recog

A package for constructive recognition of permutation and matrix groups

1.4.4

22 January 2025

Max Neunhöffer

Ákos Seress

Max Neunhöffer

Email: max@9hoeffer.de

Copyright

© 2005-2014 by Max Neunhöffer and Ákos Seress

Contents

Chapter 1

Introduction

1.1 Philosophy

This package is about group recognition. It provides a generic framework to implement methods of group recognition, regardless of what computational representation is used. This means, that the code in this package is useful at least for permutation groups, and matrix groups and projective groups over finite fields. The setup is described in [NS06].

The framework allows to build composition trees and handles the builtup and usage of these trees in a generic way. It also contains a method selection (described in Chapter 4) that allows to install recognition methods in a convenient way and that automatically tries to try the different available methods in a sensible order.

1.2 Overview over this manual

Chapter 2 describes the installation of this package.

Chapter 3 describes the generic, recursive procedure used for group recognition throughout this package. At the heart of this procedure is the definition of "FindHomomorphism" methods, which is also described in that chapter. For the choice of the right method for finding a homomorphism (or an isomorphism) we use another generic procedure, the "method selection" which is not to be confused with the GAP method selection.

Our own method selection system is described in detail in Chapter 4, because it is interesting in its own right and might be useful in other circumstances.

Chapter 6 describes the available "FindHomomorphism" methods.

Chapter 5 explains what one can do with a completed recognition tree.

Finally, Chapter 7 shows some instructive examples of the usage of this package.

1.3 Feedback and support

If you have any bug reports, feature requests, or suggestions, then please tell us via the issue tracker on GitHub.

In addition, the recog package has a mailing list, at recog@gap-system.org, which can be used for holding discussions, sharing information, and asking questions about the package. You can find more information, and register to receive the mail sent to this list, at

`https://mail.gap-system.org/mailman/listinfo/recog`.

1.4 Literature (temporary)

TODO Find the appropriate place to put these references, so that they appear in the bibliography: [LNPS06] [BLS97] [JLNP13] [DLGLO13] [LGO09] [BLGN$^+$05] [BLGN$^+$03] [NP98] [NP99] [NP97] [Pra99] [CLG97a] [CLG97b] [CLGM$^+$95] [BNS06] [CNRD09] [BHLGO15] [Neu09] [Ser03] [KK15] [LO16] [LO07] [HLO$^+$08] [BB99] [BBS09] [BS01] [Bro01] [Bro08] [Bro03] [BK06] [BK01] [CLG98] [CLGO06] [CLG01] [CFL97] [DLGO15] [GH97] [GLGO06] [HLGOR96a] [HLGOR96b] [HR94] [IL00] [KM13] [KM15] [LGO97a] [LGO02] [LGO97b] [LG01] [LMO07] [NP92] [Nie05] [O'B11] [O'B06] [Pak00] [Par84]

Chapter 2

Installation of the recog package

To install this package, just extract the package's archive file to the GAP pkg directory.

If the recog package is not automatically loaded when GAP is started, then you must load the package with `LoadPackage("recog");` before its functions become available.

Note that the recog package needs the AtlasRep, FactInt, Forms, genss, and orb packages to work. Recompiling the documentation is possible by the command gap `makedoc.g` in the recog directory. But this should not be necessary.

Chapter 3

Group recognition

This chapter describes a generic framework for group recognition. The basic problem is, we want to solve the constructive membership problem: given any $g \in G$, $G = \langle X \rangle$, write a straight line program (SLP) from X to g, for $g \notin G$ (in the situation that G is naturally embedded into some bigger group), the algorithm should fail. This is usually done by constructing some nice generators (and then writing an SLP from the nice generators to g and concatenating with an SLP from X to the nice generators). Often, for efficiency reasons, we will just store the nice generators and then only be interested in the SLP from those to g. The framework presented here deals with exactly this process.

The generic framework was designed having three situations in mind: permutation groups, matrix groups and projective groups. Although the methods used are quite different for those cases, there is a common pattern in the procedure of recognition. Namely, first we have to find a homomorphism, solve the constructive membership problem recursively in image and kernel, then put it together. The recursion ends in groups where we can solve the constructive membership problem directly. The general framework reflects this idea and separates it from the rest of the recognition methods.

Solution of the constructive membership problem comes in two stages: first a "recognition phase" and then a "verification phase". The recognition phase usually consists of randomised algorithms with certain error or failure probabilities. The result is some kind of "recognition information" that will describe the group already very well, but which is not yet proven to be correct. However, one can already write arbitrary elements in the group as product of the given generators. In the verification phase a presentation of the group is calculated, thereby proving that the group generated by the given generators is in fact isomorphic to the group described by the recognition information. In many cases the verification phase will be much more expensive than the recognition phase.

In the following sections, we describe the generic framework. We begin in Section 3.1 with a technical description of the recursive procedure behind our main function `RecogniseGroup` (3.1.5). In Section 3.2 we describe the return type of `RecogniseGroup` (3.1.5) which we call "recognition nodes". The methods to find homomorphisms are described in Section 3.3. Finally, we have three sections in which we collect conventions for the recognition of different types of groups.

3.1 The recursive procedure

At the heart of the recognition procedure is a function called `RecogniseGeneric` (3.1.1) which gets a GAP group object and returns a so-called "recognition node" (see Subsection 3.2 for details). Success or failure will be indicated by this record being in the filter `IsReady` (3.2.5) or not.

To know how to find homomorphisms the function gets as another argument a database of methods

(see Section 3.3 for a description of the setup for methods for finding homomorphisms and Section 4.1 in Chapter 4 for details about method databases). This database will be different according to the type of group in question.

To describe the algorithm executed by `RecogniseGeneric` (3.1.1) we first summarise it in steps:

1. Create a new, empty recognition node.

2. Use the database of `FindHomomorphism` methods and the method selection procedure described in Chapter 4 to try to find a homomorphism onto a smaller group or an isomorphism onto another known group. Terminate with failure if this does not work.

3. If an isomorphism is found or a method somehow else recognises the group in question, such that we can write elements as straight line programs in the generators from now on, then make the recognition node a leaf of the recognition tree and return success.

4. Otherwise the function sets up all the data for the homomorphism and calls itself with the image of the homomorphism. Note that this might use another database of recognition methods because the homomorphism might change the representation of the group.

5. After successful recognition of the image group the procedure has to recognise the kernel of the homomorphism. The first step for this is to find generators. If they are not already known from the `FindHomomorphism` method, they are created by producing random elements in the group, mapping them through the homomorphism, writing them as a straight line program in the images of the generators and applying this straight line program to the original generators. The quotient of the random element and the result of the straight line program lies in the kernel of the homomorphism. After creating 20 (FIXME: is 20 correct?) random generators of the kernel we assume for the moment that they generate the kernel.

6. The function `RecogniseGeneric` (3.1.1) can now call itself for the kernel. After successful recognition of the kernel all the data for the node is completed and success is returned.

7. The function `RecogniseGeneric` (3.1.1) now acquires preimages of the nice generators behind the homomorphism and appends the nice generators of the kernel. This list of generators is now the list of nice generators for the current node.

Note that with the collected data one can write arbitrary elements of the group as a straight line program in the generators as follows:

1. Map the element through the homomorphism.

2. Write the element in the image group as a product of the nice generators in the image group.

3. Apply the resulting straight line program to the preimages of those nice generators and calculate the quotient, which will now lie in the kernel.

4. Write the kernel element as a straight line program in the kernel generators.

5. Assemble both straight line programs to one bigger straight line program (which is now in terms of our own nice generators) and return it.

If this procedure fails in the fourth step, this indicates that our random generators for the kernel did not yet generate the full kernel and makes further recognition steps necessary. This will not happen after a successful verification phase.

The latter procedure to write elements as straight line programs in the generators is implemented in the function SLPforElementGeneric (3.3.2) which will be called automatically if one calls the SLPforElement (3.2.15) function of the resulting recognition node (see slpforelement (3.2.14)).

It is now high time to give you the calling details of the main recursive recognition function:

3.1.1 RecogniseGeneric

▷ RecogniseGeneric(*H, methoddb, depthString, knowledge*)   (function)
▷ RecognizeGeneric(*H, methoddb, depthString, knowledge*)   (function)
    Returns: fail for failure or a recognition node.

H must be a GAP group object, *methoddb* must be a method database in the sense of Section 4.2 containing FindHomomorphism methods in the sense of Section 3.3. *depthString* is a string whose length measures the depth in the recognition tree. It will be increased by one character for each step we go into the tree, namely by F for a image node, and K for a kernel. The top level begins with an empty string. *knowledge* is an optional record the components of which are copied into the new recognition node which is created for the group *H*. Especially the component hints can contain a list of additional find homomorphism methods (described by records as in Section 4.2). The methods in hints and in *methoddb* are merged and sorted into rank–descending order. The result is passed to CallMethods (4.3.1). This feature is intended to give hints about prior knowledge about which find homomorphism method might succeed.

The function performs the algorithm described above and returns either fail in case of failure or a recognition node in case of success. For the content and definition of recognition nodes see Section 3.2.

The user will usually not call this function directly, but will use the following convenience functions:

3.1.2 RecognisePermGroup

▷ RecognisePermGroup(*H*)   (function)
▷ RecognizePermGroup(*H*)   (function)
    Returns: fail for failure or a recognition node.

H must be a GAP permutation group object. This function calls RecogniseGeneric (3.1.1) with the method database used for permutation groups, which is stored in the global variable FindHomDbPerm (4.2.2), and no prior knowledge.

3.1.3 RecogniseMatrixGroup

▷ RecogniseMatrixGroup(*H*)   (function)
▷ RecognizeMatrixGroup(*H*)   (function)
    Returns: fail for failure or a recognition node.

H must be a GAP matrix group object over a finite field. This function calls RecogniseGeneric (3.1.1) with the method database used for matrix groups, which is stored in the global variable FindHomDbMatrix (4.2.3), and no prior knowledge.

3.1.4 RecogniseProjectiveGroup

▷ RecogniseProjectiveGroup(*H*) (function)
▷ RecognizeProjectiveGroup(*H*) (function)

Returns: `fail` for failure or a recognition node.

H must be a GAP matrix group object over a finite field. Since as of now no actual projective groups are implemented in the GAP library we use matrix groups instead. The recognition will however view the group as the projective group, i.e. the matrix group modulo its scalar matrices. This function calls `RecogniseGeneric` (3.1.1) with the method database used for projective groups, which is stored in the global variable `FindHomDbProjective` (4.2.4), and no prior knowledge.

3.1.5 RecogniseGroup

▷ RecogniseGroup(*H*) (function)
▷ RecognizeGroup(*H*) (function)

Returns: `fail` for failure or a recognition node.

H must be a GAP group object. This function automatically dispatches to one of the two previous functions `RecognisePermGroup` (3.1.2), or `RecogniseMatrixGroup` (3.1.3), according to the type of the group *H*. Note that since currently there is no implementation of projective groups in the GAP library, one cannot recognise a matrix group *H* as a projective group using this function.

3.1.6 TryFindHomMethod

▷ TryFindHomMethod(*H, method, projective*) (function)

Returns: `fail` or `false` or a recognition node.

Use this function to try to run a given find homomorphism method *method* on a group *H*. Indicate by the boolean *projective* whether or not the method works in projective mode. For permutation groups, set this to `false`. The result is either `fail` or `false` if the method fails or a recognition node `ri`. If the method created a leaf then `ri` will be a leaf, otherwise it will have the attribute `Homom` (3.2.7) set, but no image or kernel have been created or recognised yet. You can use for example the methods in `FindHomMethodsPerm` (4.4.1) or `FindHomMethodsMatrix` (4.4.2) or `FindHomMethodsProjective` (4.4.3) as the *method* argument.

GAP homomorphisms are not required to give a sensible answer when given a value not in their source, and in practice often enter the break loop, or return an incorrect answer. This causes problems when checking if a value is not in the represented group. To avoid this problem, `validatehomominput` (3.2.18) can be set to a function. This function is used to filter possible group elements, before they are passed to `Homom` (3.2.7).

3.2 Recognition nodes

A recognition node is a GAP component object. It is a member of the family

3.2.1 RecogNodeFamily

▷ RecogNodeFamily (family)

and is in the category

3.2.2 IsRecogNode

▷ `IsRecogNode`   (Category)

and is `IsAttributeStoringRep` (**Reference: IsAttributeStoringRep**), such that we can de-
fine attributes for it, the values of which are stored once they are known. A recognition node al-
ways represents a whole binary tree of such records, see the attributes `ImageRecogNode` (3.2.10) and
`KernelRecogNode` (3.2.11) below.

Recognition nodes can be created via:

3.2.3 RecogNode

▷ `RecogNode(H[, projective][, r])`   (operation)
▷ `RecogNode(r, H, projective)`   (operation)

Returns: a recognition node.

Create an `IsRecogNode` (3.2.2) object `node` representing the group `H`. The optional boolean
`projective` defaults to false and specifies, in the case that `H` is a matrix group, whether `H` is to
be interpreted as a projective group. The optional record `r` defaults to an empty record and is used to
initialize the returned `node`.

For backwards-compatibility, also the order of arguments `r`, `H`, `projective` is accepted.

The following filters are defined for recognition nodes:

3.2.4 IsLeaf

▷ `IsLeaf`   (Flag)

This flag indicates, whether or not a recognition node represents a leaf in the recognition tree. If
it is not set, one finds at least one of the attributes `ImageRecogNode` (3.2.10) and `KernelRecogNode`
(3.2.11) set for the corresponding node. This flag is normally reset and has to be set by a find homo-
morphism method to indicate a leaf.

3.2.5 IsReady

▷ `IsReady`   (Flag)

This flag is set for a `IsRecogNode` (3.2.2) object `node` by `RecogniseGeneric` (3.1.1), if recog-
nition of the *subtree* rooted in `node` finished successfully. Recognition of a node is considered suc-
cessful, if two conditions hold. First, the call of `CallMethods` (4.3.1) for this node reports `Success`,
that is a method from the respective method database (see Section 4.2) was successful. Secondly, the
construction of the kernel generators was successful.

Thus, if the `IsReady` flag is set, this does not necessarily mean, that the result of the recognition
procedure was verified and proven to be mathematically correct!

In particular, any computations using the datastructure set up by the recognition procedure, like
`Size` (5.1.3) and membership testing via `\in` (5.1.2), will error if `IsReady` is not set.

The following attributes are defined for recognition nodes:

3.2.6 Grp

▷ Grp(`ri`) (attribute)

The value of this attribute is the group that is to be recognised by this recognition node `ri`. This attribute is always present during recognition and after completion. Note that the generators of the group object stored here always have a memory attached to them, such that elements that are generated from them remember, how they were acquired.

3.2.7 Homom

▷ Homom(`ri`) (attribute)

The value of this attribute is the homomorphism that was found from the group described by the recognition node `ri` as a GAP object. It is set by a find homomorphism method that succeeded to find a homomorphism (or isomorphism). It does not have to be set in leaf nodes of the recognition tree.

3.2.8 NiceGens

▷ NiceGens(`ri`) (attribute)

The value of this attribute must be set for all nodes and contains the nice generators. The SLPforElement (3.2.15) function of the node will write its straight line program in terms of these nice generators. For leaf nodes, the find homomorphism method is responsible to set the value of NiceGens. By default, the original generators of the group at this node are taken. For a homomorphism (or isomorphism), the NiceGens will be the concatenation of preimages of the NiceGens of the image group (see pregensfac (3.2.9)) and the NiceGens of the kernel. A find homomorphism method does not have to set NiceGens if it finds a homomorphism. Note however, that such a find homomorphism method has to ensure somehow, that preimages of the NiceGens of the image group can be acquired. See calcnicegens (3.2.20), CalcNiceGens (3.2.23) and slptonice (3.2.24) for instructions.

3.2.9 pregensfac

▷ pregensfac(`ri`) (attribute)

The value of this attribute is only set for homomorphism nodes. In that case it contains preimages of the nice generators in the image group. This attribute is set automatically by the generic recursive recognition function using the mechanism described with the attribute calcnicegens (3.2.20) below. A find homomorphism does not have to touch this attribute.

3.2.10 ImageRecogNode

▷ ImageRecogNode(`ri`) (attribute)

The value of this attribute is the recognition node of the image of the homomorphism that was found from the group described by the recognition node `ri`. It is set by the generic recursive procedure after a find homomorphism method has succeeded to find a homomorphism (or isomorphism). It does

not have to be set in leaf nodes of the recognition tree. This attribute value provides the link to the "image" subtree of the recognition tree.

3.2.11 KernelRecogNode

▷ KernelRecogNode(*ri*) (attribute)

The value of this attribute is the recognition node of the kernel of the homomorphism that was found from the group described by the recognition node *ri*. It is set by the generic recursive procedure after a find homomorphism method has succeeded to find a homomorphism (or isomorphism). It does not have to be set in leaf nodes of the recognition tree or if the homomorphism is known to be an isomorphism. In the latter case the value of the attribute is set to `fail`. This attribute value provides the link to the "kernel" subtree of the recognition tree.

3.2.12 ParentRecogNode

▷ ParentRecogNode(*ri*) (attribute)

The value of this attribute is the recognition node of the parent of this node in the recognition tree. The top node does not have this attribute set.

3.2.13 fhmethsel

▷ fhmethsel(*ri*) (attribute)

The value of this attribute is the record returned by the method selection (see Section 4.3) after it ran to find a homomorphism (or isomorphism). It is there to be able to see which methods were tried until the recognition of the node was completed.

3.2.14 slpforelement

▷ slpforelement(*ri*) (attribute)

After the recognition phase is completed for the node *ri*, we are by definition able to write arbitrary elements in the group described by this node as a straight line program (SLP) in terms of the nice generators stored in `NiceGens` (3.2.8). This attribute value is a function taking the node *ri* and a group element as its arguments and returning the above mentioned straight line program. For the case that a find homomorphism method succeeds in finding a homomorphism, the generic recursive function sets this attribute to the function `SLPforElementGeneric` (3.3.2) which does the job for the generic homomorphism situation. In all other cases the successful find homomorphism method has to set this attribute to a function doing the job. The find homomorphism method is free to store additional data in the recognition node or the group object such that the `SLPforElement` (3.2.15) function can work.

3.2.15 SLPforElement

▷ SLPforElement(*ri*, *x*) (function)

Returns: a straight line program expressing *x* in the nice generators.

This is a wrapper function which extracts the value of the attribute `slpforelement` (3.2.14) and calls that function with the arguments `ri` and `x`.

3.2.16 StdPresentation

▷ StdPresentation(`ri`) (attribute)

After the verification phase, the presentation is stored here. Details have still to be decided upon.

3.2.17 methodsforimage

▷ methodsforimage(`ri`) (attribute)

This attribute is initialised at the beginning of the recursive recognition function with the database of find homomorphism methods that was used to recognise the group corresponding to the recognition node `ri`. If the found homomorphism changes the representation of the group (going for example from a matrix group to a permutation group), the find homomorphism method can report this by exchanging the database of find homomorphism methods to be used in the recognition of the image of the homomorphism by setting the value of this attribute to something different. It lies in the responsibility of the find homomorphism method to do so, if the representation changes through the homomorphism.

3.2.18 validatehomominput

▷ validatehomominput(`ri, x`) (attribute)

The value of this attribute, if there is any, must be a function with two arguments: a recognition record `ri`, and an element `x`. The function must return a boolean. If it returns `false`, then this means that `x` is not in the source of the homomorphism returned by Homom (3.2.7). If `true` is returned, then either `x` is in the source of that homomorphism, or passing `x` to the homomorphism returns `fail`.

For example, if `ri` represents a matrix group that preserves a subspace, then the source of Homom (3.2.7) will be matrices which preserve that subspace, and passing matrices which do not preserve this subspace to Homom (3.2.7) may produce incorrect answers. `validatehomominput` can be used to filter out such elements. The function ValidateHomomInput (3.2.19) provides a simple wrapper to this attribute, which calls `validatehomominput` unless it is not defined, in which case ValidateHomomInput (3.2.19) returns `true`.

3.2.19 ValidateHomomInput

▷ ValidateHomomInput(`ri, x`) (function)

Returns: a boolean.

This is a wrapper function which calls `validatehomominput` (3.2.18) of `ri` with `x`, or returns `true` if `ri` does not define `validatehomominput` (3.2.18).

The following two attributes are concerned with the relation between the original generators and the nice generators for a node. They are used to transport this information from a successful find homomorphism method up to the recursive recognition function:

3.2.20   calcnicegens

▷ calcnicegens(*ri*)   (attribute)

To make the recursion work, we have to acquire preimages of the nice generators in image groups under the homomorphism found. But we want to keep the information, how the nice generators were found, locally at the node where they were found. This attribute solves this problem of acquiring preimages in the following way: Its value must be a function, taking the recognition node *ri* as first argument, and a list *origgens* of preimages of the original generators of the current node, and has to return corresponding preimages of the nice generators. Usually this task can be done by storing a straight line program writing the nice generators in terms of the original generators and executing this with inputs *origgens*. Therefore the default value of this attribute is the function CalcNiceGensGeneric (3.2.21) described below.

3.2.21   CalcNiceGensGeneric

▷ CalcNiceGensGeneric(*ri, origgens*)   (function)
   Returns: a list of preimages of the nice generators
   This is the default function for leaf nodes for the attribute calcnicegens (3.2.20) described above. It does the following: If the value of the attribute slptonice (3.2.24) is set, then it must be a straight line program expressing the nice generators in terms of the original generators of this node. In that case, this straight line program is executed with *origgens* as inputs and the result is returned. Otherwise, *origgens* is returned as is. Therefore a leaf node just has to do nothing if the nice generators are equal to the original generators, or can simply store the right straight line program into the attribute slptonice (3.2.24) to fulfill its duties.

3.2.22   CalcNiceGensHomNode

▷ CalcNiceGensHomNode(*ri, origgens*)   (function)
   Returns: a list of preimages of the nice generators
   This is the default function for homomorphism node for the attribute calcnicegens (3.2.20). It just delegates to image and kernel of the homomorphism, as the nice generators of a homomorphism (or isomorphism) node are just the concatenation of the preimages of the nice generators of the image with the nice generators of the kernel. A find homomorphism method finding a homomorphism or isomorphism does not have to do anything with respect to nice generators.

3.2.23   CalcNiceGens

▷ CalcNiceGens(*ri, origgens*)   (function)
   Returns: a list of preimages of the nice generators
   This is a wrapper function which extracts the value of the attribute calcnicegens (3.2.20) and calls that function with the arguments *ri* and *origgens*.

3.2.24   slptonice

▷ slptonice(*ri*)   (attribute)

As described above, the value, if set, must be a straight line program expressing the nice generators at this node in terms of the original generators. This is for leaf nodes, that choose to use the default function `CalcNiceGensGeneric` (3.2.21) installed in the `calcnicegens` (3.2.20) attribute.

The following three attributes are concerned with the administration of the kernel of a found homomorphism. Find homomorphism methods use them to report to the main recursive recognition function their knowledge about the kernel:

3.2.25 gensN

▷ gensN(`ri`)   (attribute)

The value of this mutable attribute is a list of generators of the kernel of the homomorphism found at the node `ri`. It is initialised as an empty list when the recursive recognition function starts. Successful find homomorphism methods may append generators of the kernel to this list if they happen to stumble on them. After successful recognition of the image of the homomorphism the main recursive recognition function will try to create a few more generators of the kernel and append them to the list which is the value of the attribute `gensN`. The exact behaviour depends on the value of the attribute `findgensNmeth` (3.2.26) below. The list of generators after that step is used to recognise the kernel. Note that the generators in `gensN` have a memory attached to them, how they were obtained in terms of the original generators of the current node.

3.2.26 findgensNmeth

▷ findgensNmeth(`ri`)   (attribute)

This attribute decides about how generators of the kernel of a found homomorphism are produced. Its value has to be a record with at least two components bound. The first is `method` which holds a function taking at least one argument `ri` and possibly more, and does not return anything. The second is `args` which holds a list of arguments for the above mentioned function. The real list of arguments is derived by prepending the recognition node to the list of arguments in `args`. That is, the following code is used to call the method:

```
    gensNmeth := findgensNmeth(ri);
    CallFuncList(gensNmeth.method,Concatenation([ri],gensNmeth.args));
```

The record is initialised upon creation of the recognition node to calling `FindKernelFastNormalClosure` (3.2.29) with arguments [6, 3] (in addition to the first argument `ri`). See below for a choice of possible find kernel methods.

3.2.27 FindKernelRandom

▷ FindKernelRandom(`ri`, `n`)   (function)
   Returns: `true` or `false`.

`n` random elements are generated, mapped through the homomorphism, written as a straight line program in the generators. Then the straight line program is executed with the original generators thereby producing elements in the same coset. The quotients are then elements of the kernel. The kernel elements created are stored in the attribute `gensN` (3.2.25). Returns `false` if the generation of the straight line program for some element fails.

3.2.28 FindKernelDoNothing

▷ FindKernelDoNothing(`ri, n1, n2`) (function)
    Returns: `true`.

Does nothing. This function is intended to be set as method for producing kernel elements if the kernel is known to be trivial or if one knows, that the attribute `gensN` (3.2.25) already contains a complete set of generators for the kernel.

3.2.29 FindKernelFastNormalClosure

▷ FindKernelFastNormalClosure(`ri, n1, n2`) (function)
    Returns: `true` or `false`.

`n1` random elements of the kernel are generated by calling FindKernelRandom. Then this function computes a probable generating set of the normal closure in `G` of the group generated by the random elements. The integer `n2` indicates how hard it should try. Returns `false` if the call to FindKernelRandom (3.2.27) returns `false`.

3.2.30 gensNslp

▷ gensNslp(`ri`) (attribute)

The recursive recognition function calculates a straight line program that computes the generators of the kernel stored in `gensN` (3.2.25) in terms of the generators of the group recognised by `ri`. This straight line program is stored in the value of this mutable attribute. It is used by the generic function SLPforElementGeneric (3.3.2).

3.2.31 immediateverification

▷ immediateverification(`ri`) (attribute)

Sometimes a find homomorphism has information that it will be difficult to create generators for the kernel, for example if it is known that the kernel will need lots of generators. In that case this attribute with the default boolean value `false` can be set to `true`. In that case, the generic recursive recognition function will perform an immediate verification phase after the kernel has been recognised. This is done as follows: A few random elements are created, mapped through the homomorphism and written as an SLP in the nice generators there. Then this SLP is executed with preimages of those nice generators. The quotient lies then in the kernel and is written as an SLP in terms of the nice generators of the would be kernel. If this is not possible, then probably the creation of kernel generators was not complete and a few more kernel elements are produced and recognition in the kernel starts all over again. This is for example done in case of the "Imprimitive" method which maps onto the action on a block system. In that case, the kernel often needs lots of generators.

The following attributes are used to give a successful find homomorphism method further possibilities to transport knowledge about the group recognised by the current recognition node to the image or kernel of the found homomorphism:

3.2.32 InitialDataForKernelRecogNode

▷ InitialDataForKernelRecogNode(*ri*)   (attribute)

This attribute is initialised to a record with only the component `hints` bound to an empty list at the beginning of the recursive recognition function. Find homomorphism methods can put acquired knowledge about the group to be recognised (like for example an invariant subspace of a matrix group) into this record. When a homomorphism is found and recognition goes on in its kernel, the value of this attribute is taken as initialisation data for the newly created recognition node for the kernel. Thus, information is transported down to the recognition process for the kernel. The component `hints` is special insofar as it has to contain records describing find homomorphism methods which might be particularly successful. They are prepended to the find homomorphism method database such that they are called before any other methods. This is a means to give hints to the recognition procedure in the kernel, because often during the finding of a homomorphism knowledge is acquired which might help the recognition of the kernel.

3.2.33 InitialDataForImageRecogNode

▷ InitialDataForImageRecogNode(*ri*)   (attribute)

This attribute is initialised to a record with only the component `hints` bound to an empty list at the beginning of the recursive recognition function. Find homomorphism methods can put acquired knowledge about the group to be recognised (like for example an invariant subspace of a matrix group) into this record. When a homomorphism is found and recognition goes on in its image, the value of this attribute is taken as initialisation data for the newly created recognition node for the image. Thus, information is transported down to the recognition process for the image. The component `hints` is special insofar as it has to contain records describing find homomorphism methods which might be particularly successful. They are prepended to the find homomorphism method database such that they are called before any other methods. This is a means to give hints to the recognition procedure in the image, because often during the finding of a homomorphism knowledge is acquired which might help the recognition of the image.

3.2.34 isone

▷ isone(*ri*)   (attribute)

This attribute returns a function that tests, whether or not an element of the group is equal to the identity or not. Usually this is just the operation `IsOne` (**Reference: IsOne**) but for projective groups it is a special function returning `true` for scalar matrices. In generic code, one should always use the result of this attribute to compare an element to the identity such that the code works also for projective groups. Find homomorphism methods usually do not have to set this attribute.

3.2.35 isequal

▷ isequal(*ri*)   (attribute)

This attribute returns a function that compares two elements of the group being recognised. Usually this is just the operation `EQ` (**Reference: equality of records**) but for projective groups it is a

special function checking for equality up to a scalar factor. In generic code, one should always use the result of this attribute to compare two elements such that the code works also for projective groups. Find homomorphism methods usually do not have to set this attribute.

3.2.36   OrderFunc

▷ OrderFunc(`ri`)   (attribute)

This attribute returns a function that computes the order of an element of the group being recognised. Usually this is just the operation `Order` (**Reference: Order**) but for projective groups it is a special function. In generic code, one should always use the result of this attribute to compute the order of an element such that the code works also for projective groups. Find homomorphism methods usually do not have to set this attribute.

3.2.37   Other components of recognition nodes

In this subsection we describe a few more components of recognition nodes that can be queried or set by find homomorphism methods. Not all of these components are bound in all cases. See the individual descriptions about the conventions. Remember to use the `!.` notation to access these components of a recognition node.

`leavegensNuntouched`
　　If this component is bound to `true` by a find homomorphism method or a find kernel generators method, the generic mechanism to remove duplicates and identities in the generator for the kernel is not used. This is important if your methods rely on the generating set of the kernel being exactly as it was when found.

3.3   Methods to find homomorphisms

A "find homomorphism method" has the objective to, given a group G, either find a homomorphism from G onto a group, or to find an isomorphism, or to solve the constructive membership problem directly for G, or to fail.

In case a homomorphism is found, it has to report that homomorphism back to the calling recursive recognition function together with as much information about the kernel as possible.

If a find homomorphism method determines that the node is a leaf in the recognition tree (by solving the constructive membership problem directly), then it has to ensure, that arbitrary elements can be written in terms of the nice generators of G. It does so by returning a function together with possible extra data, that can perform this job.

Of course, the find homomorphism method also has to report, how the nice generators were acquired in terms of the original generators.

If the find homomorphism method fails, it has to report, whether it has failed forever or if it possibly makes sense to try to call this method again later.

Find homomorphism methods have to fit into the framework for method selection described in Chapter 4. We now begin to describe the technical details of how a find homomorphism method has to look like and what it has to do and what it is not allowed to do. We first explain the calling convention by means of a hypothetical function:

3.3.1 FindHomomorphism

▷ FindHomomorphism(`ri, G`)   (function)

Returns: One of the values `Success`, `NeverApplicable`, `TemporaryFailure`, or `NotEnoughInformation`.

Find homomorphism methods take two arguments `ri` and `G`, of which `ri` is a recognition node and `G` is a GAP group object. The return value is one of the four possible values in the framework for method selection described in Chapter 4 indicating success, failure, or (temporary) non-applicability. The above mentioned additional information in case of success are all returned by changing the recognition node `ri`. For the conventions about what a find homomorphism method has to do and return see below.

A failed or not applicable find homomorphism method does not have to report or do anything in the recognition node `ri`. However, it can collect information and store it either in the group object or in the recognition node. Note that for example it might be that a failed find homomorphism method acquires additional information that allows another find homomorphism method to become applicable.

A not applicable find homomorphism method should find out so relatively quickly, because otherwise the whole process might be slowed down, because a find homomorphism method repeatedly ponders about its applicability. Usually no big calculations should be triggered just to decide applicability.

A successful find homomorphism method has the following duties:

for leaves:

First it has to report whether the current node is a leaf or not in the recognition tree. That is, in case a leaf was found the method has to do `SetFilterObj(ri,IsLeaf);` thereby setting the `IsLeaf` (3.2.4) flag.

A method finding a homomorphism which is not an isomorphism indicates so by not touching the flags. *FIXME: What does that mean? Which flags? The IsLeaf filter? But then this sounds as if isomorphisms require settings some flag.. but which?!? perhaps remove that sentence?*

for leaves: `SLPforElement` **(3.2.15) function**

If a find homomorphism method has produced a leaf in the recognition tree, then it has to set the attribute `slpforelement` (3.2.14) to a function like `SLPforElementGeneric` (3.3.2) that can write an arbitrary element in `G` as a straight line program in the nice generators of `G`. The method may store additional data into the recognition node for this to work. It does not have to set any other value in `ri`.

for leaves: information about nice generators

If a find homomorphism method has produced a leaf in the recognition tree, then it has to report what are the nice generators of the group described by the leaf. To this end, it has three possibilities: Firstly to do nothing, which means, that the original generators are the nice generators. Secondly to store a straight line program expressing the nice generators in terms of the original generators into the attribute `slptonice` (3.2.24). In that case, the generic frame work takes care of the rest. The third possibility is to store a function into the value of the attribute `calcnicegens` (3.2.20) which can calculate preimages of the nice generators in terms of preimages of the original generators. See the function `CalcNiceGensGeneric` (3.2.21) for an example of such a function.

for non-leaves: the homomorphism itself

If a find homomorphism method has found a homomorphism, it has to store it as a GAP ho-

momorphism object from `G` to the image group in the attribute `Homom` (3.2.7). Note that if your homomorphism changes the representation (for example going from matrix groups to permutation groups), you will have to set the attribute `methodsforimage` (3.2.17) accordingly. Also, `ValidateHomomInput` (3.2.19) may be set to a function which returns `false` for values which may cause `Homom` (3.2.7) to produce the wrong answer, or error.

for non-leaves: kernel generators

If a find homomorphism method has found a homomorphism, it has to provide information about already known generators of the kernel. This is done firstly by appending known generators of the kernel to the attribute value of `gensN` (3.2.25) and secondly by leaving or changing the attribute `findgensNmeth` (3.2.26) to a record describing the method that should be used (for details see `findgensNmeth` (3.2.26)). If one does not change the default value, the recursive recognition function will generate 20 (FIXME: is 20 correct?) random elements in `G` and produce random generators of the kernel by dividing by a preimage of an image under the homomorphism. Note that generators in `gensN` (3.2.25) have to have a memory attached to them that stores, how they were acquired from the generators of `G`.

additional information

A find homomorphism method may store any data into the attributes `InitialDataForKernelRecogNode` (3.2.32) and `InitialDataForImageRecogNode` (3.2.33), which both are records. Components in these record that are bound during the recognition will be copied into the recognition node of the kernel and image respectively of a found homomorphism upon creation and thus are available to all find homomorphism methods called for the kernel and image. This feature might be interesting to transport information that is relevant for the recognition of the kernel or image and was acquired during the recognition of `G` itself.

A special role is played by the component `hints` in both of the above records, which can hold a list of records describing find homomorphism methods that shall be tried first when recognising the kernel or image.

In addition, a find homomorphism method might set the attribute `immediateverification` (3.2.31) to `true`, if it considers the problem of finding kernel generators particularly difficult.

To explain the calling conventions for `SLPforElement` (3.2.15) functions and for the sake of completeness we present now the function `SLPforElementGeneric` (3.3.2) which is used for the case of a "homomorphism node":

3.3.2 SLPforElementGeneric

▷ SLPforElementGeneric(`ri`, `x`) (function)

Returns: A GAP straight line program.

This function takes as arguments a recognition node `ri` and a group element `x`. It returns a GAP straight line program that expresses the element `x` in terms of the nice generators of the group `G` recognised by `ri`.

This generic function here does exactly this job for the generic situation that we found a homomorphism from G to some other group say H with kernel N. It first maps `x` via the homomorphism to H and uses the recognition information there to write it as a straight line program in terms of the nice generators of H. Then it applies this straight line program to the preimages of those nice generators (see `pregensfac` (3.2.9)) thereby finding an element y of G with $x \cdot y^{-1}$ lying in the kernel N.

Then the function writes this element as a straight line program in the nice generators of N again using the recursively acquired recognition info about N. In the end a concatenated straight line program for x is built, which is in terms of the nice generators of the current node.

Chapter 4

Method selection

The setup described in this chapter is intended for situations, in which lots of different methods are available to fulfill a certain task, but in which it is not possible in the beginning to decide, which one to use. Therefore this setup regulates, rather than just which method to choose, in which order the various methods are tried. The methods themselves return whether they were successful and, if not, whether it is sensible to try them again at a later stage.

The design is intentionally kept as simple as possible and at the same time as versatile as possible, thereby providing a useful framework for many situations as described above.

Note the differences to the GAP method selection, which is designed with the idea in mind that it will be quite clear in most situations, which one is "the best" method for a given set of input data, and that we do not want to try different things. On the other hand, the GAP method selection is quite complicated, which is to some extend necessary to make sure, that lots of different information about the objects in question can be used to really find the best method.

Our setup here in particular has to fulfill the requirement, that in the end, with lots of methods installed, one still has to be able to have an overview and to "prove", that the whole system always does the right thing.

4.1 What are methods?

Recognition methods lie in the filter IsRecogMethod (4.1.1) and can be created via the function RecogMethod (4.1.2).

4.1.1 IsRecogMethod

▷ IsRecogMethod   (Category)

The category of recognition methods, that is of the objects created via RecogMethod (4.1.2).

4.1.2 RecogMethod

▷ RecogMethod(*stamp, comment, func*)   (function)

Return a recognition method method in the filter IsRecogMethod (4.1.1), where *stamp* is a string describing the method uniquely, *comment* is a string explaining how the method works, and *func*

is the method itself. The components *stamp* and *comment* can be accessed via the attributes Stamp (4.1.4) and Comment (4.1.5).

A recognition method returns one of the following four values:

Success
> means that the method was successful and no more methods have to be tried.

NeverApplicable
> means that the method was not successful and that there is no point to call the method again in this situation whatsoever.

TemporaryFailure
> means that the method temporarily failed, that it however could be sensible to call it again in this situation at a later stage. This value is typical for a Las Vegas algorithm using randomised methods, which has failed, but which may succeed when called again.

NotEnoughInformation
> means that the method for some reason refused to do its work. However, it is possible that it will become applicable later such that it makes sense to call it again, for example when more information is available.

A recognition method method should always be stored into the component Stamp(method) of one of the following records: FindHomMethodsGeneric (4.4.4), FindHomMethodsPerm (4.4.1), FindHomMethodsMatrix (4.4.2), and FindHomMethodsProjective (4.4.3). To this end one can use the function BindRecogMethod (4.1.3).

4.1.3 BindRecogMethod

▷ BindRecogMethod(*r*, *arg*)   (function)

Create the recognition method method by calling RecogMethod (4.1.2) with arguments *arg*. Then bind the component Stamp(method) of *r* to *method*.

4.1.4 Stamp

▷ Stamp(*method*)   (attribute)

The stamp of *method*, see RecogMethod (4.1.2). The argument *method* must lie in IsRecogMethod (4.1.1).

4.1.5 Comment

▷ Comment(*method*)   (attribute)

The comment of *method*, see RecogMethod (4.1.2). The argument *method* must lie in IsRecogMethod (4.1.1).

4.1.6 CallRecogMethod

▷ CallRecogMethod(*m, args*)   (function)

Call UnpackRecogMethod(m) with arguments `args` and return the return value. The argument `m` must lie in IsRecogMethod (4.1.1).

4.2 Method Databases

A *method database* is a list of records, where each record has the following components:

method

    A recognition method created with RecogMethod (4.1.2).

rank

    An integer used to sort the various methods. Higher numbers mean that the method is tried earlier. See CallMethods (4.3.1) for information on how the methods are called.

The databases are always sorted such that the ranks are decreasing. Use AddMethod (4.2.1) to add a method to a database according to its rank.

4.2.1 AddMethod

▷ AddMethod(*methodDb, method, rank*)   (function)

Add the recognition method `method` with rank `rank` to the method database `methodDb`. Return nothing. `method` is inserted into `methodDb` such that the ranks of its entries are in decreasing order. For information on recognition methods and method databases see RecogMethod (4.1.2) and Section 4.2, respectively.

The following databases contain the methods for finding homomorphisms for permutation, matrix, and projective groups.

4.2.2 FindHomDbPerm

▷ FindHomDbPerm   (global variable)

The method database for permutation groups.

4.2.3 FindHomDbMatrix

▷ FindHomDbMatrix   (global variable)

The method database for matrix groups.

4.2.4 FindHomDbProjective

▷ FindHomDbProjective   (global variable)

The method database for projective matrix groups.

4.3 How methods are called

Whenever the method selection shall be used, one calls the following function:

4.3.1 CallMethods

▷ CallMethods(*db, limit[, furtherargs]*) (function)

> **Returns:** a record `ms` describing this method selection procedure.

> The argument *db* must be a method database in the sense of Section 4.2. *limit* must be a non-negative integer. *furtherargs* stands for an arbitrary number of additional arguments, which are handed down to the called methods. Of course they must fulfill the conventions defined for the methods in the database *db*.

> The function first creates a "method selection" record keeping track of the things that happened during the method trying procedure, which is also used during this procedure. Then it calls methods with the algorithm described below and in the end returns the method selection record in its final state.

> The method selection record has the following components:

inapplicableMethods

> a record, in which for every method that returned `NeverApplicable` the value 1 is bound to the component with name the stamp of the method.

failedMethods

> a record, in which for every time a method returned `TemporaryFailure` the value bound to the component with name the stamp of the method is increased by 1 (not being bound means zero).

successMethod

> the stamp of the method that succeeded, if one did. This component is only bound after successful completion.

result

> a boolean value which is either `Success` or `TemporaryFailure` depending on whether a successful method was found or the procedure gave up respectively. This component is only bound after completion of the method selection procedure.

tolerance

> the number of times all methods failed until one succeeded. See below.

The algorithm used by `CallMethods` is extremely simple: It sets a counter `tolerance` to zero. The main loop starts at the beginning of the method database and runs through the methods in turn. Provided a method did not yet return `NeverApplicable` and did not yet return `TemporaryFailure` more than `tolerance` times before, it is tried. According to the value returned by the method, the following happens:

NeverApplicable

> this is marked in the method selection record and the main loop starts again at the beginning of the method database.

TemporaryFailure

> this is counted in the method selection record and the main loop starts again at the beginning of the method database.

`NotEnoughInformation`
     the main loop goes to the next method in the method database.

`Success`
     this is marked in the method selection record and the procedure returns successfully.

If the main loop reaches the end of the method database without calling a method (because all methods have already failed or are not applicable), then the counter `tolerance` is increased by one and everything starts all over again. This is repeated until `tolerance` is greater than the `limit` which is the second argument of `CallMethods`. The last value of the `tolerance` counter is returned in the component `tolerance` of the method selection record.

     Note that the main loop starts again at the beginning of the method database after each failed method call! However, this does not lead to an infinite loop, because the failure is recorded in the method selection record such that the method is skipped until the `tolerance` increases. Once the `tolerance` has been increased methods having returned `TemporaryFailure` will be called again. The idea behind this approach is that even failed methods can collect additional information about the arguments changing them accordingly. This might give methods that come earlier and were not applicable up to now the opportunity to begin working. Therefore one can install very good methods that depend on some already known knowledge which will only be acquired during the method selection procedure by other methods, with a high rank.

4.4   Global records storing functions

The following global records store the methods for finding homomorphisms for group recognition. We collect them in these records such that we do not use up too many global variable names.

4.4.1   FindHomMethodsPerm

▷ FindHomMethodsPerm   (global variable)

     Stores recog methods for permutation groups.

4.4.2   FindHomMethodsMatrix

▷ FindHomMethodsMatrix   (global variable)

     Stores recog methods for matrix groups.

4.4.3   FindHomMethodsProjective

▷ FindHomMethodsProjective   (global variable)

     Stores recog methods for projective groups.

4.4.4   FindHomMethodsGeneric

▷ FindHomMethodsGeneric   (global variable)

In this global record the functions that are methods for finding homomorphisms for generic group recognition are stored. We collect them all in this record such that we do not use up too many global variable names.

The following global records hold the functions for writing group elements as straight line programs (SLPs) in terms of the generators after successful group recognition. We collect them in these records such that we do not use up too many global variable names.

4.4.5 SLPforElementFuncsPerm

▷ SLPforElementFuncsPerm (global variable)

Stores the SLP functions for permutation groups.

4.4.6 SLPforElementFuncsMatrix

▷ SLPforElementFuncsMatrix (global variable)

Stores the SLP functions for matrix groups.

4.4.7 SLPforElementFuncsProjective

▷ SLPforElementFuncsProjective (global variable)

Stores the SLP functions for projective groups.

4.4.8 SLPforElementFuncsGeneric

▷ SLPforElementFuncsGeneric (global variable)

Stores the SLP functions for generic groups.

Chapter 5

After successful recognition

This chapter explains, what one can do with recognition nodes after a successful recognition (and possibly verification).

Of course, one can inspect the whole tree of recognition nodes just by looking at the stored attribute values. Moreover, constructive membership tests can be performed using the function SLPforElement (3.2.15), thereby writing an arbitrary element in terms of the nice generators, which are stored in the attribute NiceGens (3.2.8). If fail is returned, then the element in question does not lie in the recognised group or the recognition made an error.

Here is an example of a successful recognition tree:

```
──────────────────── Example ────────────────────
gap> g := DirectProduct(SymmetricGroup(12),SymmetricGroup(5));
Group([ (1,2,3,4,5,6,7,8,9,10,11,12), (1,2), (13,14,15,16,17), (13,14) ])
gap> ri := RecogniseGroup(g);
#I  Finished rank 90 method "NonTransitive": success.
#I  Going to the image (depth=0, try=1).
#I  Finished rank 95 method "MovesOnlySmallPoints": success.
#I  Back from image (depth=0).
#I  Calculating preimages of nice generators.
#I  Creating 20 random generators for kernel.
...................
#I  Going to the kernel (depth=0).
#I  Finished rank 80 method "Giant": success.
#I  Back from kernel (depth=0).
<recognition node NonTransitive
 F:<recognition node MovesOnlySmallPoints Size=120>
 K:<recognition node Giant Size=479001600>>
```

One sees that the recursive process runs, first it finds that the permutation action is not transitive, a homomorphism is found by mapping onto the action on one of the orbits. The image is recognised to permute only a few points. The kernel is recognised to be a full symmetric group in its natural action on at least 10 points (recognised as "Giant").

After this, we can write arbitrary group elements in the group g in terms of the nice generators:

```
──────────────────── Example ────────────────────
gap> x := PseudoRandom(g);
(1,12)(2,5,9,11,10,3,4)(7,8)(13,14,16,15,17)
gap> slp := SLPforElement(ri,x);
```

```
  <straight line program>
gap> ResultOfStraightLineProgram(slp,NiceGens(ri));
  (1,12)(2,5,9,11,10,3,4)(7,8)(13,14,16,15,17)
```

Note that this example only works by using also the recog package which contains the necessary recognition methods.

5.1 Functions and methods for recognition nodes

If you need an element explicitly written in terms of the original generators, you can use the following function:

5.1.1 SLPforNiceGens

▷ SLPforNiceGens(*ri*)   (function)

Returns: an SLP expressing the nice generators in the original ones

This function assembles a possibly quite large straight line program expressing the nice generators in terms of the original ones by using the locally stored information in the recognition tree recursively.

You can concatenate straight line programs in the nice generators with the result of this function to explicitly write an element in terms of the original generators.

5.1.2 \in

▷ \in(*x, ri*)   (method)

Returns: `true` or `false`

This method tests, whether the element *x* lies in the group recognised by the recognition node *ri*. Note that this is only a convenience method, in fact `SLPforElement` (3.2.15) is used and the resulting straight line program is thrown away.

5.1.3 Size

▷ Size(*ri*)   (method)

Returns: the size of the recognised group

This method calculates the size of the recognised group by multiplying the size of the image and the kernel recursively. It is assumed that leaf nodes know already or can calculate the size of their group.

5.1.4 DisplayCompositionFactors

▷ DisplayCompositionFactors(*ri*)   (function)

Returns: nothing

This function displays a composition series by using the recursive recognition tree. It only works, if the usual operation `CompositionSeries` (**Reference: CompositionSeries**) works for all leaves. THIS DOES CURRENTLY NOT WORK FOR PROJECTIVE GROUPS AND THUS FOR MATRIX GROUPS!

Chapter 6

Methods for recognition

6.1 Methods for generic groups

The following methods can be equally applied to permutation, matrix and projective groups. We do not refer to them as black-box groups here, as they are allowed to contain code that only works for inputs of the listed types.

6.1.1 FewGensAbelian

This method is used for recognizing permutation groups.

If there are not too may generators (right now that means at most 200), check whether they commute; if yes, dispatch to 'KnownNilpotent', otherwise return NeverApplicable.

6.1.2 KnownNilpotent

This method is unused!

Hint to this method if you know G to be nilpotent or call it directly if you find out so. Note that it will return NeverApplicable if G is a p-group for some prime p. Make sure that the !.projective component is set correctly such that we can set the right Order method.

6.1.3 SnAnUnknownDegree

This method is unused!

This method tries to determine whether the input group given by `ri` is isomorphic to a symmetric group Sn or alternating group An with $11 \leq n$. It is an implementation of [JLNP13].

If `Grp(ri)` is a permutation group, we assume that it is primitive and not a giant (a giant is Sn or An in natural action).

6.1.4 TrivialGroup

This method is used for recognizing permutation, matrix, and projective groups.

This method is successful if and only if all generators of a group `G` are equal to the identity. Otherwise, it returns NeverApplicable indicating that it will never succeed. This method is only installed to handle the trivial case such that we do not have to take this case into account in the other methods.

6.2 Methods for permutation groups

The following table gives an overview over the installed methods and their rank (higher rank means higher priority, the method is tried earlier, see Chapter 4).

1250	`FewGensAbelian`	if very few generators, check IsAbelian and if yes, do KnownNilpotent	6.1.1
1050	`FewGensAbelian`	if very few generators, check IsAbelian and if yes, do KnownNilpotent	6.1.1
300	`TrivialGroup`	go through generators and compare to the identity	6.1.4
100	`ThrowAwayFixedPoints`	try to find a huge amount of (possible internal) fixed points	6.2.11
99	`FewGensAbelian`	if very few generators, check IsAbelian and if yes, do KnownNilpotent	6.1.1
97	`Pcgs`	use a Pcgs to calculate a stabilizer chain	6.2.7
95	`MovesOnlySmallPoints`	calculate a stabilizer chain if only small points are moved	6.2.5
90	`NonTransitive`	try to find non-transitivity and restrict to orbit	6.2.6
80	`Giant`	tries to find Sn and An in their natural actions	6.2.2
70	`Imprimitive`	for a imprimitive permutation group, restricts to block system	6.2.3
60	`LargeBasePrimitive`	recognises large-base primitive permutation groups	6.2.4
55	`StabilizerChainPerm`	for a permutation group using a stabilizer chain via the genss package	6.2.10
50	`StabChain`	for a permutation group using a stabilizer chain	6.2.9

Table: Permutation group find homomorphism methods

6.2.1 `BalTreeForBlocks`

This method is unused!

This method creates a balanced composition tree for the kernel of an imprimitive group. This is guaranteed as the method is just called from `FindHomMethodsPerm`.`'Imprimitive'` and itself. The homomorphism for the split in the composition tree used is induced by the action of `G` on half of its blocks.

6.2.2 `Giant`

This method is used for recognizing permutation groups.

The method tries to determine whether the input group `G` is a giant (that is, A_n or S_n in its natural action on n points). The output is either a data structure D containing nice generators for `G` and a procedure to write an SLP for arbitrary elements of `G` from the nice generators; or `NeverApplicable` if `G` is not transitive; or `fail`, in the case that no evidence was found that `G` is a giant, or evidence was found, but the construction of D was unsuccessful. If the method constructs D then the calling node becomes a leaf.

6.2.3 `Imprimitive`

This method is used for recognizing permutation groups.

If the input group is not known to be transitive then this method returns `NotEnoughInformation`. If the input group is known to be transitive and primitive then the method returns `NeverApplicable`; otherwise, the method tries to compute a nontrivial block system. If successful then a homomorphism to the action on the blocks is defined; otherwise, the method returns `NeverApplicable`.

If the method is successful then it also gives a hint for the children of the node by determining whether the kernel of the action on the block system is solvable. If the answer is yes then the default value 20 for the number of random generators in the kernel construction is increased by the number of blocks.

6.2.4 `LargeBasePrimitive`

This method is used for recognizing permutation groups.

This method tries to determine whether the input group `G` is a fixed-point-free large-base primitive group that neither is a symmetric nor an alternating group in its natural action. This method is an implementation of [LNPS06].

A primitive group H acting on N points is called *large* if there exist n, k, and r with $\binom{N=\{n\}}{k\}^r}$, and up to a permutational isomorphism H is a subgroup of the product action wreath product $S_n \wr S_r$, and an overgroup of $(A_n)^r$ where S_n and A_n act on the k-subsets of $\{1, ..., n\}$. This algorithm recognises fixed-point-free large primitive groups with $r \cdot k > 1$ and $2 \cdot r \cdot k^2 \leq n$.

A large primitive group H of the above type which does have fixed points is handled as follows: if the group H does not know yet that it is primitive, then `ThrowAwayFixedPoints` (6.2.11) returns `NotEnoughInformation`. After the first call to `LargeBasePrimitive`, the group H knows that it is primitive, but since it has fixed points `LargeBasePrimitive` returns `NeverApplicable`. Since `ThrowAwayFixedPoints` (6.2.11) previously returned `NotEnoughInformation`, it will be called again. Then it will use the new information about H being primitive, and is guaranteed to prune away the fixed points and set up a reduction homomorphism. `LargeBasePrimitive` is then applicable to the image of that homomorphism.

If `G` is imprimitive then the output is `NeverApplicable`. If `G` is primitive then the output is either a homomorphism into the natural imprimitive action of `G` on *nr* points with *r* blocks of size *n*, or `TemporaryFailure`, or `NeverApplicable` if no parameters *n*, *k*, and *r* as above exist.

6.2.5 `MovesOnlySmallPoints`

This method is used for recognizing permutation groups.

If a permutation group moves only small points (currently, this means that its largest moved point is at most 10), then this method computes a stabilizer chain for the group via '`StabChain`'. This is because the most convenient way of solving constructive membership in such a group is via a stabilizer chain. In this case, the calling node becomes a leaf node of the composition tree.

If the input group moves a large point (currently, this means a point larger than 10), then this method returns `NeverApplicable`.

6.2.6 `NonTransitive`

This method is used for recognizing permutation groups.

If a permutation group `G` acts nontransitively then this method computes a homomorphism to the action of `G` on the orbit of the largest moved point. If `G` is transitive then the method returns `NeverApplicable`.

6.2.7 `Pcgs`

This method is used for recognizing permutation groups.

This is the GAP library function to compute a stabiliser chain for a solvable permutation group. If the method is successful then the calling node becomes a leaf node in the recursive scheme. If the input group is not solvable then the method returns `NeverApplicable`.

6.2.8 `PcgsForBlocks`

This method is unused!

This method is called after a hint is set in `FindHomMethodsPerm.`'`Imprimitive`'. Therefore, the group `G` preserves a non-trivial block system. This method checks whether or not the restriction of `G` on one block is solvable. If so, then `FindHomMethodsPerm.`'`Pcgs`' is called, and otherwise `NeverApplicable` is returned.

6.2.9 `StabChain`

This method is used for recognizing permutation groups.

This is the randomized GAP library function for computing a stabiliser chain. The method selection process ensures that this function is called only with small-base inputs, where the method works efficiently.

6.2.10 `StabilizerChainPerm`

This method is used for recognizing permutation groups.

TODO

6.2.11 `ThrowAwayFixedPoints`

This method is used for recognizing permutation groups.

This method defines a homomorphism of a permutation group `G` to the action on the moved points of `G` if `G` has any fixed points, and is either known to be primitive or the ratio of fixed points to moved points exceeds a certain threshold. If `G` has fixed points but is not primitive, then it returns `NotEnoughInformation` so that it may be called again at a later time. In all other cases, it returns `NeverApplicable`.

In the current setup, the homomorphism is defined if the number n of moved points is at most $1/3$ of the largest moved point of `G`, or n is at most half of the number of points on which `G` is stored internally by GAP.

The fact that this method returns `NotEnoughInformation` if `G` has fixed points but does not know whether it is primitive, is important for the efficient handling of large-base primitive groups by `LargeBasePrimitive` (6.2.4).

6.3 Methods for matrix groups

The following table gives an overview over the installed methods and their rank (higher rank means higher priority, the method is tried earlier, see Chapter 4). Note that there are not that many methods for matrix groups since the system can switch to projective groups by dividing out the subgroup of scalar matrices. The bulk of the recognition methods are then installed es methods for projective groups.

3100	`TrivialGroup`	go through generators and compare to the identity	6.1.4
1175	`KnownStabilizerChain`	use an already known stabilizer chain for this group	6.3.6
1100	`DiagonalMatrices`	check whether all generators are diagonal matrices	6.3.4
1000	`ReducibleIso`	use the MeatAxe to find invariant subspaces	6.3.9
900	`GoProjective`	divide out scalars and recognise projectively	6.3.5

Table: Matrix group find homomorphism methods

6.3.1 `BlockDiagonal`

This method is unused!

This method is only called when a hint was passed down from the method ‘`BlockLowerTriangular`’. In that case, it knows that the group is in block diagonal form. The method is used both in the matrix- and the projective case.

The method immediately delegates to projective methods handling all the diagonal blocks projectively. This is done by giving a hint to the image to use the method ‘`BlocksModScalars`’. The method for the kernel then has to deal with only scalar blocks, either projectively or with scalars, which is again done by giving a hint to either use ‘`BlockScalar`’ or ‘`BlockScalarProj`’ respectively.

Note that this method is implemented in a way such that it can also be used as a method for a projective group `G`. In that case the recognition node has the `!.projective` component bound to `true` and this information is passed down to image and kernel.

6.3.2 `BlockLowerTriangular`

This method is unused!

This method is only called when a hint was passed down from the method ‘`ReducibleIso`’. In that case, it knows that a base change to block lower triangular form has been performed. The method can then immediately find a homomorphism by mapping to the diagonal blocks. It sets up this homomorphism and gives hints to image and kernel. For the image, the method ‘`BlockDiagonal`’ is used and for the kernel, the method ‘`LowerLeftPGroup`’ is used.

Note that this method is implemented in a way such that it can also be used as a method for a projective group `G`. In that case the recognition node has the `!.projective` component bound to `true` and this information is passed down to image and kernel.

6.3.3 `BlockScalar`

This method is unused!

This method is only called by a hint. Alongside with the hint it gets a block decomposition respected by the matrix group `G` to be recognised and the promise that all diagonal blocks of all group elements will only be scalar matrices. This method recursively builds a balanced tree and does scalar recognition in each leaf.

6.3.4 `DiagonalMatrices`

This method is used for recognizing matrix groups.

This method is successful if and only if all generators of a matrix group `G` are diagonal matrices. Otherwise, it returns `NeverApplicable`.

6.3.5 `GoProjective`

This method is used for recognizing matrix groups.

This method defines a homomorphism from a matrix group `G` into the projective group `G` modulo scalar matrices. In fact, since projective groups in GAP are represented as matrix groups, the homomorphism is the identity mapping and the only difference is that in the image the projective group methods can be applied. The bulk of the work in matrix recognition is done in the projective group setting.

6.3.6 `KnownStabilizerChain`

This method is used for recognizing matrix groups.

If a stabilizer chain is already known, then the kernel node is given knowledge about this known stabilizer chain, and the image node is told to use homomorphism methods from the database for permutation groups. If a stabilizer chain of a parent node is already known this is used for the computation of a stabilizer chain of this node. This stabilizer chain is then used in the same way as above.

6.3.7 `LowerLeftPGroup`

This method is unused!

This method is only called by a hint from '`BlockLowerTriangular`' as the kernel of the homomorphism mapping to the diagonal blocks. The method uses the fact that this kernel is a p-group where p is the characteristic of the underlying field. It exploits this fact and uses this special structure to find nice generators and a method to express group elements in terms of these.

6.3.8 `NaturalSL`

This method is unused!
    TODO

6.3.9 `ReducibleIso`

This method is used for recognizing matrix and projective groups.

This method determines whether a matrix group `G` acts irreducibly. If yes, then it returns `NeverApplicable`. If `G` acts reducibly then a composition series of the underlying module is computed and a base change is performed to write `G` in a block lower triangular form. Also, the method passes a hint to the image group that it is in block lower triangular form, so the image immediately can make recursive calls for the actions on the diagonal blocks, and then to the lower p-part. For the image the method '`BlockLowerTriangular`' is used.

Note that this method is implemented in a way such that it can also be used as a method for a projective group `G`. In that case the recognition node has the `!.projective` component bound to `true` and this information is passed down to image and kernel.

6.3.10 `Scalar`

This method is unused!
    TODO

6.4 Methods for projective groups

The following table gives an overview over the installed methods and their rank (higher rank means higher priority, the method is tried earlier, see Chapter 4). Note that the recognition for matrix group switches to projective recognition rather soon in the recognition process such that most recognition methods in fact are installed as methods for projective groups.

3000	TrivialGroup	go through generators and compare to the identity	6.1.4
1300	ProjDeterminant	find homomorphism to non-zero scalars mod d–th powers	6.4.20
1200	ReducibleIso	use the MeatAxe to find invariant subspaces	6.3.9
1100	NotAbsolutelyIrred	write over a bigger field with smaller degree	6.4.19
1050	ClassicalNatural	check whether it is a classical group in its natural representation	6.4.9
1000	Subfield	write over a smaller field with same degree	6.4.24
900	C3C5	compute a normal subgroup of derived and resolve C3 and C5	6.4.7
850	C6	find either an (imprimitive) action or a symplectic one	6.4.8
840	D247	play games to find a normal subgroup	6.4.11
810	AltSymBBByDegree	try BB recognition for dim+1 and/or dim+2 if sensible	6.4.1
800	TensorDecomposable	find a tensor decomposition	6.4.25
700	FindElmOfEvenNormal	find D2, D4 or D7 by finding an element of an even normal subgroup	6.4.13
600	LowIndex	find an (imprimitive) action on subspaces	6.4.17
580	NameSporadic	generate maximal orders	6.4.18
550	ComputeSimpleSocle	compute simple socle of almost simple group	6.4.10
500	ThreeLargeElOrders	recognise Lie type groups and get its characteristic	6.4.26
400	LieTypeNonConstr	do non-constructive recognition of Lie type groups	6.4.16
100	StabilizerChainProj	last resort: compute a stabilizer chain (projectively)	6.4.23

Table: Projective group find homomorphism methods

6.4.1 AltSymBBByDegree

This method is used for recognizing projective groups.

This method is a black box constructive (?) recognition of alternating and symmetric groups.

This algorithm is probably based on the paper [BLGN$^+$05].

6.4.2 BiggerScalarsOnly

This method is unused!

TODO

6.4.3 BlockScalarProj

This method is unused!

This method is only called by a hint. Alongside with the hint it gets a block decomposition respected by the matrix group `G` to be recognised and the promise that all diagonal blocks of all group elements will only be scalar matrices. This method simply norms the last diagonal block in all generators by multiplying with a scalar and then delegates to `BlockScalar` (see 6.3.3) and matrix group mode to do the recognition.

6.4.4 `Blocks`

This method is unused!
   TODO

6.4.5 `BlocksBackToMats`

This method is unused!
   TODO

6.4.6 `BlocksModScalars`

This method is unused!
   This method is only called when hinted from above. In this method it is understood that G should *neither* be recognised as a matrix group *nor* as a projective group. Rather, it treats all diagonal blocks modulo scalars which means that two matrices are considered to be equal, if they differ only by a scalar factor in *corresponding* diagonal blocks, and this scalar can be different for each diagonal block. This means that the kernel of the homomorphism mapping to a node which is recognised using this method will have only scalar matrices in all diagonal blocks.
   This method does the balanced tree approach mapping to subsets of the diagonal blocks and finally using projective recognition to recognise single diagonal block groups.

6.4.7 `C3C5`

This method is used for recognizing projective groups.
   TODO

6.4.8 `C6`

This method is used for recognizing projective groups.
   This method is designed for the handling of the Aschbacher class C6 (normaliser of an extraspecial group). If the input $G \leq PGL(d, q)$ does not satisfy $d = r^n$ and $r|q - 1$ for some prime r and integer n then the method returns `NeverApplicable`. Otherwise, it returns either a homomorphism of G into $Sp(2n, r)$, or a homomorphism into the C2 permutation action of G on a decomposition of $GF(q)^d$, or `fail`.

6.4.9 `ClassicalNatural`

This method is used for recognizing projective groups.
   TODO

6.4.10 `ComputeSimpleSocle`

This method is used for recognizing projective groups.
   This method randomly computes the non-abelian simple socle and stores it along with additional information if it is called for an almost simple group. Once the non-abelian simple socle is computed the function does not need to be called again for this node and therefore returns `NeverApplicable`.

6.4.11 `D247`

This method is used for recognizing projective groups.
TODO

6.4.12 `DoBaseChangeForBlocks`

This method is unused!
TODO

6.4.13 `FindElmOfEvenNormal`

This method is used for recognizing projective groups.
TODO

6.4.14 `KroneckerKernel`

This method is unused!
TODO

6.4.15 `KroneckerProduct`

This method is unused!
TODO

6.4.16 `LieTypeNonConstr`

This method is used for recognizing projective groups.

Recognise quasi-simple group of Lie type when characteristic is given. Based on [BKPS02] and [AB01].

6.4.17 `LowIndex`

This method is used for recognizing projective groups.

This method is designed for the handling of the Aschbacher class C2 (stabiliser of a decomposition of the underlying vector space), but may succeed on other types of input as well. Given $G \leq PGL(d,q)$, the output is either the permutation action of G on a short orbit of subspaces or `fail`. In the current setup, "short orbit" is defined to have length at most $4d$.

6.4.18 `NameSporadic`

This method is used for recognizing projective groups.

This method returns a list of sporadic simple groups that the group underlying `ri` could be. It does not recognise extensions of sporadic simple groups nor the Monster and the Baby Monster group. It is based on the Magma v2.24.10 function `RecognizeSporadic`.

6.4.19 `NotAbsolutelyIrred`

This method is used for recognizing projective groups.

If an irreducible projective group `G` acts absolutely irreducibly then this method returns `NeverApplicable`. If `G` is not absolutely irreducible then a homomorphism into a smaller dimensional representation over an extension field is defined. A hint is handed down to the image that no test for absolute irreducibility has to be done any more. Another hint is handed down to the kernel indicating that the only possible kernel elements can be elements in the centraliser of `G` in $PGL(d,q)$ that come from scalar matrices in the extension field.

6.4.20 `ProjDeterminant`

This method is used for recognizing projective groups.

The method defines a homomorphism from a projective group $G \leq PGL(d,q)$ to the cyclic group $GF(q)^*/D$, where D is the set of dth powers in $GF(q)^*$. The image of a group element $g \in G$ is the determinant of a matrix representative of g, modulo D.

6.4.21 `PrototypeForC2C4`

This method is unused!
TODO/FIXME: PrototypeForC2C4 is not used anywhere

6.4.22 `SporadicsByOrders`

This method is unused!
This method returns a list of sporadic simple groups that `G` possibly could be. Therefore it checks whether `G` has elements of orders that do not appear in sporadic groups and otherwise checks whether the most common ("killer") orders of the sporadic groups appear. Afterwards it creates hints that come out of a table for the sporadic simple groups.

6.4.23 `StabilizerChainProj`

This method is used for recognizing projective groups.

This method computes a stabiliser chain and a base and strong generating set using projective actions. This is a last resort method since for bigger examples no short orbits can be found in the natural action. The strong generators are the nice generator in this case and expressing group elements in terms of the nice generators ist just sifting along the stabiliser chain.

6.4.24 `Subfield`

This method is used for recognizing projective groups.
TODO

6.4.25 `TensorDecomposable`

This method is used for recognizing projective groups.
TODO/FIXME: it is unclear if the following description actually belongs to this method, so be cautious!

This method currently tries to find one tensor factor by powering up commutators of random elements to elements of prime order. This seems to work quite well provided that the two tensor factors are not "linked" too much such that there exist enough elements that act with different orders on both tensor factors.

This method and its description needs some improvement.

6.4.26 `ThreeLargeElOrders`

This method is used for recognizing projective groups.

In the case when the input group $G \leq PGL(d, p^e)$ is suspected to be simple but not alternating, this method takes the three largest element orders from a sample of pseudorandom elements of G. From these element orders, it tries to determine whether G is of Lie type and the characteristic of G if it is of Lie type. In the case when G is of Lie type of characteristic different from p, the method also provides a short list of the possible isomorphism types of G.

This method assumes that its input is neither alternating nor sporadic and that `ComputeSimpleSocle` (6.4.10) has already been called.

This recognition method is based on the paper [KS09].

6.5 Unused methods

The following table gives an overview over the methods which are currently unused.

KnownNilpotent	FindHomMethodsGeneric	6.1.2
SnAnUnknownDegree	FindHomMethodsGeneric	6.1.3
PcgsForBlocks	FindHomMethodsPerm	6.2.8
BalTreeForBlocks	FindHomMethodsPerm	6.2.1
BlockScalar	FindHomMethodsMatrix	6.3.3
NaturalSL	FindHomMethodsMatrix	6.3.8
Scalar	FindHomMethodsMatrix	6.3.10
BlockLowerTriangular	FindHomMethodsMatrix	6.3.2
BlockDiagonal	FindHomMethodsMatrix	6.3.1
LowerLeftPGroup	FindHomMethodsMatrix	6.3.7
DoBaseChangeForBlocks	FindHomMethodsProjective	6.4.12
Blocks	FindHomMethodsProjective	6.4.4
BlocksModScalars	FindHomMethodsProjective	6.4.6
BlocksBackToMats	FindHomMethodsProjective	6.4.5
KroneckerProduct	FindHomMethodsProjective	6.4.15
KroneckerKernel	FindHomMethodsProjective	6.4.14
BiggerScalarsOnly	FindHomMethodsProjective	6.4.2
PrototypeForC2C4	FindHomMethodsProjective	6.4.21
SporadicsByOrders	FindHomMethodsProjective	6.4.22
BlockScalarProj	FindHomMethodsProjective	6.4.3

Table: Unused group find homomorphism methods

Chapter 7

Examples

TODO

Chapter 8

Renaming

Many of the names in the recog package were found to be unintuitive or inconsistent with other names. For these reasons the names were changed to be more descriptive and to follow the internal consistency of GAP. In this chapter a dictionary from old to new names is included. The meanings of the old names for components of a recognition node are described in Section 2.6 "Components of the recognition node" of the article [NS06].

Old Name	*New Name*
`RecognitionInfoFamily`	`RecogNodeFamily` (3.2.1)
`IsRecognitionInfo`	`IsRecogNode` (3.2.2)
`RIFac`	`ImageRecogNode` (3.2.10)
`RIKer`	`KernelRecogNode` (3.2.11)
`RIParent`	`ParentRecogNode` (3.2.12)
`methodsforfactor`	`methodsforimage` (3.2.17)
`forfactor`	`InitialDataForImageRecogNode` (3.2.33)
`forkernel`	`InitialDataForKernelRecogNode` (3.2.32)

Table: Renaming

Chapter 9

How to write a recognition method

This chapter explains how to integrate a newly developed group recognition method into the framework provided by the recog package.

TODO: Refer to Chapter 4 for an explanation of methods. There are leaf methods and split methods. The next two sections describe how to implement leaf and split methods respectively, and include example code.

9.1 Leaf methods

A leaf method must at the very least do the following, examples will be provided below (TODO: add a reference):

- Provide the order of the recognized group via `SetSize(ri, NNN)`.

- Provide a set of SLPs which map the original generators X to the nice generators Y, as entry for the attribute `slptonice`.

- Provide a function which maps any element $g \in G$ to a corresponding SLP in terms of the nice generators Y, as entry for the attribute `slpforelement`.

- Call `SetFilterObj(ri, IsLeaf);` to mark the node as a leaf node.

There are further values that can be provided, in particular to speed up computations; we'll come back to that later. Let's first look at an example: The following method is used by recog to recognize trivial groups, as a base case for the recursive group recognition algorithm. It works for arbitrary groups.

TODO: AutoDoc inserts extra paragraph commands here:

```
———————————————————————————————— Code ————————————————————————————————
  BindRecogMethod(FindHomMethodsGeneric, "TrivialGroup",
  "go through generators and compare to the identity",
  function(ri, G)
    local gens;
    # get the generators of the group
    gens := GeneratorsOfGroup(G);
<P/>
    # check whether all generators are trivial
    # ri!.isone is explained below
```

```
    if not ForAll(gens, ri!.isone) then
      # NeverApplicable because it makes
      # no sense to call this method again
      return NeverApplicable;
    fi;
<P/>
    # The group is trivial! Provide required information:
<P/>
    # size of the group
    SetSize(ri, 1);
<P/>
    # explained below
    Setslpforelement(ri, SLPforElementFuncsGeneric.TrivialGroup);
<P/>
    # SLP from given generators to nice generators
    Setslptonice(ri, StraightLineProgramNC([[[1,0]]],
                     Length(gens)));
<P/>
    # We have reached a leaf node.
    SetFilterObj(ri, IsLeaf);
    return Success;
end);
```

The input is in the format described above (TODO), and the return value is "Success".
Two more comments:

- When we check whether all generators are the identity, we call `ri!.isone`, instead of `IsOne`. The reason for this is the need to support *projective groups*. For permutation groups and matrix groups, `ri!.isone` is simply defined to be `IsOne`. For projective groups, it is set to `IsOneProjective`, which can be read as "is one modulo scalars".

- The function `SLPforElementFuncs.TrivialGroup` takes `ri` as well as an element g as input. If $g \in G$, then it is supposed to return an SLP for g in terms of the nice gens Y. Otherwise it returns `fail`. Here is the concrete implementation:

```
──────── Code ────────
SLPforElementFuncsGeneric.TrivialGroup := function(ri,g)
    if not ri!.isone(g) then
        return fail;
    fi;
    return StraightLineProgramNC( [ [1,0] ], 1 );
end;
```

Finally, we need to let recog know about this new recognition method. This is done via the `AddMethod` function. Another example!

```
──────── Code ────────
AddMethod(FindHomDbPerm, FindHomMethodsGeneric.TrivialGroup, 300);
```

TODO: refer to the `AddMethod` documentation instead. Also this is outdated now. The function `AddMethod` takes four mandatory arguments `db`, `meth`, `rank`, `stamp`, and an optional fifth argument `comment`. Their meaning is as follows:

- `db` is the "method database", and determines to which type of groups the methods should be applied. Allowed values are:

  - `FindHomDbPerm`

  - `FindHomDbMatrix`

  - `FindHomDbProj`

- `meth` is the recognition method we have defined. In our example this is `FindHomMethodsGeneric.TrivialGroup`.

- `rank` is the relative rank of the recognition method, given as an integer. The idea is that methods with a high rank get called before methods with a low rank, so [recog] tries recognition methods starting from the highest rank. What the "right" rank for a given method is depends on which other methods exist and what their ranks are. As a rule of thumb, methods which are either very fast or very likely to be applicable should be tried before slower methods, or methods which are less likely to be relevant.

- `stamp` holds a string value that uniquely describes the method. This is used for bookkeeping. It is also used in the manual, for printing the recognition tree, and for debugging purposes.

- `comment` is a string valued comment which in the example above has been used to explain what the method does. This argument is optional and can be left out.

Note that above, we only installed our method into `FindHomDbPerm`. But in recog, it is actually also installed for matrix and projective groups. We reproduce the corresponding `AddMethod` calls here. Note that the ranks differ, so the same method can be called with varying priority depending on the type of group.

```
————————————————— Code —————————————————
AddMethod(FindHomDbMatrix, FindHomMethodsGeneric.TrivialGroup, 3100);
```

```
————————————————— Code —————————————————
AddMethod(FindHomDbProjective, FindHomMethodsGeneric.TrivialGroup, 3000);
```

- TODO: more advanced example?

- TODO: also explain how verification works

- TODO: we need something that demonstrates the other two return values (Oh yes, good point.)

9.2 Elements with memory

When using the memory of group elements, one currently has to always access ri!.gensHmem instead of doing `GroupWithMemory(Grp(ri))`. Namely, many functions for objects with memory assume that, if the elements live in the same group, then their `!.slp` components are identical.

9.3 Splitting methods

Recall that splitting recognition methods produce an epimorphism $\phi : G \to H$ and then delegate the work to the image H and the kernel $N := \ker(\phi)$. This means that now N and H have to be constructively recognized. Such a splitting recognition method only needs to provide a homomorphism, by calling SetHomom(ri, hom);. However, in practice one will want to provide additional data.

We start with an example, similar to a method used in recog. This refers to permutation groups only!

```
————————————————————————— Code ————————————————————————
  BindRecogMethod(FindHomMethodsPerm, "NonTransitive",
  "try to find non-transitivity and restrict to orbit",
  rec(validatesOrAlwaysValidInput := true),
  function(ri, G)
      local hom,la,o;
  <P/>
      # test whether we can do something:
      if IsTransitive(G) then
          return NeverApplicable;
      fi;
  <P/>
      # compute orbit of the largest moved point
      la := LargestMovedPoint(G);
      o := Orb(G,la,OnPoints);
      Enumerate(o);
      # compute homomorphism into Sym(o), i.e, restrict
      # the permutation action of G to the orbit o
      hom := OrbActionHomomorphism(G,o);
      # TODO: explanation
      Setvalidatehomominput(ri, {ri,p} -> ForAll(o, x -> (x^p in o)));
      # store the homomorphism into the recognition node
      SetHomom(ri,hom);
  <P/>
      # TODO: explanation
      Setimmediateverification(ri, true);
  <P/>
      # indicate success
      return Success;
  end);
```

TODO Alternatively use this:

```
———————————————————————— Example ———————————————————————
  FindHomMethodsPerm.NonTransitive := function(ri, G)
    local hom, la, o;

    # test whether we can do something:
    if IsTransitive(G) then
      # the action is transitive, so we can't do
      # anything, and there is no point in calling us again.
      return NeverApplicable;
    fi;
```

```
   # compute orbit of the largest moved point
   la := LargestMovedPoint(G);
   o := Orbit(G, la, OnPoints);

   # compute homomorphism into Sym(o), i.e, restrict
   # the permutation action of G to the orbit o
   hom := ActionHomomorphism(G, o);

   # store the homomorphism into the recognition node
   SetHomom(ri, hom);

   # indicate success
   return Success;
end;
```

———————————————————— Code ————————————————————
```
AddMethod(FindHomDbPerm, FindHomMethodsPerm.NonTransitive, 90);
```

TODO: More complex example:

———————————————————— Example ————————————————————
```
FindHomMethodsMatrix.BlockLowerTriangular := function(ri, G)
  # This is only used coming from a hint, we know what to do:
  # A base change was done to get block lower triangular shape.
  # We first do the diagonal blocks, then the lower p-part:
  local H, data, hom, newgens;

  # we need to construct a homomorphism, but to defined it,
  # we need the image, but of course the image is defined in
  # terms of the homomorphism... to break this cycle, we do
  # the following: we first map the input generators using
  # the helper function RECOG.HomOntoBlockDiagonal; this
  # function is later also used as the underlying mapping
  # of the homomorphism.
  data := rec( blocks := ri!.blocks );
  newgens := List(GeneratorsOfGroup(G),
                  x -> RECOG.HomOntoBlockDiagonal(data, x));
  Assert(0, not fail in newgens);

  # now that we have the images of the generators, we can
  # defined the image group
  H := Group(newgens);

  # finally, we define the homomorphism
  hom := GroupHomByFuncWithData(G, H, RECOG.HomOntoBlockDiagonal, data);

  # ... and store it in the recognition node
  SetHomom(ri, hom);

  # since we know exactly what kind of group we are looking
  # at, we don't want to run generic recognition on the
  # image group and the kernel. So we provide "hints" to
  # ensure more appropriate recognition methods are applied
  # first.
```

```
  # Give hint to image
  InitialDataForImageRecogNode(ri).blocks := ri!.blocks;
  Add(InitialDataForImageRecogNode(ri).hints,
      rec( method := FindHomMethodsMatrix.BlockDiagonal,
           rank := 2000,
           stamp := "BlockDiagonal" ) );

  # Tell recog that we have a better method for finding kernel
  findgensNmeth(ri).method := FindKernelLowerLeftPGroup;
  findgensNmeth(ri).args := [];

  # Give hint to kernel N
  Add(InitialDataForKernelRecogNode(ri).hints,
      rec( method := FindHomMethodsMatrix.LowerLeftPGroup,
           rank := 2000,
           stamp := ''LowerLeftPGroup" ));
  InitialDataForKernelRecogNode(ri).blocks := ri!.blocks;

  # This function always succeeds, because it is only
  # called for inputs for which it is known to apply.
  return Success;
end;
```

References

[AB01]   Christine Altseimer and Alexandre V. Borovik. Probabilistic recognition of orthogo-
              nal and symplectic groups. In *Groups and computation, III (Columbus, OH, 1999)*,
              volume 8, page 1–20. de Gruyter, Berlin, 2001. 40

[BB99]   László Babai and Robert Beals. A polynomial-time theory of black box groups. I. In
              Groups St. Andrews 1997 in Bath, I, volume 260 of *London Math. Soc. Lecture Note
              Ser.*, page 30–64. Cambridge Univ. Press, Cambridge, 1999. 6

[BBS09]   László Babai, Robert Beals, and Ákos Seress. Polynomial-time theory of matrix
              groups. In *STOC'09—Proceedings of the 2009 ACM International Symposium on The-
              ory of Computing*, page 55–64. ACM, New York, 2009. 6

[BHLGO15]   Henrik Bäärnhielm, Derek Holt, C. R. Leedham-Green, and E. A. O'Brien. A practical
              model for computation with matrix groups. *J. Symbolic Comput.*, 68(part 1):27–60,
              2015. https://doi.org/10.1016/j.jsc.2014.08.006. 6

[BK01]   Peter A. Brooksbank and William M. Kantor. On constructive recognition of a black
              box PSL(d,q). In *Groups and computation, III (Columbus, OH, 1999)*, volume 8 of
              Ohio State Univ. Math. Res. Inst. Publ., page 95–111. de Gruyter, Berlin, 2001. 6

[BK06]   Peter A. Brooksbank and William M. Kantor. Fast constructive recogni-
              tion of black box orthogonal groups. *J. Algebra*, 300(1):256–288, 2006.
              https://doi.org/10.1016/j.jalgebra.2006.02.024. 6

[BKPS02]   László Babai, William M. Kantor, Péter P. Pálfy, and Ákos Seress. Black-box recogni-
              tion of finite simple groups of lie type by statistics of element orders. *J. Group Theory*,
              5(4):383–401, 2002. https://doi.org/10.1515/jgth.2002.010. 40

[BLGN⁺03]   Robert Beals, Charles R. Leedham-Green, Alice C. Niemeyer, Cheryl E. Praeger,
              and Ákos Seress. A black-box group algorithm for recognizing finite symmet-
              ric and alternating groups. I. *Trans. Amer. Math. Soc.*, 355(5):2097–2113, 2003.
              https://doi.org/10.1090/S0002-9947-03-03040-X. 6

[BLGN⁺05]   Robert Beals, Charles R. Leedham-Green, Alice C. Niemeyer, Cheryl E. Praeger, and
              Ákos Seress. Constructive recognition of finite alternating and symmetric groups act-
              ing as matrix groups on their natural permutation modules. *J. Algebra*, 292(1):4–46,
              2005. https://doi.org/10.1016/j.jalgebra.2005.01.035. 6, 38

[BLS97]   László Babai, Eugene M. Luks, and Ákos Seress. Fast management
              of permutation groups. I. *SIAM J. Comput.*, 26(5):1310–1342, 1997.
              https://doi.org/10.1137/S0097539794229417. 6

[BNS06]   Peter Brooksbank, Alice C. Niemeyer, and Ákos Seress. A reduction algorithm for matrix groups with an extraspecial normal subgroup. In *Finite geometries, groups, and computation*, page 1–16. Walter de Gruyter, Berlin, 2006. 6

[Bro01]   Peter A. Brooksbank. A constructive recognition algorithm for the matrix group $\Omega(d,q)$. In *Groups and computation, III (Columbus, OH, 1999)*, volume 8 of *Ohio State Univ. Math. Res. Inst. Publ.*, page 79–93. de Gruyter, Berlin, 2001. 6

[Bro03]   Peter A. Brooksbank. Fast constructive recognition of black-box unitary groups. *LMS J. Comput. Math.*, 6:162–197, 2003. https://doi.org/10.1112/S1461157000000437. 6

[Bro08]   Peter A. Brooksbank. Fast constructive recognition of black box symplectic groups. *J. Algebra*, 320(2):885–909, 2008. https://doi.org/10.1016/j.jalgebra.2008.03.021. 6

[BS01]   László Babai and Aner Shalev. Recognizing simplicity of black-box groups and the frequency of *p*-singular elements in affine groups. In *Groups and computation, III (Columbus, OH, 1999)*, volume 8 of *Ohio State Univ. Math. Res. Inst. Publ.*, page 39–62. de Gruyter, Berlin, 2001. 6

[CFL97]   Gene Cooperman, Larry Finkelstein, and Steve Linton. Constructive recognition of a black box group isomorphic to GL($n,2$). In *Groups and computation, II (New Brunswick, NJ, 1995)*, volume 28 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, page 85–100. Amer. Math. Soc., Providence, RI, 1997. 6

[CLG97a]   Frank Celler and C. R. Leedham-Green. Calculating the order of an invertible matrix. In *Groups and computation, II (New Brunswick, NJ, 1995)*, volume 28 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, page 55–60. Amer. Math. Soc., Providence, RI, 1997. 6

[CLG97b]   Frank Celler and C. R. Leedham-Green. A non-constructive recognition algorithm for the special linear and other classical groups. In *Groups and computation, II (New Brunswick, NJ, 1995)*, volume 28 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, page 61–67. Amer. Math. Soc., Providence, RI, 1997. 6

[CLG98]   F. Celler and C. R. Leedham-Green. A constructive recognition algorithm for the special linear group. In *The atlas of finite groups: ten years on (Birmingham, 1995)*, volume 249 of *London Math. Soc. Lecture Note Ser.*, page 11–26. Cambridge Univ. Press, Cambridge, 1998. https://doi.org/10.1017/CBO9780511565830.007. 6

[CLG01]   Marston Conder and Charles R. Leedham-Green. Fast recognition of classical groups over large fields. In *Groups and computation, III (Columbus, OH, 1999)*, volume 8 of *Ohio State Univ. Math. Res. Inst. Publ.*, page 113–121. de Gruyter, Berlin, 2001. 6

[CLGM$^+$95]   Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995. https://doi.org/10.1080/00927879508825509. 6

[CLGO06]   M. D. E. Conder, C. R. Leedham-Green, and E. A. O'Brien. Constructive recognition of PSL(2,q). *Trans. Amer. Math. Soc.*, 358(3):1203–1221, 2006. https://doi.org/10.1090/S0002-9947-05-03756-6. 6

[CNRD09]   Jon F. Carlson, Max Neunhöffer, and Colva M. Roney-Dougal. A polynomial-time reduction algorithm for groups of semilinear or subfield class. *J. Algebra*, 322(3):613–637, 2009. https://doi.org/10.1016/j.jalgebra.2009.04.022. 6

[DLGLO13] Heiko Dietrich, C. R. Leedham-Green, Frank Lübeck, and E. A. O'Brien. Constructive recognition of classical groups in even characteristic. *J. Algebra*, 391:227–255, 2013. https://doi.org/10.1016/j.jalgebra.2013.04.031. 6

[DLGO15]   Heiko Dietrich, C. R. Leedham-Green, and E. A. O'Brien. Effective black-box constructive recognition of classical groups. *J. Algebra*, 421:460–492, 2015. https://doi.org/10.1016/j.jalgebra.2014.08.039. 6

[GH97]   S. P. Glasby and R. B. Howlett. Writing representations over minimal fields. *Comm. Algebra*, 25(6):1703–1711, 1997. https://doi.org/10.1080/00927879708825947. 6

[GLGO06]   S. P. Glasby, C. R. Leedham-Green, and E. A. O'Brien. Writing projective representations over subfields. *J. Algebra*, 295(1):51–61, 2006. https://doi.org/10.1016/j.jalgebra.2005.03.037. 6

[HLGOR96a] Derek F. Holt, C. R. Leedham-Green, E. A. O'Brien, and Sarah Rees. Computing matrix group decompositions with respect to a normal subgroup. *J. Algebra*, 184(3):818–838, 1996. https://doi.org/10.1006/jabr.1996.0286. 6

[HLGOR96b] Derek F. Holt, C. R. Leedham-Green, E. A. O'Brien, and Sarah Rees. Testing matrix groups for primitivity. *J. Algebra*, 184(3):795–817, 1996. https://doi.org/10.1006/jabr.1996.0285. 6

[HLO+08]   P. E. Holmes, S. A. Linton, E. A. O'Brien, A. J. E. Ryba, and R. A. Wilson. Constructive membership in black-box groups. *J. Group Theory*, 11(6):747–763, 2008. https://doi.org/10.1515/JGT.2008.047. 6

[HR94]   Derek F. Holt and Sarah Rees. Testing modules for irreducibility. *J. Austral. Math. Soc. Ser. A*, 57(1):1–16, 1994. 6

[IL00]   Gábor Ivanyos and Klaus Lux. Treating the exceptional cases of the MeatAxe. *Experiment. Math.*, 9(3):373–381, 2000. http://projecteuclid.org/euclid.em/1045604672. 6

[JLNP13]   Sebastian Jambor, Martin Leuner, Alice C. Niemeyer, and Wilhelm Plesken. Fast recognition of alternating groups of unknown degree. *J. Algebra*, 392:315–335, 2013. https://doi.org/10.1016/j.jalgebra.2013.06.005. 6, 32

[KK15]   William M. Kantor and Martin Kassabov. Black box groups isomorphic to $PGL(2, 2^e)$. *J. Algebra*, 421:16–26, 2015. https://doi.org/10.1016/j.jalgebra.2014.08.014. 6

[KM13]   W. M. Kantor and K. Magaard. Black box exceptional groups of Lie type. *Trans. Amer. Math. Soc.*, 365(9):4895–4931, 2013. https://doi.org/10.1090/S0002-9947-2013-05822-9. 6

[KM15]   William M. Kantor and Kay Magaard. Black box exceptional groups of Lie type II. *J. Algebra*, 421:524–540, 2015. https://doi.org/10.1016/j.jalgebra.2014.09.003. 6

[KS09]   William M. Kantor and Ákos Seress.   Large element orders and the char-
          acteristic of Lie-type simple groups.   *J. Algebra*, 322(3):802–832, 2009.
          https://doi.org/10.1016/j.jalgebra.2009.05.004. 42

[LG01]   Charles R. Leedham-Green. The computational matrix group project. In *Groups and
          computation, III (Columbus, OH, 1999)*, volume 8 of *Ohio State Univ. Math. Res. Inst.
          Publ.*, page 229–247. de Gruyter, Berlin, 2001. 6

[LGO97a] C. R. Leedham-Green and E. A. O'Brien.   Recognising tensor products
          of matrix groups.   *Internat. J. Algebra Comput.*, 7(5):541–559, 1997.
          https://doi.org/10.1142/S0218196797000241. 6

[LGO97b] C. R. Leedham-Green and E. A. O'Brien. Tensor products are projective geometries.
          J. Algebra, 189(2):514–528, 1997. https://doi.org/10.1006/jabr.1996.6881. 6

[LGO02]   C. R. Leedham-Green and E. A. O'Brien. Recognising tensor-induced matrix groups.
          J. Algebra, 253(1):14–30, 2002. https://doi.org/10.1016/S0021-8693(02)00041-8. 6

[LGO09]   C. R. Leedham-Green and E. A. O'Brien.   Constructive recognition of
          classical groups in odd characteristic.   *J. Algebra*, 322(3):833–881, 2009.
          https://doi.org/10.1016/j.jalgebra.2009.04.028. 6

[LMO07]   F. Lübeck, K. Magaard, and E. A. O'Brien. Constructive recognition of $SL_3(q)$. *J.
          Algebra*, 316(2):619–633, 2007. https://doi.org/10.1016/j.jalgebra.2007.01.020. 6

[LNPS06] Maska Law, Alice C. Niemeyer, Cheryl E. Praeger, and Ákos Seress. A reduction algo-
          rithm for large-base primitive permutation groups. *LMS J. Comput. Math.*, 9:159–173,
          2006. https://doi.org/10.1112/S1461157000001236. 6, 34

[LO07]   Martin W. Liebeck and E. A. O'Brien. Finding the characteristic of a group of Lie type.
          J. Lond. Math. Soc. (2), 75(3):741–754, 2007. https://doi.org/10.1112/jlms/jdm028. 6

[LO16]   Martin W. Liebeck and E. A. O'Brien.   Recognition of finite exceptional
          groups of Lie type.   *Trans. Amer. Math. Soc.*, 368(9):6189–6226, 2016.
          https://doi.org/10.1090/tran/6534. 6

[Neu09]   Max Neunhöffer. *Constructive Recognition of Finite Groups*. Habilitation thesis,
          RWTH Aachen, 2009. https://github.com/neunhoef/habil. 6

[Nie05]   Alice C. Niemeyer.   Constructive recognition of normalizers of small extra-
          special matrix groups.   *Internat. J. Algebra Comput.*, 15(2):367–394, 2005.
          https://doi.org/10.1142/S021819670500230X. 6

[NP92]   Peter M. Neumann and Cheryl E. Praeger.   A recognition algorithm for
          special linear groups.   *Proc. London Math. Soc. (3)*, 65(3):555–603, 1992.
          https://doi.org/10.1112/plms/s3-65.3.555. 6

[NP97]   Alice C. Niemeyer and Cheryl E. Praeger. Implementing a recognition algorithm for
          classical groups. In *Groups and computation, II (New Brunswick, NJ, 1995)*, volume 28
          of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, page 273–296. Amer. Math.
          Soc., Providence, RI, 1997. 6

[NP98]   Alice C. Niemeyer and Cheryl E. Praeger. A recognition algorithm for classical groups over finite fields. *Proc. London Math. Soc. (3)*, 77(1):117–169, 1998. https://doi.org/10.1112/S0024611598000422. 6

[NP99]   Alice C. Niemeyer and Cheryl E. Praeger. A recognition algorithm for non-generic classical groups over finite fields. *J. Austral. Math. Soc. Ser. A*, 67(2):223–253, 1999. 6

[NS06]   Max Neunhöffer and Ákos Seress. A data structure for a uniform approach to computations with finite groups. In *ISSAC 2006*, page 254–261. ACM, New York, 2006. https://doi.org/10.1145/1145768.1145811. 5, 44

[O'B06]   E. A. O'Brien. Towards effective algorithms for linear groups. In *Finite geometries, groups, and computation*, page 163–190. Walter de Gruyter, Berlin, 2006. 6

[O'B11]   E. A. O'Brien. Algorithms for matrix groups. In *Groups St Andrews 2009 in Bath. Volume 2*, volume 388 of *London Math. Soc. Lecture Note Ser.*, page 297–323. Cambridge Univ. Press, Cambridge, 2011. 6

[Pak00]   Igor Pak. The product replacement algorithm is polynomial. In *41st Annual Symposium on Foundations of Computer Science (Redondo Beach, CA, 2000)*, page 476–485. IEEE Comput. Soc. Press, Los Alamitos, CA, 2000. https://doi.org/10.1109/SFCS.2000.892135. 6

[Par84]   R. A. Parker. The computer calculation of modular characters (the meat-axe). In *Computational group theory (Durham, 1982)*, page 267–274. Academic Press, London, 1984. 6

[Pra99]   Cheryl E. Praeger. Primitive prime divisor elements in finite classical groups. In *Groups St. Andrews 1997 in Bath, II*, volume 261 of *London Math. Soc. Lecture Note Ser.*, page 605–623. Cambridge Univ. Press, Cambridge, 1999. https://doi.org/10.1017/CBO9780511666148.024. 6

[Ser03]   Ákos Seress. *Permutation group algorithms*, volume 152 of *Cambridge Tracts in Mathematics*. Cambridge University Press, Cambridge, 2003. https://doi.org/10.1017/CBO9780511546549. 6

Index