fuse to 9AB and 9A'B'.

3. Now we define class names for *general upward extensions* *G.A* of a simple group *G*. Each element *g* of such a group lies in an upward extension *G.a* by a cyclic group, and the class names w. r. t. *G.a* are already defined. The name of the class of *g* in *G.A* is obtained by concatenating the names of the classes in the orbit of *G.A* on the classes of cyclic upward extensions of *G*, after ordering the names lexicographically and removing the element order part from all of them except the first one. An *exception* is the situation where dashed and non-dashed class names appear in an orbit; in this case, the dashed names are omitted.

   For example, the classes 21A and 21B of the group $U_3(5).3$ fuse in $U_3(5).S_3$ to the class 21AB, and the class 2B of $U_3(5).2$ fuses with the involution classes 2B', 2B'' in the groups $U_3(5).2'$ and $U_3(5).2''$ to the class 2B of $U_3(5).S_3$.

   It may happen that some names in the outputs component of a record returned by AtlasProgram (3.5.4) do not uniquely determine the classes of the corresponding elements. For example, the (algebraically conjugate) classes 39A and 39B of the group Co_1 have not been distinguished yet. In such cases, the names used contain a minus sign -, and mean "one of the classes in the range described by the name before and the name after the minus sign"; the element order part of the name does not appear after the minus sign. So the name 39A-B for the group Co_1 means 39A or 39B, and the name 20A-B''' for the group $Sz(32).5$ means one of the classes of element order 20 in this group (these classes lie outside the simple group *Sz*).

4. For a *downward extension* *m.G.A* of an almost simple group *G.A* by a cyclic group of order *m*, let π denote the natural epimorphism from *m.G.A* onto *G.A*. Each class name of *m.G.A* has the form nX_0, nX_1 etc., where nX is the class name of the image under π, and the indices 0, 1 etc. are chosen according to the position of the class in the lifting order rows for *G*, see [CCN+85, Chapter 7, Section 7, and the example in Section 8]).

For example, if $m = 6$ then `1A_1` and `1A_5` denote the classes containing the generators of the kernel of π, that is, central elements of order 6.

3.4.2 AtlasClassNames

▷ `AtlasClassNames(tbl)` (function)

Returns: a list of class names.

Let *tbl* be the ordinary or modular character table of a group G, say, that is almost simple or a downward extension of an almost simple group and such that *tbl* is an ATLAS table from the GAP Character Table Library, according to its `InfoText` (**Reference: InfoText**) value. Then `AtlasClassNames` returns the list of class names for G, as defined in Section 3.4.1. The ordering of class names is the same as the ordering of the columns of *tbl*.

(The function may work also for character tables that are not ATLAS tables, but then clearly the class names returned are somewhat arbitrary.)

Example

```
gap> AtlasClassNames( CharacterTable( "L3(4).3" ) );
[ "1A", "2A", "3A", "4ABC", "5A", "5B", "7A", "7B", "3B", "3B'",
  "3C", "3C'", "6B", "6B'", "15A", "15A'", "15B", "15B'", "21A",
  "21A'", "21B", "21B'" ]
gap> AtlasClassNames( CharacterTable( "U3(5).2" ) );
[ "1A", "2A", "3A", "4A", "5A", "5B", "5CD", "6A", "7AB", "8AB",
  "10A", "2B", "4B", "6D", "8C", "10B", "12B", "20A", "20B" ]
gap> AtlasClassNames( CharacterTable( "L2(27).6" ) );
[ "1A", "2A", "3AB", "7ABC", "13ABC", "13DEF", "14ABC", "2B", "4A",
  "26ABC", "26DEF", "28ABC", "28DEF", "3C", "3C'", "6A", "6A'",
  "9AB", "9A'B'", "6B", "6B'", "12A", "12A'" ]
gap> AtlasClassNames( CharacterTable( "L3(4).3.2_2" ) );
[ "1A", "2A", "3A", "4ABC", "5AB", "7A", "7B", "3B", "3C", "6B",
  "15A", "15B", "21A", "21B", "2C", "4E", "6E", "8D", "14A", "14B" ]
gap> AtlasClassNames( CharacterTable( "3.A6" ) );
[ "1A_0", "1A_1", "1A_2", "2A_0", "2A_1", "2A_2", "3A_0", "3B_0",
  "4A_0", "4A_1", "4A_2", "5A_0", "5A_1", "5A_2", "5B_0", "5B_1",
  "5B_2" ]
gap> AtlasClassNames( CharacterTable( "2.A5.2" ) );
[ "1A_0", "1A_1", "2A_0", "3A_0", "3A_1", "5AB_0", "5AB_1", "2B_0",
  "4A_0", "4A_1", "6A_0", "6A_1" ]
```

3.4.3 AtlasCharacterNames

▷ `AtlasCharacterNames(tbl)` (function)

Returns: a list of character names.

Let *tbl* be the ordinary or modular character table of a simple group. `AtlasCharacterNames` returns a list of strings, the i-th entry being the name of the i-th irreducible character of *tbl*; this name consists of the degree of this character followed by distinguishing lowercase letters.

Example

```
gap> AtlasCharacterNames( CharacterTable( "A5" ) );
[ "1a", "3a", "3b", "4a", "5a" ]
```

3.5 Accessing Data via AtlasRep

The examples shown in this section refer to the situation that no extensions have been notified, and to a perhaps outdated table of contents. That is, the current version of the database may contain more information than is shown here.

3.5.1 DisplayAtlasInfo

▷ `DisplayAtlasInfo(`*`[listofnames][,] [std][,] ["contents", sources][,] [...]`*`)` (function)

▷ `DisplayAtlasInfo(`*`gapname[, std][, ...]`*`)` (function)

This function lists the information available via the AtlasRep package, for the given input. There are essentially three ways of calling this function.

- If there is no argument or if the first argument is a list *listofnames* of strings that are GAP names of groups, `DisplayAtlasInfo` shows an overview of the known information.
- If the first argument is a string *gapname* that is a GAP name of a group, `DisplayAtlasInfo` shows an overview of the information that is available for this group.
- If the string `"contents"` is the only argument then the function shows which parts of the database are available; these are at least the `"core"` part, which means the data from the ATLAS of Group Representations, and the `"internal"` part, which means the data that are distributed with the AtlasRep package. Other parts can become available by calls to `AtlasOfGroupRepresentationsNotifyData` (5.1.1). Note that the shown numbers of locally available files depend on what has already been downloaded.

In each case, the information will be printed to the screen or will be fed into a pager, see Section 4.2.11. An interactive alternative to `DisplayAtlasInfo` is the function `BrowseAtlasInfo` (BrowseAtlasInfo???), see [BL18].

The following paragraphs describe the structure of the output in the two cases. Examples can be found in Section 3.5.2.

Called without arguments, `DisplayAtlasInfo` shows a general overview for all groups. If some information is available for the group G, say, then one line is shown for G, with the following columns.

`group`
: the GAP name of G (see Section 3.2),

`#`
: the number of faithful representations stored for G that satisfy the additional conditions given (see below),

`maxes`
: the number of available straight line programs for computing generators of maximal subgroups of G,

`cl`
: a + sign if at least one program for computing representatives of conjugacy classes of elements of G is stored,

`cyc`
: a + sign if at least one program for computing representatives of classes of maximally cyclic subgroups of G is stored,

`out` descriptions of outer automorphisms of G for which at least one program is stored,

`fnd` a + sign if at least one program is available for finding standard generators,

`chk` a + sign if at least one program is available for checking whether a set of generators is a set of standard generators, and

`prs` a + sign if at least one program is available that encodes a presentation.

Called with a list *listofnames* of strings that are GAP names of some groups, `DisplayAtlasInfo` prints the overview described above but restricted to the groups in this list.

In addition to or instead of *listofnames*, the string `"contents"` and a description *sources* of the data may be given about which the overview is formed. See below for admissible values of *sources*.

Called with a string *gapname* that is a GAP name of a group, `DisplayAtlasInfo` prints an overview of the information that is available for this group. One line is printed for each faithful representation, showing the number of this representation (which can be used in calls of `AtlasGenerators` (3.5.3)), and a string of one of the following forms; in both cases, *id* is a (possibly empty) string.

`G <= Sym(`*nid*`)`
denotes a permutation representation of degree n, for example `G <= Sym(40a)` and `G <= Sym(40b)` denote two (nonequivalent) representations of degree 40.

`G <= GL(`*nid*`,`*descr*`)`
denotes a matrix representation of dimension n over a coefficient ring described by *descr*, which can be a prime power, $\mathbb{Z}$ (denoting the ring of integers), a description of an algebraic extension field, $\mathbb{C}$ (denoting an unspecified algebraic extension field), or $\mathbb{Z}/m\mathbb{Z}$ for an integer m (denoting the ring of residues mod m); for example, `G <= GL(2a,4)` and `G <= GL(2b,4)` denote two (nonequivalent) representations of dimension 2 over the field with four elements.

After the representations, the programs available for *gapname* are listed. The following optional arguments can be used to restrict the overviews.

std must be a positive integer or a list of positive integers; if it is given then only those representations are considered that refer to the *std*-th set of standard generators or the *i*-th set of standard generators, for *i* in *std* (see Section 3.3),

`"contents"` **and** *sources*
for a string or a list of strings *sources*, restrict the data about which the overview is formed; if *sources* is the string `"core"` then only data from the ATLAS of Group Representations are considered, if *sources* is a string that denotes a data extension in the sense of a `dirid` argument of `AtlasOfGroupRepresentationsNotifyData` (5.1.1) then only the data that belong to this data extension are considered; also a list of such strings may be given, then the union of these data is considered,

`Identifier` **and** *id*
restrict to representations with `id` component in the list *id* (note that this component is itself a list, entering this list is not admissible), or satisfying the function *id*,

`IsPermGroup` **and** `true` **(or** `false`**)**
restrict to permutation representations (or to representations that are not permutation representations),

`NrMovedPoints` **and** *n*
: for a positive integer, a list of positive integers, or a property *n*, restrict to permutation representations of degree equal to *n*, or in the list *n*, or satisfying the function *n*,

`NrMovedPoints` **and the string** `"minimal"`
: restrict to faithful permutation representations of minimal degree (if this information is available),

`IsTransitive` **and a boolean value**
: restrict to transitive or intransitive permutation representations where this information is available (if the value `true` or `false` is given), or to representations for which this information is not available (if the value `fail` is given),

`IsPrimitive` **and a boolean value**
: restrict to primitive or imprimitive permutation representations where this information is available (if the value `true` or `false` is given), or to representations for which this information is not available (if the value `fail` is given),

`Transitivity` **and** *n*
: for a nonnegative integer, a list of nonnegative integers, or a property *n*, restrict to permutation representations for which the information is available that the transitivity is equal to *n*, or is in the list *n*, or satisfies the function *n*; if *n* is `fail` then restrict to all permutation representations for which this information is not available,

`RankAction` **and** *n*
: for a nonnegative integer, a list of nonnegative integers, or a property *n*, restrict to permutation representations for which the information is available that the rank is equal to *n*, or is in the list *n*, or satisfies the function *n*; if *n* is `fail` then restrict to all permutation representations for which this information is not available,

`IsMatrixGroup` **and** `true` **(or** `false`**)**
: restrict to matrix representations (or to representations that are not matrix representations),

`Characteristic` **and** *p*
: for a prime integer, a list of prime integers, or a property *p*, restrict to matrix representations over fields of characteristic equal to *p*, or in the list *p*, or satisfying the function *p* (representations over residue class rings that are not fields can be addressed by entering `fail` as the value of *p*),

`Dimension` **and** *n*
: for a positive integer, a list of positive integers, or a property *n*, restrict to matrix representations of dimension equal to *n*, or in the list *n*, or satisfying the function *n*,

`Characteristic`**,** *p*`,` `Dimension`**, and the string** `"minimal"`
: for a prime integer *p*, restrict to faithful matrix representations over fields of characteristic *p* that have minimal dimension (if this information is available),

`Ring` **and** *R*
: for a ring or a property *R*, restrict to matrix representations for which the information is available that the ring spanned by the matrix entries is contained in this ring or satisfies this property (note that the representation might be defined over a proper subring); if *R* is `fail` then restrict to all matrix representations for which this information is not available,

`Ring`**,** *R***,** `Dimension`**, and the string** `"minimal"`
: for a ring *R*, restrict to faithful matrix representations over this ring that have minimal dimension (if this information is available),

`Character` **and** *chi*
: for a class function or a list of class functions *chi*, restrict to representations with these characters (note that the underlying characteristic of the class function, see Section (**Reference: UnderlyingCharacteristic**), determines the characteristic of the representation),

`Character` **and** *name*
: for a string *name*, restrict to representations for which the character is known to have this name, according to the information shown by `DisplayAtlasInfo`; if the characteristic is not specified then it defaults to zero,

`Character` **and** *n*
: for a positive integer *n*, restrict to representations for which the character is known to be the *n*-th irreducible character in GAP's library character table of the group in question; if the characteristic is not specified then it defaults to zero,

`IsStraightLineProgram` **and** `true`
: restrict to straight line programs, straight line decisions (see Section 6.1), and black box programs (see Section 6.2), and

`IsStraightLineProgram` **and** `false`
: restrict to representations.

Note that the above conditions refer only to the information that is available without accessing the representations. For example, if it is not stored in the table of contents whether a permutation representation is primitive then this representation does not match an `IsPrimitive` condition in `DisplayAtlasInfo`.

If "minimality" information is requested and no available representation matches this condition then either no minimal representation is available or the information about the minimality is missing. See `MinimalRepresentationInfo` (6.3.1) for checking whether the minimality information is available for the group in question. Note that in the cases where the string `"minimal"` occurs as an argument, `MinimalRepresentationInfo` (6.3.1) is called with third argument `"lookup"`; this is because the stored information was precomputed just for the groups in the ATLAS of Group Representations, so trying to compute non-stored minimality information (using other available databases) will hardly be successful.

The representations are ordered as follows. Permutation representations come first (ordered according to their degrees), followed by matrix representations over finite fields (ordered first according to the field size and second according to the dimension), matrix representations over the integers, and then matrix representations over algebraic extension fields (both kinds ordered according to the dimension), the last representations are matrix representations over residue class rings (ordered first according to the modulus and second according to the dimension).

The maximal subgroups are ordered according to decreasing group order. For an extension $G.p$ of a simple group G by an outer automorphism of prime order p, this means that G is the first maximal subgroup and then come the extensions of the maximal subgroups of G and the novelties; so the n-th maximal subgroup of G and the n-th maximal subgroup of $G.p$ are in general not related. (This coincides with the numbering used for the `Maxes` (**CTblLib: Maxes**) attribute for character tables.)

3.5.2 Examples for DisplayAtlasInfo

Here are some examples how DisplayAtlasInfo (3.5.1) can be called, and how its output can be interpreted.

Example

```
gap> DisplayAtlasInfo( "contents" );
- AtlasRepAccessRemoteFiles: false

- AtlasRepDataDirectory: /home/you/gap/pkg/atlasrep/

ID       | address, version, files
---------+-----------------------------------------------
core     | http://atlas.math.rwth-aachen.de/Atlas/,
         | version 2019-04-08,
         | 10586 files locally available.
---------+-----------------------------------------------
internal | atlasrep/datapkg,
         | version 2019-05-06,
         | 276 files locally available.
---------+-----------------------------------------------
mfer     | http://www.math.rwth-aachen.de/~mfer/datagens/,
         | version 2015-10-06,
         | 34 files locally available.
---------+-----------------------------------------------
ctblocks | ctblocks/atlas/,
         | version 2019-04-08,
         | 121 files locally available.
```

Note: The above output does not fit to the rest of the manual examples, since data extensions except internal have been removed at the beginning of Chapter 2.

The output tells us that two data extensions have been notified in addition to the core data from the ATLAS of Group Representations and the (local) internal data distributed with the AtlasRep package. The files of the extension mfer must be downloaded before they can be read (but note that the access to remote files is disabled), and the files of the extension ctblocks are locally available in the ctblocks/atlas subdirectory of the GAP package directory. This table (in particular the numbers of locally available files) depends on your installation of the package and how many files you have already downloaded.

Example

```
gap> DisplayAtlasInfo( [ "M11", "A5" ] );
group |  #  | maxes | cl | cyc | out | fnd | chk | prs
------+-----+-------+----+-----+-----+-----+-----+----
M11   | 42  |     5 |  + |  +  |     |  +  |  +  |  +
A5*   | 18  |     3 |  + |     |     |     |  +  |  +
```

The above output means that the database provides 42 representations of the Mathieu group M_{11}, straight line programs for computing generators of representatives of all five classes of maximal subgroups, for computing representatives of the conjugacy classes of elements and of generators of maximally cyclic subgroups, contains no straight line program for applying outer automorphisms (well, in fact M_{11} admits no nontrivial outer automorphism), and contains straight line decisions that check a set

of generators or a set of group elements for being a set of standard generators. Analogously, 18 representations of the alternating group A_5 are available, straight line programs for computing generators of representatives of all three classes of maximal subgroups, and no straight line programs for computing representatives of the conjugacy classes of elements, of generators of maximally cyclic subgroups, and no for computing images under outer automorphisms; straight line decisions for checking the standardization of generators or group elements are available.

Example

```
gap> DisplayAtlasInfo( [ "M11", "A5" ], NrMovedPoints, 11 );
group | # | maxes | cl | cyc | out | fnd | chk | prs
------+---+-------+----+-----+-----+-----+-----+----
M11   | 1 |     5 |  + |  +  |     |  +  |  +  |  +
```

The given conditions restrict the overview to permutation representations on 11 points. The rows for all those groups are omitted for which no such representation is available, and the numbers of those representations are shown that satisfy the given conditions. In the above example, we see that no representation on 11 points is available for A_5, and exactly one such representation is available for M_{11}.

Example

```
gap> DisplayAtlasInfo( "A5", IsPermGroup, true );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
1: G <= Sym(5)  3-trans., on cosets of A4 (1st max.)
2: G <= Sym(6)  2-trans., on cosets of D10 (2nd max.)
3: G <= Sym(10) rank 3, on cosets of S3 (3rd max.)
gap> DisplayAtlasInfo( "A5", NrMovedPoints, [ 4 .. 9 ] );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
1: G <= Sym(5) 3-trans., on cosets of A4 (1st max.)
2: G <= Sym(6) 2-trans., on cosets of D10 (2nd max.)
```

The first three representations stored for A_5 are (in fact primitive) permutation representations.

Example

```
gap> DisplayAtlasInfo( "A5", Dimension, [ 1 .. 3 ] );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
 8: G <= GL(2a,4)                character 2a
 9: G <= GL(2b,4)                character 2b
10: G <= GL(3,5)                 character 3a
12: G <= GL(3a,9)                character 3a
13: G <= GL(3b,9)                character 3b
17: G <= GL(3a,Field([Sqrt(5)])) character 3a
18: G <= GL(3b,Field([Sqrt(5)])) character 3b
gap> DisplayAtlasInfo( "A5", Characteristic, 0 );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
14: G <= GL(4,Z)                 character 4a
15: G <= GL(5,Z)                 character 5a
16: G <= GL(6,Z)                 character 3ab
17: G <= GL(3a,Field([Sqrt(5)])) character 3a
18: G <= GL(3b,Field([Sqrt(5)])) character 3b
```

The representations with number between 4 and 13 are (in fact irreducible) matrix representations over various finite fields, those with numbers 14 to 16 are integral matrix representations, and the last two are matrix representations over the field generated by $\sqrt{5}$ over the rational number field.

Example

```
gap> DisplayAtlasInfo( "A5", Identifier, "a" );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
 4: G <= GL(4a,2)                character 4a
 8: G <= GL(2a,4)                character 2a
12: G <= GL(3a,9)                character 3a
17: G <= GL(3a,Field([Sqrt(5)])) character 3a
```

Each of the representations with the numbers $4, 8, 12$, and 17 is labeled with the distinguishing letter `a`.

Example

```
gap> DisplayAtlasInfo( "A5", NrMovedPoints, IsPrimeInt );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
1: G <= Sym(5) 3-trans., on cosets of A4 (1st max.)
gap> DisplayAtlasInfo( "A5", Characteristic, IsOddInt );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
 6: G <= GL(4,3)  character 4a
 7: G <= GL(6,3)  character 3ab
10: G <= GL(3,5)  character 3a
11: G <= GL(5,5)  character 5a
12: G <= GL(3a,9) character 3a
13: G <= GL(3b,9) character 3b
gap> DisplayAtlasInfo( "A5", Dimension, IsPrimeInt );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
 8: G <= GL(2a,4)                character 2a
 9: G <= GL(2b,4)                character 2b
10: G <= GL(3,5)                 character 3a
11: G <= GL(5,5)                 character 5a
12: G <= GL(3a,9)                character 3a
13: G <= GL(3b,9)                character 3b
15: G <= GL(5,Z)                 character 5a
17: G <= GL(3a,Field([Sqrt(5)])) character 3a
18: G <= GL(3b,Field([Sqrt(5)])) character 3b
gap> DisplayAtlasInfo( "A5", Ring, IsFinite and IsPrimeField );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
 4: G <= GL(4a,2) character 4a
 5: G <= GL(4b,2) character 2ab
 6: G <= GL(4,3)  character 4a
 7: G <= GL(6,3)  character 3ab
10: G <= GL(3,5)  character 3a
11: G <= GL(5,5)  character 5a
```

The above examples show how the output can be restricted using a property (a unary function that returns either `true` or `false`) that follows `NrMovedPoints` (**Reference: NrMovedPoints for a per-**

mutation), `Characteristic` (**Reference: Characteristic**), `Dimension` (**Reference: Dimension**), or `Ring` (**Reference: Ring**) in the argument list of `DisplayAtlasInfo` (3.5.1).

Example

```
gap> DisplayAtlasInfo( "A5", IsStraightLineProgram, true );
Programs for G = A5:    (all refer to std. generators 1)
--------------------
- class repres.*
- presentation
- maxes (all 3):
  1:  A4
  2:  D10
  3:  S3
- std. gen. checker:
  (check)
  (pres)
```

Straight line programs are available for computing generators of representatives of the three classes of maximal subgroups of A_5, and a straight line decision for checking whether given generators are in fact standard generators is available as well as a presentation in terms of standard generators, see `AtlasProgram` (3.5.4).

3.5.3 AtlasGenerators

▷ `AtlasGenerators(`*gapname, repnr[, maxnr]*`)` (function)

▷ `AtlasGenerators(`*identifier*`)` (function)

Returns: a record containing generators for a representation, or `fail`.

In the first form, *gapname* must be a string denoting a GAP name (see Section 3.2) of a group, and *repnr* a positive integer. If at least *repnr* representations for the group with GAP name *gapname* are available then `AtlasGenerators`, when called with *gapname* and *repnr*, returns an immutable record describing the *repnr*-th representation; otherwise `fail` is returned. If a third argument *maxnr*, a positive integer, is given then an immutable record describing the restriction of the *repnr*-th representation to the *maxnr*-th maximal subgroup is returned.

The result record has at least the following components.

`contents`
: the identifier of the part of the database to which the generators belong, for example `"core"` or `"internal"`,

`generators`
: a list of generators for the group,

`groupname`
: the GAP name of the group (see Section 3.2),

`identifier`
: a GAP object (a list of filenames plus additional information) that uniquely determines the representation, see Section 7.7; the value can be used as `identifier` argument of `AtlasGenerators`.

`repname`
: a string that is an initial part of the filenames of the generators.

`repnr`
: the number of the representation in the current session, equal to the argument *repnr* if this is given.

`standardization`
: the positive integer denoting the underlying standard generators,

`type`
: a string that describes the type of the representation (`"perm"` for a permutation representation, `"matff"` for a matrix representation over a finite field, `"matint"` for a matrix representation over the ring of integers, `"matalg"` for a matrix representation over an algebraic number field).

Additionally, the following *describing components* may be available if they are known, and depending on the data type of the representation.

`size`
: the group order,

`id`
: the distinguishing string as described for `DisplayAtlasInfo` (3.5.1),

`charactername`
: a string that describes the character of the representation,

`constituents`
: a list of positive integers denoting the positions of the irreducible constituents of the character of the representation,

`p` **(for permutation representations)**
: for the number of moved points,

`dim` **(for matrix representations)**
: the dimension of the matrices,

`ring` **(for matrix representations)**
: the ring generated by the matrix entries,

`transitivity` **(for permutation representations)**
: a nonnegative integer, see `Transitivity` (**Reference: Transitivity**),

`orbits` **(for intransitive permutation representations)**
: the sorted list of orbit lengths on the set of moved points,

`rankAction` **(for transitive permutation representations)**
: the number of orbits of the point stabilizer on the set of moved points, see `RankAction` (**Reference: RankAction**),

`stabilizer` **(for transitive permutation representations)**
: a string that describes the structure of the point stabilizers,

`isPrimitive` **(for transitive permutation representations)**
: `true` if the point stabilizers are maximal subgroups, and `false` otherwise,

`maxnr` **(for primitive permutation representations)**
: the number of the class of maximal subgroups that contains the point stabilizers, w. r. t. the `Maxes` (**CTblLib: Maxes**) list.

It should be noted that the number *repnr* refers to the number shown by `DisplayAtlasInfo` (3.5.1) *in the current session*; it may be that after the addition of new representations (for example after loading a package that provides some), *repnr* refers to another representation.

The alternative form of `AtlasGenerators`, with only argument *identifier*, can be used to fetch the result record with `identifier` value equal to *identifier*. The purpose of this variant is to access the *same* representation also in *different* GAP sessions.

Example

```
gap> gens1:= AtlasGenerators( "A5", 1 );
rec( charactername := "1a+4a", constituents := [ 1, 4 ],
  contents := "core", generators := [ (1,2)(3,4), (1,3,5) ],
  groupname := "A5", id := "",
  identifier := [ "A5", [ "A5G1-p5B0.m1", "A5G1-p5B0.m2" ], 1, 5 ],
  isPrimitive := true, maxnr := 1, p := 5, rankAction := 2,
  repname := "A5G1-p5B0", repnr := 1, size := 60, stabilizer := "A4",
  standardization := 1, transitivity := 3, type := "perm" )
gap> gens8:= AtlasGenerators( "A5", 8 );
rec( charactername := "2a", constituents := [ 2 ], contents := "core",
  dim := 2,
  generators := [ [ [ Z(2)^0, 0*Z(2) ], [ Z(2^2), Z(2)^0 ] ],
      [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, Z(2)^0 ] ] ], groupname := "A5",
  id := "a",
  identifier := [ "A5", [ "A5G1-f4r2aB0.m1", "A5G1-f4r2aB0.m2" ], 1,
      4 ], repname := "A5G1-f4r2aB0", repnr := 8, ring := GF(2^2),
  size := 60, standardization := 1, type := "matff" )
gap> gens17:= AtlasGenerators( "A5", 17 );
rec( charactername := "3a", constituents := [ 2 ], contents := "core",
  dim := 3,
  generators :=
    [ [ [ -1, 0, 0 ], [ 0, -1, 0 ], [ -E(5)-E(5)^4, -E(5)-E(5)^4, 1 ]
        ], [ [ 0, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ] ] ],
  groupname := "A5", id := "a",
  identifier := [ "A5", "A5G1-Ar3aB0.g", 1, 3 ],
  polynomial := [ -1, 1, 1 ], repname := "A5G1-Ar3aB0", repnr := 17,
  ring := NF(5,[ 1, 4 ]), size := 60, standardization := 1,
  type := "matalg" )
```

Each of the above pairs of elements generates a group isomorphic to A_5.

Example

```
gap> gens1max2:= AtlasGenerators( "A5", 1, 2 );
rec( charactername := "1a+4a", constituents := [ 1, 4 ],
  contents := "core", generators := [ (1,2)(3,4), (2,3)(4,5) ],
  groupname := "D10", id := "",
  identifier := [ "A5", [ "A5G1-p5B0.m1", "A5G1-p5B0.m2" ], 1, 5, 2 ],
  isPrimitive := true, maxnr := 1, p := 5, rankAction := 2,
```

```
    repname := "A5G1-p5B0", repnr := 1, size := 10, stabilizer := "A4",
    standardization := 1, transitivity := 3, type := "perm" )
gap> id:= gens1max2.identifier;;
gap> gens1max2 = AtlasGenerators( id );
true
gap> max2:= Group( gens1max2.generators );;
gap> Size( max2 );
10
gap> IdGroup( max2 ) = IdGroup( DihedralGroup( 10 ) );
true
```

The elements stored in `gens1max2.generators` describe the restriction of the first representation of A_5 to a group in the second class of maximal subgroups of A_5 according to the list in the ATLAS of Finite Groups [CCN+85]; this subgroup is isomorphic to the dihedral group D_{10}.

3.5.4 AtlasProgram

▷ `AtlasProgram(`*gapname[, std][, "contents", sources][, "version", vers], ...*`)` (function)

▷ `AtlasProgram(`*identifier*`)` (function)

Returns: a record containing a program, or `fail`.

In the first form, *gapname* must be a string denoting a GAP name (see Section 3.2) of a group G, say. If the database contains a straight line program (see Section (**Reference: Straight Line Programs**)) or straight line decision (see Section 6.1) or black box program (see Section 6.2) as described by the arguments indicated by `...` (see below) then `AtlasProgram` returns an immutable record containing this program. Otherwise `fail` is returned.

If the optional argument *std* is given, only those straight line programs/decisions are considered that take generators from the *std*-th set of standard generators of G as input, see Section 3.3.

If the optional arguments `"contents"` and *sources* are given then the latter must be either a string or a list of strings, with the same meaning as described for `DisplayAtlasInfo` (3.5.1).

If the optional arguments `"version"` and *vers* are given then the latter must be either a number or a list of numbers, and only those straight line programs/decisions are considered whose version number fits to *vers*.

The result record has at least the following components.

`groupname`
: the string *gapname*,

`identifier`
: a GAP object (a list of filenames plus additional information) that uniquely determines the program; the value can be used as *identifier* argument of `AtlasProgram` (see below),

`program`
: the required straight line program/decision, or black box program,

`standardization`
: the positive integer denoting the underlying standard generators of G,

`version`
: the substring of the filename of the program that denotes the version of the program.

If the program computes generators of the restriction to a maximal subgroup then also the following components are present.

`size`
: the order of the maximal subgroup,

`subgroupname`
: a string denoting a name of the maximal subgroup.

In the first form, the arguments indicated by . . . must be as follows.

(the string `"maxes"` and) a positive integer *maxnr*
: the required program computes generators of the *maxnr*-th maximal subgroup of the group with GAP name *gapname*.

  In this case, the result record of `AtlasProgram` also may contain a component `size`, whose value is the order of the maximal subgroup in question.

the string `"maxes"` and two positive integers *maxnr* **and** *std*2
: the required program computes standard generators of the *maxnr*-th maximal subgroup of the group with GAP name *gapname*, w. r. t. the standardization *std*2.

  A prescribed `"version"` parameter refers to the straight line program for computing the restriction, not to the program for standardizing the result of the restriction.

  The meaning of the component `size` in the result, if present, is the same as in the previous case.

the string `"maxstd"` and three positive integers *maxnr*, *vers*, *substd*
: the required program computes standard generators of the *maxnr*-th maximal subgroup of the group with GAP name *gapname* w. r. t. standardization *substd*; in this case, the inputs of the program are *not* standard generators of the group with GAP name *gapname* but the outputs of the straight line program with version *vers* for computing generators of its *maxnr*-th maximal subgroup.

the string `"kernel"` and a string *factname*
: the required program computes generators of the kernel of an epimorphism from *G* to a group with GAP name *factname*.

one of the strings `"classes"` or `"cyclic"`
: the required program computes representatives of conjugacy classes of elements or representatives of generators of maximally cyclic subgroups of *G*, respectively.

  See [BSWW01] and [SWW00] for the background concerning these straight line programs. In these cases, the result record of `AtlasProgram` also contains a component `outputs`, whose value is a list of class names of the outputs, as described in Section 3.4.

the string `"cyc2ccl"` (and the string *vers***)**
: the required program computes representatives of conjugacy classes of elements from representatives of generators of maximally cyclic subgroups of *G*. Thus the inputs are the outputs of the program of type `"cyclic"` whose version is *vers*.

the strings `"cyc2ccl"`**,** *vers*1**,** `"version"`**,** *vers*2
the required program computes representatives of conjugacy classes of elements from representatives of generators of maximally cyclic subgroups of G, where the inputs are the outputs of the program of type `"cyclic"` whose version is *vers*1 and the required program itself has version *vers*2.

the strings `"automorphism"` **and** *autname*
the required program computes images of standard generators under the outer automorphism of G that is given by this string.

Note that a value `"2"` of *autname* means that the square of the automorphism is an inner automorphism of G (not necessarily the identity mapping) but the automorphism itself is not.

the string `"check"`
the required result is a straight line decision that takes a list of generators for G and returns `true` if these generators are standard generators of G w. r. t. the standardization *`std`*, and `false` otherwise.

the string `"presentation"`
the required result is a straight line decision that takes a list of group elements and returns `true` if these elements are standard generators of G w. r. t. the standardization *`std`*, and `false` otherwise.

See `StraightLineProgramFromStraightLineDecision` (6.1.9) for an example how to derive defining relators for G in terms of the standard generators from such a straight line decision.

the string `"find"`
the required result is a black box program that takes G and returns a list of standard generators of G, w. r. t. the standardization *`std`*.

the string `"restandardize"` **and an integer** *std*2
the required result is a straight line program that computes standard generators of G w. r. t. the *std*2-th set of standard generators of G; in this case, the argument *`std`* must be given.

the strings `"other"` **and** *descr*
the required program is described by *descr*.

The second form of `AtlasProgram`, with only argument the list *`identifier`*, can be used to fetch the result record with `identifier` value equal to *`identifier`*.

Example

```
gap> prog:= AtlasProgram( "A5", 2 );
rec( groupname := "A5", identifier := [ "A5", "A5G1-max2W1", 1 ],
  program := <straight line program>, size := 10,
  standardization := 1, subgroupname := "D10", version := "1" )
gap> StringOfResultOfStraightLineProgram( prog.program, [ "a", "b" ] );
"[ a, bbab ]"
gap> gens1:= AtlasGenerators( "A5", 1 );
rec( charactername := "1a+4a", constituents := [ 1, 4 ],
  contents := "core", generators := [ (1,2)(3,4), (1,3,5) ],
  groupname := "A5", id := "",
  identifier := [ "A5", [ "A5G1-p5B0.m1", "A5G1-p5B0.m2" ], 1, 5 ],
  isPrimitive := true, maxnr := 1, p := 5, rankAction := 2,
```

```
  repname := "A5G1-p5B0", repnr := 1, size := 60, stabilizer := "A4",
  standardization := 1, transitivity := 3, type := "perm" )
gap> maxgens:= ResultOfStraightLineProgram( prog.program,
>                  gens1.generators );
[ (1,2)(3,4), (2,3)(4,5) ]
gap> maxgens = gens1max2.generators;
true
```

The above example shows that for restricting representations given by standard generators to a maximal subgroup of A_5, we can also fetch and apply the appropriate straight line program. Such a program (see (**Reference: Straight Line Programs**)) takes standard generators of a group –in this example A_5– as its input, and returns a list of elements in this group –in this example generators of the D_{10} subgroup we had met above– which are computed essentially by evaluating structured words in terms of the standard generators.

Example

```
gap> prog:= AtlasProgram( "J1", "cyclic" );
rec( groupname := "J1", identifier := [ "J1", "J1G1-cycW1", 1 ],
  outputs := [ "6A", "7A", "10B", "11A", "15B", "19A" ],
  program := <straight line program>, standardization := 1,
  version := "1" )
gap> gens:= GeneratorsOfGroup( FreeGroup( "x", "y" ) );;
gap> ResultOfStraightLineProgram( prog.program, gens );
[ (x*y)^2*((y*x)^2*y^2*x)^2*y^2, x*y, (x*(y*x*y)^2)^2*y,
  (x*y*x*(y*x*y)^3*x*y^2)^2*x*y*x*(y*x*y)^2*y, x*y*x*(y*x*y)^2*y,
  (x*y)^2*y ]
```

The above example shows how to fetch and use straight line programs for computing generators of representatives of maximally cyclic subgroups of a given group.

3.5.5 AtlasProgramInfo

▷ `AtlasProgramInfo(`*gapname[, std][, "contents", sources][, "version", vers], ...*`)` (function)

Returns: a record describing a program, or `fail`.

`AtlasProgramInfo` takes the same arguments as `AtlasProgram` (3.5.4), and returns a similar result. The only difference is that the records returned by `AtlasProgramInfo` have no components `program` and `outputs`. The idea is that one can use `AtlasProgramInfo` for testing whether the program in question is available at all, but without downloading files. The `identifier` component of the result of `AtlasProgramInfo` can then be used to fetch the program with `AtlasProgram` (3.5.4).

Example

```
gap> AtlasProgramInfo( "J1", "cyclic" );
rec( groupname := "J1", identifier := [ "J1", "J1G1-cycW1", 1 ],
  standardization := 1, version := "1" )
```

3.5.6 OneAtlasGeneratingSetInfo

▷ `OneAtlasGeneratingSetInfo(`*[gapname][,] [std][,] [...]*`)` (function)

Returns: a record describing a representation that satisfies the conditions, or `fail`.

Let *gapname* be a string denoting a GAP name (see Section 3.2) of a group G, say. If the database contains at least one representation for G with the required properties then OneAtlasGeneratingSetInfo returns a record r whose components are the same as those of the records returned by AtlasGenerators (3.5.3), except that the component generators is not contained, and an additional component givenRing is present if Ring is one of the arguments in the function call.

The information in givenRing can be used later to construct the matrices over the prescribed ring. Note that this ring may be for example a domain constructed with AlgebraicExtension (**Reference: AlgebraicExtension**) instead of a field of cyclotomics or of a finite field constructed with GF (**Reference: GF for field size**).

The component identifier of r can be used as input for AtlasGenerators (3.5.3) in order to fetch the generators. If no representation satisfying the given conditions is available then fail is returned.

If the argument *std* is given then it must be a positive integer or a list of positive integers, denoting the sets of standard generators w. r. t. which the representation shall be given (see Section 3.3).

The argument *gapname* can be missing (then all available groups are considered), or a list of group names can be given instead.

Further restrictions can be entered as arguments, with the same meaning as described for DisplayAtlasInfo (3.5.1). The result of OneAtlasGeneratingSetInfo describes the first generating set for G that matches the restrictions, in the ordering shown by DisplayAtlasInfo (3.5.1).

Note that even in the case that the user preference AtlasRepAccessRemoteFiles has the value true (see Section 4.2.1), OneAtlasGeneratingSetInfo does *not* attempt to *transfer* remote data files, just the table of contents is evaluated. So this function (as well as AllAtlasGeneratingSetInfos (3.5.7)) can be used to check for the availability of certain representations, and afterwards one can call AtlasGenerators (3.5.3) for those representations one wants to work with.

In the following example, we try to access information about permutation representations for the alternating group A_5.

Example

```
gap> info:= OneAtlasGeneratingSetInfo( "A5" );
rec( charactername := "1a+4a", constituents := [ 1, 4 ],
  contents := "core", groupname := "A5", id := "",
  identifier := [ "A5", [ "A5G1-p5B0.m1", "A5G1-p5B0.m2" ], 1, 5 ],
  isPrimitive := true, maxnr := 1, p := 5, rankAction := 2,
  repname := "A5G1-p5B0", repnr := 1, size := 60, stabilizer := "A4",
  standardization := 1, transitivity := 3, type := "perm" )
gap> gens:= AtlasGenerators( info.identifier );
rec( charactername := "1a+4a", constituents := [ 1, 4 ],
  contents := "core", generators := [ (1,2)(3,4), (1,3,5) ],
  groupname := "A5", id := "",
  identifier := [ "A5", [ "A5G1-p5B0.m1", "A5G1-p5B0.m2" ], 1, 5 ],
  isPrimitive := true, maxnr := 1, p := 5, rankAction := 2,
  repname := "A5G1-p5B0", repnr := 1, size := 60, stabilizer := "A4",
  standardization := 1, transitivity := 3, type := "perm" )
gap> info = OneAtlasGeneratingSetInfo( "A5", IsPermGroup, true );
true
gap> info = OneAtlasGeneratingSetInfo( "A5", NrMovedPoints, "minimal" );
true
gap> info = OneAtlasGeneratingSetInfo( "A5", NrMovedPoints, [ 1 .. 10 ] );
```

```
true
gap> OneAtlasGeneratingSetInfo( "A5", NrMovedPoints, 20 );
fail
```

Note that a permutation representation of degree 20 could be obtained by taking twice the primitive representation on 10 points; however, the database does not store this imprimitive representation (cf. Section 3.1).

We continue this example. Next we access matrix representations of A_5.

Example

```
gap> info:= OneAtlasGeneratingSetInfo( "A5", IsMatrixGroup, true );
rec( charactername := "4a", constituents := [ 4 ], contents := "core",
  dim := 4, groupname := "A5", id := "a",
  identifier := [ "A5", [ "A5G1-f2r4aB0.m1", "A5G1-f2r4aB0.m2" ], 1,
      2 ], repname := "A5G1-f2r4aB0", repnr := 4, ring := GF(2),
  size := 60, standardization := 1, type := "matff" )
gap> gens:= AtlasGenerators( info.identifier );
rec( charactername := "4a", constituents := [ 4 ], contents := "core",
  dim := 4,
  generators := [ <an immutable 4x4 matrix over GF2>,
      <an immutable 4x4 matrix over GF2> ], groupname := "A5",
  id := "a",
  identifier := [ "A5", [ "A5G1-f2r4aB0.m1", "A5G1-f2r4aB0.m2" ], 1,
      2 ], repname := "A5G1-f2r4aB0", repnr := 4, ring := GF(2),
  size := 60, standardization := 1, type := "matff" )
gap> info = OneAtlasGeneratingSetInfo( "A5", Dimension, 4 );
true
gap> info = OneAtlasGeneratingSetInfo( "A5", Characteristic, 2 );
true
gap> info2:= OneAtlasGeneratingSetInfo( "A5", Ring, GF(2) );;
gap> info.identifier = info2.identifier;
true
gap> OneAtlasGeneratingSetInfo( "A5", Characteristic, [2,5], Dimension, 2 );
rec( charactername := "2a", constituents := [ 2 ], contents := "core",
  dim := 2, groupname := "A5", id := "a",
  identifier := [ "A5", [ "A5G1-f4r2aB0.m1", "A5G1-f4r2aB0.m2" ], 1,
      4 ], repname := "A5G1-f4r2aB0", repnr := 8, ring := GF(2^2),
  size := 60, standardization := 1, type := "matff" )
gap> OneAtlasGeneratingSetInfo( "A5", Characteristic, [2,5], Dimension, 1 );
fail
gap> info:= OneAtlasGeneratingSetInfo( "A5", Characteristic, 0,
>                                            Dimension, 4 );
rec( charactername := "4a", constituents := [ 4 ], contents := "core",
  dim := 4, groupname := "A5", id := "",
  identifier := [ "A5", "A5G1-Zr4B0.g", 1, 4 ],
  repname := "A5G1-Zr4B0", repnr := 14, ring := Integers, size := 60,
  standardization := 1, type := "matint" )
gap> gens:= AtlasGenerators( info.identifier );
rec( charactername := "4a", constituents := [ 4 ], contents := "core",
  dim := 4,
  generators :=
    [
      [ [ 1, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 1, 0, 0 ],
```

```
          [ -1, -1, -1, -1 ] ],
      [ [ 0, 1, 0, 0 ], [ 0, 0, 0, 1 ], [ 0, 0, 1, 0 ],
          [ 1, 0, 0, 0 ] ] ], groupname := "A5", id := "",
  identifier := [ "A5", "A5G1-Zr4B0.g", 1, 4 ],
  repname := "A5G1-Zr4B0", repnr := 14, ring := Integers, size := 60,
  standardization := 1, type := "matint" )
gap> info = OneAtlasGeneratingSetInfo( "A5", Ring, Integers );
true
gap> info2:= OneAtlasGeneratingSetInfo( "A5", Ring, CF(37) );;
gap> info = info2;
false
gap> Difference( RecNames( info2 ), RecNames( info ) );
[ "givenRing" ]
gap> info2.givenRing;
CF(37)
gap> OneAtlasGeneratingSetInfo( "A5", Ring, Integers mod 77 );
fail
gap> info:= OneAtlasGeneratingSetInfo( "A5", Ring, CF(5), Dimension, 3 );
rec( charactername := "3a", constituents := [ 2 ], contents := "core",
  dim := 3, givenRing := CF(5), groupname := "A5", id := "a",
  identifier := [ "A5", "A5G1-Ar3aB0.g", 1, 3 ],
  polynomial := [ -1, 1, 1 ], repname := "A5G1-Ar3aB0", repnr := 17,
  ring := NF(5,[ 1, 4 ]), size := 60, standardization := 1,
  type := "matalg" )
gap> gens:= AtlasGenerators( info );
rec( charactername := "3a", constituents := [ 2 ], contents := "core",
  dim := 3,
  generators :=
    [ [ [ -1, 0, 0 ], [ 0, -1, 0 ], [ -E(5)-E(5)^4, -E(5)-E(5)^4, 1 ]
        ], [ [ 0, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ] ] ],
  givenRing := CF(5), groupname := "A5", id := "a",
  identifier := [ "A5", "A5G1-Ar3aB0.g", 1, 3 ],
  polynomial := [ -1, 1, 1 ], repname := "A5G1-Ar3aB0", repnr := 17,
  ring := NF(5,[ 1, 4 ]), size := 60, standardization := 1,
  type := "matalg" )
gap> gens2:= AtlasGenerators( info.identifier );;
gap> Difference( RecNames( gens ), RecNames( gens2 ) );
[ "givenRing" ]
gap> OneAtlasGeneratingSetInfo( "A5", Ring, GF(17) );
fail
```

3.5.7 AllAtlasGeneratingSetInfos

▷ `AllAtlasGeneratingSetInfos(`*`[gapname][,] [std][,] [...]`*`)` (function)

Returns: the list of all records describing representations that satisfy the conditions.

`AllAtlasGeneratingSetInfos` is similar to `OneAtlasGeneratingSetInfo` (3.5.6). The difference is that the list of *all* records describing the available representations with the given properties is returned instead of just one such component. In particular an empty list is returned if no such representation is available.

Example

```
gap> AllAtlasGeneratingSetInfos( "A5", IsPermGroup, true );
```

```
[ rec( charactername := "1a+4a", constituents := [ 1, 4 ],
      contents := "core", groupname := "A5", id := "",
      identifier := [ "A5", [ "A5G1-p5B0.m1", "A5G1-p5B0.m2" ], 1, 5 ]
        , isPrimitive := true, maxnr := 1, p := 5, rankAction := 2,
      repname := "A5G1-p5B0", repnr := 1, size := 60,
      stabilizer := "A4", standardization := 1, transitivity := 3,
      type := "perm" ),
  rec( charactername := "1a+5a", constituents := [ 1, 5 ],
      contents := "core", groupname := "A5", id := "",
      identifier := [ "A5", [ "A5G1-p6B0.m1", "A5G1-p6B0.m2" ], 1, 6 ]
        , isPrimitive := true, maxnr := 2, p := 6, rankAction := 2,
      repname := "A5G1-p6B0", repnr := 2, size := 60,
      stabilizer := "D10", standardization := 1, transitivity := 2,
      type := "perm" ),
  rec( charactername := "1a+4a+5a", constituents := [ 1, 4, 5 ],
      contents := "core", groupname := "A5", id := "",
      identifier := [ "A5", [ "A5G1-p10B0.m1", "A5G1-p10B0.m2" ], 1,
          10 ], isPrimitive := true, maxnr := 3, p := 10,
      rankAction := 3, repname := "A5G1-p10B0", repnr := 3,
      size := 60, stabilizer := "S3", standardization := 1,
      transitivity := 1, type := "perm" ) ]
```

Note that a matrix representation in any characteristic can be obtained by reducing a permutation representation or an integral matrix representation; however, the database does not *store* such a representation (cf. Section 3.1).

3.5.8 AtlasGroup

▷ `AtlasGroup(`*`[gapname][,] [std][,] [...]`*`)` (function)
▷ `AtlasGroup(`*`identifier`*`)` (function)

Returns: a group that satisfies the conditions, or `fail`.

`AtlasGroup` takes the same arguments as `OneAtlasGeneratingSetInfo` (3.5.6), and returns the group generated by the `generators` component of the record that is returned by `OneAtlasGeneratingSetInfo` (3.5.6) with these arguments; if `OneAtlasGeneratingSetInfo` (3.5.6) returns `fail` then also `AtlasGroup` returns `fail`.

Example

```
gap> g:= AtlasGroup( "A5" );
Group([ (1,2)(3,4), (1,3,5) ])
```

Alternatively, it is possible to enter exactly one argument, a record *identifier* as returned by `OneAtlasGeneratingSetInfo` (3.5.6) or `AllAtlasGeneratingSetInfos` (3.5.7), or the `identifier` component of such a record.

Example

```
gap> info:= OneAtlasGeneratingSetInfo( "A5" );
rec( charactername := "1a+4a", constituents := [ 1, 4 ],
  contents := "core", groupname := "A5", id := "",
  identifier := [ "A5", [ "A5G1-p5B0.m1", "A5G1-p5B0.m2" ], 1, 5 ],
  isPrimitive := true, maxnr := 1, p := 5, rankAction := 2,
  repname := "A5G1-p5B0", repnr := 1, size := 60, stabilizer := "A4",
  standardization := 1, transitivity := 3, type := "perm" )
```

```
gap> AtlasGroup( info );
Group([ (1,2)(3,4), (1,3,5) ])
gap> AtlasGroup( info.identifier );
Group([ (1,2)(3,4), (1,3,5) ])
```

In the groups returned by AtlasGroup, the value of the attribute AtlasRepInfoRecord (3.5.10) is set. This information is used for example by AtlasSubgroup (3.5.9) when this function is called with second argument a group created by AtlasGroup.

3.5.9 AtlasSubgroup

▷ AtlasSubgroup(*gapname[, std][, ...], maxnr*) (function)
▷ AtlasSubgroup(*identifier, maxnr*) (function)
▷ AtlasSubgroup(*G, maxnr*) (function)

Returns: a group that satisfies the conditions, or fail.

The arguments of AtlasSubgroup, except the last argument *maxnr*, are the same as for AtlasGroup (3.5.8). If the database provides a straight line program for restricting representations of the group with name *gapname* (given w. r. t. the *std*-th standard generators) to the *maxnr*-th maximal subgroup and if a representation with the required properties is available, in the sense that calling AtlasGroup (3.5.8) with the same arguments except *maxnr* yields a group, then AtlasSubgroup returns the restriction of this representation to the *maxnr*-th maximal subgroup.

In all other cases, fail is returned.

Note that the conditions refer to the group and not to the subgroup. It may happen that in the restriction of a permutation representation to a subgroup, fewer points are moved, or that the restriction of a matrix representation turns out to be defined over a smaller ring. Here is an example.

Example

```
gap> g:= AtlasSubgroup( "A5", NrMovedPoints, 5, 1 );
Group([ (1,5)(2,3), (1,3,5) ])
gap> NrMovedPoints( g );
4
```

Alternatively, it is possible to enter exactly two arguments, the first being a record *identifier* as returned by OneAtlasGeneratingSetInfo (3.5.6) or AllAtlasGeneratingSetInfos (3.5.7), or the identifier component of such a record, or a group *G* constructed with AtlasGroup (3.5.8).

Example

```
gap> info:= OneAtlasGeneratingSetInfo( "A5" );
rec( charactername := "1a+4a", constituents := [ 1, 4 ],
  contents := "core", groupname := "A5", id := "",
  identifier := [ "A5", [ "A5G1-p5B0.m1", "A5G1-p5B0.m2" ], 1, 5 ],
  isPrimitive := true, maxnr := 1, p := 5, rankAction := 2,
  repname := "A5G1-p5B0", repnr := 1, size := 60, stabilizer := "A4",
  standardization := 1, transitivity := 3, type := "perm" )
gap> AtlasSubgroup( info, 1 );
Group([ (1,5)(2,3), (1,3,5) ])
gap> AtlasSubgroup( info.identifier, 1 );
Group([ (1,5)(2,3), (1,3,5) ])
gap> AtlasSubgroup( AtlasGroup( "A5" ), 1 );
Group([ (1,5)(2,3), (1,3,5) ])
```

3.5.10 AtlasRepInfoRecord (for a group)

▷ AtlasRepInfoRecord(*G*) (attribute)
▷ AtlasRepInfoRecord(*name*) (attribute)

Returns: the record stored in the group *G* when this was constructed with AtlasGroup (3.5.8), or a record with information about the group with name *name*.

For a group *G* that has been constructed with AtlasGroup (3.5.8), the value of this attribute is the info record that describes *G*, in the sense that this record was the first argument of the call to AtlasGroup (3.5.8), or it is the result of the call to OneAtlasGeneratingSetInfo (3.5.6) with the conditions that were listed in the call to AtlasGroup (3.5.8).

Example

```
gap> AtlasRepInfoRecord( AtlasGroup( "A5" ) );
rec( charactername := "1a+4a", constituents := [ 1, 4 ],
  contents := "core", groupname := "A5", id := "",
  identifier := [ "A5", [ "A5G1-p5B0.m1", "A5G1-p5B0.m2" ], 1, 5 ],
  isPrimitive := true, maxnr := 1, p := 5, rankAction := 2,
  repname := "A5G1-p5B0", repnr := 1, size := 60, stabilizer := "A4",
  standardization := 1, transitivity := 3, type := "perm" )
```

For a string *name* that is a GAP name of a group *G*, say, AtlasRepInfoRecord returns a record that contains information about *G* which is used by DisplayAtlasInfo (3.5.1). The following components may be bound in the record.

name
: the string *name*,

nrMaxes
: the number of conjugacy classes of maximal subgroups of *G*,

size
: the order of *G*,

sizesMaxes
: a list which contains at position *i*, if bound, the order of a subgroup in the *i*-th class of maximal subgroups of *G*,

slpMaxes
: a list of length two; the first entry is a list of positions *i* such that a straight line program for computing the restriction of representations of *G* to a subgroup in the *i*-th class of maximal subgroups is available via AtlasRep; the second entry is the corresponding list of standardizations of the generators of *G* for which these straight line programs are available,

structureMaxes
: a list which contains at position *i*, if bound, a string that describes the structure of the subgroups in the *i*-th class of maximal subgroups of *G*.

Example

```
gap> AtlasRepInfoRecord( "A5" );
rec( name := "A5", nrMaxes := 3, size := 60,
  sizesMaxes := [ 12, 10, 6 ],
  slpMaxes := [ [ 1 .. 3 ], [ [ 1 ], [ 1 ], [ 1 ] ] ],
```

```
    structureMaxes := [ "A4", "D10", "S3" ] )
gap> AtlasRepInfoRecord( "J5" );
rec(  )
```

3.5.11 EvaluatePresentation

▷ EvaluatePresentation(*G, gapname[, std]*) (operation)
▷ EvaluatePresentation(*gens, gapname[, std]*) (operation)

Returns: a list of group elements or `fail`.

The first argument must be either a group *G* or a list *gens* of group generators, and *gapname* must be a string that is a GAP name (see Section 3.2) of a group H, say. The optional argument *std*, if given, must be a positive integer that denotes a standardization of generators of H, the default is 1.

`EvaluatePresentation` returns `fail` if no presentation for H w. r. t. the standardization *std* is stored in the database, and otherwise returns the list of results of evaluating the relators of a presentation for H at *gens* or the `GeneratorsOfGroup` (**Reference: GeneratorsOfGroup**) value of *G*, respectively. (An error is signalled if the number of generators is not equal to the number of inputs of the presentation.)

The result can be used as follows. Let N be the normal closure of the the result in *G*. The factor group G/N is an epimorphic image of H. In particular, if all entries of the result have order 1 then *G* itself is an epimorphic image of H. Moreover, an epimorphism is given by mapping the *std*-th standard generators of H to the N-cosets of the given generators of *G*.

Example

```
gap> g:= MathieuGroup( 12 );;
gap> gens:= GeneratorsOfGroup( g );;  # switch to 2 generators
gap> g:= Group( gens[1] * gens[3], gens[2] * gens[3] );;
gap> EvaluatePresentation( g, "J0" );  # no pres. for group "J0"
fail
gap> relimgs:= EvaluatePresentation( g, "M11" );;
gap> List( relimgs, Order );  # wrong group
[ 3, 1, 5, 4, 10 ]
gap> relimgs:= EvaluatePresentation( g, "M12" );;
gap> List( relimgs, Order );  # generators are not standard
[ 3, 4, 5, 4, 4 ]
gap> g:= AtlasGroup( "M12" );;
gap> relimgs:= EvaluatePresentation( g, "M12", 1 );;
gap> List( relimgs, Order );  # right group, std. generators
[ 1, 1, 1, 1, 1 ]
gap> g:= AtlasGroup( "2.M12" );;
gap> relimgs:= EvaluatePresentation( g, "M12", 1 );;
gap> List( relimgs, Order );  # std. generators for extension
[ 1, 2, 1, 1, 2 ]
gap> Size( NormalClosure( g, SubgroupNC( g, relimgs ) ) );
2
```

3.5.12 StandardGeneratorsData

▷ StandardGeneratorsData(*G, gapname[, std]*) (operation)
▷ StandardGeneratorsData(*gens, gapname[, std]*) (operation)

Returns: a record that describes standard generators of the group in question, or `fail`, or the string `"timeout"`.

The first argument must be either a group *G* or a list *gens* of group generators, and *gapname* must be a string that is a GAP name (see Section 3.2) of a group H, say. The optional argument *std*, if given, must be a positive integer that denotes a standardization of generators of H, the default is 1.

If the global option `projective` is given then the group elements must be matrices over a finite field, and the group must be a central extension of the group H by a normal subgroup that consists of scalar matrices. In this case, all computations will be carried out modulo scalar matrices (in particular, element orders will be computed using `ProjectiveOrder` (**Reference: ProjectiveOrder**)), and the returned standard generators will belong to H.

`StandardGeneratorsData` returns

`fail`
: if no black box program for computing standard generators of H w. r. t. the standardization *std* is stored in the database, or if the black box program returns `fail` because a runtime error occurred or the program has proved that the given group or generators cannot generate a group isomorphic to H,

`"timeout"`
: if the black box program returns `"timeout"`, typically because some elements of a given order were not found among a reasonable number of random elements, or

a record containing standard generators
: otherwise.

When the result is not a record then either the group is not isomorphic to H (modulo scalars if applicable), or we were unlucky with choosing random elements.

When a record is returned *and* *G* or the group generated by *gens*, respectively, is isomorphic to H (or to a central extension of H by a group of scalar matrices if the global option `projective` is given) then the result describes the desired standard generators.

If *G* or the group generated by *gens*, respectively, is *not* isomorphic to H then it may still happen that `StandardGeneratorsData` returns a record. For a proof that the returned record describes the desired standard generators, one can use a presentation of H whose generators correspond to the *std*-th standard generators, see `EvaluatePresentation` (3.5.11).

A returned record has the following components.

`gapname`
: the string *gapname*,

`givengens`
: the list of group generators from which standard generators were computed, either *gens* or the `GeneratorsOfGroup` (**Reference: GeneratorsOfGroup**) value of *G*,

`stdgens`
: a list of standard generators of the group,

`givengenstostdgens`
: a straight line program that takes `givengens` as inputs, and returns `stdgens`,

`std` the underlying standardization *std*.

The first examples show three cases of failure, due to the unavailability of a suitable black box program or to a wrong choice of *gapname*. (In the search for standard generators of M_{11} in the group M_{12}, one may or may not find an element whose order does not appear in M_{11}; in the first case, the result is `fail`, whereas a record is returned in the second case. Both cases occur.)

Example

```
gap> StandardGeneratorsData( MathieuGroup( 11 ), "J0" );
fail
gap> StandardGeneratorsData( MathieuGroup( 11 ), "M12" );
"timeout"
gap> repeat
>      res:= StandardGeneratorsData( MathieuGroup( 12 ), "M11" );
>    until res = fail;
```

The next example shows a computation of standard generators for the Mathieu group M_{12}. Using a presentation of M_{12} w. r. t. these standard generators, we prove that the given group is isomorphic to M_{12}.

Example

```
gap> gens:= GeneratorsOfGroup( MathieuGroup( 12 ) );;
gap> std:= 1;;
gap> res:= StandardGeneratorsData( gens, "M12", std );;
gap> Set( RecNames( res ) );
[ "gapname", "givengens", "givengenstostdgens", "std", "stdgens" ]
gap> gens = res.givengens;
true
gap> ResultOfStraightLineProgram( res.givengenstostdgens, gens )
>    = res.stdgens;
true
gap> evl:= EvaluatePresentation( res.stdgens, "M12", std );;
gap> ForAll( evl, IsOne );
true
```

The next example shows the use of the global option `projective`. We take an irreducible matrix representation of the double cover of the Mathieu group M_{12} (thus the center is represented by scalar matrices) and compute standard generators of the factor group M_{12}. Using a presentation of M_{12} w. r. t. these standard generators, we prove that the given group is modulo scalars isomorphic to M_{12}, and we get generators for the kernel.

Example

```
gap> g:= AtlasGroup( "2.M12", IsMatrixGroup, Characteristic, IsPosInt );;
gap> gens:= Permuted( GeneratorsOfGroup( g ), (1,2) );;
gap> res:= StandardGeneratorsData( gens, "M12", std : projective );;
gap> gens = res.givengens;
true
gap> ResultOfStraightLineProgram( res.givengenstostdgens, gens )
>    = res.stdgens;
true
gap> evl:= EvaluatePresentation( res.stdgens, "M12", std );;
gap> ForAll( evl, IsOne );
false
gap> ForAll( evl, x -> IsCentral( g, x ) );
true
```

3.6 Browse Applications Provided by AtlasRep

The functions BrowseMinimalDegrees (3.6.1), BrowseBibliographySporadicSimple (3.6.2), and BrowseAtlasInfo (BrowseAtlasInfo???) (an alternative to DisplayAtlasInfo (3.5.1)) are available only if the GAP package Browse (see [BL18]) is loaded.

3.6.1 BrowseMinimalDegrees

▷ BrowseMinimalDegrees(*[gapnames]*) (function)

Returns: the list of info records for the clicked representations.

If the GAP package Browse (see [BL18]) is loaded then this function is available. It opens a browse table whose rows correspond to the groups for which AtlasRep knows some information about minimal degrees, whose columns correspond to the characteristics that occur, and whose entries are the known minimal degrees.

Example

```
gap> if IsBound( BrowseMinimalDegrees ) then
>   down:= NCurses.keys.DOWN;;  DOWN:= NCurses.keys.NPAGE;;
>   right:= NCurses.keys.RIGHT;;  END:= NCurses.keys.END;;
>   enter:= NCurses.keys.ENTER;;  nop:= [ 14, 14, 14 ];;
>   # just scroll in the table
>   BrowseData.SetReplay( Concatenation( [ DOWN, DOWN, DOWN,
>          right, right, right ], "sedddrrrddd", nop, nop, "Q" ) );
>   BrowseMinimalDegrees();;
>   # restrict the table to the groups with minimal ordinary degree 6
>   BrowseData.SetReplay( Concatenation( "scf6",
>        [ down, down, right, enter, enter ] , nop, nop, "Q" ) );
>   BrowseMinimalDegrees();;
>   BrowseData.SetReplay( false );
> fi;
```

If an argument *gapnames* is given then it must be a list of GAP names of groups. The browse table is then restricted to the rows corresponding to these group names and to the columns that are relevant for these groups. A perhaps interesting example is the subtable with the data concerning sporadic simple groups and their covering groups, which has been published in [Jan05]. This table can be shown as follows.

Example

```
gap> if IsBound( BrowseMinimalDegrees ) then
>   # just scroll in the table
>   BrowseData.SetReplay( Concatenation( [ DOWN, DOWN, DOWN, END ],
>          "rrrrrrrrrrrrr", nop, nop, "Q" ) );
>   BrowseMinimalDegrees( BibliographySporadicSimple.groupNamesJan05 );;
> fi;
```

The browse table does *not* contain rows for the groups $6.M_{22}$, $12.M_{22}$, $6.Fi_{22}$. Note that in spite of the title of [Jan05], the entries in Table 1 of this paper are in fact the minimal degrees of faithful *irreducible* representations, and in the above three cases, these degrees are larger than the minimal degrees of faithful representations. The underlying data of the browse table is about the minimal faithful (but not necessarily irreducible) degrees.

The return value of BrowseMinimalDegrees is the list of OneAtlasGeneratingSetInfo (3.5.6) values for those representations that have been "clicked" in visual mode.

The variant without arguments of this function is also available in the menu shown by BrowseGapData (BrowseGapData???).

3.6.2 BrowseBibliographySporadicSimple

▷ BrowseBibliographySporadicSimple() (function)

Returns: a record as returned by ParseBibXMLExtString (**GAPDoc: ParseBibXMLextString**).

If the GAP package Browse (see [BL18]) is loaded then this function is available. It opens a browse table whose rows correspond to the entries of the bibliographies in the ATLAS of Finite Groups [CCN+85] and in the ATLAS of Brauer Characters [JLPW95].

The function is based on BrowseBibliography (BrowseBibliography???), see the documentation of this function for details, e.g., about the return value.

The returned record encodes the bibliography entries corresponding to those rows of the table that are "clicked" in visual mode, in the same format as the return value of ParseBibXMLExtString (**GAPDoc: ParseBibXMLextString**), see the manual of the GAP package GAPDoc [LN18] for details.

BrowseBibliographySporadicSimple can be called also via the menu shown by BrowseGapData (BrowseGapData???).

Example

```
gap> if IsBound( BrowseBibliographySporadicSimple ) then
>   enter:= NCurses.keys.ENTER;;  nop:= [ 14, 14, 14 ];;
>   BrowseData.SetReplay( Concatenation(
>     # choose the application
>     "/Bibliography of Sporadic Simple Groups", [ enter, enter ],
>     # search in the title column for the Atlas of Finite Groups
>     "scr/Atlas of finite groups", [ enter,
>     # and quit
>     nop, nop, nop, nop ], "Q" ) );
>   BrowseGapData();;
>   BrowseData.SetReplay( false );
> fi;
```

The bibliographies contained in the ATLAS of Finite Groups [CCN+85] and in the ATLAS of Brauer Characters [JLPW95] are available online in HTML format, see http://www.math.rwth-aachen.de/~Thomas.Breuer/atlasrep/bibl/index.html.

The source data in BibXMLext format, which are used by BrowseBibliographySporadicSimple, are distributed with the AtlasRep package, in four files with suffix xml in the package's bibl directory. Note that each of the two books contains two bibliographies.

Details about the BibXMLext format, including information how to transform the data into other formats such as BibTeX, can be found in the GAP package GAPDoc (see [LN18]).

Chapter 4

Customizations of the AtlasRep Package

4.1 Installing the AtlasRep Package

To install the package, unpack the archive file in a directory in the `pkg` directory of your local copy of GAP 4. This might be the `pkg` directory of the GAP 4 root directory, see (**Reference: Installing a GAP Package**) for details. It is however also possible to keep an additional `pkg` directory somewhere else, see Section (**Reference: GAP Root Directories**). The latter possibility *must* be chosen if you do not have write access to the GAP root directory.

If it is likely that you will work offline, it makes sense to install the "starter archive" that can be downloaded from the package's homepage.

The package consists entirely of GAP code, no external binaries need to be compiled for the package itself.

After unpacking the package archive, the write permissions for those directories should be checked into which users will download files. Every user can customize these paths via a user preference, see Section 4.2.2, the defaults are the subdirectories `data*` of the package directory. The recommended permissions under UNIX for the default directories are set as follows.

Example

```
you@unix> chmod 1777 atlasrep/data*
you@unix> ls -ld atlasrep/data*
drwxrwxrwt   3 you      you          1024 Apr 12 12:34 dataext
drwxrwxrwt   3 you      you          1024 Apr 12 12:34 datagens
drwxrwxrwt   3 you      you          1024 Apr 12 12:34 datapkg
drwxrwxrwt   3 you      you          1024 Apr 12 12:34 dataword
```

For checking the installation of the package, you should start GAP and call

Example

```
gap> ReadPackage( "atlasrep", "tst/testinst.g" );
```

If the installation is o.k. then the GAP prompt appears without anything else being printed; otherwise the output lines tell you what should be changed.

PDF, HTML, and text versions of the package manual are available in the `doc` directory of the package.

4.2 User Preferences of the AtlasRep Package

This section describes global parameters for which it might make sense to change their defaults, using GAP's user preferences (see (**Reference: Configuring User preferences**)).

- Is access to remote data allowed (see Section 4.2.1)? If yes then also the following parameters are of interest.
  - From where can the data be fetched (see Section 4.2.3)?
  - Where are local copies of these data stored (see Section 4.2.2)?
  - Shall files be compressed after they have been downloaded (see Section 4.2.4)?
- The following parameters influence reading and writing of local files.
  - What shall actually happen when data are requested by the interface functions (see Section 4.2.5)?
  - If the value of the user preference `FileAccessFunctions` contains `"direct access to a local server"`, what is its path (see Section 4.2.6)?
  - Shall `ScanMeatAxeFile` (7.3.1) focus on small runtime or on small space when reading MeatAxe text files (see Section 4.2.7)?
  - Which kind of headers shall `MeatAxeString` (7.3.2) create (see Section 4.2.8)?
  - Shall `MeatAxeString` (7.3.2) interpret permutation matrices more as permutations (mode 2) or as matrices (mode 1 or 6) (see Section 4.2.9)?
  - Shall the default for `CMtxBinaryFFMatOrPerm` (7.3.4) be to write binary files of zero-based or one-based permutations (see Section 4.2.10)?
- Which function is used by `DisplayAtlasInfo` (3.5.1) for printing to the screen (see Section 4.2.11)?
- How does `DisplayAtlasInfo` (3.5.1) mark data that do not belong to the core database (see Section 4.2.12)?
- Shall debug messages be printed when local data files are read (see Section 4.2.13)?

4.2.1 User preference `AtlasRepAccessRemoteFiles`

The value `true` (the default) allows the AtlasRep package to fetch data files that are not yet locally available. If the value is `false` then only those data files can be used that are available locally.

If you are working offline then you should set the value to `false`.

Changing the value in a running GAP session does not affect the information shown by `DisplayAtlasInfo` (3.5.1), this information depends on the value of the preference at the time when the AtlasRep package and its data extensions get loaded.

4.2.2 User preference AtlasRepDataDirectory

The value must be a string that is either empty or the filename of a directory (in the sense of IsDirectoryPath (**Reference: IsDirectoryPath**)) that contains the directories in which downloaded data will be stored.

An empty string means that downloaded data are just kept in the GAP session but not saved to local files.

The default depends on the user's permissions for the subdirectories dataext, datagens, dataword of the AtlasRep directory: If these directories are writable for the user then the installation path of the AtlasRep package (including a trailing slash symbol) is taken, otherwise the default is an empty string.

4.2.3 User preference AtlasRepTOCData

The value must be a list of strings of the form "ID|address" where ID is the id of a part of the database and address is an URL or a file path (as an absolute path or relative to the user's home directory, cf. Directory (**Reference: Directory**)) of a readable JSON format file that contain the table of contents of this part, see StringOfAtlasTableOfContents (5.1.3).

The default lists four entries: the core database, the data distributed with the AtlasRep package, and the data that belong to the packages MFER and CTBlocks.

4.2.4 User preference CompressDownloadedMeatAxeFiles

When used with UNIX, GAP can read gzipped files, see (**Reference: Saving and Loading a Workspace**). If the package's user preference CompressDownloadedMeatAxeFiles has the value true then each MeatAxe format text file that is downloaded from the internet is afterwards compressed with gzip. The default value is false.

Compressing files saves a lot of space if many MeatAxe format files are accessed. (Note that data files in other formats are very small.) For example, at the time of the release of version 2.0 the core database contained about 8 400 data files in MeatAxe format, which needed about 1 400 MB in uncompressed text format and about 275 MB in compressed text format.

4.2.5 User preference FileAccessFunctions

This preference allows one to customize what actually happens when data are requested by the interface functions: Is it necessary to download some files? If yes then which files are downloaded? If no then which files are actually read into GAP?

Currently one can choose among the following features.

1. Download/read MeatAxe text files.
2. Prefer downloading/reading MeatAxe binary files.
3. Prefer reading locally available data files.

(Of course files can be downloaded only if the user preference AtlasRepAccessRemoteFiles has the value true, see Section 4.2.1.)

This feature could be used more generally, see Section 7.2 for technical details and the possibility to add other features.

4.2.6 User preference AtlasRepLocalServerPath

If the data of the core database are available locally (for example because one has access to a local mirror of the data) then one may prefer reading these files instead of downloading data. In order to achieve this, one can set the user preference AtlasRepLocalServerPath and add "direct access to a local server" to the user preference FileAccessFunctions, see Section 4.2.5.

The value must be a string that is the filename of a directory (in the sense of IsDirectoryPath (**Reference: IsDirectoryPath**)) that contains the data of the ATLAS of Group Representations, in the same directory tree structure as on the ATLAS server.

4.2.7 User preference HowToReadMeatAxeTextFiles

The value "fast" means that ScanMeatAxeFile (7.3.1) reads text files via StringFile (**GAPDoc: StringFile**). Otherwise each file containing a matrix over a finite field is read line by line via ReadLine (**Reference: ReadLine**), and the GAP matrix is constructed line by line, in a compressed representation (see (**Reference: Row Vectors over Finite Fields**) and (**Reference: Matrices over Finite Fields**)); this makes it possible to read large matrices in a reasonable amount of space.

The StringFile (**GAPDoc: StringFile**) approach is faster but needs more intermediate space when text files containing matrices over finite fields are read. For example, a 4370 by 4370 matrix over the field with two elements (as occurs for an irreducible representation of the Baby Monster) requires less than 3 MB space in GAP but the corresponding MeatAxe format text file is more than 19 MB large. This means that when one reads the file with the fast variant, GAP will temporarily grow by more than this value.

Note that this parameter has an effect only when ScanMeatAxeFile (7.3.1) is used. It has no effect for example if MeatAxe binary files are read, cf. FFMatOrPermCMtxBinary (7.3.5).

4.2.8 User preference WriteHeaderFormatOfMeatAxeFiles

This user preference determines the format of the header lines of MeatAxe format strings created by MeatAxeString (7.3.2), see the C-MeatAxe manual [Rin] for details. The following values are supported.

"numeric"
: means that the header line of the returned string consists of four integers (in the case of a matrix these are mode, row number, column number and field size),

"numeric (fixed)"
: means that the header line of the returned string consists of four integers as in the case "numeric", but additionally each integer is right aligned in a substring of length (at least) six,

"textual"
: means that the header line of the returned string consists of assignments such as matrix field=2.

4.2.9 User preference WriteMeatAxeFilesOfMode2

The value true means that the function MeatAxeString (7.3.2) will encode permutation matrices via mode 2 descriptions, that is, the first entry in the header line is 2, and the following lines contain

the positions of the nonzero entries. If the value is `false` (the default) then `MeatAxeString` (7.3.2) encodes permutation matrices via mode 1 or mode 6 descriptions, that is, the lines contain the matrix entries.

4.2.10 User preference `BaseOfMeatAxePermutation`

The value 0 means that the function `CMtxBinaryFFMatOrPerm` (7.3.4) writes zero-based permutations, that is, permutations acting on the points from 0 to the degree minus one; this is achieved by shifting down all images of the GAP permutation by one. The value 1 (the default) means that the permutation stored in the binary file acts on the points from 1 to the degree.

Up to version 2.3 of the `C`-MeatAxe, permutations in binary files were always one-based. Zero-based permutations were introduced in version 2.4.

4.2.11 User preference `DisplayFunction`

The way how `DisplayAtlasInfo` (3.5.1) shows the requested overview is controlled by the package AtlasRep's user preference `DisplayFunction`. The value must be a string that evaluates to a GAP function. The default value is `"Print"` (see `Print` (**Reference: Print**)), other useful values are `"PrintFormattedString"` (see `PrintFormattedString` (**GAPDoc: PrintFormattedString**)) and `"AGR.Pager"`; the latter means that `Pager` (**Reference: Pager**) is called with the `formatted` option, which is necessary for switching off GAP's automatic line breaking.

4.2.12 User preference `AtlasRepMarkNonCoreData`

The value is a string (the default is a star '*') that is used in `DisplayAtlasInfo` (3.5.1) to mark data that do not belong to the core database, see Section 5.2.

4.2.13 User preference `DebugFileLoading`

If the value is `true` then debug messages are printed before and after data files get loaded. The default value is `false`.

4.2.14 User preference `AtlasRepJsonFilesAddresses`

The value, if set, must be a list of length two, the first entry being an URL describing a directory that contains Json format files of the available matrix representations in characteristic zero, and the second being a directory path where these files shall be stored locally. If the value is set (this is the default) then the functions of the package use the Json format files instead of the GAP format files.

4.3 Web Contents for the AtlasRep Package

The home page of the AtlasRep package provides

- package archives,
- introductory package information,
- the current *table of contents* of core data in the file `atlasprm.json` of the package, cf. `StringOfAtlasTableOfContents` (5.1.3),

- the list of changes of remote core data files,
- a starter archive containing many small representations and programs, and
- an overview of the core data in a similar format as the information shown by the function `DisplayAtlasInfo` (3.5.1) of the package; more details can be found on the home page of the ATLAS of Group Representations.

4.4 Extending the ATLAS Database

Users who have computed new representations that might be interesting for inclusion into the ATLAS of Group representations can send the data in question to `R.A.Wilson@qmul.ac.uk`.

It is also possible to make additional representations and programs accessible for the GAP interface, and to use these "private" data in the same way as the core data. See Chapter 5 for details.

Chapter 5

Extensions of the AtlasRep Package

It may be interesting to use the functions of the GAP interface also for representations or programs that are *not* part of the ATLAS of Group Representations. This chapter describes how to achieve this.

The main idea is that users can notify collections of "private" data files, which may consist of

1. new faithful representations and programs for groups that are declared already in the core part of the database that belongs to the "official" ATLAS of Group Representations (see Section 5.1),
2. the declaration of groups that are not declared in the ATLAS of Group Representations, and representations and programs for them (see Section 5.2), and
3. the definition of new kinds of representations and programs (see Section 7.5).

A test example of a local extension is given in Section 5.3. Another such example is the small collection of data that is distributed together with the package, in its `datapkg` directory; its contents can be listed by calling `DisplayAtlasInfo( "contents", "internal" )`.

Examples of extensions by files that can be downloaded from the internet can be found in the GAP packages MFER [BHM09] and CTBlocks [Bre14]. These extensions are automatically notified as soon as AtlasRep is available, via the default value of the user preference `AtlasRepTOCData`, see Section 4.2.3; their contents can be listed by calling `DisplayAtlasInfo( "contents", "mfer" )` and `DisplayAtlasInfo( "contents", "ctblocks" )`, respectively.

Several of the sanity checks for the core part of the AtlasRep data make sense also for data extensions, see Section 7.9 for more information.

5.1 Notify Additional Data

After the AtlasRep package has been loaded into the GAP session, one can extend the data which the interface can access by own representations and programs. The following two variants are supported.

- The additional data files are locally available in some directory. Information about the declaration of new groups or about additional information such as the character names of representations can be provided in an optional JSON format file named `toc.json` in this directory.
- The data files can be downloaded from the internet. Both the list of available data and additional information as in the above case are given by either a local JSON format file or the URL of a JSON format file. This variant requires the user preference `AtlasRepAccessRemoteFiles` (see Section 4.2.1) to have the value `true`.

In both cases, AtlasOfGroupRepresentationsNotifyData (5.1.1) can be used to make the private data available to the interface.

5.1.1 AtlasOfGroupRepresentationsNotifyData

▷ AtlasOfGroupRepresentationsNotifyData(*dir, id[, test]*) (function)
▷ AtlasOfGroupRepresentationsNotifyData(*filename[, id][, test]*) (function)
▷ AtlasOfGroupRepresentationsNotifyData(*url[, id][, test]*) (function)

Returns: true if the overview of the additional data can be evaluated and if the names of the data files in the extension are compatible with the data files that had been available before the call, otherwise false.

The following variants are supported for notifying additional data.

Contents of a local directory

The first argument *dir* must be either a local directory (see (**Reference: Directories**)) or a string denoting the path of a local directory, such that the GAP object describing this directory can be obtained by calling Directory (**Reference: Directory**) with the argument *dir*; in the latter case, *dir* can be an absolute path or a path relative to the user's home directory (starting with a tilde character ~) or a path relative to the directory where GAP was started. The files contained in this directory or in its subdirectories (only one level deep) are considered. If the directory contains a JSON document in a file with the name toc.json then this file gets evaluated; its purpose is to provide additional information about the data files.

Calling AtlasOfGroupRepresentationsNotifyData means to evaluate the contents of the directory and (if available) of the file toc.json.

Accessing data means to read the locally available data files.

The argument *id* must be a string. It will be used in the identifier components of the records that are returned by interface functions (see Section 3.5) for data contained in the directory *dir*. (Note that the directory name may be different in different GAP sessions or for different users who want to access the same data, whereas the identifier components shall be independent of such differences.)

An example of a local extension is the contents of the datapkg directory of the AtlasRep package. This extension gets notified automatically when AtlasRep gets loaded. For restricting data collections to this extension, one can use the identifier "internal".

Local file describing the contents of a local or remote directory

The first argument *filename* must be the name of a local file whose content is a JSON document that lists the available data, additional information about these data, and an URL from where the data can be downloaded. The data format of this file is defined by the JSON schema file doc/atlasreptoc_schema.json of the AtlasRep package.

Calling AtlasOfGroupRepresentationsNotifyData means to evaluate the contents of the file *filename*, without trying to access the remote data. The *id* is then either given implicitly by the ID component of the JSON document or can be given as the second argument.

Downloaded data files are stored in the subdirectory dataext/*id* of the directory that is given by the user preference AtlasRepDataDirectory, see Section 4.2.2.

Accessing data means to download remote files if necessary but to prefer files that are already locally available.

An example of such an extension is the set of permutation representations provided by the MFER package [BHM09]; due to the file sizes, these representations are *not* distributed together with the MFER package. For restricting data collections to this extension, one can use the identifier `"mfer"`.

Another example is given by some of the data that belong to the CTBlocks package [Bre14]. These data are also distributed with that package, and notifying the extension in the situation that the CTBlocks package is available will make its local data available, via the component `LocalDirectory` of the JSON document `ctblocks.json`; notifying the extension in the situation that the CTBlocks package is *not* available will make the remote files available, via the component `DataURL` of this JSON document. For restricting data collections to this extension, one can use the identifier `"ctblocks"`.

URL of a file

(This variant works only if the IO package [Neu14] is available.)

The first argument *url* must be the URL of a JSON document as in the previous case.

Calling `AtlasOfGroupRepresentationsNotifyData` in *online mode* (that is, the user preference `AtlasRepAccessRemoteFiles` has the value `true`) means to download this file and to evaluate it; the *id* is then given implicitly by the `ID` component of the JSON document, and the contents of the document gets stored in a file with name `dataext/`*id*`/toc.json`, relative to the directory given by the value of the user preference `AtlasRepDataDirectory`. Also downloaded files for this extension will be stored in the directory `dataext/`*id*.

Calling `AtlasOfGroupRepresentationsNotifyData` in *offline mode* requires that the argument *id* is explicitly given. In this case, it is checked whether the `dataext` subdirectory contains a subdirectory with name *id*; if not then `false` is returned, if yes then the contents of this local directory gets notified via the first form described above.

Accessing data in online mode means the same as in the case of a remote directory. Accessing data in offline mode means the same as in the case of a local directory.

Examples of such extension are again the data from the packages CTBlocks and MFER described above, but in the situation that these packages are *not* loaded, and that just the web URLs of their JSON documents are entered which describe the contents.

In all three cases, if the optional argument *test* is given then it must be either `true` or `false`. In the `true` case, consistency checks are switched on during the notification. The default for *test* is `false`.

The notification of an extension may happen as a side-effect when a GAP package gets loaded that provides the data in question. Besides that, one may collect the notifications of data extensions in one's `gaprc` file (see Section (**Reference: The gap.ini and gaprc files**)).

5.1.2 AtlasOfGroupRepresentationsForgetData

▷ `AtlasOfGroupRepresentationsForgetData(`*dirid*`)` (function)

If *dirid* is the identifier of a database extension that has been notified with `AtlasOfGroupRepresentationsNotifyData` (5.1.1) then `AtlasOfGroupRepresentationsForgetData` undoes the notification; this means that from then on, the data of this extension cannot be accessed anymore in the current session.

5.1.3 StringOfAtlasTableOfContents

▷ StringOfAtlasTableOfContents(*inforec*) (function)

For a record *inforec* with at least the component ID, with value "core" or the identifier of a data extension (see AtlasOfGroupRepresentationsNotifyData (5.1.1)), this function returns a string that describes the part of AtlasRep data belonging to *inforec*.ID.

Printed to a file, the returned string can be used as the table of contents of this part of the data. For that purpose, also the following components of *inforec* must be bound (all strings). Version, SelfURL (the internet address of the table of contents file itself). At least one of the following two components must be bound. DataURL is the internet address of the directory from where the data in question can be downloaded. LocalDirectory is a path relative to GAP's pkg directory where the data may be stored locally (depending on whether some GAP package is installed). If the component DataURL is bound then the returned string contains the information about the data files; this is not necessary if the data are *only* locally available. If both DataURL and LocalDirectory are bound then locally available data will be prefered at runtime.

Alternatively, *inforec* can also be the ID string; in this case, the values of those of the supported components mentioned above that are defined in an available JSON file for this ID are automatically inserted. (If there is no such file yet then entering the ID string as *inforec* does not make sense.)

For an example how to use the function, see Section 5.3.

5.2 The Effect of Extensions on the User Interface

First suppose that only new groups or new data for known groups or for new groups are added.

In this case, DisplayAtlasInfo (3.5.1) lists the additional representations and programs in the same way as other data known to AtlasRep, except that parts outside the core database are marked with the string that is the value of the user preference AtlasRepMarkNonCoreData, see Section 4.2.12. The ordering of representations listed by DisplayAtlasInfo (3.5.1) (and referred to by AtlasGenerators (3.5.3)) will in general change whenever extensions get notified. For the other interface functions described in Chapter 3, the only difference is that also the additional data can be accessed.

If also new data types are introduced in an extension (see Section 7.5) then additional columns or rows can appear in the output of DisplayAtlasInfo (3.5.1), and new inputs can become meaningful for all interface functions.

5.3 An Example of Extending the AtlasRep Data

This section shows an extension by a few *locally available* files.

We set the info level of InfoAtlasRep (7.1.1) to 1 in this section.

Example

```
gap> locallevel:= InfoLevel( InfoAtlasRep );;
gap> SetInfoLevel( InfoAtlasRep, 1 );
```

Let us assume that the local directory privdir contains data for the cyclic group C_4 of order 4 and for the alternating group A_5 on 5 points, respectively. Note that it is obvious what the term "standard generators" means for the group C_4.

Further let us assume that privdir contains the following files.

`C4G1-p4B0.m1`
: a faithful permutation representation of C_4 on 4 points,

`C4G1-max1W1`
: the straight line program that returns the square of its unique input,

`C4G1-a2W1`
: the straight line program that raises its unique input to the third power,

`C4G1-XtestW1`
: the straight line program that returns the square of its unique input,

`A5G1-p60B0.m1` **and** `A5G1-p60B0.m2`
: standard generators for A_5 in its regular permutation representation.

The directory and the files can be created as follows.

Example

```
gap> prv:= DirectoryTemporary( "privdir" );;
gap> FileString( Filename( prv, "C4G1-p4B0.m1" ),
>                MeatAxeString( [ (1,2,3,4) ], 4 ) );;
gap> FileString( Filename( prv, "C4G1-max1W1" ),
>                "inp 1\npwr 2 1 2\noup 1 2\n" );;
gap> FileString( Filename( prv, "C4G1-XtestW1" ),
>                "inp 1\npwr 2 1 2\noup 1 2\n" );;
gap> FileString( Filename( prv, "C4G1-a2W1" ),
>                "inp 1\npwr 3 1 2\noup 1 2\n" );;
gap> FileString( Filename( prv, "C4G1-Ar1aB0.g" ),
>                "return rec( generators:= [ [[E(4)]] ] );\n" );;
gap> points:= Elements( AlternatingGroup( 5 ) );;
gap> FileString( Filename( prv, "A5G1-p60B0.m1" ),
>      MeatAxeString( [ Permutation( (1,2)(3,4), points, OnRight ) ], 60 ) );;
gap> FileString( Filename( prv, "A5G1-p60B0.m2" ),
>      MeatAxeString( [ Permutation( (1,3,5), points, OnRight ) ], 60 ) );;
```

(We could also introduce intermediate directories `C4` and `A5`, say, each with the data for one group only.)

The core part of the AtlasRep data does not contain information about C_4, so we first notify this group, in the file `privdir/toc.json`. Besides the name of the group, we store the following information: the group order, the number of (classes of) maximal subgroups, their orders, their structures, and describing data about the three representations. The group A_5 is already known with name `A5` in the core part of the AtlasRep data, so it need not and cannot be notified again.

Example

```
gap> FileString( Filename( prv, "toc.json" ), Concatenation( [ "{\n",
>        "\"ID\":\"priv\",\n",
>        "\"Data\":[\n",
>        "[\"GNAN\",[\"C4\",\"C4\"]],\n",
>        "[\"GRS\",[\"C4\",4]],\n",
>        "[\"MXN\",[\"C4\",1]],\n",
>        "[\"MXO\",[\"C4\",[2]]],\n",
>        "[\"MXS\",[\"C4\",[\"C2\"]]],\n",
>        "[\"RNG\",[\"C4G1-Ar1aB0\",\"CF(4)\",",
```

```
>                   "[\"QuadraticField\",-1],[1,0,1]]],\n",
>         "[\"API\",[\"C4G1-p4B0\",[1,4,\"imprim\",\"1 < C2\"]]],\n",
>         "[\"API\",[\"A5G1-p60B0\",[1,60,\"imprim\",\"1 < S3\"]]]\n",
>         "]\n",
>         "}\n" ] ) );;
```

Then we notify the extension.

Example

```
gap> AtlasOfGroupRepresentationsNotifyData( prv, "priv", true );
true
```

Now we can use the interface functions for accessing the additional data.

Example

```
gap> DisplayAtlasInfo( [ "C4" ] );
group | # | maxes | cl | cyc | out | fnd | chk | prs
------+---+------+----+-----+-----+-----+-----+----
C4*   | 2 |    1 |    |     |   2 |     |     |
gap> DisplayAtlasInfo( "C4" );
Representations for G = C4:    (all refer to std. generators 1)
---------------------------
1: G <= Sym(4)*       rank 4, on cosets of 1 < C2
2: G <= GL(1a,CF(4))*

Programs for G = C4:    (all refer to std. generators 1)
--------------------
- automorphisms*:
  2*
- maxes (all 1):
  1*:  C2
- other scripts*:
  "test"*
gap> DisplayAtlasInfo( "C4", IsPermGroup, true );
Representations for G = C4:    (all refer to std. generators 1)
---------------------------
1: G <= Sym(4)* rank 4, on cosets of 1 < C2
gap> DisplayAtlasInfo( "C4", IsMatrixGroup );
Representations for G = C4:    (all refer to std. generators 1)
---------------------------
2: G <= GL(1a,CF(4))*
gap> DisplayAtlasInfo( "C4", Dimension, 2 );
gap> DisplayAtlasInfo( "A5", NrMovedPoints, 60 );
Representations for G = A5:    (all refer to std. generators 1)
---------------------------
4: G <= Sym(60)* rank 60, on cosets of 1 < S3
gap> info:= OneAtlasGeneratingSetInfo( "C4" );
rec( contents := "priv", groupname := "C4", id := "",
  identifier := [ "C4", [ [ "priv", "C4G1-p4B0.m1" ] ], 1, 4 ],
  isPrimitive := false, p := 4, rankAction := 4,
  repname := "C4G1-p4B0", repnr := 1, size := 4,
  stabilizer := "1 < C2", standardization := 1, transitivity := 1,
  type := "perm" )
```

```
gap> AtlasGenerators( info.identifier );
rec( contents := "priv", generators := [ (1,2,3,4) ],
  groupname := "C4", id := "",
  identifier := [ "C4", [ [ "priv", "C4G1-p4B0.m1" ] ], 1, 4 ],
  isPrimitive := false, p := 4, rankAction := 4,
  repname := "C4G1-p4B0", repnr := 1, size := 4,
  stabilizer := "1 < C2", standardization := 1, transitivity := 1,
  type := "perm" )
gap> AtlasProgram( "C4", 1 );
rec( groupname := "C4",
  identifier := [ "C4", [ [ "priv", "C4G1-max1W1" ] ], 1 ],
  program := <straight line program>, size := 2, standardization := 1,
  subgroupname := "C2", version := "1" )
gap> AtlasProgram( "C4", "maxes", 1 );
rec( groupname := "C4",
  identifier := [ "C4", [ [ "priv", "C4G1-max1W1" ] ], 1 ],
  program := <straight line program>, size := 2, standardization := 1,
  subgroupname := "C2", version := "1" )
gap> AtlasProgram( "C4", "maxes", 2 );
fail
gap> AtlasGenerators( "C4", 1 );
rec( contents := "priv", generators := [ (1,2,3,4) ],
  groupname := "C4", id := "",
  identifier := [ "C4", [ [ "priv", "C4G1-p4B0.m1" ] ], 1, 4 ],
  isPrimitive := false, p := 4, rankAction := 4,
  repname := "C4G1-p4B0", repnr := 1, size := 4,
  stabilizer := "1 < C2", standardization := 1, transitivity := 1,
  type := "perm" )
gap> AtlasGenerators( "C4", 2 );
rec( contents := "priv", dim := 1, generators := [ [ [ E(4) ] ] ],
  groupname := "C4", id := "a",
  identifier := [ "C4", [ [ "priv", "C4G1-Ar1aB0.g" ] ], 1, 1 ],
  polynomial := [ 1, 0, 1 ], repname := "C4G1-Ar1aB0", repnr := 2,
  ring := GaussianRationals, size := 4, standardization := 1,
  type := "matalg" )
gap> AtlasGenerators( "C4", 3 );
fail
gap> AtlasProgram( "C4", "other", "test" );
rec( groupname := "C4",
  identifier := [ "C4", [ [ "priv", "C4G1-XtestW1" ] ], 1 ],
  program := <straight line program>, standardization := 1,
  version := "1" )
```

We can restrict the data shown by `DisplayAtlasInfo` (3.5.1) to our extension, as follows.

Example

```
gap> DisplayAtlasInfo( "contents", "priv" );
group | # | maxes | cl | cyc | out | fnd | chk | prs
------+---+-------+----+-----+-----+-----+-----+----
A5*   | 1 |       |    |     |     |     |     |
C4*   | 2 |     1 |    |     |   2 |     |     |
```

For checking the data in the extension, we apply the relevant sanity checks (see Section 7.9).

Example

```
gap> AGR.Test.Words( "priv" );
true
gap> AGR.Test.FileHeaders( "priv" );
true
gap> AGR.Test.Files( "priv" );
true
gap> AGR.Test.BinaryFormat( "priv" );
true
gap> AGR.Test.Primitivity( "priv" : TryToExtendData );
true
gap> AGR.Test.Characters( "priv" : TryToExtendData );
#I  AGR.Test.Character:
#I  add new info
["CHAR",["A5","A5G1-p60B0",
0,[1,[2,3],[3,3],[4,4],[5,5]],"1a+3a^3b^3+4a^4+5a^5"]],
#I  AGR.Test.Character:
#I  add new info
["CHAR",["C4","C4G1-p4B0",0,[1,2,3,4],"1abcd"]],
true
```

We did not store the character information in the file `privdir/toc.json`, and GAP was able to identify the characters of the two permutation representations. (The identification of the character for the matrix representation fails because we cannot distinguish between the two Galois conjugate faithful characters.)

If we store the character information as proposed by GAP, this information will for example become part of the records returned by `OneAtlasGeneratingSetInfo` (3.5.6). (Note that we have to enter `"priv"` as the last argument of `AGR.CHAR` when we call the function interactively, in order to assign the information to the right context.)

Example

```
gap> AGR.CHAR("A5","A5G1-p60B0",
> 0,[1,[2,3],[3,3],[4,4],[5,5]],"1a+3a^3b^3+4a^4+5a^5", "priv" );
gap> AGR.CHAR("C4","C4G1-p4B0",0,[1,2,3,4],"1abcd", "priv" );
gap> AGR.Test.Characters( "priv" );
true
gap> OneAtlasGeneratingSetInfo( "C4" );
rec( charactername := "1abcd", constituents := [ 1, 2, 3, 4 ],
  contents := "priv", groupname := "C4", id := "",
  identifier := [ "C4", [ [ "priv", "C4G1-p4B0.m1" ] ], 1, 4 ],
  isPrimitive := false, p := 4, rankAction := 4,
  repname := "C4G1-p4B0", repnr := 1, size := 4,
  stabilizer := "1 < C2", standardization := 1, transitivity := 1,
  type := "perm" )
```

A string that describes the JSON format overview of the data extension can be created with `StringOfAtlasTableOfContents` (5.1.3).

Example

```
gap> Print( StringOfAtlasTableOfContents( "priv" ) );
{
"ID":"priv",
```

```
"Data":[
["GNAN",["C4","C4"]],

["GRS",["C4",4]],

["MXN",["C4",1]],

["MXO",["C4",[2]]],

["MXS",["C4",["C2"]]],

["RNG",["C4G1-Ar1aB0","CF(4)",["QuadraticField",-1],[1,0,1]]],

["API",["A5G1-p60B0",[1,60,"imprim","1 < S3"]]],
["API",["C4G1-p4B0",[1,4,"imprim","1 < C2"]]],

["CHAR",["A5","A5G1-p60B0",0,[1,[2,3],[3,3],[4,4],[5,5]],"1a+3a^3b^3+4\
a^4+5a^5"]],
["CHAR",["C4","C4G1-p4B0",0,[1,2,3,4],"1abcd"]]
]
}
```

If we prescribe a "DataURL" component that starts with "http" then also the "TOC" lines are listed, in order to enable remote access to the data.

Example

```
gap> Print( StringOfAtlasTableOfContents(
>               rec( ID:= "priv", DataURL:= "http://someurl" ) ) );
{
"ID":"priv",
"DataURL":"http://someurl",
"Data":[
["GNAN",["C4","C4"]],

["GRS",["C4",4]],

["MXN",["C4",1]],

["MXO",["C4",[2]]],

["MXS",["C4",["C2"]]],

["TOC",["perm","A5G1-p60B0.m",[118815263,24584221]]],
["TOC",["matalg","C4G1-Ar1aB0.g",[49815028]]],
["TOC",["otherscripts","C4G1-XtestW1",[-27672877]]],
["TOC",["out","C4G1-a2W1",[126435524]]],
["TOC",["maxes","C4G1-max1W1",[-27672877]]],
["TOC",["perm","C4G1-p4B0.m",[102601978]]],

["RNG",["C4G1-Ar1aB0","CF(4)",["QuadraticField",-1],[1,0,1]]],

["API",["A5G1-p60B0",[1,60,"imprim","1 < S3"]]],
["API",["C4G1-p4B0",[1,4,"imprim","1 < C2"]]],
```

```
["CHAR",["A5","A5G1-p60B0",0,[1,[2,3],[3,3],[4,4],[5,5]],"1a+3a^3b^3+4\
a^4+5a^5"]],
["CHAR",["C4","C4G1-p4B0",0,[1,2,3,4],"1abcd"]]
]
}
```

Finally, we "uninstall" our extension, and reset the info level that had been set to 1 in the beginning. (Also the group name `C4` is removed this way, which is an advantage of using a `toc.json` file over calling `AGR.GNAN` directly.),

Example

```
gap> AtlasOfGroupRepresentationsForgetData( "priv" );
gap> SetInfoLevel( InfoAtlasRep, locallevel );
```

We need not care about removing the temporary directory and the files in it. GAP will try to remove directories created with `DirectoryTemporary` (**Reference: DirectoryTemporary**) at the end of the GAP session.

Chapter 6

New GAP Objects and Utility Functions provided by the AtlasRep Package

This chapter describes GAP objects and functions that are provided by the AtlasRep package but that might be of general interest.

The new objects are straight line decisions (see Section 6.1) and black box programs (see Section 6.2).

The new functions are concerned with representations of minimal degree, see Section 6.3, and a JSON interface, see Section 6.4.

6.1 Straight Line Decisions

Straight line decisions are similar to straight line programs (see Section (**Reference: Straight Line Programs**)) but return `true` or `false`. A straight line decision checks whether its inputs have some property. An important example is to check whether a given list of group generators is in fact a list of standard generators (cf. Section3.3) for this group.

A straight line decision in GAP is represented by an object in the filter `IsStraightLineDecision` (6.1.1) that stores a list of "lines" each of which has one of the following three forms.

1. a nonempty dense list l of integers,
2. a pair $[l, i]$ where l is a list of form 1. and i is a positive integer,
3. a list [`"Order"`, i, n] where i and n are positive integers.

The first two forms have the same meaning as for straight line programs (see Section (**Reference: Straight Line Programs**)), the last form means a check whether the element stored at the i-th label has the order n.

For the meaning of the list of lines, see `ResultOfStraightLineDecision` (6.1.6).

Straight line decisions can be constructed using `StraightLineDecision` (6.1.5), defining attributes for straight line decisions are `NrInputsOfStraightLineDecision` (6.1.3) and `LinesOfStraightLineDecision` (6.1.2), an operation for straight line decisions is `ResultOfStraightLineDecision` (6.1.6).

Special methods applicable to straight line decisions are installed for the operations `Display` (**Reference: Display**), `IsInternallyConsistent` (**Reference: IsInternallyConsistent**), `PrintObj` (**Reference: PrintObj**), and `ViewObj` (**Reference: ViewObj**).

For a straight line decision *prog*, the default `Display` (**Reference: Display**) method prints the interpretation of *prog* as a sequence of assignments of associative words and of order checks; a record with components `gensnames` (with value a list of strings) and `listname` (a string) may be entered as second argument of `Display` (**Reference: Display**), in this case these names are used, the default for `gensnames` is `[ g1, g2, ... ]`, the default for *listname* is *r*.

6.1.1 IsStraightLineDecision

▷ `IsStraightLineDecision(`*obj*`)` (category)

Each straight line decision in GAP lies in the filter `IsStraightLineDecision`.

6.1.2 LinesOfStraightLineDecision

▷ `LinesOfStraightLineDecision(`*prog*`)` (operation)

Returns: the list of lines that define the straight line decision.

This defining attribute for the straight line decision *prog* (see `IsStraightLineDecision` (6.1.1)) corresponds to `LinesOfStraightLineProgram` (**Reference: LinesOfStraightLineProgram**) for straight line programs.

Example

```
gap> dec:= StraightLineDecision( [ [ [ 1, 1, 2, 1 ], 3 ],
> [ "Order", 1, 2 ], [ "Order", 2, 3 ], [ "Order", 3, 5 ] ] );
<straight line decision>
gap> LinesOfStraightLineDecision( dec );
[ [ [ 1, 1, 2, 1 ], 3 ], [ "Order", 1, 2 ], [ "Order", 2, 3 ],
  [ "Order", 3, 5 ] ]
```

6.1.3 NrInputsOfStraightLineDecision

▷ `NrInputsOfStraightLineDecision(`*prog*`)` (operation)

Returns: the number of inputs required for the straight line decision.

This defining attribute corresponds to `NrInputsOfStraightLineProgram` (**Reference: NrInputsOfStraightLineProgram**).

Example

```
gap> NrInputsOfStraightLineDecision( dec );
2
```

6.1.4 ScanStraightLineDecision

▷ `ScanStraightLineDecision(`*string*`)` (function)

Returns: a record containing the straight line decision, or `fail`.

Let *string* be a string that encodes a straight line decision in the sense that it consists of the lines listed for `ScanStraightLineProgram` (7.4.1), except that `oup` lines are not allowed, and instead lines of the following form may occur.

`chor` *a b*

means that it is checked whether the order of the element at label *a* is *b*.

`ScanStraightLineDecision` returns a record containing as the value of its component `program` the corresponding GAP straight line decision (see `IsStraightLineDecision` (6.1.1)) if the input string satisfies the syntax rules stated above, and returns `fail` otherwise. In the latter case, information about the first corrupted line of the program is printed if the info level of `InfoCMeatAxe` (7.1.2) is at least 1.

Example

```
gap> str:= "inp 2\nchor 1 2\nchor 2 3\nmu 1 2 3\nchor 3 5";;
gap> prg:= ScanStraightLineDecision( str );
rec( program := <straight line decision> )
gap> prg:= prg.program;;
gap> Display( prg );
# input:
r:= [ g1, g2 ];
# program:
if Order( r[1] ) <> 2 then  return false;  fi;
if Order( r[2] ) <> 3 then  return false;  fi;
r[3]:= r[1]*r[2];
if Order( r[3] ) <> 5 then  return false;  fi;
# return value:
true
```

6.1.5 StraightLineDecision

▷ `StraightLineDecision(`*lines[, nrgens]*`)` (function)

▷ `StraightLineDecisionNC(`*lines[, nrgens]*`)` (function)

Returns: the straight line decision given by the list of lines.

Let *lines* be a list of lists that defines a unique straight line decision (see `IsStraightLineDecision` (6.1.1)); in this case `StraightLineDecision` returns this program, otherwise an error is signalled. The optional argument *nrgens* specifies the number of input generators of the program; if a list of integers (a line of form 1. in the definition above) occurs in *lines* then this number is not determined by *lines* and therefore *must* be specified by the argument *nrgens*; if not then `StraightLineDecision` returns `fail`.

`StraightLineDecisionNC` does the same as `StraightLineDecision`, except that the internal consistency of the program is not checked.

6.1.6 ResultOfStraightLineDecision

▷ `ResultOfStraightLineDecision(`*prog, gens[, orderfunc]*`)` (operation)

Returns: `true` if all checks succeed, otherwise `false`.

`ResultOfStraightLineDecision` evaluates the straight line decision (see `IsStraightLineDecision` (6.1.1)) *prog* at the group elements in the list *gens*.

The function for computing the order of a group element can be given as the optional argument *orderfunc*. For example, this may be a function that gives up at a certain limit if one has to be aware of extremely huge orders in failure cases.

The *result* of a straight line decision with lines $p_1, p_2, \ldots, p_k$ when applied to *gens* is defined as follows.

(a) First a list r of intermediate values is initialized with a shallow copy of `gens`.

(b) For $i \leq k$, before the i-th step, let r be of length n. If p_i is the external representation of an associative word in the first n generators then the image of this word under the homomorphism that is given by mapping r to these first n generators is added to r. If p_i is a pair $[l, j]$, for a list l, then the same element is computed, but instead of being added to r, it replaces the j-th entry of r. If p_i is a triple $[$`"Order"`$, i, n]$ then it is checked whether the order of $r[i]$ is n; if not then `false` is returned immediately.

(c) If all k lines have been processed and no order check has failed then `true` is returned.

Here are some examples.

Example

```
gap> dec:= StraightLineDecision( [ ], 1 );
<straight line decision>
gap> ResultOfStraightLineDecision( dec, [ () ] );
true
```

The above straight line decision `dec` returns `true` –for *any* input of the right length.

Example

```
gap> dec:= StraightLineDecision( [ [ [ 1, 1, 2, 1 ], 3 ],
>       [ "Order", 1, 2 ], [ "Order", 2, 3 ], [ "Order", 3, 5 ] ] );
<straight line decision>
gap> LinesOfStraightLineDecision( dec );
[ [ [ 1, 1, 2, 1 ], 3 ], [ "Order", 1, 2 ], [ "Order", 2, 3 ],
  [ "Order", 3, 5 ] ]
gap> ResultOfStraightLineDecision( dec, [ (), () ] );
false
gap> ResultOfStraightLineDecision( dec, [ (1,2)(3,4), (1,4,5) ] );
true
```

The above straight line decision admits two inputs; it tests whether the orders of the inputs are 2 and 3, and the order of their product is 5.

6.1.7 Semi-Presentations and Presentations

We can associate a *finitely presented group* F/R to each straight line decision `dec`, say, as follows. The free generators of the free group F are in bijection with the inputs, and the defining relators generating R as a normal subgroup of F are given by those words w^k for which `dec` contains a check whether the order of w equals k.

So if `dec` returns `true` for the input list $[g_1, g_2, \ldots, g_n]$ then mapping the free generators of F to the inputs defines an epimorphism Φ from F to the group G, say, that is generated by these inputs, such that R is contained in the kernel of Φ.

(Note that "satisfying `dec`" is a stronger property than "satisfying a presentation". For example, $\langle x \mid x^2 = x^3 = 1 \rangle$ is a presentation for the trivial group, but the straight line decision that checks whether the order of x is both 2 and 3 clearly always returns `false`.)

AtlasRep supports the following two kinds of straight line decisions.

- A *presentation* is a straight line decision `dec` that is defined for a set of standard generators of a group G and that returns `true` if and only if the list of inputs is in fact a sequence of such standard generators for G. In other words, the relators derived from the order checks in the way described above are defining relators for G, and moreover these relators are words in terms of standard generators. (In particular the kernel of the map Φ equals R whenever `dec` returns `true`.)

- A *semi-presentation* is a straight line decision `dec` that is defined for a set of standard generators of a group G and that returns `true` for a list of inputs *that is known to generate a group isomorphic with* G if and only if these inputs form in fact a sequence of standard generators for G. In other words, the relators derived from the order checks in the way described above are *not necessarily defining relators* for G, but if we assume that the g_i generate G then they are standard generators. (In particular, F/R may be a larger group than G but in this case Φ maps the free generators of F to standard generators of G.)

  More about semi-presentations can be found in [NW05].

Available presentations and semi-presentations are listed by `DisplayAtlasInfo` (3.5.1), they can be accessed via `AtlasProgram` (3.5.4). (Clearly each presentation is also a semi-presentation. So a semi-presentation for some standard generators of a group is regarded as available whenever a presentation for these standard generators and this group is available.)

Note that different groups can have the same semi-presentation. We illustrate this with an example that is mentioned in [NW05]. The groups $L_2(7) \cong L_3(2)$ and $L_2(8)$ are generated by elements of the orders 2 and 3 such that their product has order 7, and no further conditions are necessary to define standard generators.

Example

```
gap> check:= AtlasProgram( "L2(8)", "check" );
rec( groupname := "L2(8)",
  identifier := [ "L2(8)", "L28G1-check1", 1, 1 ],
  program := <straight line decision>, standardization := 1,
  version := "1" )
gap> gens:= AtlasGenerators( "L2(8)", 1 );
rec( charactername := "1a+8a", constituents := [ 1, 6 ],
  contents := "core",
  generators := [ (1,2)(3,4)(6,7)(8,9), (1,3,2)(4,5,6)(7,8,9) ],
  groupname := "L2(8)", id := "",
  identifier := [ "L2(8)", [ "L28G1-p9B0.m1", "L28G1-p9B0.m2" ], 1, 9
     ], isPrimitive := true, maxnr := 1, p := 9, rankAction := 2,
  repname := "L28G1-p9B0", repnr := 1, size := 504,
  stabilizer := "2^3:7", standardization := 1, transitivity := 3,
  type := "perm" )
gap> ResultOfStraightLineDecision( check.program, gens.generators );
true
gap> gens:= AtlasGenerators( "L3(2)", 1 );
rec( contents := "core", generators := [ (2,4)(3,5), (1,2,3)(5,6,7) ],
  groupname := "L3(2)", id := "a",
  identifier := [ "L3(2)", [ "L27G1-p7aB0.m1", "L27G1-p7aB0.m2" ], 1,
      7 ], isPrimitive := true, maxnr := 1, p := 7, rankAction := 2,
  repname := "L27G1-p7aB0", repnr := 1, size := 168,
  stabilizer := "S4", standardization := 1, transitivity := 2,
  type := "perm" )
```

```
gap> ResultOfStraightLineDecision( check.program, gens.generators );
true
```

6.1.8 AsStraightLineDecision

▷ AsStraightLineDecision(*bbox*) (attribute)

Returns: an equivalent straight line decision for the given black box program, or `fail`.

For a black box program (see `IsBBoxProgram` (6.2.1)) *bbox*, `AsStraightLineDecision` returns a straight line decision (see `IsStraightLineDecision` (6.1.1)) with the same output as *bbox*, in the sense of `AsBBoxProgram` (6.2.5), if such a straight line decision exists, and `fail` otherwise.

Example

```
gap> lines:= [ [ "Order", 1, 2 ], [ "Order", 2, 3 ],
>              [ [ 1, 1, 2, 1 ], 3 ], [ "Order", 3, 5 ] ];;
gap> dec:= StraightLineDecision( lines, 2 );
<straight line decision>
gap> bboxdec:= AsBBoxProgram( dec );
<black box program>
gap> asdec:= AsStraightLineDecision( bboxdec );
<straight line decision>
gap> LinesOfStraightLineDecision( asdec );
[ [ "Order", 1, 2 ], [ "Order", 2, 3 ], [ [ 1, 1, 2, 1 ], 3 ],
  [ "Order", 3, 5 ] ]
```

6.1.9 StraightLineProgramFromStraightLineDecision

▷ StraightLineProgramFromStraightLineDecision(*dec*) (operation)

Returns: the straight line program associated to the given straight line decision.

For a straight line decision *dec* (see `IsStraightLineDecision` (6.1.1)), `StraightLineProgramFromStraightLineDecision` returns the straight line program (see `IsStraightLineProgram` (**Reference: IsStraightLineProgram**) obtained by replacing each line of type 3. (i.e, each order check) by an assignment of the power in question to a new slot, and by declaring the list of these elements as the return value.

This means that the return value describes exactly the defining relators of the presentation that is associated to the straight line decision, see 6.1.7.

For example, one can use the return value for printing the relators with `StringOfResultOfStraightLineProgram` (**Reference: StringOfResultOfStraightLineProgram**), or for explicitly constructing the relators as words in terms of free generators, by applying `ResultOfStraightLineProgram` (**Reference: ResultOfStraightLineProgram**) to the program and to these generators.

Example

```
gap> dec:= StraightLineDecision( [ [ [ 1, 1, 2, 1 ], 3 ],
> [ "Order", 1, 2 ], [ "Order", 2, 3 ], [ "Order", 3, 5 ] ] );
<straight line decision>
gap> prog:= StraightLineProgramFromStraightLineDecision( dec );
<straight line program>
gap> Display( prog );
# input:
r:= [ g1, g2 ];
```

```
# program:
r[3]:= r[1]*r[2];
r[4]:= r[1]^2;
r[5]:= r[2]^3;
r[6]:= r[3]^5;
# return values:
[ r[4], r[5], r[6] ]
gap> StringOfResultOfStraightLineProgram( prog, [ "a", "b" ] );
"[ a^2, b^3, (ab)^5 ]"
gap> gens:= GeneratorsOfGroup( FreeGroup( "a", "b" ) );
[ a, b ]
gap> ResultOfStraightLineProgram( prog, gens );
[ a^2, b^3, (a*b)^5 ]
```

6.2 Black Box Programs

Black box programs formalize the idea that one takes some group elements, forms arithmetic expressions in terms of them, tests properties of these expressions, executes conditional statements (including jumps inside the program) depending on the results of these tests, and eventually returns some result.

A specification of the language can be found in [Nic06], see also
`http://atlas.math.rwth-aachen.de/Atlas/info/blackbox.html`.

The *inputs* of a black box program may be explicit group elements, and the program may also ask for random elements from a given group. The *program steps* form products, inverses, conjugates, commutators, etc. of known elements, *tests* concern essentially the orders of elements, and the *result* is a list of group elements or `true` or `false` or `fail`.

Examples that can be modeled by black box programs are

***straight line programs*,**
: which require a fixed number of input elements and form arithmetic expressions of elements but do not use random elements, tests, conditional statements and jumps; the return value is always a list of elements; these programs are described in Section (**Reference: Straight Line Programs**).

***straight line decisions*,**
: which differ from straight line programs only in the sense that also order tests are admissible, and that the return value is `true` if all these tests are satisfied, and `false` as soon as the first such test fails; they are described in Section 6.1.

***scripts for finding standard generators*,**
: which take a group and a function to generate a random element in this group but no explicit input elements, admit all control structures, and return either a list of standard generators or `fail`; see `ResultOfBBoxProgram` (6.2.4) for examples.

In the case of general black box programs, currently GAP provides only the possibility to read an existing program via `ScanBBoxProgram` (6.2.2), and to run the program using `RunBBoxProgram` (6.2.3). It is not our aim to write such programs in GAP.

The special case of the "find" scripts mentioned above is also admissible as an argument of `ResultOfBBoxProgram` (6.2.4), which returns either the set of found generators or `fail`.

Contrary to the general situation, more support is provided for straight line programs and straight line decisions in GAP, see Section (**Reference: Straight Line Programs**) for functions that manipulate them (compose, restrict etc.).

The functions `AsStraightLineProgram` (6.2.6) and `AsStraightLineDecision` (6.1.8) can be used to transform a general black box program object into a straight line program or a straight line decision if this is possible.

Conversely, one can create an equivalent general black box program from a straight line program or from a straight line decision with `AsBBoxProgram` (6.2.5).

Computing a straight line program related to a given straight line decision is supported in the sense of `StraightLineProgramFromStraightLineDecision` (6.1.9).

Note that none of these three kinds of objects is a special case of another: Running a black box program with `RunBBoxProgram` (6.2.3) yields a record, running a straight line program with `ResultOfStraightLineProgram` (**Reference: ResultOfStraightLineProgram**) yields a list of elements, and running a straight line decision with `ResultOfStraightLineDecision` (6.1.6) yields `true` or `false`.

6.2.1 IsBBoxProgram

▷ `IsBBoxProgram(`*obj*`)` (category)

Each black box program in GAP lies in the filter `IsBBoxProgram`.

6.2.2 ScanBBoxProgram

▷ `ScanBBoxProgram(`*string*`)` (function)

Returns: a record containing the black box program encoded by the input string, or `fail`.

For a string *string* that describes a black box program, e.g., the return value of `StringFile` (**GAPDoc: StringFile**), `ScanBBoxProgram` computes this black box program. If this is successful then the return value is a record containing as the value of its component `program` the corresponding GAP object that represents the program, otherwise `fail` is returned.

As the first example, we construct a black box program that tries to find standard generators for the alternating group A_5; these standard generators are any pair of elements of the orders 2 and 3, respectively, such that their product has order 5.

Example

```
gap> findstr:= "\
>   set V 0\n\
> lbl START1\n\
>   rand 1\n\
>   ord 1 A\n\
>   incr V\n\
>   if V gt 100 then timeout\n\
>   if A notin 1 2 3 5 then fail\n\
>   if A noteq 2 then jmp START1\n\
> lbl START2\n\
>   rand 2\n\
>   ord 2 B\n\
>   incr V\n\
>   if V gt 100 then timeout\n\
>   if B notin 1 2 3 5 then fail\n\
```

```
>   if B noteq 3 then jmp START2\n\
>   # The elements 1 and 2 have the orders 2 and 3, respectively.\n\
>   set X 0\n\
> lbl CONJ\n\
>   incr X\n\
>   if X gt 100 then timeout\n\
>   rand 3\n\
>   cjr 2 3\n\
>   mu 1 2 4   # ab\n\
>   ord 4 C\n\
>   if C notin 2 3 5 then fail\n\
>   if C noteq 5 then jmp CONJ\n\
>   oup 2 1 2";;
gap> find:= ScanBBoxProgram( findstr );
rec( program := <black box program> )
```

The second example is a black box program that checks whether its two inputs are standard generators for A_5.

Example

```
gap> checkstr:= "\
> chor 1 2\n\
> chor 2 3\n\
> mu 1 2 3\n\
> chor 3 5";;
gap> check:= ScanBBoxProgram( checkstr );
rec( program := <black box program> )
```

6.2.3 RunBBoxProgram

▷ RunBBoxProgram(*prog*, *G*, *input*, *options*) (function)

Returns: a record describing the result and the statistics of running the black box program *prog*, or `fail`, or the string `"timeout"`.

For a black box program *prog*, a group *G*, a list *input* of group elements, and a record *options*, `RunBBoxProgram` applies *prog* to *input*, where *G* is used only to compute random elements.

The return value is `fail` if a syntax error or an explicit `fail` statement is reached at runtime, and the string `"timeout"` if a `timeout` statement is reached. (The latter might mean that the random choices were unlucky.) Otherwise a record with the following components is returned.

`gens`
: a list of group elements, bound if an `oup` statement was reached,

`result`
: `true` if a `true` statement was reached, `false` if either a `false` statement or a failed order check was reached,

The other components serve as statistical information about the numbers of the various operations (`multiply`, `invert`, `power`, `order`, `random`, `conjugate`, `conjugateinplace`, `commutator`), and the runtime in milliseconds (`timetaken`).

The following components of *options* are supported.

`randomfunction`
: the function called with argument `G` in order to compute a random element of `G` (default `PseudoRandom` (**Reference: PseudoRandom**))

`orderfunction`
: the function for computing element orders (default `Order` (**Reference: Order**)),

`quiet`
: if `true` then ignore `echo` statements (default `false`),

`verbose`
: if `true` then print information about the line that is currently processed, and about order checks (default `false`),

`allowbreaks`
: if `true` then call `Error` (**Reference: Error**) when a `break` statement is reached, otherwise ignore `break` statements (default `true`).

As an example, we run the black box programs constructed in the example for `ScanBBoxProgram` (6.2.2).

Example

```
gap> g:= AlternatingGroup( 5 );;
gap> res:= RunBBoxProgram( find.program, g, [], rec() );;
gap> IsBound( res.gens );  IsBound( res.result );
true
false
gap> List( res.gens, Order );
[ 2, 3 ]
gap> Order( Product( res.gens ) );
5
gap> res:= RunBBoxProgram( check.program, "dummy", res.gens, rec() );;
gap> IsBound( res.gens );  IsBound( res.result );
false
true
gap> res.result;
true
gap> othergens:= GeneratorsOfGroup( g );;
gap> res:= RunBBoxProgram( check.program, "dummy", othergens, rec() );;
gap> res.result;
false
```

6.2.4 ResultOfBBoxProgram

▷ `ResultOfBBoxProgram(`*prog, G[, options]*`)` (function)

Returns: a list of group elements or `true`, `false`, `fail`, or the string `"timeout"`.

This function calls `RunBBoxProgram` (6.2.3) with the black box program *prog* and second argument either a group or a list of group elements; if *options* is not given then the default options of `RunBBoxProgram` (6.2.3) are assumed. The return value is `fail` if this call yields `fail`, otherwise the `gens` component of the result, if bound, or the `result` component if not.

Note that a *group* `G` is used as the second argument in the call of `RunBBoxProgram` (6.2.3) (the source for random elements), whereas a *list* `G` is used as the third argument (the inputs).

As an example, we run the black box programs constructed in the example for ScanBBoxProgram (6.2.2).

Example

```
gap> g:= AlternatingGroup( 5 );;
gap> res:= ResultOfBBoxProgram( find.program, g );;
gap> List( res, Order );
[ 2, 3 ]
gap> Order( Product( res ) );
5
gap> res:= ResultOfBBoxProgram( check.program, res );
true
gap> othergens:= GeneratorsOfGroup( g );;
gap> res:= ResultOfBBoxProgram( check.program, othergens );
false
```

6.2.5 AsBBoxProgram

▷ AsBBoxProgram(*slp*) (attribute)

Returns: an equivalent black box program for the given straight line program or straight line decision.

Let *slp* be a straight line program (see IsStraightLineProgram (**Reference: IsStraightLineProgram**)) or a straight line decision (see IsStraightLineDecision (6.1.1)). Then AsBBoxProgram returns a black box program *bbox* (see IsBBoxProgram (6.2.1)) with the "same" output as *slp*, in the sense that ResultOfBBoxProgram (6.2.4) yields the same result for *bbox* as ResultOfStraightLineProgram (**Reference: ResultOfStraightLineProgram**) or ResultOfStraightLineDecision (6.1.6), respectively, for *slp*.

Example

```
gap> f:= FreeGroup( "x", "y" );;  gens:= GeneratorsOfGroup( f );;
gap> slp:= StraightLineProgram( [ [1,2,2,3], [3,-1] ], 2 );
<straight line program>
gap> ResultOfStraightLineProgram( slp, gens );
y^-3*x^-2
gap> bboxslp:= AsBBoxProgram( slp );
<black box program>
gap> ResultOfBBoxProgram( bboxslp, gens );
[ y^-3*x^-2 ]
gap> lines:= [ [ "Order", 1, 2 ], [ "Order", 2, 3 ],
>              [ [ 1, 1, 2, 1 ], 3 ], [ "Order", 3, 5 ] ];;
gap> dec:= StraightLineDecision( lines, 2 );
<straight line decision>
gap> ResultOfStraightLineDecision( dec, [ (1,2)(3,4), (1,3,5) ] );
true
gap> ResultOfStraightLineDecision( dec, [ (1,2)(3,4), (1,3,4) ] );
false
gap> bboxdec:= AsBBoxProgram( dec );
<black box program>
gap> ResultOfBBoxProgram( bboxdec, [ (1,2)(3,4), (1,3,5) ] );
true
gap> ResultOfBBoxProgram( bboxdec, [ (1,2)(3,4), (1,3,4) ] );
false
```

6.2.6 AsStraightLineProgram

▷ AsStraightLineProgram(*bbox*) (attribute)

Returns: an equivalent straight line program for the given black box program, or fail.

For a black box program (see AsBBoxProgram (6.2.5)) *bbox*, AsStraightLineProgram returns a straight line program (see IsStraightLineProgram (**Reference: IsStraightLineProgram**)) with the same output as *bbox* if such a straight line program exists, and fail otherwise.

Example

```
gap> Display( AsStraightLineProgram( bboxslp ) );
# input:
r:= [ g1, g2 ];
# program:
r[3]:= r[1]^2;
r[4]:= r[2]^3;
r[5]:= r[3]*r[4];
r[3]:= r[5]^-1;
# return values:
[ r[3] ]
gap> AsStraightLineProgram( bboxdec );
fail
```

6.3 Representations of Minimal Degree

This section deals with minimal degrees of permutation and matrix representations. We do not provide an algorithm that computes these degrees for an arbitrary group, we only provide some tools for evaluating known databases, mainly concerning "bicyclic extensions" (see [CCN+85, Section 6.5]) of simple groups, in order to derive the minimal degrees, see Section 6.3.4.

In the AtlasRep package, this information can be used for prescribing "minimality conditions" in DisplayAtlasInfo (3.5.1), OneAtlasGeneratingSetInfo (3.5.6), and AllAtlasGeneratingSetInfos (3.5.7). An overview of the stored minimal degrees can be shown with BrowseMinimalDegrees (3.6.1).

6.3.1 MinimalRepresentationInfo

▷ MinimalRepresentationInfo(*grpname, conditions*) (function)

Returns: a record with the components value and source, or fail

Let *grpname* be the GAP name of a group G, say. If the information described by *conditions* about minimal representations of this group can be computed or is stored then MinimalRepresentationInfo returns a record with the components value and source, otherwise fail is returned.

The following values for *conditions* are supported.

- If *conditions* is NrMovedPoints (**Reference: NrMovedPoints for a permutation**) then value, if known, is the degree of a minimal faithful (not necessarily transitive) permutation representation for G.
- If *conditions* consists of Characteristic (**Reference: Characteristic**) and a prime integer p then value, if known, is the dimension of a minimal faithful (not necessarily irreducible) matrix representation in characteristic p for G.

- If *conditions* consists of `Size` (**Reference: Size**) and a prime power `q` then `value`, if known, is the dimension of a minimal faithful (not necessarily irreducible) matrix representation over the field of size `q` for *G*.

In all cases, the value of the component `source` is a list of strings that describe sources of the information, which can be the ordinary or modular character table of *G* (see [CCN+85], [JLPW95], [HL89]), the table of marks of *G*, or [Jan05]. For an overview of minimal degrees of faithful matrix representations for sporadic simple groups and their covering groups, see also

`http://www.math.rwth-aachen.de/~MOC/mindeg/`.

Note that `MinimalRepresentationInfo` cannot provide any information about minimal representations over prescribed fields in characteristic zero.

Information about groups that occur in the AtlasRep package is precomputed in `MinimalRepresentationInfoData` (6.3.2), so the packages CTblLib and TomLib are not needed when `MinimalRepresentationInfo` is called for these groups. (The only case that is not covered by this list is that one asks for the minimal degree of matrix representations over a prescribed field in characteristic coprime to the group order.)

One of the following strings can be given as an additional last argument.

`"cache"`
: means that the function tries to compute (and then store) values that are not stored in `MinimalRepresentationInfoData` (6.3.2), but stored values are preferred; this is also the default.

`"lookup"`
: means that stored values are returned but the function does not attempt to compute values that are not stored in `MinimalRepresentationInfoData` (6.3.2).

`"recompute"`
: means that the function always tries to compute the desired value, and checks the result against stored values.

Example

```
gap> MinimalRepresentationInfo( "A5", NrMovedPoints );
rec(
  source := [ "computed (alternating group)",
      "computed (char. table)", "computed (subgroup tables)",
      "computed (subgroup tables, known repres.)",
      "computed (table of marks)" ], value := 5 )
gap> MinimalRepresentationInfo( "A5", Characteristic, 2 );
rec( source := [ "computed (char. table)" ], value := 2 )
gap> MinimalRepresentationInfo( "A5", Size, 2 );
rec( source := [ "computed (char. table)" ], value := 4 )
```

6.3.2 MinimalRepresentationInfoData

▷ `MinimalRepresentationInfoData` (global variable)

This is a record whose components are GAP names of groups for which information about minimal permutation and matrix representations were known in advance or have been computed in the current GAP session. The value for the group *G*, say, is a record with the following components.

`NrMovedPoints`
: a record with the components `value` (the degree of a smallest faithful permutation representation of G) and `source` (a string describing the source of this information).

`Characteristic`
: a record whose components are at most `0` and strings corresponding to prime integers, each bound to a record with the components `value` (the degree of a smallest faithful matrix representation of G in this characteristic) and `source` (a string describing the source of this information).

`CharacteristicAndSize`
: a record whose components are strings corresponding to prime integers p, each bound to a record with the components `sizes` (a list of powers q of p), `dimensions` (the corresponding list of minimal dimensions of faithful matrix representations of G over a field of size q), `sources` (the corresponding list of strings describing the source of this information), and `complete` (a record with the components `val` (`true` if the minimal dimension over *any* finite field in characteristic p can be derived from the values in the record, and `false` otherwise) and `source` (a string describing the source of this information)).

The values are set by `SetMinimalRepresentationInfo` (6.3.3).

6.3.3 SetMinimalRepresentationInfo

▷ `SetMinimalRepresentationInfo(`*grpname, op, value, source*`)` (function)

Returns: `true` if the values were successfully set, `false` if stored values contradict the given ones.

This function sets an entry in `MinimalRepresentationInfoData` (6.3.2) for the group G, say, with GAP name *grpname*.

Supported values for *op* are

- `"NrMovedPoints"` (see `NrMovedPoints` (**Reference: NrMovedPoints for a permutation**)), which means that *value* is the degree of minimal faithful (not necessarily transitive) permutation representations of G,
- a list of length two with first entry `"Characteristic"` (see `Characteristic` (**Reference: Characteristic**)) and second entry *char* either zero or a prime integer, which means that *value* is the dimension of minimal faithful (not necessarily irreducible) matrix representations of G in characteristic *char*,
- a list of length two with first entry `"Size"` (see `Size` (**Reference: Size**)) and second entry a prime power q, which means that *value* is the dimension of minimal faithful (not necessarily irreducible) matrix representations of G over the field with q elements, and
- a list of length three with first entry `"Characteristic"` (see `Characteristic` (**Reference: Characteristic**)), second entry a prime integer p, and third entry the string `"complete"`, which means that the information stored for characteristic p is complete in the sense that for any given power q of p, the minimal faithful degree over the field with q elements equals that for the largest stored field size of which q is a power.

In each case, *source* is a string describing the source of the data; *computed* values are detected from the prefix "comp" of *source*.

If the intended value is already stored and differs from *value* then an error message is printed.

Example

```
gap> SetMinimalRepresentationInfo( "A5", "NrMovedPoints", 5,
>      "computed (alternating group)" );
true
gap> SetMinimalRepresentationInfo( "A5", [ "Characteristic", 0 ], 3,
>      "computed (char. table)" );
true
gap> SetMinimalRepresentationInfo( "A5", [ "Characteristic", 2 ], 2,
>      "computed (char. table)" );
true
gap> SetMinimalRepresentationInfo( "A5", [ "Size", 2 ], 4,
>      "computed (char. table)" );
true
gap> SetMinimalRepresentationInfo( "A5", [ "Size", 4 ], 2,
>      "computed (char. table)" );
true
gap> SetMinimalRepresentationInfo( "A5", [ "Characteristic", 3 ], 3,
>      "computed (char. table)" );
true
```

6.3.4 Criteria Used to Compute Minimality Information

The information about the minimal degree of a faithful *matrix representation* of G in a given characteristic or over a given field in positive characteristic is derived from the relevant (ordinary or modular) character table of G, except in a few cases where this table itself is not known but enough information about the degrees is available in [HL89] and [Jan05].

The following criteria are used for deriving the minimal degree of a faithful *permutation representation* of G from the information in the GAP libraries of character tables and of tables of marks.

- If the name of G has the form "An" or "An.2" (denoting alternating and symmetric groups, respectively) then the minimal degree is n, except if n is smaller than 3 or 2, respectively.
- If the name of G has the form "L2(q)" (denoting projective special linear groups in dimension two) then the minimal degree is $q+1$, except if $q \in \{2,3,5,7,9,11\}$, see [Hup67, Satz II.8.28].
- If the largest maximal subgroup of G is core-free then the index of this subgroup is the minimal degree. (This is used when the two character tables in question and the class fusion are available in GAP's Character Table Library [Bre22]; this happens for many character tables of simple groups.)
- If G has a unique minimal normal subgroup then each minimal faithful permutation representation is transitive. (Note that the core of each point stabilizer is either trivial or contains the unique minimal normal subgroup.)

  In this case, the minimal degree can be computed directly from the information in the table of marks of G if this is available in GAP's Library of Tables of Marks [NMP18].

  Suppose that the largest maximal subgroup of G is not core-free but simple and normal in G, and that the other maximal subgroups of G are core-free. In this case, we take the minimum of

the indices of the core-free maximal subgroups and of the product of index and minimal degree of the normal maximal subgroup. (This suffices since no core-free subgroup of the whole group can contain a nontrivial normal subgroup of a normal maximal subgroup.)

Let N be the unique minimal normal subgroup of G, and assume that G/N is simple and has minimal degree n, say. If there is a subgroup U of index $n \cdot |N|$ in G that intersects N trivially then the minimal degree of G is $n \cdot |N|$. (This is used for the case that N is central in G and $N \times U$ occurs as a subgroup of G.)

- If we know a subgroup of G whose minimal degree is n, say, and if we know either (a class fusion from) a core-free subgroup of index n in G or a faithful permutation representation of degree n for G then n is the minimal degree for G. (This happens often for tables of almost simple groups.)

6.4 A JSON Interface

We define a mapping between certain GAP objects and JSON (JavaScript Object Notation) texts (see [JSO14]), as follows.

- The three GAP values `true`, `false`, and `fail` correspond to the JSON texts `true`, `false`, and `null`, respectively.
- GAP strings correspond to JSON strings; special characters in a GAP string (control characters ASCII 0 to 31, backslash and double quote) are mapped as defined in JSON's specification, and other ASCII characters are kept as they are; if a GAP string contains non-ASCII characters, it is assumed that it is UTF-8 encoded, and one may choose either to keep non-ASCII characters as they are, or to create an ASCII only JSON string, using JSON's syntax for Unicode code points ("`\uXXXX`"); in the other direction, JSON strings are assumed to be UTF-8 encoded, and are mapped to UTF-8 encoded GAP strings, by keeping the non-ASCII characters and converting substrings of the form `\uXXXX` accordingly.
- GAP integers (in the sense of `IsInt` (**Reference: IsInt**)) are mapped to JSON numbers that consist of digits and optionally a leading sign character `-`; in the other direction, JSON numbers of this form and also JSON numbers that involve no decimal dots and have no negative exponent (for example `"2e3"`) are mapped to GAP integers.
- GAP rationals (in the sense of `IsRat` (**Reference: IsRat**)) which are not integers are represented by JSON floating point numbers; the JSON representation (and hence the precision) is given by first applying `Float` (**Reference: Float**) and then `String` (**Reference: String**).
- GAP floats (in the sense of Chapter (**Reference: Floats**) in the GAP Reference Manual) are mapped to JSON floating point numbers; the JSON representation (and hence the precision) is given by applying `String` (**Reference: String**); in the other direction, JSON numbers that involve a decimal dot or a negative exponent are mapped to GAP floats.
- (Nested and not self-referential) dense GAP lists of objects correspond to JSON arrays such that the list entries correspond to each other. (Note that JSON does not support non-dense arrays.)
- (Nested and not self-referential) GAP records correspond to JSON objects such that both labels (which are strings in GAP and JSON) and values correspond to each other.

The GAP functions AGR.JsonText (6.4.2) and AGR.GapObjectOfJsonText (6.4.3) can be used to create a JSON text from a suitable GAP object and the GAP object that corresponds to a given JSON text, respectively.

Note that the composition of the two functions is in general *not* the identity mapping, because AGR.JsonText (6.4.2) accepts non-integer rationals, whereas AGR.GapObjectOfJsonText (6.4.3) does not create such objects.

Note also that the results of AGR.JsonText (6.4.2) do not contain information about dependencies between common subobjects. This is another reason why applying first AGR.JsonText (6.4.2) and then AGR.GapObjectOfJsonText (6.4.3) may yield a GAP object with different behaviour.

Applying AGR.JsonText (6.4.2) to a self-referential object such as [~] will raise a "recursion depth trap" error.

6.4.1 Why JSON?

The aim of this JSON interface is to read and write certain data files with GAP such that these files become easily accessible independent of GAP. The function AGR.JsonText (6.4.2) is intended just as a prototype, variants of this function are very likely to appear in other contexts, for example in order to force certain line formatting or ordering of record components.

It is *not* the aim of the JSON interface to provide self-contained descriptions of arbitrary GAP objects, in order to read them into a GAP session. Note that those GAP objects for which a JSON equivalent exists (and many more) can be easily written to files as they are, and GAP can read them efficiently. On the other hand, more complicated GAP objects can be written and read via the so-called *pickling*, for which a framework is provided by the GAP package IO [Neu14].

Here are a few situations which are handled well by pickling but which cannot be addressed with a JSON interface.

- Pickling and unpickling take care of common subobjects of the given GAP object. The following example shows that the applying first AGR.JsonText (6.4.2) and then AGR.GapObjectOfJsonText (6.4.3) may yield an object which behaves differently.

Example

```
gap> l:= [ [ 1 ] ];; l[2]:= l[1];;  l;
[ [ 1 ], [ 1 ] ]
gap> new:= AGR.GapObjectOfJsonText( AGR.JsonText( l ) ).value;
[ [ 1 ], [ 1 ] ]
gap> Add( l[1], 2 );  l;
[ [ 1, 2 ], [ 1, 2 ] ]
gap> Add( new[1], 2 );  new;
[ [ 1, 2 ], [ 1 ] ]
```

- GAP admits self-referential objects, for example as follows.

Example

```
gap> l:= [];;  l[1]:= l;;
```

Pickling and unpickling take care of self-referential objects, but AGR.JsonText (6.4.2) does not support the conversion of such objects.

6.4.2 AGR.JsonText

▷ AGR.JsonText(*obj[, mode]*) (function)

Returns: a new mutable string that describes *obj* as a JSON text, or fail.

If *obj* is a GAP object for which a corresponding JSON text exists, according to the mapping described above, then such a JSON text is returned. Otherwise, fail is returned.

If the optional argument *mode* is given and has the value "ASCII" then the result in an ASCII string, otherwise the encoding of strings that are involved in *obj* is kept.

Example

```
gap> AGR.JsonText( [] );
"[]"
gap> AGR.JsonText( "" );
"\"\""
gap> AGR.JsonText( "abc\ndef\cghi" );
"\"abc\\ndef\\u0003ghi\""
gap> AGR.JsonText( rec() );
"{}"
gap> AGR.JsonText( [ , 2 ] );
fail
gap> str:= [ '\303', '\266' ];;  # umlaut o
gap> json:= AGR.JsonText( str );;  List( json, IntChar );
[ 34, 195, 182, 34 ]
gap> AGR.JsonText( str, "ASCII" );
"\"\\u00F6\""
```

6.4.3 AGR.GapObjectOfJsonText

▷ AGR.GapObjectOfJsonText(*string*) (function)

Returns: a new mutable record whose value component, if bound, contains a mutable GAP object that represents the JSON text *string*.

If *string* is a string that represents a JSON text then the result is a record with the components value (the corresponding GAP object in the sense of the above interface) and status (value true). Otherwise, the result is a record with the components status (value false) and errpos (the position in *string* where the string turns out to be not valid JSON).

Example

```
gap> AGR.GapObjectOfJsonText( "{ \"a\": 1 }" );
rec( status := true, value := rec( a := 1 ) )
gap> AGR.GapObjectOfJsonText( "{ \"a\": x }" );
rec( errpos := 8, status := false )
```

Chapter 7

Technicalities of the AtlasRep Package

This chapter describes those parts of the GAP interface to the ATLAS of Group Representations that do not belong to the user interface (cf. Chapter 3).

Besides global variables used for administrational purposes (see Section 7.1) and several sanity checks (see Section 7.9), they can be regarded as the interface between the data actually contained in the files and the corresponding GAP objects (see Section 7.2, 7.3, 7.4, and 7.5), and the interface between the remote and the local version of the database (see Section 7.6 and 7.8). The former interface contains functions to read and write files in MeatAxe format, which may be interesting for users familiar with MeatAxe standalones (see for example [Rin]). Other low level functions may be undocumented in the sense that they are not described in this manual. Users interested in them may look at the actual implementation in the `gap` directory of the package, but it may happen that this will be changed in future versions of the package.

7.1 Global Variables Used by the AtlasRep Package

For debugging purposes, AtlasRep functions print information depending on the info level of the info classes `InfoAtlasRep` (7.1.1), `InfoCMeatAxe` (7.1.2), and `InfoBBox` (7.1.3) (cf. (**Reference: Info Functions**)).

The info level of an info class can be changed using `SetInfoLevel` (**Reference: InfoLevel**). For example, the info level of `InfoAtlasRep` (7.1.1) can be set to the nonnegative integer n using `SetInfoLevel( InfoAtlasRep, `n` )`.

7.1.1 InfoAtlasRep

▷ `InfoAtlasRep` (info class)

If the info level of `InfoAtlasRep` is at least 1 then information about `fail` results of AtlasRep functions is printed. If the info level is at least 2 then also information about calls to external programs is printed. The default level is 0, no information is printed on this level.

7.1.2 InfoCMeatAxe

▷ `InfoCMeatAxe` (info class)

If the info level of InfoCMeatAxe is at least 1 then information about fail results of C-MeatAxe functions (see Section 7.3) is printed. The default level is zero, no information is printed on this level.

7.1.3 InfoBBox

▷ InfoBBox (info class)

If the info level of InfoBBox is at least 1 then information about fail results of functions dealing with black box programs (see Section 6.2) is printed. The default level is 0, no information is printed on this level.

7.1.4 AGR

▷ AGR (global variable)

is a record whose components are functions and data that are used by the high level interface functions. Some of the components are documented, see for example the index of the package manual.

7.1.5 AtlasOfGroupRepresentationsInfo

▷ AtlasOfGroupRepresentationsInfo (global variable)

This is a record that is defined in the file gap/types.g of the package, with the following components.

GAPnames
: a list of pairs, each containing the GAP name and the ATLAS-file name of a group, see Section 3.2,

notified
: a list used for administrating extensions of the database (see Chapter 5); the value is changed by AtlasOfGroupRepresentationsNotifyData (5.1.1) and AtlasOfGroupRepresentationsForgetData (5.1.2),

characterinfo, permrepinfo, ringinfo
: additional information about representations, concerning the afforded characters, the point stabilizers of permutation representations, and the rings of definition of matrix representations; this information is used by DisplayAtlasInfo (3.5.1),

TableOfContents
: a record with at most the components core, internal, local, merged, types, and the identifiers of database extensions. The value of the component types is set in AGR.DeclareDataType (7.5.1), and the values of the other components are created by AtlasOfGroupRepresentationsNotifyData (5.1.1).

accessFunctions
: a list of records, each describing how to access the data files, see Sections 4.2.5 and 7.2, and

7.2 How to Customize the Access to Data files

By default, locally available data files are stored in prescribed directories, and the files are exactly the text files that have been downloaded from appropriate places in the internet. However, a more flexible approach may be useful.

First, one may want to use *different file formats*, for example MeatAxe binary files may be provided parallel to MeatAxe text files. Second, one may want to use *a different directory structure*, for example the same structure as used on some server –this makes sense for example if a local mirror of a server is available, because then one can read the server files directly, without transferring/copying them to another directory.

In order to achieve this (and perhaps more), we admit to customize the meaning of the following three access steps.

Are the required data locally available?
: There may be different file formats available, such as text or binary files, and it may happen that the data are available in one file or are distributed to several files.

How can a file be made locally available?
: A different remote file may be fetched, or some postprocessing may be required.

How is the data of a file accessed by GAP?
: A different function may be needed to evaluate the file contents.

For creating an overview of the locally available data, the first of these steps must be available independent of actually accessing the file in question. For updating the local copy of the server data, the second of the above steps must be available independent of the third one. Therefore, the package provides the possibility to extend the default behaviour by adding new records to the `accessFunctions` component of `AtlasOfGroupRepresentationsInfo` (7.1.5). The relevant record components are as follows.

`description`
: This must be a short string that describes for which kinds of files the functions in the current record are intended, which file formats are supported etc. The value is used as key in the user preference `FileAccessFunctions`, see Section 4.2.5.

`location(` *files*,*type* `)`
: Let *files* be a list of pairs `[ dirname, filename ]`, and *type* be the data type (see `AGR.DeclareDataType` (7.5.1)) to which the files belong. This function must return either the absolute paths where the mechanism implemented by the current record expects the local version of the given files, or `fail` if this function does not feel responsible for these files.

  The files are regarded as not locally available if all installed `location` functions return either `fail` or paths of nonexisting files, in the sense of `IsExistingFile` (**Reference: IsExistingFile**).

`fetch(` *filepath*, *filename*,*dirname*,*type* `)`
: This function is called if a file is not locally available and if the `location` function in the current record has returned a list of paths. The argument *type* must be the same as for the `location` function, and *filepath* and *filename* must be strings (*not* lists of strings).

The return value must be `true` if the function succeeded with making the file locally available (including postprocessing if applicable), a string with the contents of the data file if the remote data were directly loaded into the GAP session (if no local caching is possible), and `false` otherwise.

`contents(` *files*,*type*,*filepaths* `)`

This function is called when the `location` function in the current record has returned the path(s) *filepath*, and if either these are paths of existing files or the `fetch` function in the current record has been called for these paths, and the return value was `true`. The first three arguments must be the same as for the `location` function.

The return value must be the contents of the file(s), in the sense that the GAP matrix, matrix list, permutation, permutation list, or program described by the file(s) is returned. This means that besides reading the file(s) via the appropriate function, interpreting the contents may be necessary.

In `AGR.FileContents` (7.6.2), those records in the `accessFunctions` component of `AtlasOfGroupRepresentationsInfo` (7.1.5) are considered –in reversed order– whose `description` component occurs in the user preference `FileAccessFunctions`, see Section 4.2.5.

7.3 Reading and Writing MeatAxe Format Files

7.3.1 ScanMeatAxeFile

▷ `ScanMeatAxeFile(`*filename[, q][, "string"]*`)` (function)

Returns: the matrix or list of permutations stored in the file or encoded by the string.

Let *filename* be the name of a GAP readable file (see (**Reference: Filename**)) that contains a matrix or a permutation or a list of permutations in MeatAxe text format (see the section about the program `zcv` in the `C`-MeatAxe documentation [Rin]), and let *q* be a prime power. `ScanMeatAxeFile` returns the corresponding GAP matrix or list of permutations, respectively.

If the file contains a matrix then the way how it is read by `ScanMeatAxeFile` depends on the value of the user preference `HowToReadMeatAxeTextFiles`, see Section 4.2.7.

If the parameter *q* is given then the result matrix is represented over the field with *q* elements, the default for *q* is the field size stored in the file.

If the file contains a list of permutations then it is read with `StringFile` (**GAPDoc: StringFile**); the parameter *q*, if given, is ignored in this case.

If the string `"string"` is entered as the third argument then the first argument must be a string as obtained by reading a file in MeatAxe text format as a text stream (see `InputTextFile` (**Reference: InputTextFile**)). Also in this case, `ScanMeatAxeFile` returns the corresponding GAP matrix or list of permutations, respectively.

7.3.2 MeatAxeString

▷ `MeatAxeString(`*mat, q*`)` (operation)

▷ `MeatAxeString(`*perms, degree*`)` (operation)

▷ `MeatAxeString(`*perm, q, dims*`)` (operation)

▷ MeatAxeString(*intmat*) (operation)

Returns: a string encoding the GAP objects given as input in C-MeatAxe text format, see [Rin].

In the first form, for a matrix *mat* whose entries lie in the finite field with *q* elements, MeatAxeString returns a string that encodes *mat* as a matrix over GF(*q*).

In the second form, for a nonempty list *perms* of permutations that move only points up to the positive integer *degree*, MeatAxeString returns a string that encodes *perms* as permutations of degree *degree*.

In the third form, for a permutation *perm* with largest moved point *n*, say, a prime power *q*, and a list *dims* of length two containing two positive integers larger than or equal to *n*, MeatAxeString returns a string that encodes *perm* as a matrix over GF(*q*), of dimensions *dims*, whose first *n* rows and columns describe the permutation matrix corresponding to *perm*, and the remaining rows and columns are zero.

In the fourth form, for a matrix *intmat* of integers, MeatAxeString returns a string that encodes *intmat* as an integer matrix.

When strings are printed to files using PrintTo (**Reference: PrintTo**) or AppendTo (**Reference: AppendTo**) then line breaks are inserted whenever lines exceed the number of characters given by the second entry of the list returned by SizeScreen (**Reference: SizeScreen**), see (**Reference: Operations for Output Streams**). This behaviour is not desirable for creating data files. So the recommended functions for printing the result of MeatAxeString to a file are FileString (**GAPDoc: FileString**) and WriteAll (**Reference: WriteAll**).

Example

```
gap> mat:= [ [ 1, -1 ], [ 0, 1 ] ] * Z(3)^0;;
gap> str:= MeatAxeString( mat, 3 );
"1 3 2 2\n12\n01\n"
gap> mat = ScanMeatAxeFile( str, "string" );
true
gap> str:= MeatAxeString( mat, 9 );
"1 9 2 2\n12\n01\n"
gap> mat = ScanMeatAxeFile( str, "string" );
true
gap> perms:= [ (1,2,3)(5,6) ];;
gap> str:= MeatAxeString( perms, 6 );
"12 1 6 1\n2\n3\n1\n4\n6\n5\n"
gap> perms = ScanMeatAxeFile( str, "string" );
true
gap> str:= MeatAxeString( perms, 8 );
"12 1 8 1\n2\n3\n1\n4\n6\n5\n7\n8\n"
gap> perms = ScanMeatAxeFile( str, "string" );
true
```

Note that the output of MeatAxeString in the case of permutation matrices depends on the user preference WriteMeatAxeFilesOfMode2.

Example

```
gap> perm:= (1,2,4);;
gap> str:= MeatAxeString( perm, 3, [ 5, 6 ] );
"2 3 5 6\n2\n4\n3\n1\n5\n"
gap> mat:= ScanMeatAxeFile( str, "string" );;  Print( mat, "\n" );
[ [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
```

```
    [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ],
    [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
    [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ] ]
gap> pref:= UserPreference( "AtlasRep", "WriteMeatAxeFilesOfMode2" );;
gap> SetUserPreference( "AtlasRep", "WriteMeatAxeFilesOfMode2", true );
gap> MeatAxeString( mat, 3 ) = str;
true
gap> SetUserPreference( "AtlasRep", "WriteMeatAxeFilesOfMode2", false );
gap> MeatAxeString( mat, 3 );
"1 3 5 6\n010000\n000100\n001000\n100000\n000010\n"
gap> SetUserPreference( "AtlasRep", "WriteMeatAxeFilesOfMode2", pref );
```

7.3.3 FFList

▷ `FFList(F)` (function)

Returns: a list of elements in the given finite field.

▷ `FFLists` (global variable)

`FFList` is a utility program for the conversion of vectors and matrices from MeatAxe format to GAP format and vice versa. It is used by `ScanMeatAxeFile` (7.3.1) and `MeatAxeString` (7.3.2).

For a finite field `F`, `FFList` returns a list l giving the correspondence between the MeatAxe numbering and the GAP numbering of the elements in `F`.

The element of `F` corresponding to MeatAxe number n is $l[n+1]$, and the MeatAxe number of the field element z is `Position( l,z ) - 1`.

The global variable `FFLists` is used to store the information about `F` once it has been computed.

Example

```
gap> FFList( GF(4) );
[ 0*Z(2), Z(2)^0, Z(2^2), Z(2^2)^2 ]
gap> IsBound( FFLists[4] );
true
```

The MeatAxe defines the bijection between the elements in the field with $q = p^d$ elements and the set $\{0,1,\ldots,q-1\}$ of integers by assigning the field element $\sum_{i=0}^{d-1} c_i z^i$ to the integer $\sum_{i=0}^{d-1} c_i p^i$, where the c_i are in the set $\{0,1,\ldots,p-1\}$ and z is the primitive root of the field with q elements that corresponds to the residue class of the indeterminate, modulo the ideal spanned by the Conway polynomial of degree d over the field with p elements.

The finite fields introduced by the StandardFF package [Lüb21] are supported by `FFList` and `FFLists`, in the sense that the bijection defined by `StandardIsomorphismGF` (StandardIsomorphismGF???) is applied automatically when `F` is a field in the filter `IsStandardFiniteField` (IsStandardFiniteField???).

7.3.4 CMtxBinaryFFMatOrPerm

▷ `CMtxBinaryFFMatOrPerm(elm, def, outfile[, base])` (function)

Let the pair `(elm,def)` be either of the form (M,q) where M is a matrix over a finite field F, say, with $q \leq 256$ elements, or of the form (π,n) where π is a permutation with largest moved point at most n. Let `outfile` be a string. `CMtxBinaryFFMatOrPerm` writes the C-MeatAxe binary format of

M, viewed as a matrix over F, or of π, viewed as a permutation on the points up to n, to the file with name *outfile*.

In the case of a permutation π, the optional argument *base* prescribes whether the binary file contains the points from 0 to *deg*-1 (*base*$=0$, supported by version 2.4 of the C-MeatAxe) or the points from 1 to *deg* (*base*$=1$, supported by older versions of the C-MeatAxe). The default for *base* is given by the value of the user preference BaseOfMeatAxePermutation, see Section 4.2.10.

(The binary format is described in the C-MeatAxe manual [Rin].)

Example

```
gap> tmpdir:= DirectoryTemporary();;
gap> mat:= Filename( tmpdir, "mat" );;
gap> q:= 4;;
gap> mats:= GeneratorsOfGroup( GL(10,q) );;
gap> CMtxBinaryFFMatOrPerm( mats[1], q, Concatenation( mat, "1" ) );
gap> CMtxBinaryFFMatOrPerm( mats[2], q, Concatenation( mat, "2" ) );
gap> prm:= Filename( tmpdir, "prm" );;
gap> n:= 200;;
gap> perms:= GeneratorsOfGroup( SymmetricGroup( n ) );;
gap> CMtxBinaryFFMatOrPerm( perms[1], n, Concatenation( prm, "1" ) );
gap> CMtxBinaryFFMatOrPerm( perms[2], n, Concatenation( prm, "2" ) );
gap> CMtxBinaryFFMatOrPerm( perms[1], n, Concatenation( prm, "1a" ), 0 );
gap> CMtxBinaryFFMatOrPerm( perms[2], n, Concatenation( prm, "2b" ), 1 );
```

7.3.5 FFMatOrPermCMtxBinary

▷ FFMatOrPermCMtxBinary(*fname*) (function)

Returns: the matrix or permutation stored in the file.

Let *fname* be the name of a file that contains the C-MeatAxe binary format of a matrix over a finite field or of a permutation, as is described in [Rin]. FFMatOrPermCMtxBinary returns the corresponding GAP matrix or permutation.

Example

```
gap> FFMatOrPermCMtxBinary( Concatenation( mat, "1" ) ) = mats[1];
true
gap> FFMatOrPermCMtxBinary( Concatenation( mat, "2" ) ) = mats[2];
true
gap> FFMatOrPermCMtxBinary( Concatenation( prm, "1" ) ) = perms[1];
true
gap> FFMatOrPermCMtxBinary( Concatenation( prm, "2" ) ) = perms[2];
true
gap> FFMatOrPermCMtxBinary( Concatenation( prm, "1a" ) ) = perms[1];
true
gap> FFMatOrPermCMtxBinary( Concatenation( prm, "2b" ) ) = perms[2];
true
```

7.4 Reading and Writing ATLAS Straight Line Programs

7.4.1 ScanStraightLineProgram

▷ ScanStraightLineProgram(*filename[, "string"]*) (function)

Returns: a record containing the straight line program, or fail.

Let `filename` be the name of a file that contains a straight line program in the sense that it consists only of lines in the following form.

`#`*anything*
: lines starting with a hash sign # are ignored,

`echo` *anything*
: lines starting with `echo` are ignored for the `program` component of the result record (see below), they are used to set up the bijection between the labels used in the program and conjugacy class names in the case that the program computes dedicated class representatives,

`inp` *n*
: means that there are *n* inputs, referred to via the labels `1`, `2`, ..., *n*,

`inp` *k a*1 *a*2 **...** *ak*
: means that the next *k* inputs are referred to via the labels *a*1, *a*2, ..., *ak*,

`cjr` *a b*
: means that *a* is replaced by *b*`^(-1) * a * b`,

`cj` *a b c*
: means that *c* is defined as *b*`^(-1) * a * b`,

`com` *a b c*
: means that *c* is defined as *a*`^(-1) * b^`(-1) `* a * b`,

`iv` *a b*
: means that *b* is defined as *a*`^(-1)`,

`mu` *a b c*
: means that *c* is defined as *a* `*` *b*,

`pwr` *a b c*
: means that *c* is defined as *b*`^`*a*,

`cp` *a b*
: means that *b* is defined as a copy of *a*,

`oup` *l*
: means that there are *l* outputs, stored in the labels `1`, `2`, ..., *l*, and

`oup` *l b*1 *b*2 **...** *bl*
: means that the next *l* outputs are stored in the labels *b*1, *b*2, ... *bl*.

Each of the labels *a*, *b*, *c* can be any nonempty sequence of digits and alphabet characters, except that the first argument of `pwr` must denote an integer.

If the `inp` or `oup` statements are missing then the input or output, respectively, is assumed to be given by the labels `1` and `2`. There can be multiple `inp` lines at the beginning of the program and multiple `oup` lines at the end of the program. Only the first `inp` or `oup` line may omit the names of the elements. For example, an empty file `filename` or an empty string `string` represent a straight line program with two inputs that are returned as outputs.

No command except cjr may overwrite its own input. For example, the line mu a b a is not legal. (This is not checked.)

ScanStraightLineProgram returns a record containing as the value of its component program the corresponding GAP straight line program (see IsStraightLineProgram (**Reference: IsStraightLineProgram**)) if the input string satisfies the syntax rules stated above, and returns fail otherwise. In the latter case, information about the first corrupted line of the program is printed if the info level of InfoCMeatAxe (7.1.2) is at least 1.

If the string "string" is entered as the second argument then the first argument must be a string as obtained by reading a file in MeatAxe text format as a text stream (see InputTextFile (**Reference: InputTextFile**)). Also in this case, ScanStraightLineProgram returns either a record with the corresponding GAP straight line program or fail.

If the input describes a straight line program that computes certain class representatives of the group in question then the result record also contains the component outputs. Its value is a list of strings, the entry at position *i* denoting the name of the class in which the *i* output of the straight line program lies; see Section 3.4 for the definition of the class names that occur.

Such straight line programs must end with a sequence of output specifications of the following form.

Example

```
echo "Classes 1A 2A 3A 5A 5B"
oup 5 3 1 2 4 5
```

This example means that the list of outputs of the program contains elements of the classes 1A, 2A, 3A, 5A, and 5B (in this order), and that inside the program, these elements are referred to by the five names 3, 1, 2, 4, and 5.

7.4.2 AtlasStringOfProgram

▷ AtlasStringOfProgram(*prog[, outputnames]*) (function)

▷ AtlasStringOfProgram(*prog, "mtx"*) (function)

Returns: a string encoding the straight line program/decision in the format used in ATLAS files.

For a straight line program or straight line decision *prog* (see IsStraightLineProgram (**Reference: IsStraightLineProgram**) and IsStraightLineDecision (6.1.1)), this function returns a string describing the input format of an equivalent straight line program or straight line decision as used in the data files, that is, the lines are of the form described in ScanStraightLineProgram (7.4.1).

A list of strings that is given as the optional second argument *outputnames* is interpreted as the class names corresponding to the outputs; this argument has the effect that appropriate echo statements appear in the result string.

If the string "mtx" is given as the second argument then the result has the format used in the C-MeatAxe (see [Rin]) rather than the format described for ScanStraightLineProgram (7.4.1). (Note that the C-MeatAxe format does not make sense if the argument *outputnames* is given, and that this format does not support inp and oup statements.)

The argument *prog* must not be a black box program (see IsBBoxProgram (6.2.1)).

Example

```
gap> str:= "inp 2\nmu 1 2 3\nmu 3 1 2\niv 2 1\noup 2 1 2";;
gap> prg:= ScanStraightLineProgram( str, "string" );
rec( program := <straight line program> )
```

```
gap> prg:= prg.program;;
gap> Display( prg );
# input:
r:= [ g1, g2 ];
# program:
r[3]:= r[1]*r[2];
r[2]:= r[3]*r[1];
r[1]:= r[2]^-1;
# return values:
[ r[1], r[2] ]
gap> StringOfResultOfStraightLineProgram( prg, [ "a", "b" ] );
"[ (aba)^-1, aba ]"
gap> AtlasStringOfProgram( prg );
"inp 2\nmu 1 2 3\nmu 3 1 2\niv 2 1\noup 2\n"
gap> prg:= StraightLineProgram( "(a^2b^3)^-1", [ "a", "b" ] );
<straight line program>
gap> Print( AtlasStringOfProgram( prg ) );
inp 2
pwr 2 1 4
pwr 3 2 5
mu 4 5 3
iv 3 4
oup 1 4
gap> prg:= StraightLineProgram( [ [2,3], [ [3,1,1,4], [1,2,3,1] ] ], 2 );
<straight line program>
gap> Print( AtlasStringOfProgram( prg ) );
inp 2
pwr 3 2 3
pwr 4 1 5
mu 3 5 4
pwr 2 1 6
mu 6 3 5
oup 2 4 5
gap> Print( AtlasStringOfProgram( prg, "mtx" ) );
# inputs are expected in 1 2
zsm pwr3 2 3
zsm pwr4 1 5
zmu 3 5 4
zsm pwr2 1 6
zmu 6 3 5
echo "outputs are in 4 5"
gap> str:= "inp 2\nchor 1 2\nchor 2 3\nmu 1 2 3\nchor 3 5";;
gap> prg:= ScanStraightLineDecision( str );;
gap> AtlasStringOfProgram( prg.program );
"inp 2\nchor 1 2\nchor 2 3\nmu 1 2 3\nchor 3 5\n"
```

7.5 Data Types Used in the AtlasRep Package

Each representation or program that is administrated by the AtlasRep package belongs to a unique *data type*. Informally, examples of data types are “permutation representation”, “matrix representation over the integers”, or “straight line program for computing class representatives”.

The idea is that for each data type, there can be

- a column of its own in the output produced by DisplayAtlasInfo (3.5.1) when called without arguments or with only argument a list of group names,
- a line format of its own for the output produced by DisplayAtlasInfo (3.5.1) when called with first argument a group name,
- an input format of its own for AtlasProgram (3.5.4),
- an input format of its own for OneAtlasGeneratingSetInfo (3.5.6), and
- specific tests for the data of this data type; these functions are used by the global tests described in Section 7.9.

Formally, a data type is defined by a record whose components are used by the interface functions. The details are described in the following.

7.5.1 AGR.DeclareDataType

▷ AGR.DeclareDataType(*kind*, *name*, *record*) (function)

Let *kind* be one of the strings "rep" or "prg", and *record* be a record. If *kind* is "rep" then AGR.DeclareDataType declares a new data type of representations, if *kind* is "prg" then it declares a new data type of programs. The string *name* is the name of the type, for example "perm", "matff", or "classes". AtlasRep stores the data for each group internally in a record whose component *name* holds the list of the data about the type with this name.

Mandatory components of *record* are

FilenameFormat
: This defines the format of the filenames containing data of the type in question. The value must be a list that can be used as the second argument of AGR.ParseFilenameFormat (7.6.1), such that only filenames of the type in question match. (It is not checked whether this "detection function" matches exactly one type, so declaring a new type needs care.)

AddFileInfo
: This defines the information stored in the table of contents for the data of the type. The value must be a function that takes three arguments (the current list of data for the type and the given group, a list returned by AGR.ParseFilenameFormat (7.6.1) for the given type, and a filename). This function adds the necessary parts of the data entry to the list, and returns true if the data belongs to the type, otherwise false is returned; note that the latter case occurs if the filename matches the format description but additional conditions on the parts of the name are not satisfied (for example integer parts may be required to be positive or prime powers).

ReadAndInterpretDefault
: This is the function that does the work for the default contents value of the accessFunctions component of AtlasOfGroupRepresentationsInfo (7.1.5), see Section 7.2. This function must take a path and return the GAP object given by this file.

`AddDescribingComponents` **(for** `rep` **only)**
: This function takes two arguments, a record (that will be returned by `AtlasGenerators` (3.5.3), `OneAtlasGeneratingSetInfo` (3.5.6), or `AllAtlasGeneratingSetInfos` (3.5.7)) and the type record *`record`*. It sets the components `p`, `dim`, `id`, and `ring` that are promised for return values of the abovementioned three functions.

`DisplayGroup` **(for** `rep` **only)**
: This defines the format of the lines printed by `DisplayAtlasInfo` (3.5.1) for a given group. The value must be a function that takes a list as returned by the function given in the component `AddFileInfo`, and returns the string to be printed for the representation in question.

Optional components of *`record`* are

`DisplayOverviewInfo`
: This is used to introduce a new column in the output of `DisplayAtlasInfo` (3.5.1) when this is called without arguments or with a list of group names as its only argument. The value must be a list of length three, containing at its first position a string used as the header of the column, at its second position one of the strings `"r"` or `"l"`, denoting right or left aligned column entries, and at its third position a function that takes two arguments (a list of tables of contents of the AtlasRep package and a group name), and returns a list of length two, containing the string to be printed as the column value and `true` or `false`, depending on whether private data is involved or not. (The default is `fail`, indicating that no new column shall be printed.)

`DisplayPRG` **(for** `prg` **only)**
: This is used in `DisplayAtlasInfo` (3.5.1) for ATLAS programs. The value must be a function that takes four arguments (a list of tables of contents to examine, a list containing the GAP name and the ATLAS name of the given group, a list of integers or `true` for the required standardization, and a list of all available standardizations), and returns the list of lines (strings) to be printed as the information about the available programs of the current type and for the given group. (The default is to return an empty list.)

`AccessGroupCondition` **(for** `rep` **only)**
: This is used in `DisplayAtlasInfo` (3.5.1) and `OneAtlasGeneratingSetInfo` (3.5.6). The value must be a function that takes two arguments (a list as returned by `OneAtlasGeneratingSetInfo` (3.5.6), and a list of conditions), and returns `true` or `false`, depending on whether the first argument satisfies the conditions. (The default value is `ReturnFalse` (**Reference: ReturnFalse**).)

  The function must support conditions such as `[ IsPermGroup, true ]` and `[ NrMovedPoints, [ 5, 6 ] ]`, in general a list of functions followed by a prescribed value, a list of prescribed values, another (unary) function, or the string `"minimal"`. For an overview of the interesting functions, see `DisplayAtlasInfo` (3.5.1).

`AccessPRG` **(for** `prg` **only)**
: This is used in `AtlasProgram` (3.5.4). The value must be a function that takes four arguments (the current table of contents, the group name, an integer or a list of integers or `true` for the required standardization, and a list of conditions given by the optional arguments of `AtlasProgram` (3.5.4)), and returns either `fail` or a list that together with the group name forms the identifier of a program that matches the conditions. (The default value is `ReturnFail` (**Reference: ReturnFail**).)

`AtlasProgram` **(for** `prg` **only)**
: This is used in `AtlasProgram` (3.5.4) to create the result value from the identifier. (The default value is `AtlasProgramDefault`, which works whenever the second entry of the identifier is the filename; this is not the case for example if the program is the composition of several programs.)

`AtlasProgramInfo` **(for** `prg` **only)**
: This is used in `AtlasProgramInfo` (3.5.5) to create the result value from the identifier. (The default value is `AtlasProgramDefault`.)

`TOCEntryString`
: This is used in `StringOfAtlasTableOfContents` (5.1.3). The value must be a function that takes two or three arguments (the name *name* of the type, a list as returned by `AGR.ParseFilenameFormat` (7.6.1), and optionally a string that indicates the "remote" format) and returns a string that describes the appropriate data format. (The default value is `TOCEntryStringDefault`.)

`PostprocessFileInfo`
: This is used in the construction of a table of contents for testing or rearranging the data of the current table of contents. The value must be a function that takes two arguments, the table of contents record and the record in it that belongs to one fixed group. (The default function does nothing.)

`SortTOCEntries`
: This is used in the construction of a table of contents for sorting the entries after they have been added and after the value of the component `PostprocessFileInfo` has been called. The value must be a function that takes a list as returned by `AGR.ParseFilenameFormat` (7.6.1), and returns the sorting key. (There is no default value, which means that no sorting is needed.)

`TestFileHeaders` **(for** `rep` **only)**
: This is used in the function `AGR.Test.FileHeaders`. The value must be a function that takes the same four arguments as `AGR.FileContents` (7.6.2), except that the third argument is a list as returned by `AGR.ParseFilenameFormat` (7.6.1). (The default value is `ReturnTrue` (**Reference: ReturnTrue**).)

`TestFiles` **(for** `rep` **only)**
: This is used in the function `AGR.Test.Files`. The format of the value and the default are the same as for the component `TestFileHeaders`.

`TestWords` **(for** `prg` **only)**
: This is used in the function `AGR.Test.Words`. The value must be a function that takes five arguments where the first four are the same arguments as for `AGR.FileContents` (7.6.2), except that the fifth argument is `true` or `false`, indicating verbose mode or not.

7.6 Filenames Used in the AtlasRep Package

AtlasRep expects that the filename of each data file describes the contents of the file. This section lists the definitions of the supported structures of filenames.

Each filename consists of two parts, separated by a minus sign `-`. The first part is always of the form *groupname*`G`*i*, where the integer *i* denotes the *i*-th set of standard generators for the group *G*, say,

with ATLAS-file name *groupname* (see 3.2). The translations of the name *groupname* to the name(s) used within GAP is given by the component `GAPnames` of `AtlasOfGroupRepresentationsInfo` (7.1.5).

The names of files that contain straight line programs or straight line decisions have one of the following forms. In each of these cases, the suffix `W`*n* means that *n* is the version number of the program.

groupname`G`*i*`-cycW`*n*

In this case, the file contains a straight line program that returns a list of representatives of generators of maximally cyclic subgroups of *G*. An example is `Co1G1-cycW1`.

groupname`G`*i*`-cclsW`*n*

In this case, the file contains a straight line program that returns a list of conjugacy class representatives of *G*. An example is `RuG1-cclsW1`.

groupname`G`*i*`cycW`*n*`-cclsW`*m*

In this case, the file contains a straight line program that takes the return value of the program in the file *groupname*`G`*i*`-cycW`*n* (see above), and returns a list of conjugacy class representatives of *G*. An example is `M11G1cycW1-cclsW1`.

groupname`G`*i*`-max`*k*`W`*n*

In this case, the file contains a straight line program that takes generators of *G* w. r. t. the *i*-th set of standard generators, and returns a list of generators (in general *not* standard generators) for a subgroup *U* in the *k*-th class of maximal subgroups of *G*. An example is `J1G1-max7W1`.

groupname`G`*i*`max`*k*`W`*n*`-`*subgroupname*`G`*j*`W`*m*

In this case, the file contains a straight line program that takes the return value of the program in the file *groupname*`G`*i*`-max`*k*`W`*n* (see above), which are generators for a group *U*, say; *subgroupname* is a name for *U*, and the return value is a list of standard generators for *U*, w. r. t. the *j*-th set of standard generators. (Of course this implies that the groups in the *k*-th class of maximal subgroups of *G* are isomorphic to the group with name *subgroupname*.) An example is `J1G1max1W1-L211G1W1`; the first class of maximal subgroups of the Janko group J_1 consists of groups isomorphic to the linear group $L_2(11)$, for which standard generators are defined.

groupname`G`*i*`-a`*outname*`W`*n*

In this case, the file contains a straight line program that takes generators of *G* w. r. t. the *i*-th set of standard generators, and returns the list of their images under the outer automorphism α of *G* given by the name *outname*; if this name is empty then α is the unique nontrivial outer automorphism of *G*; if it is a positive integer *k* then α is a generator of the unique cyclic order *k* subgroup of the outer automorphism group of *G*; if it is of the form `2_1` or `2a`, `4_2` or `4b`, `3_3` or `3c` ... then α generates the cyclic group of automorphisms induced on *G* by $G.2_1$, $G.4_2$, $G.3_3$ ...; finally, if it is of the form *k*`p`*d*, with *k* one of the above forms and *d* an integer then *d* denotes the number of dashes appended to the automorphism described by *k*; if $d = 1$ then *d* can be omitted. Examples are `A5G1-aW1`, `L34G1-a2_1W1`, `U43G1-a2_3pW1`, and `O8p3G1-a2_2p5W1`; these file names describe the outer order 2 automorphism of A_5 (induced by the action of S_5) and the order 2 automorphisms of $L_3(4)$, $U_4(3)$, and $O_8^+(3)$ induced by the actions of $L_3(4).2_1$, $U_4(3).2_2'$, and $O_8^+(3).2_2''''$, respectively.

*groupname*G*i*-ker*factgroupname*W*n*
: In this case, the file contains a straight line program that takes generators of G w. r. t. the i-th set of standard generators, and returns generators of the kernel of an epimorphism that maps G to a group with ATLAS-file name $factgroupname$. An example is `2A5G1-kerA5W1`.

*groupname*G*i*-G*j*W*n*
: In this case, the file contains a straight line program that takes generators of G w. r. t. the i-th set of standard generators, and returns standard generators of G w. r. t. the j-th set of standard generators. An example is `L35G1-G2W1`.

*groupname*G*i*-check*n*
: In this case, the file contains a straight line decision that takes generators of G, and returns `true` if these generators are standard generators w. r. t. the i-th standardization, and `false` otherwise.

*groupname*G*i*-P*n*
: In this case, the file contains a straight line decision that takes some group elements, and returns `true` if these elements are standard generators for G, w. r. t. the i-th standardization, and `false` otherwise.

*groupname*G*i*-find*n*
: In this case, the file contains a black box program that takes a group, and returns (if it is successful) a set of standard generators for G, w. r. t. the i-th standardization.

*groupname*G*i*-X*descr*W*n*
: In this case, the file contains a straight line program that takes generators of G w. r. t. the i-th set of standard generators, and whose return value corresponds to *descr*. This format is used only in private extensions (see Chapter 5), such a script can be accessed with *descr* as the third argument of `AtlasProgram` (3.5.4).

The names of files that contain group generators have one of the following forms. In each of these cases, *id* is a (possibly empty) string that starts with a lowercase alphabet letter (see `IsLowerAlphaChar` (**Reference: IsLowerAlphaChar**)), and m is a nonnegative integer, meaning that the generators are written w. r. t. the m-th basis (the meaning is defined by the ATLAS developers).

*groupname*G*i*-f*q*r*dim*B*m*.m*nr*
: a file in MeatAxe text file format containing the nr-th generator of a matrix representation over the field with q elements, of dimension *dim*. An example is `S5G1-f2r4aB0.m1`.

*groupname*G*i*-p*n*B*m*.m*nr*
: a file in MeatAxe text file format containing the nr-th generator of a permutation representation on n points. An example is `M11G1-p11B0.m1`.

*groupname*G*i*-Ar*dim*B*m*.g
: a GAP readable file containing all generators of a matrix representation of dimension *dim* over an algebraic number field not specified further. An example is `A5G1-Ar3aB0.g`.

*groupname*G*i*-Zr*dim*B*m*.**g**
: a GAP readable file containing all generators of a matrix representation over the integers, of dimension *dim*. An example is `A5G1-Zr4B0.g`.

*groupname*Gi-Hr*dimid*Bm.g
: a GAP readable file containing all generators of a matrix representation over a quaternion algebra over an algebraic number field, of dimension *dim*. An example is 2A6G1-Hr2aB0.g.

*groupname*Gi-Znr*dimid*Bm.g
: a GAP readable file containing all generators of a matrix representation of dimension *dim* over the ring of integers mod *n*. An example is 2A8G1-Z4r4aB0.g.

7.6.1 AGR.ParseFilenameFormat

▷ AGR.ParseFilenameFormat(*string*, *format*) (function)

Returns: a list of strings and integers if *string* matches *format*, and fail otherwise.

Let *string* be a filename, and *format* be a list $[[c_1, c_2, \ldots, c_n], [f_1, f_2, \ldots, f_n]]$ such that each entry c_i is a list of strings and of functions that take a character as their argument and return true or false, and such that each entry f_i is a function for parsing a filename, such as the currently undocumented functions ParseForwards and ParseBackwards.

AGR.ParseFilenameFormat returns a list of strings and integers such that the concatenation of their String (**Reference: String**) values yields *string* if *string* matches *format*, and fail otherwise. Matching is defined as follows. Splitting *string* at each minus character (-) yields m parts $s_1, s_2, \ldots, s_m$. The string *string* matches *format* if s_i matches the conditions in c_i, for $1 \leq i \leq n$, in the sense that applying f_i to s_i and c_i yields a non-fail result.

Example

```
gap> format:= [ [ [ IsChar, "G", IsDigitChar ],
>                 [ "p", IsDigitChar, AGR.IsLowerAlphaOrDigitChar,
>                   "B", IsDigitChar, ".m", IsDigitChar ] ],
>               [ ParseBackwards, ParseForwards ] ];;
gap> AGR.ParseFilenameFormat( "A6G1-p10B0.m1", format );
[ "A6", "G", 1, "p", 10, "", "B", 0, ".m", 1 ]
gap> AGR.ParseFilenameFormat( "A6G1-p15aB0.m1", format );
[ "A6", "G", 1, "p", 15, "a", "B", 0, ".m", 1 ]
gap> AGR.ParseFilenameFormat( "A6G1-f2r16B0.m1", format );
fail
```

7.6.2 AGR.FileContents

▷ AGR.FileContents(*files*, *type*) (function)

Returns: the GAP object obtained from reading and interpreting the file(s) given by *files*.

Let *files* be a list of pairs of the form [dirname, filename], where dirname and filename are strings, and let *type* be a data type (see AGR.DeclareDataType (7.5.1)). Each dirname must be one of "datagens", "dataword", or the dirid value of a data extension (see AtlasOfGroupRepresentationsNotifyData (5.1.1)). If the contents of each of the files in question is accessible and their data belong to the data type type then AGR.FileContents returns the contents of the files; otherwise fail is returned.

Note that if some file is already stored in the *dirname* directory then AGR.FileContents does *not* check whether the relevant table of contents actually contains *filename*.

7.7 The record component identifier used by the AtlasRep Package

The functions AtlasGenerators (3.5.3), AtlasProgram (3.5.4), AtlasProgramInfo (3.5.5), OneAtlasGeneratingSetInfo (3.5.6), and AllAtlasGeneratingSetInfos (3.5.7) return records which have a component identifier. The value of this component describes the record in the sense that one can reconstruct the whole record from it, and the identifier value can be used as an input for AtlasGenerators (3.5.3), AtlasProgram (3.5.4), AtlasProgramInfo (3.5.5), AtlasGroup (3.5.8), and AtlasSubgroup (3.5.9).

The identifier component has the following format.

- For records describing representations, it is a list of the form [gapname, files, std, info].
- For records describing straight line programs and straight line decisions, it is a list of the form [gapname, files, std].

Here gapname is the GAP name of the group in question, files defines the data files, std is the standardization of its generators, and info is some information that depends on the type of the representation, for example the number of moved points in the case of a permutation representation.

The files entry has one of the following formats:

- a string, in the case that exactly one file is needed that does not belong to a private extension; an example of such an identifier value is ["J1", "J1G1-cycW1", 1]
- a list whose entries are strings (which refer to files from the core part of the database) and pairs of the form [tocid, file] (which refer to files from the extension given by tocid); examples of identifier values are ["A5", ["A5G1-p5B0.m1", "A5G1-p5B0.m2"], 1, 5], ["2.M12", [["mfer", "2M12G1-cclsW1"]], 1], ["2.M12", ["M12G1-max1W1", ["internal", "2M12G1-kerM12W1"]], 1], ["2.M12", [["mfer", "2M12G1-p24bB0.m1"], ["mfer", "2M12G1-p24bB0.m2"]], 1, 24].

Up to version 1.5 of the AtlasRep package, a different identifier format was used for files from extensions of the database. Namely, the first entry of the list was a pair [tocid, groupname], and the second entry was either a string or a list of strings. Note that with that old format, it was not possible to describe a combination of several files from different sources (core part and extension, or different extensions). The function AtlasRepIdentifier (7.7.1) can be used to convert between the two formats.

7.7.1 AtlasRepIdentifier

▷ AtlasRepIdentifier(*oldid*) (function)
▷ AtlasRepIdentifier(*id, "old"*) (function)

This function converts between the "old format" (the one used up to version 1.5.1 of the package) and the "new format" (the one used since version 2.0) of the identifier component of the records returned by AtlasRep functions. Note that the two formats differ only for identifier components that describe data from non-core parts of the database.

If the only argument is a list *oldid* that is an identifier in old format then the function returns the corresponding identifier in new format. If there are two arguments, a list *id* that is

an `identifier` in new format and the string *`"old"`*, then the function returns the corresponding `identifier` in old format if this is possible, and `fail` otherwise.

Example

```
gap> id:= [ "A5", [ "A5G1-p5B0.m1", "A5G1-p5B0.m2" ], 1, 5 ];;
gap> AtlasRepIdentifier( id ) = id;
true
gap> id:= [ "L2(8)", "L28G1-check1", 1, 1 ];;
gap> AtlasRepIdentifier( id ) = id;
true
gap> oldid:= [ [ "priv", "C4" ], [ "C4G1-p4B0.m1" ], 1, 4 ];;
gap> newid:= AtlasRepIdentifier( oldid );
[ "C4", [ [ "priv", "C4G1-p4B0.m1" ] ], 1, 4 ]
gap> oldid = AtlasRepIdentifier( newid, "old" );
true
gap> oldid:= [ [ "priv", "C4" ], "C4G1-max1W1", 1 ];;
gap> newid:= AtlasRepIdentifier( oldid );
[ "C4", [ [ "priv", "C4G1-max1W1" ] ], 1 ]
gap> oldid = AtlasRepIdentifier( newid, "old" );
true
gap> oldid:= [ [ "priv", "C4" ], "C4G1-Ar1aB0.g", 1, 1 ];;
gap> newid:= AtlasRepIdentifier( oldid );
[ "C4", [ [ "priv", "C4G1-Ar1aB0.g" ] ], 1, 1 ]
gap> oldid = AtlasRepIdentifier( newid, "old" );
true
gap> oldid:= [ [ "priv", "C4" ], "C4G1-XtestW1", 1 ];;
gap> newid:= AtlasRepIdentifier( oldid );
[ "C4", [ [ "priv", "C4G1-XtestW1" ] ], 1 ]
gap> oldid = AtlasRepIdentifier( newid, "old" );
true
gap> oldid:= [ [ "mfer", "2.M12" ],
>  [ "2M12G1-p264aB0.m1", "2M12G1-p264aB0.m2" ], 1, 264 ];;
gap> newid:= AtlasRepIdentifier( oldid );
[ "2.M12",
  [ [ "mfer", "2M12G1-p264aB0.m1" ], [ "mfer", "2M12G1-p264aB0.m2" ] ]
    , 1, 264 ]
gap> oldid = AtlasRepIdentifier( newid, "old" );
true
```

7.8 The Tables of Contents of the AtlasRep Package

The list of AtlasRep data is stored in several *tables of contents*, which are given essentially by JSON documents, one for the core data and one for each data extension in the sense of Chapter 5. The only exception are data extensions by locally available files in a given directory, where the contents of this directory itself describes the data in question. One can create such a JSON document for the contents of a given local data directory with the function `StringOfAtlasTableOfContents` (5.1.3).

Here are the administrational functions that are called when a data extension gets notified with `AtlasOfGroupRepresentationsNotifyData` (5.1.1). In each case, *gapname* and *atlasname* denote the GAP and ATLAS name of the group in question (see Section 3.2), and *dirid* denotes the identifier of the data extension.

The following functions define group names, available representations, and straight line programs.

`AGR.GNAN(` *gapname*, *atlasname*[, *dirid*] `)`

Called with two strings *gapname* (the GAP name of the group) and *atlasname* (the ATLAS name of the group), `AGR.GNAN` stores the information in the list `AtlasOfGroupRepresentationsInfo.GAPnames`, which defines the name mapping between the ATLAS names and GAP names of the groups.

An example of a valid call is `AGR.GNAN("A5.2","S5")`.

`AGR.TOC(` *typename*, *filename*, *crc*[, *dirid*] `)`

`AGR.TOC` notifies an entry to the `TableOfContents.(` *dirid* `)` component of `AtlasOfGroupRepresentationsInfo` (7.1.5). The string *typename* must be the name of the data type to which the entry belongs, the string *filename* must be the prefix of the data file(s), and *crc* must be a list that contains the checksums of the data files, which are either integers (see `CrcFile` (**Reference: CrcFile**)) or strings (see `HexSHA256`). In particular, the number of files that are described by the entry equals the length of *crc*.

The optional argument *dirid* is equal to the argument with the same name in the corresponding call of `AtlasOfGroupRepresentationsNotifyData` (5.1.1). If no *dirid* argument is given then the current value of `AGR.DIRID` is taken as the default; this value is set automatically before a `toc.json` file gets evaluated by `AtlasOfGroupRepresentationsNotifyData` (5.1.1), and is reset afterwards. If `AGR.DIRID` is not bound and *dirid* is not given then this function has no effect.

An example of a valid call is `AGR.TOC("perm","alt/A5/mtx/S5G1-p5B0.m", [-3581724,115937465])`.

The following functions add data about the groups and their standard generators. The function calls must be executed after the corresponding `AGR.GNAN` calls.

`AGR.GRS(` *gapname*, *size*[, *dirid*] `)`

The integer *size* is stored as the order of the group with GAP name *gapname*, in `AtlasOfGroupRepresentationsInfo.GAPnames`.

An example of a valid call is `AGR.GRS("A5.2",120)`.

`AGR.MXN(` *gapname*, *nrMaxes*[, *dirid*] `)`

The integer *nrMaxes* is stored as the number of classes of maximal subgroups of the group with GAP name *gapname*, in `AtlasOfGroupRepresentationsInfo.GAPnames`.

An example of a valid call is `AGR.MXN("A5.2",4)`.

`AGR.MXO(` *gapname*, *sizesMaxes*[, *dirid*] `)`

The list *sizesMaxes* of subgroup orders of the classes of maximal subgroups of the group with GAP name *gapname* (not necessarily dense, in non-increasing order) is stored in `AtlasOfGroupRepresentationsInfo.GAPnames`.

An example of a valid call is `AGR.MXO("A5.2",[60,24,20,12])`.

`AGR.MXS(` *gapname*, *structureMaxes*[, *dirid*] `)`

Called with the string The list *structureMaxes* of strings describing the structures of the maximal subgroups of the group with GAP name *gapname* (not necessarily dense), is stored in `AtlasOfGroupRepresentationsInfo.GAPnames`.

An example of a valid call is `AGR.MXS("A5.2",["A5","S4","5:4","S3x2"])`.

`AGR.STDCOMP(` *gapname*, *factorCompatibility*[, *dirid*] `)`

The list *factorCompatibility* (with entries the standardization of the group with GAP name *gapname* , the GAP name of a factor group, the standardization of this factor group, and `true` or `false`, indicating whether mapping the standard generators for *gapname* to those of *factgapname* defines an epimorphism) is stored in `AtlasOfGroupRepresentationsInfo.GAPnames`.

An example of a valid call is `AGR.STDCOMP("2.A5.2",[1,"A5.2",1,true])`.

The following functions add data about representations or straight line programs that are already known. The function calls must be executed after the corresponding `AGR.TOC` calls.

`AGR.RNG(` *repname*, *descr*[, *dirid*] `)`

Called with two strings *repname* (denoting the name of a file containing the generators of a matrix representation over a ring that is not determined by the filename) and *descr* (describing this ring *R*, say), `AGR.RNG` adds the triple [*repname*, *descr*, *R*] to the list stored in the `ringinfo` component of `AtlasOfGroupRepresentationsInfo` (7.1.5).

An example of a valid call is `AGR.RNG("A5G1-Ar3aBO","Field([Sqrt(5)])")`.

`AGR.TOCEXT(` *atlasname*, *std*, *maxnr*, *files*[, *dirid*] `)`

Called with *atlasname*, the positive integers *std* (the standardization) and *maxnr* (the number of the class of maximal subgroups), and the list *files* (of filenames of straight line programs for computing generators of the *maxnr*-th maximal subgroup, using a straight line program for a factor group plus perhaps some straight line program for computing kernel generators), `AGR.TOCEXT` stores the information in `AtlasOfGroupRepresentationsInfo.GAPnames`.

An example of a valid call is `AGR.TOCEXT("2A5",1,3,["A5G1-max3W1"])`.

`AGR.API(` *repname*, *info*[, *dirid*] `)`

Called with the string *repname* (denoting the name of a permutation representation) and the list *info* (describing the point stabilizer of this representation), `AGR.API` binds the component *repname* of the record `AtlasOfGroupRepresentationsInfo.permrepinfo` to a record that describes the contents of *info*.

info has the following entries.

- At position 1, the transitivity is stored.
- If the transitivity is zero then *info* has length two, and the second entry is the list of orbit lengths.
- If the transitivity is positive then *info* has length four or five, and the second entry is the rank of the action.
- If the transitivity is positive then the third entry is one of the strings `"prim"`, `"imprim"`, denoting primitivity or not.
- If the transitivity is positive then the fourth entry is either the string `"???"` or a string that describes the structure of the point stabilizer. If the third entry is `"imprim"` then this description consists of a subgroup part and a maximal subgroup part, separated by `" < "`.
- If the third entry is `"prim"` then the fifth entry is either the string `"???"` or the number of the class of maximal subgroups that are the point stabilizers.

An example of a valid call is `AGR.API("A5G1-p5B0",[3,2,"prim","A4",1])`.

`AGR.CHAR(` *gapname*, *repname*, *char*, *pos*[, *charname*[, *dirid*]] `)`

Called with the strings *gapname* and *repname* (denoting the name of the representation), the integer *char* (the characteristic of the representation), and *pos* (the position or list of positions of the irreducible constituent(s)), `AGR.CHAR` stores the information in `AtlasOfGroupRepresentationsInfo.characterinfo`.

A string describing the character can be entered as *charname*.

If *dirid* is given but no *charname* is known then one can enter `fail` as the fifth argument.

An example of a valid call is `AGR.CHAR("M11","M11G1-p11B0",0,[1,2],"1a+10a")`.

7.9 Sanity Checks for the AtlasRep Package

The file `tst/testall.g` of the package contains `Test` (**Reference: Test**) statements for checking whether the AtlasRep functions behave as documented. One can run these tests by calling `ReadPackage( "AtlasRep", "tst/testall.g" )`. The examples in the package manual form a part of the tests, they are collected in the file `tst/docxpl.tst` of the package.

The remainder of this section deals with consistency checks of the data. The tests described in Section 7.9.1 can be used for data from any extension of the database (see Chapter 5), Section 7.9.2 lists tests which apply only to the core part of the database.

All these tests apply only to *locally* available files (see Section 7.8), no files are downloaded during the tests. Thus the required space and time for running these tests depend on the amount of locally available data.

Some of the tests compute and verify additional data, such as information about point stabilizers of permutation representations. In these cases, output lines starting with `#E` are error messages that point to inconsistencies, whereas output lines starting with `#I` inform about data that have been computed and were not yet stored, or about stored data that were not verified. These tests are experimental in the sense that they involve several heuristics. Depending on the data to which they are applied, it may happen that the tests run out of space or do not finish in acceptable time. Please inform the package maintainer if you run into such problems.

7.9.1 Sanity Checks for a Table of Contents

The following tests can be used to check the data that belong to a given part of the database (core data or extension). Each of these tests is given by a function with optional argument *tocid*, the identifying string that had been entered as the second argument of `AtlasOfGroupRepresentationsNotifyData` (5.1.1). The contents of the core part can be checked by entering `"core"`, which is also the default for *tocid*. The function returns `false` if an error occurs, otherwise `true`. Currently the following tests of this kind are available. (For some of them, the global option `TryToExtendData` can be entered in order to try the computation of not yet stored data.)

`AGR.Test.GroupOrders()`

checks whether the group orders stored in the `GAPnames` component of `AtlasOfGroupRepresentationsInfo` (7.1.5) coincide with the group orders computed from an ATLAS permutation representation of degree up to `AGR.Test.MaxTestDegree`, from the available character table or table of marks with the given name, or from the structure of the

name, in the sense that splitting the name at the first dot (`.`) or colon (`:`) and applying the same criteria to derive the group order from the two parts may yield enough information.

`AGR.Test.Words( [`*tocid*`] )`
processes the straight line programs that belong to *tocid*, using the function stored in the `TestWords` component of the data type in question.

The straight line programs for the cases listed in `AGR.Test.HardCases.TestWords` are omitted.

`AGR.Test.ClassScripts( [`*tocid*`] )`
checks whether the straight line programs that belong to *tocid* and that compute representatives of certain conjugacy classes are consistent with information stored on the GAP character table of the group in question, in the sense that the given class names really occur in the character table and that the element orders and centralizer orders for the classes are correct.

`AGR.Test.CycToCcls( [`*tocid*`][:TryToExtendData] )`
checks whether all straight line programs that belong to *tocid* and that compute class representatives from representatives of cyclic subgroups possess a corresponding straight line program (*anywhere* in the database) for computing representatives of cyclic subgroups.

`AGR.Test.FileHeaders( [`*tocid*`] )`
checks whether the MeatAxe text files that belong to *tocid* have a header line that is consistent with the filename, and whether the contents of all GAP format data files that belong to *tocid* is consistent with the filename.

`AGR.Test.Files( [`*tocid*`] )`
checks whether the MeatAxe text files that belong to *tocid* can be read with `ScanMeatAxeFile` (7.3.1) such that the result is not `fail`. The function does not check whether the first line of a MeatAxe text file is consistent with the filename, since this can be tested with `AGR.Test.FileHeaders`.

`AGR.Test.BinaryFormat( [`*tocid*`] )`
checks whether all MeatAxe text files that belong to *tocid* satisfy that applying first `CMtxBinaryFFMatOrPerm` (7.3.4) and then `FFMatOrPermCMtxBinary` (7.3.5) yields the same object.

`AGR.Test.Primitivity( [`*tocid*`][:TryToExtendData] )`
checks the stored primitivity information for the permutation representations that belong to *tocid*. That is, the number of orbits, in case of a transitive action the transitivity, the rank, the information about the point stabilizers are computed if possible, and compared with the stored information.

`AGR.Test.Characters( [`*tocid*`][:TryToExtendData] )`
checks the character information (that belongs to *tocid*) for the matrix and permutation representations.

`AGR.Test.StdCompatibility( [`*tocid*`][:TryToExtendData] )`
checks whether the information about the compatibility of standard generators of a group and its factor groups that is stored in the `GAPnames` component of

`AtlasOfGroupRepresentationsInfo` (7.1.5) and belongs to *tocid* coincides with computed values.

The following criterion is used for computing the value for a group G. Use the GAP Character Table Library to determine factor groups F of G for which standard generators are defined and moreover a presentation in terms of these standard generators is known. Evaluate the relators of the presentation in the standard generators of G, and let N be the normal closure of these elements in G. Then mapping the standard generators of F to the N-cosets of the standard generators of G is an epimorphism. If $|G/N| = |F|$ holds then G/N and F are isomorphic, and the standard generators of G and F are compatible in the sense that mapping the standard generators of G to their N-cosets yields standard generators of F.

`AGR.Test.KernelGenerators( [`*tocid*`][:TryToExtendData] )`

checks whether the straight line programs (that belong to *tocid*) for computing generators of kernels of natural epimorphisms between ATLAS groups compute generators of normal subgroups of the right group orders. If it is known that the given standard generators of the given group are compatible with some standard generators of the factor group in question (see the section about `AGR.Test.StdCompatibility`) then it is also checked whether evaluating the straight line program at these standard generators of the factor group yields only the identity.

Note that the verification of normal subgroups of matrix groups may be *very* time and space consuming if the package recog [NSA+18] is not available.

The function also tries to *find* words for computing kernel generators of those epimorphisms for which no straight line programs are stored; the candidates are given by stored factor fusions between the character tables from the GAP Character Table Library.

`AGR.Test.MaxesOrders( [`*tocid*`] )`

checks whether the orders of maximal subgroups stored in the component `GAPnames` of `AtlasOfGroupRepresentationsInfo` (7.1.5) coincide with the orders computed from the restriction of an ATLAS permutation representation of degree up to `AGR.Test.MaxTestDegree` (using a straight line program that belongs to *tocid*), from the character table, or the table of marks with the given name, or from the information about maximal subgroups of the factor group modulo a normal subgroup that is contained in the Frattini subgroup.

`AGR.Test.MaxesStructure()`

checks whether the names of maximal subgroups stored in the component `GAPnames` of `AtlasOfGroupRepresentationsInfo` (7.1.5) coincide with the names computed from the GAP character table with the given name.

`AGR.Test.MaxesStandardization( [`*tocid*`] )`

checks whether the straight line programs (that belong to *tocid*) for standardizing the generators of maximal subgroups are correct: If a semi-presentation is available for the maximal subgroup and the standardization in question then it is used, otherwise an explicit isomorphism is tried.

`AGR.Test.CompatibleMaxes( [`*tocid*`][:TryToExtendData] )`

checks whether the information about deriving straight line programs for restricting to subgroups from straight line programs that belong to a factor group coincide with computed values.

The following criterion is used for computing the value for a group G. If F is a factor group of G such that the standard generators of G and F are compatible (see the test function `AGR.Test.StdCompatibility`) and if there are a presentation for F and a permutation

representation of G then it is checked whether the "maxes" type straight line programs for F can be used to compute generators for the maximal subgroups of G; if not then generators of the kernel of the natural epimorphism from G to F, must be added.

7.9.2 Other Sanity Checks

The tests described in this section are intended for checking data that do not belong to a particular part of the AtlasRep database. Therefore *all* locally available data are used in these tests. Each of the tests is given by a function without arguments that returns `false` if a contradiction was found during the test, and `true` otherwise. Additionally, certain messages are printed when contradictions between stored and computed data are found, when stored data cannot be verified computationally, or when the computations yield improvements of the stored data. Currently the following tests of this kind are available.

`AGR.Test.Standardization()`
: checks whether all generating sets corresponding to the same set of standard generators have the same element orders; for the case that straight line programs for computing certain class representatives are available, also the orders of these representatives are checked w. r. t. all generating sets.

`AGR.Test.StdTomLib()`
: checks whether the standard generators are compatible with those that occur in the TomLib package.

`AGR.Test.MinimalDegrees()`
: checks that the (permutation and matrix) representations available in the database do not have smaller degree than the minimum claimed in Section 6.3.

Finally, we reset the user preference and the info level which had been set at the beginning of Chapter 2.

Example

```
gap> SetUserPreference( "AtlasRep", "DisplayFunction", origpref );
gap> SetInfoLevel( InfoAtlasRep, globallevel );
```

References

[BGH+22] T. Breuer, S. Gutsche, M. Horn, A. Hulpke, S. Kohl, F. Lübeck, and C. Wensley. utils, utility functions in gap, Version 0.77. https://gap-packages.github.io/utils, Aug 2022. GAP package. 6

[BHM09] T. Breuer, I. Höhler, and J. Müller. MFER, multiplicity-free endomorphism rings of permutation modules of the sporadic simple groups and their cyclic and bicyclic extensions, Version 1.0.0. https://www.math.rwth-aachen.de/~MFER, Jul 2009. GAP package. 70, 72

[BL18] T. Breuer and F. Lübeck. Browse, ncurses interface and browsing applications, Version 1.8.9. https://www.math.rwth-aachen.de/~Browse, Jun 2018. GAP package. 12, 13, 39, 62, 63

[BN95] T. Breuer and S. P. Norton. *Improvements to the Atlas*, page 297–327. Volume 11 of *London Mathematical Society Monographs. New Series* [JLPW95], 1995. Appendix 2 by T. Breuer and S. Norton, Oxford Science Publications. 6

[Bre14] T. Breuer. CTBlocks, Blocks of Character Tables, Version 0.9.3. https://www.math.rwth-aachen.de/~Thomas.Breuer/ctblocks, Feb 2014. GAP package. 70, 72

[Bre22] T. Breuer. The GAP Character Table Library, Version 1.3.3. https://www.math.rwth-aachen.de/~Thomas.Breuer/ctbllib, Mar 2022. GAP package. 14, 18, 35, 94

[BSWW01] J. N. Bray, I. A. I. Suleiman, P. G. Walsh, and R. A. Wilson. Generating maximal subgroups of sporadic simple groups. *Comm. Algebra*, 29(3):1325–1337, 2001. 5, 50

[CCN+85] J. H. Conway, R. T. Curtis, S. P. Norton, R. A. Parker, and R. A. Wilson. *Atlas of finite groups*. Oxford University Press, Eynsham, 1985. Maximal subgroups and ordinary characters for simple groups, With computational assistance from J. G. Thackray. 6, 12, 18, 24, 35, 36, 37, 49, 63, 91, 92

[CP96] J. J. Cannon and C. Playoust. An introduction to algebraic programming in Magma. http://www.math.usyd.edu.au:8000/u/magma, 1996. 6

[GAP19] GAP – Groups, Algorithms, and Programming, Version 4.10.2. http://www.gap-system.org, Jun 2019. 6

[HL89] G. Hiss and K. Lux. *Brauer trees of sporadic groups*. Oxford Science Publications. The Clarendon Press, Oxford University Press, New York, 1989. 92, 94

[Hup67] B. Huppert. *Endliche Gruppen. I*. Die Grundlehren der Mathematischen Wissenschaften, Band 134. Springer-Verlag, Berlin, 1967. 94

[Jan05] C. Jansen. The minimal degrees of faithful representations of the sporadic simple groups and their covering groups. *LMS J. Comput. Math.*, 8:122–144 (electronic), 2005. 62, 92, 94

[JLPW95] C. Jansen, K. Lux, R. Parker, and R. Wilson. *An atlas of Brauer characters*, volume 11 of *London Mathematical Society Monographs. New Series*. The Clarendon Press Oxford University Press, New York, 1995. Appendix 2 by T. Breuer and S. Norton, Oxford Science Publications. 12, 63, 92, 122

[JSO14] The javascript object notation (json) data interchange format. `http://www.rfc-editor.org/info/rfc7159`, Mar 2014. 95

[LN18] F. Lübeck and M. Neunhöffer. GAPDoc, a Meta Package for GAP Documentation, Version 1.6.2. `https://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc`, Oct 2018. GAP package. 14, 15, 63

[Lüb21] F. Lübeck. StandardFF, a GAP package for constructing finite fields. `https://github.com/frankluebeck/StandardFF/`, 2021. GAP package. 103

[Neu14] M. Neunhöffer. IO, bindings for low level C library IO, Version 4.3.1. `http://www-groups.mcs.st-and.ac.uk/~neunhoef/Computer/Software/Gap/io.html`, Apr 2014. GAP package. 13, 15, 72, 96

[Nic06] S. J. Nickerson. *An Atlas of Characteristic Zero Representations*. Phd thesis, School of Mathematics, University of Birmingham, 2006. 5, 86

[NMP18] L. Naughton, T. Merkwitz, and G. Pfeiffer. TomLib, the GAP library of tables of marks, Version 1.2.7. `http://schmidt.nuigalway.ie/tomlib`, Oct 2018. GAP package. 29, 94

[NSA+18] M. Neunhöffer, Á. Seress, N. Ankaralioglu, P. Brooksbank, F. Celler, S. Howe, M. Law, S. Linton, G. Malle, A. Niemeyer, E. O'Brien, C. M. Roney-Dougal, and M. Horn. recog, a collection of group recognition methods, Version 1.3.1. `https://gap-packages.github.io/recog`, Sep 2018. GAP package. 120

[NW05] S. J. Nickerson and R. A. Wilson. Semi-presentations for the sporadic simple groups. *Experiment. Math.*, 14(3):359–371, 2005. 84

[Rin] M. Ringe. The C MeatAxe, Version 2.4. `https://www.math.rwth-aachen.de/~MTX`. 6, 67, 98, 101, 102, 104, 106

[SWW00] I. A. I. Suleiman, P. G. Walsh, and R. A. Wilson. Conjugacy classes in sporadic simple groups. *Comm. Algebra*, 28(7):3209–3222, 2000. 5, 50

[Wil96] R. A. Wilson. Standard generators for sporadic simple groups. *J. Algebra*, 184(2):505–515, 1996. 5, 36

[WWT+] R. A. Wilson, P. Walsh, J. Tripp, I. Suleiman, R. A. Parker, S. P. Norton, S. Nickerson, S. Linton, J. Bray, and R. Abbott. ATLAS of Finite Group Representations. `http://atlas.math.rwth-aachen.de/Atlas/v3`. 5, 6, 18

Index